Лабораторная работа №4
Тема: «Управление потоками»

Теоретическая часть
Создание потоков в приложении

Все потоки мультипрограммного вычислительного процесса должны иметь доступ к системным ресурсам — кучам, портам, файлам, окнам и т.д. Потоки взаимодействуют друг с другом в двух основных случаях: если совместно используют разделяемый ресурс либо уведомляют друг друга о завершении каких-либо операций. С вопросами синхронизации потоков связана обширная тематика, которую мы частично рассмотрим. Предварительно нужно научиться создавать многопоточные процессы. Наиболее просто это сделать, используя современные системы программирования, например MS Visual Studio 2017 и языковые средства, предоставляемые этой системой.

Для создания нового потока в среде .NET нужно создать экземпляр класса Thread. Конструктор этого класса принимает один параметр — экземпляр делегата ThreadStart. Этот делегат представляет метод, который будет вызываться для выполнения операций созданного потока. Шаблон для создания потока выполнения выглядит следующим образом:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;
namespace ConsoleApplication1
{
    class Create_Thread
    {
        // Метод Countdown считает от 1000 до 1
        public static void Countdown
        {
            for (int counter = 1000; counter > 0; counter--)
                Console.Write(counter.ToString() + "|");
            Console.WriteLine();
        }
        public static void Method()
        {
            {
                // Тело метода
            }
        }
        static void Main()
        {
            // Пример создания потоков
                Thread t1 = new Thread(new ThreadStart(Method));
                // Объявление потока с «привязкой» к методу Method()
                // Создание второго потока
                Thread t2 = new Thread(new ThreadStart(Countdown));     // Привязка потока к методу
                // Запуск потока
                t1.Start();
                t2.Start();
                Countdown();  // В это же время метод Countdown вызывается в основном потоке
                t1.Abort();
                t2.Abort();
            }
        }
    }
```

В этой программе создается поток с именем «t2» и вызывается метод Start. После того как метод Start вызван, начинается выполнение метода Countdown. В то же время метод Countdown вызывается из основного потока. Основная идея этого примера — заставить два метода Countdown выполняться одновременно. Однако результаты с каждым запуском программы будут отличаться от предыдущих (рис. 1, 2). Связано это с тем, что поток и метод используют общий ресурс – дисплей компьютера.

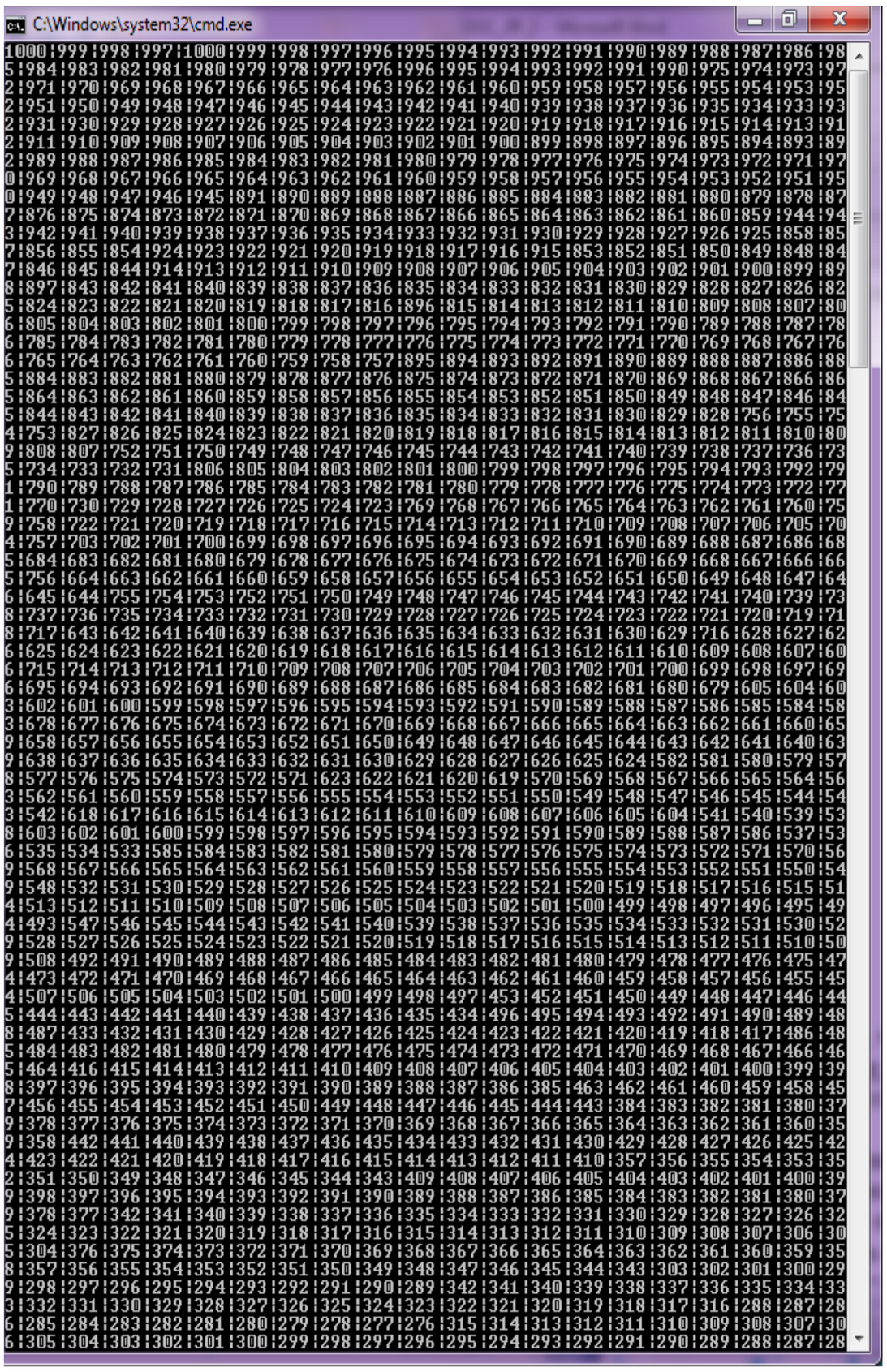

Рис. 1

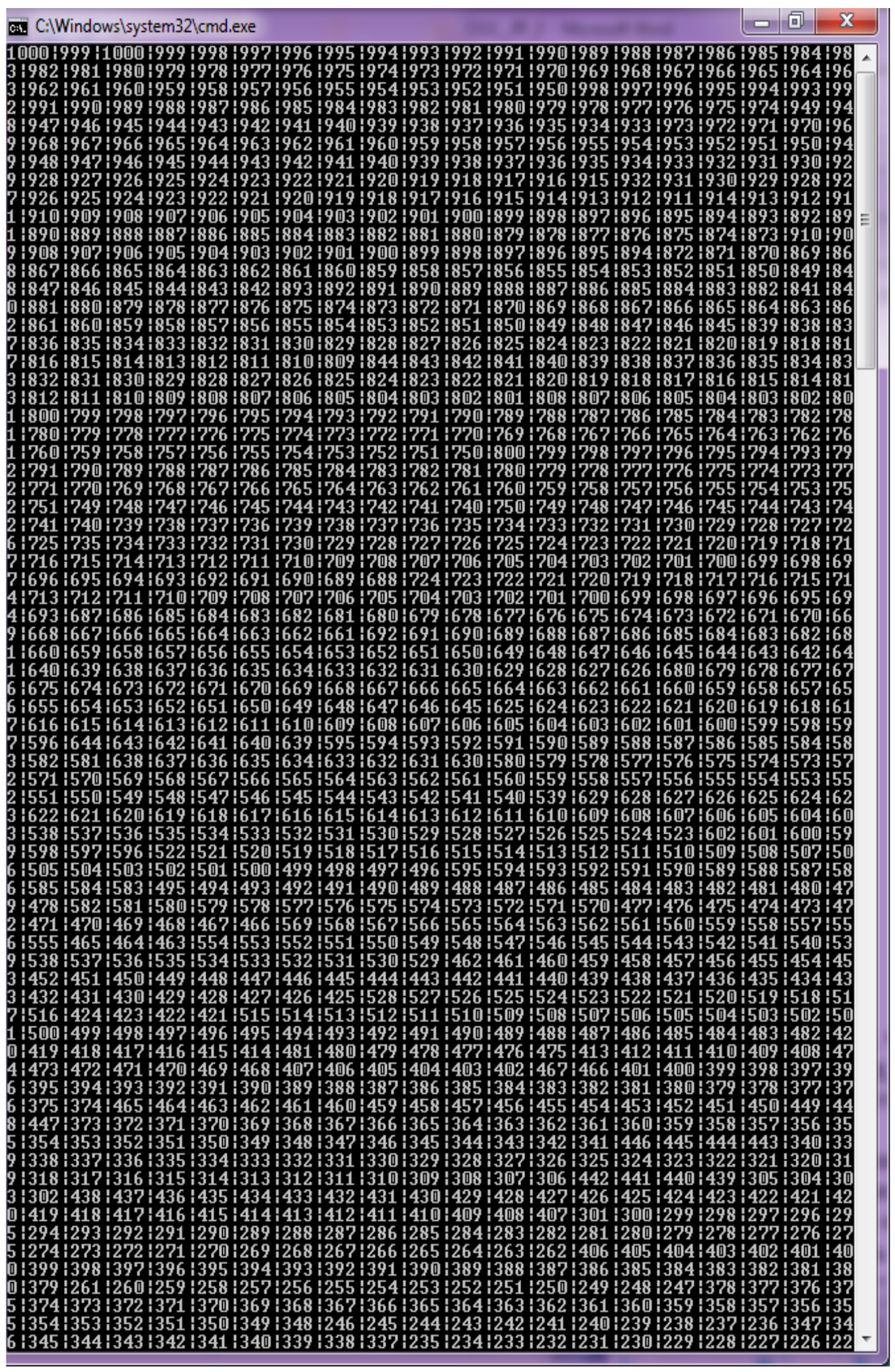

Рис. 2

Приоритеты потоков

Один из основных принципов многопоточной архитектуры заключается в том, что не все потоки имеют одинаковые приоритеты. Несмотря на то что все потоки создаются одинаковыми (с приоритетом Normal), не всегда целесообразно оставлять приоритеты потоков, установленные при их создании.

Например, если производится фоновая печать одновременно с редактированием документа, то ясно, что отдавать половину вычислительных ресурсов процессу печати невыгодно. Намного лучше отдавать большее количество ресурсов основному приложению — в этом случае оно будет быстрее реагировать на запросы пользователя, а печать все равно закончится, может быть несколько позже.

Чтобы выполнить подобное перераспределение ресурсов, используется свойство Priority, которое имеет пять допустимых значений:

1) Lowest (низший);
2) BelowNormal (ниже нормального);
3) Normal (нормальный);
4) AboveNormal (выше нормального);
5) Highest (наивысший).

Следует иметь в виду, что среда .NET изначально проектировалась в расчете на перенос на несколько платформ, поэтому перечисленные значения приоритетов могут не совпадать со значениями конкретной ОС. Например, в Windows 2000 существует шесть основных приоритетов, а

в Windows CE 3.0 — 256 градаций приоритета, обозначаемых числами от 0 до 255.

Если изменить приоритет потока t2 (из предыдущей программы), к примеру, на «Высокий» (Highest), тогда он будет выполняться быстрее потока, который является основным (рис. 3, 4)

```
    // Пример потоков с приоритетами
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;
namespace ConsoleApplication1
{
   class Prior  Thread
{
   public static void Countdown() // Метод Countdown считает от 1000 до 1
   {
      for (int counter = 400; counter > 0; counter--)
         Console.Write(counter.ToString() + " | ");
      Console.WriteLine();
   }
   static void Main()
   {
      // Создание потока
      Thread t2 = new Thread(new ThreadStart(Countdown));    // Привязка потока к методу
      // Установка потоку t2 высшего приоритета
      t2.Priority = ThreadPriority.Highest;
      // Установка текущему потоку низшего приоритета
      Thread.CurrentThread.Priority = ThreadPriority.Lowest;
      // Запуск потока
      t2.Start();
      // В это же время метод Countdown вызывается в основном потоке
      for (int counter = 400; counter > 0; counter--)
         Console.Write(counter.ToString() + " C ");
      Console.WriteLine();
      t2.Abort();
      }
   }
}
```

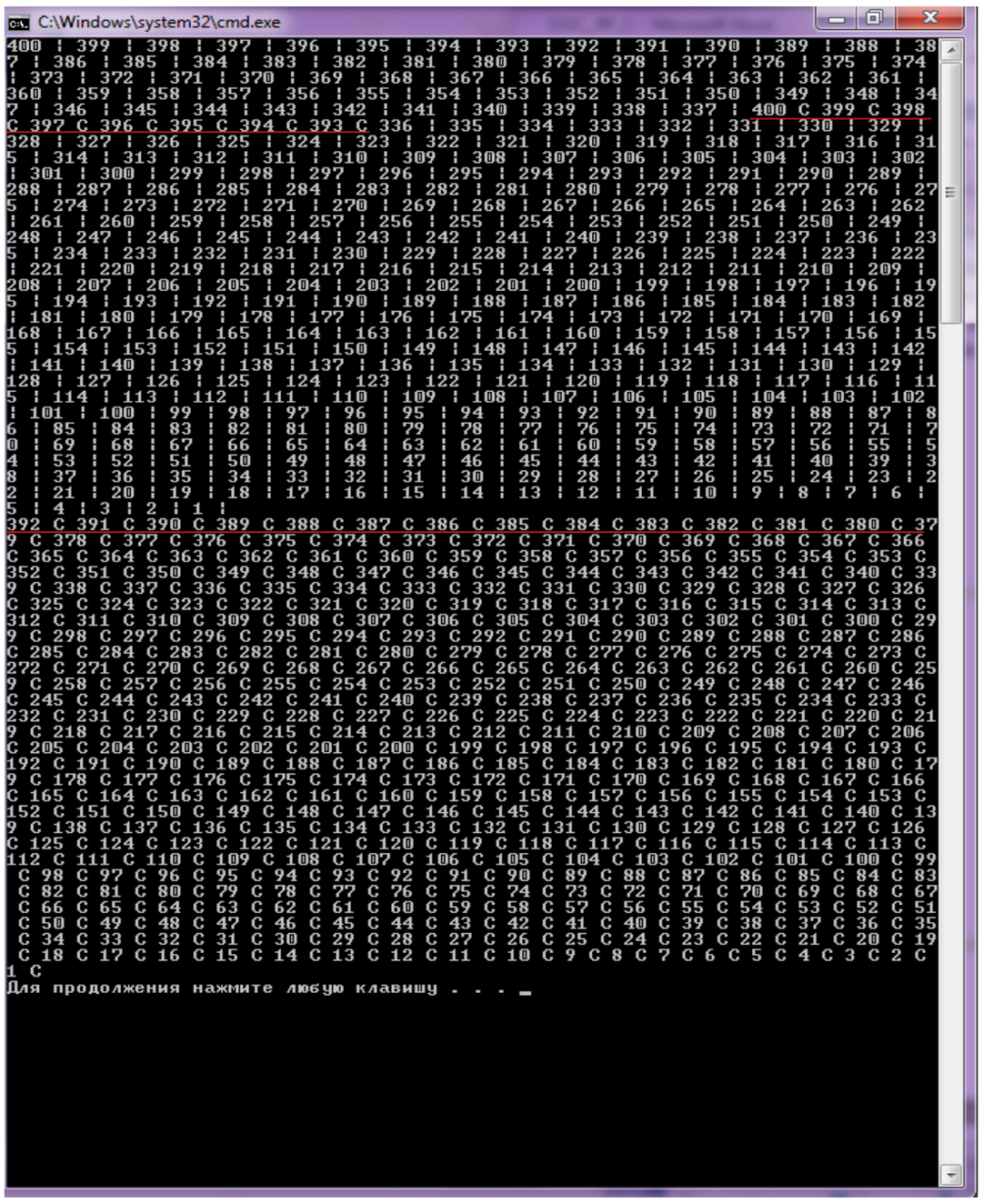

Рис. 3

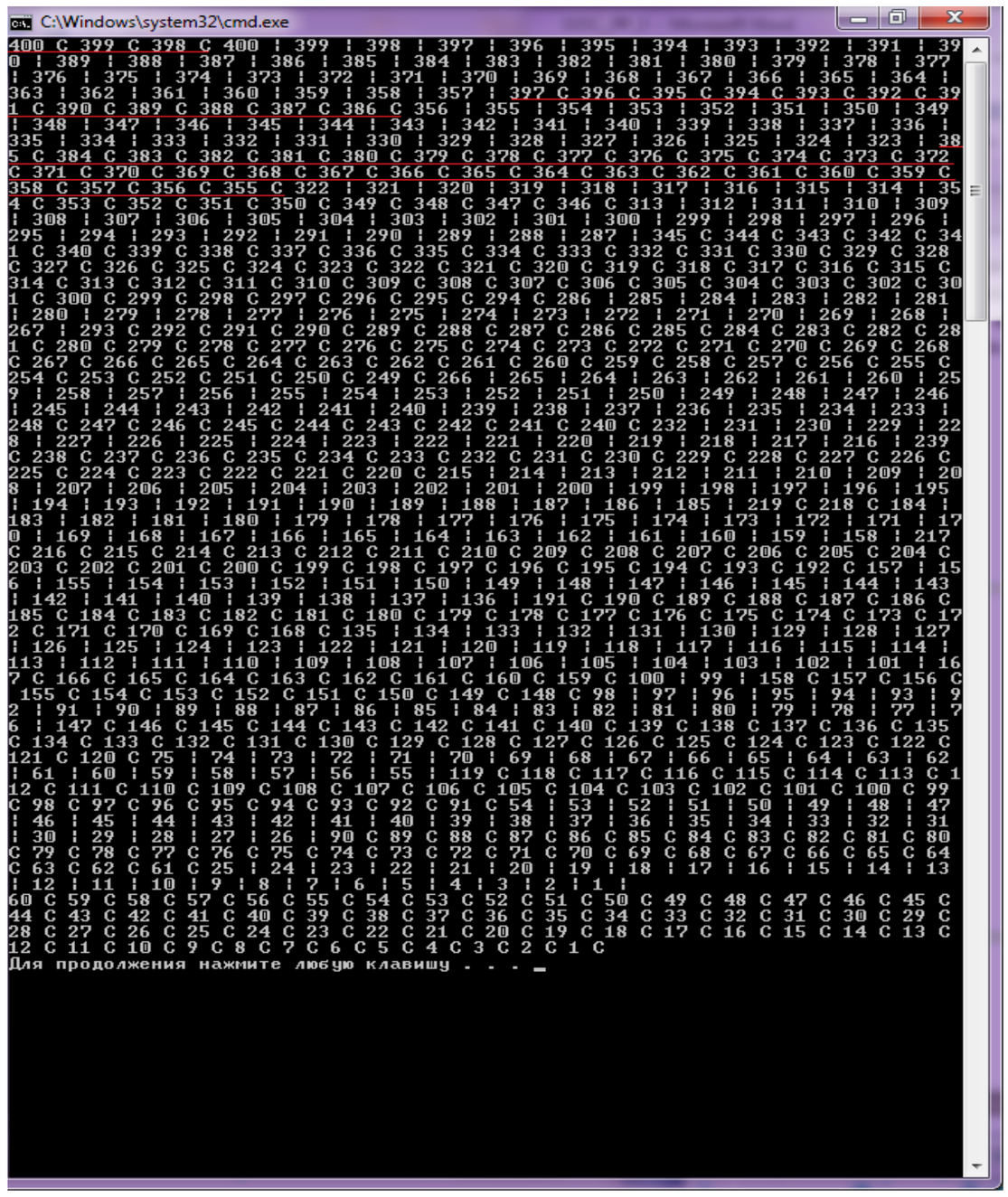

Рис. 4

Состояния потоков

Объекты класса Thread за время своего существования могут пребывать в разных состояниях. Для получения текущего состояния потока используется свойство ThreadState. Оно возвращает одно из десяти возможных значений:

1) Unstarted — не запущен;
2) Running — выполняется;
3) Background — фоновый;
4) WaitSleepJoin — заблокирован в результате вызова методов Wait, Sleep или Join;
5) SuspendRequested — запрос на приостановку;
6) Suspended — приостановлен;
7) StopRequested — запрос на остановку;
8) Stopped — остановлен;
9) AbortRequested — запрос на отмену (был вызван метод Abort, но поток еще не получил исключение ThreadAbortException);
10) Aborted — отменен.

Обычно сразу после создания объект класса Thread находится в состоянии Unstarted. После вызова метода Thread.Start он переходит в состояние Running. Если после этого свойству IsBackground присвоить значение true, то поток перейдет в состояние Background, и т.д.

Поток может находиться в нескольких состояниях в один и тот же момент времени. Например, поток, ждущий освобождения ресурса, может быть одновременно в состоянии WaitSleepJoin и, если в ходе ожидания был вызван метод Abort, в состоянии AbortRequested. Поэтому для проверки состояния потока часто приходится использовать логические конструкции. Такой подход показан в следующей программе:

```
// Анализ состояния потока
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;
```

```
namespace ConsoleApp4
{
    class Program
    {
        // Метод Countdown считает от 10 до 1
        public static void Countdown()
        {
            for (int counter = 10; counter > 0; counter--)
                Console.Write(counter.ToString() + " ");
            Console.WriteLine();
        }
        // Метод DumpThreadState выводит текущее состояние потока
        // ThreadState — битовая маска состояния потока (принимает потоковую переменную t)
        public static void DumpThreadState(Thread t)
        {
            Console.Write("Текущее состояние: ");
            if ((t.ThreadState & ThreadState.Aborted) == ThreadState.Aborted)
                Console.Write("Отменен");
            if ((t.ThreadState & ThreadState.AbortRequested) == ThreadState.AbortRequested)
                Console.Write("Запрос на отмену");
            if ((t.ThreadState & ThreadState.Background) == ThreadState.Background)
                Console.Write("Выполняется в фоновом режиме");
            if ((t.ThreadState & (ThreadState.Stopped | ThreadState.Unstarted | ThreadState.Aborted)) == 0)
                Console.Write("Выполняется");
            if ((t.ThreadState & ThreadState.Stopped) == ThreadState.Stopped)
                Console.Write("Остановлен");
            if ((t.ThreadState & ThreadState.StopRequested) == ThreadState.StopRequested)
                Console.Write("Запрос на остановку");
            if ((t.ThreadState & ThreadState.Suspended) == ThreadState.Suspended)
                Console.Write("Приостановлен");
            if ((t.ThreadState & ThreadState.SuspendRequested) == ThreadState.SuspendRequested)
                Console.Write("Запрос на приостановку");
            if ((t.ThreadState & ThreadState.Unstarted) == ThreadState.Unstarted)
                Console.Write("Не запущен");
            if ((t.ThreadState & ThreadState.WaitSleepJoin) == ThreadState.WaitSleepJoin)
                Console.Write("Ожидает освобождения ресурсов");
            Console.WriteLine();
        }
        static void Main()
        {
            // Создание еще одного потока
            Thread t2 = new Thread(new ThreadStart(Countdown)); // Привязка потока к методу Countdown()
            // Вызов функции DumpThreadState (возвращает состояние потока) с передачей параметра t2
            DumpThreadState(t2);          // Запуск нового потока
            t2.Start();
            // Вызов функции DumpThreadState (возвращает состояние потока) с передачей параметра t2
            DumpThreadState(t2);
            // В это же время метод Countdown вызывается в основном потоке
            Countdown();                  // Отмена второго потока
            t2.Abort();
            // Вызов функции DumpThreadState (возвращает состояние потока) с передачей параметра t2
            DumpThreadState(t2);
        }
    }
}
```

После нескольких «запусков» этой программы могут получиться такие результаты (не одинаковые, рис. 3, 4, 5).

Рис. 3

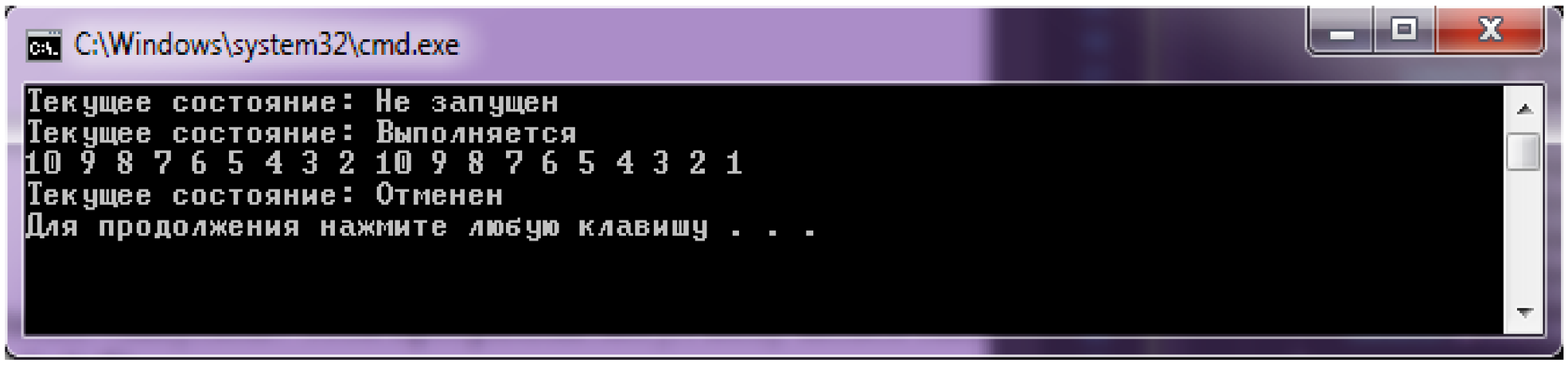

Рис. 4

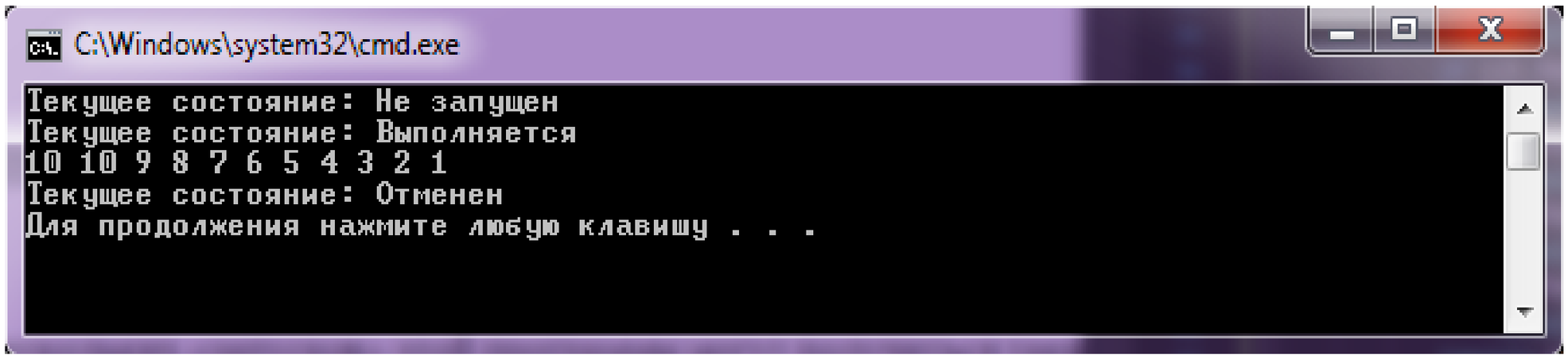

Рис. 5

Порядок выполнения потоков

Как было показано в предыдущих примерах, обычно порядок выполнения потоков не определен. Однако существуют способы управления потоками, позволяющие сделать их поведение более предсказуемым.

Класс ***Timer*** позволяет выполнять потоки через повторяющиеся промежутки времени, класс ***Join*** позволяет одному потоку ждать завершения другого. Классы ***Lock, Interlocked, Monitor*** и ***Mutex*** позволяют координировать действия нескольких потоков и использование ресурсов несколькими потоками.

При помощи класса Timer можно добавить в программу поток, работающий по принципу «вызвали и забыли». При создании экземпляра класса Timer указывается четыре параметра:

1) **Callback.** Делегат типа TimerCallback, представляющий метод, который будет вызываться таймером;
2) **State.** Объект, который передается методу TimerCallback. Его можно использовать для того, чтобы таймеру было доступно некоторое постоянное состояние. Если такой необходимости нет, то этот параметр может быть равным *null*;
3) **DueTime.** Количество миллисекунд перед первым срабатыванием таймера;
4) **Period.** Количество миллисекунд между срабатываниями таймера.

В следующем листинге показан пример использования класса Timer:

```
// Использование класса Timer
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;
```

```
namespace ConsoleApp5
{
    class Program
    {
        // Метод CheckTime вызывается по таймеру
        public static void CheckTime(Object state)
        {
            Console.WriteLine(DateTime.Now);    // Вывод даты-времени
        }
        static void Main()
        {
            // Создание делегата, который будет вызываться объектом Timer
            TimerCallback tc = new TimerCallback(CheckTime);   // Привязка к методу CheckTime()
            // Создание таймера, срабатывающего дважды в секунду, первый запуск произойдет через одну секунду
            Timer t = new Timer(tc, null, 1000, 500);
            Console.WriteLine("Нажмите Enter для завершения программы...");  // Ожидание ввода пользователя
            Console.Read();
            // Освобождение ресурсов
            t.Dispose();
            t = null;
        }
    }
}
```

Результат работы программы представлен на рис. 6.

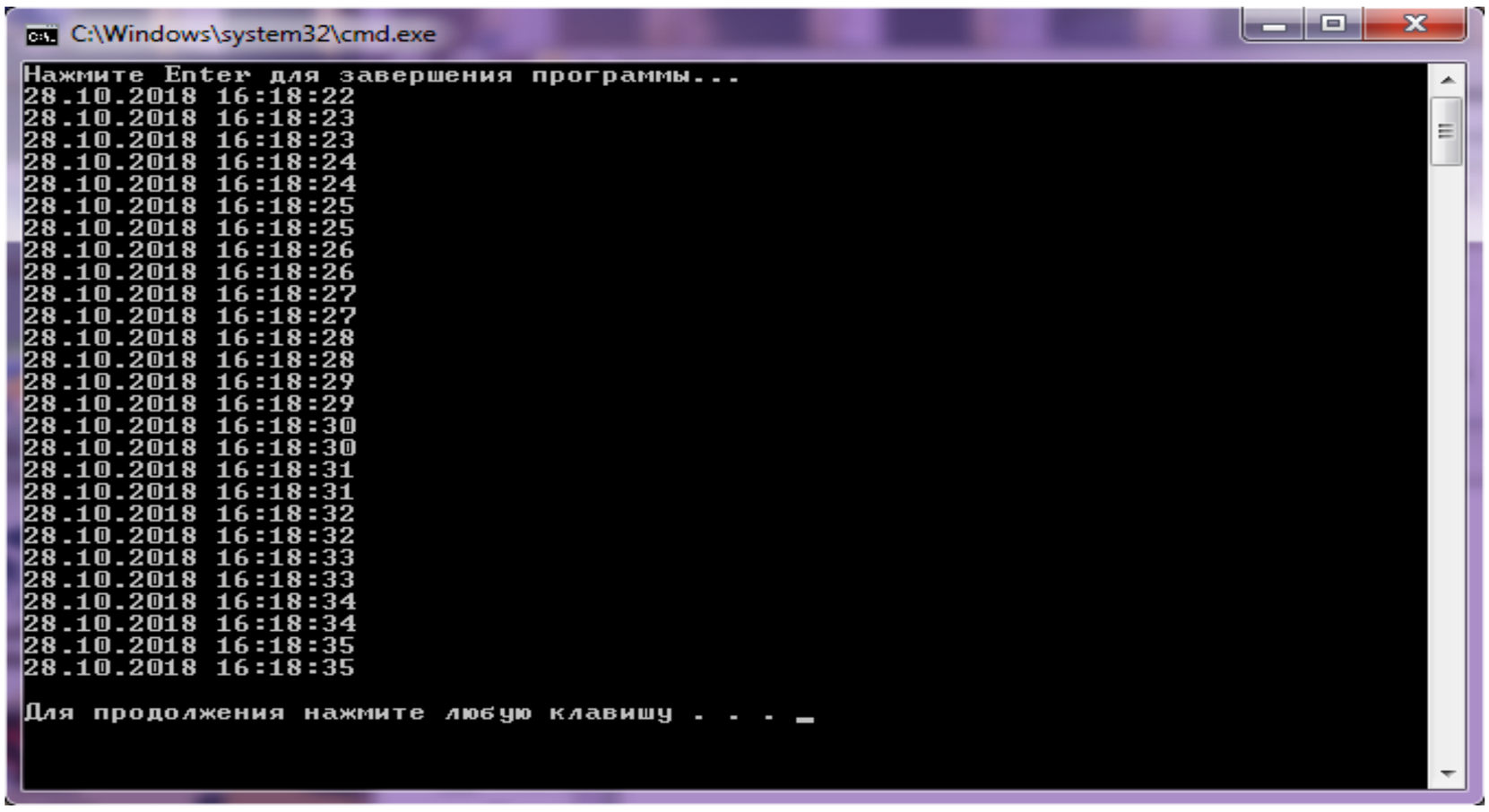

Рис. 6

Последовательное выполнение потоков

Метод, указанный в конструкторе объекта Timer, выполняется в отдельном потоке, созданном системой, а не в потоке, создавшем объект Timer. Обратите внимание на вызов метода Dispose объекта Timer. Это делается для того, чтобы корректно освободить уже ненужные ресурсы, выделенные отдельному потоку.

Метод Thread.Join позволяет «прикрепить» один поток к другому. Это означает, что, например, первый поток, будет ждать завершения второго, после чего будет запущен. Использование этого метода показано в следующем листинге.

```
// Использование метода Join
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;

namespace ConsoleApp5
{
    class Program
    {
        public static void Countdown()          // Метод Countdown считает от 100 до 1
{
    for (int counter = 100; counter > 0; counter--)
        Console.Write(counter.ToString() + " ");
}
public static void Countdown1()         // Метод Countdown 1 считает от 100 до 1 с буквой A
{
    for (int counter = 100; counter > 0; counter--)
        Console.Write(counter.ToString() + " A ");
}
static void Main()
{
    // Создание второго потока
    Thread t2 = new Thread(new ThreadStart(Countdown)); // Привязка потока к методу Countdown
    t2.Start();                                 // Запуск второго потока
    t2.Join();                                  // Блокировка первого потока до завершения второго
            Console.WriteLine("");              // Вызов метода Countdown1 из первого потока
            Countdown1();                       // Вызов метода Countdown1 из первого потока
        }
    }
}
```

Если выполнить эту программу (рис. 7), то можно заметить, что вывод двух методов Countdown не пересекается, несмотря на то, что оба потока выполняются с нормальным приоритетом. Это произошло потому, что вызов t2.Join приостанавливает основной поток до тех пор, пока не завершится выполнение метода Countdown во втором потоке.

Метод Join имеет перегруженную версию, которая позволяет указать максимальное время ожидания. Например, для того чтобы ждать завершения второго потока не дольше 5 с., можно использовать оператор t2.Join(5000).

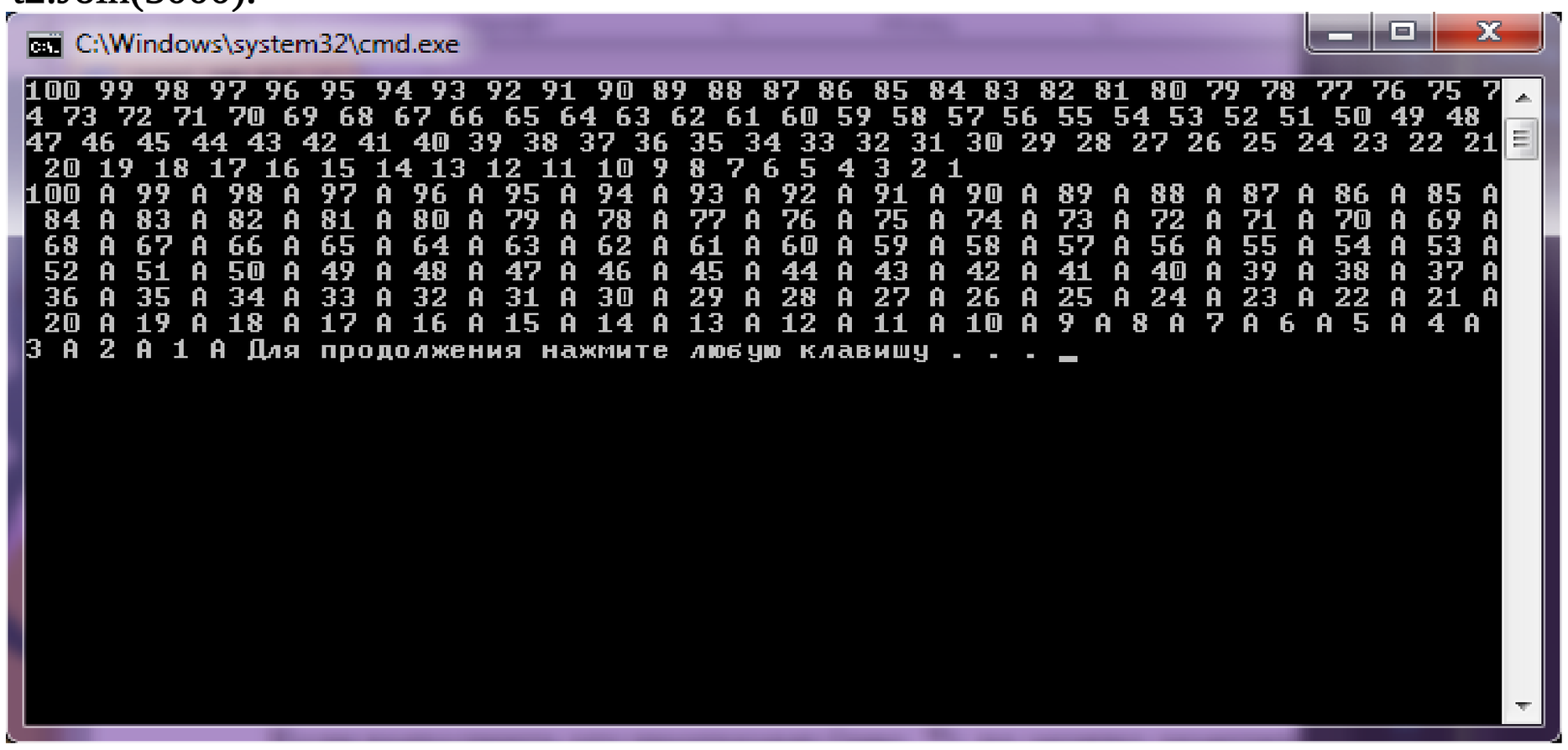

Рис. 7

Практическая часть

1. Модифицировать любую из программ, представленных в лабораторной работе, организовав три и более потока.
2. Организовать программу с последовательной работой потоков (три и более потока).

Контрольные вопросы

1. Имеют ли все потоки одинаковые приоритеты? Сколько категорий приоритета существует?
2. Опишите класс Timer.
3. Как можно осуществить последовательное выполнение потоков?
4. Будет ли отличаться результат каждого выполнения программы, в которой: а) два потока, выполняющиеся параллельно; б) два потока, выполняющихся последовательно. Рассмотреть две ситуации для а) и б): 1) два потока привязаны к методам, которые прибавляют и вычитают какое-либо число (к примеру, первый метод постоянно прибавляет к числу N константу C1, второй метод вычитает из числа N константу C2); 2) два потока привязаны к методам, которые умножают и делят какое-либо число. Будет ли результат на выходе один и тот же или нет?

Содержание отчета

В отчет о выполненной работе включить следующие материалы:

1. Тему и цель работы.
2. Результаты выполнения заданий: исследуемые схемы, полученные таблицы переходов.
3. Анализ полученных результатов.
4. Ответы на контрольные вопросы.
5. Выводы по работе.