Лабораторная работа №5
Тема: «Проблемы многопоточных программ»

Теоретическая часть
Критические секции

При использовании многопоточной программы возникает ряд проблем, которые не существуют в однопоточных программах. Например, как поведет себя программа, если два потока используют одну и ту же переменную? Результат может отличаться от ожидаемого. В следующем листинге показан многопоточный код с проблемой такого рода — критической секцией, которая представляет собой участок кода, требующий монопольного доступа к каким-то общим данным.

```
// Многопоточная работа с проблемами критической секции
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;

namespace Critical_Section
{
    class Program
    {
        // Общий счетчик
        static int Runs = 0;
        // Метод CountUp увеличивает значение общего счетчика
        public static void CountUp()
        {
    while (Runs <= 10)
    {
        int Temp = Runs;
        Temp++;
        Console.WriteLine(Thread.CurrentThread.Name + " " + Temp);
        // Остановка потока на 1 с. (освобождение ресурсов)
        Thread.Sleep(1000);
        Runs = Temp;
    }
}
static void Main()
{
    // Создание и запуск двух потоков
    Thread t2 = new Thread(new ThreadStart(CountUp));
    // Задаем имя потока
    t2.Name = "Thread_2:";
            Thread t3 = new Thread(new ThreadStart(CountUp));
            t3.Name = "Thread_3:";
            // Запуск потоков
            t2.Start();
            t3.Start();
        }
    }
}
```

Эта программа выводит строки, изображенные на рис. 1. Если вы ожидали увидеть десять повторений цикла в методе CountUp, то очень удивитесь, увидев результат. Проблема состоит в том, что увеличение переменной Runs производится не за один шаг. Дело в том, что в процессе получения

значения переменной, увеличения его на единицу и присвоения нового значения могут происходить другие действия. Метод Thread.Sleep указывает объекту Thread освободить процессор на указанное количество миллисекунд.

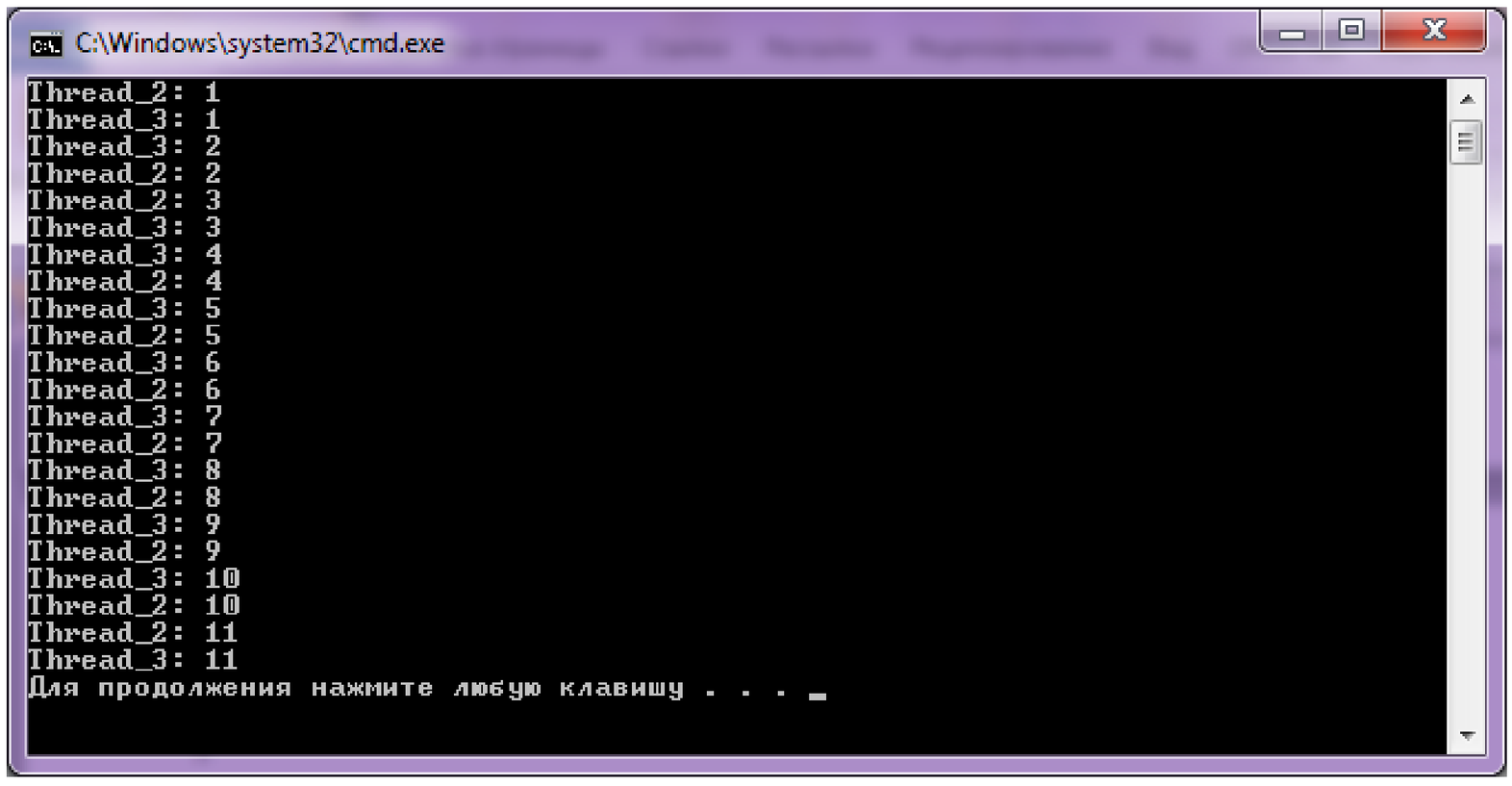


Рис. 1

Что же произошло в нашем случае? Приведем последовательность выполняемых действий:

1) Поток «Thread_2» присваивает переменной «Temp» значение переменной «Runs» и увеличивает «Temp» на единицу;
2) Поток «Thread_2» приостанавливает свое выполнение на 1 с.;
3) Планировщик передает управление потоку «Thread_3»;
4) Поток «Thread_3» присваивает переменной «Temp» значение переменной «Runs» и увеличивает «Temp» на единицу. Здесь используются те же значения, что и в потоке «Thread_2», так как поток «Thread_2» еще не успел изменить значение общего счетчика (он «заснул» на 1000 мс);
5) Поток «Thread_3» приостанавливает свое выполнение на 1 с.;
6) Поток «Thread_2» возобновляет выполнение и присваивает новое значение переменной «Runs»;
7) Поток «Thread_3» возобновляет выполнение и присваивает новое значение переменной «Runs», изменяя значение, присвоенное потоком «Thread_2»;
8) Цикл повторяется.

Заметим, что в этом случае метод Sleep используется для того, чтобы сделать борьбу за ресурсы очевидной. Подобная ситуация может сложиться и без использования метода Sleep, так как планировщик может переключиться на выполнение другого потока в любом месте цикла, перед тем как переменной Runs присвоено новое значение.

Блокировка

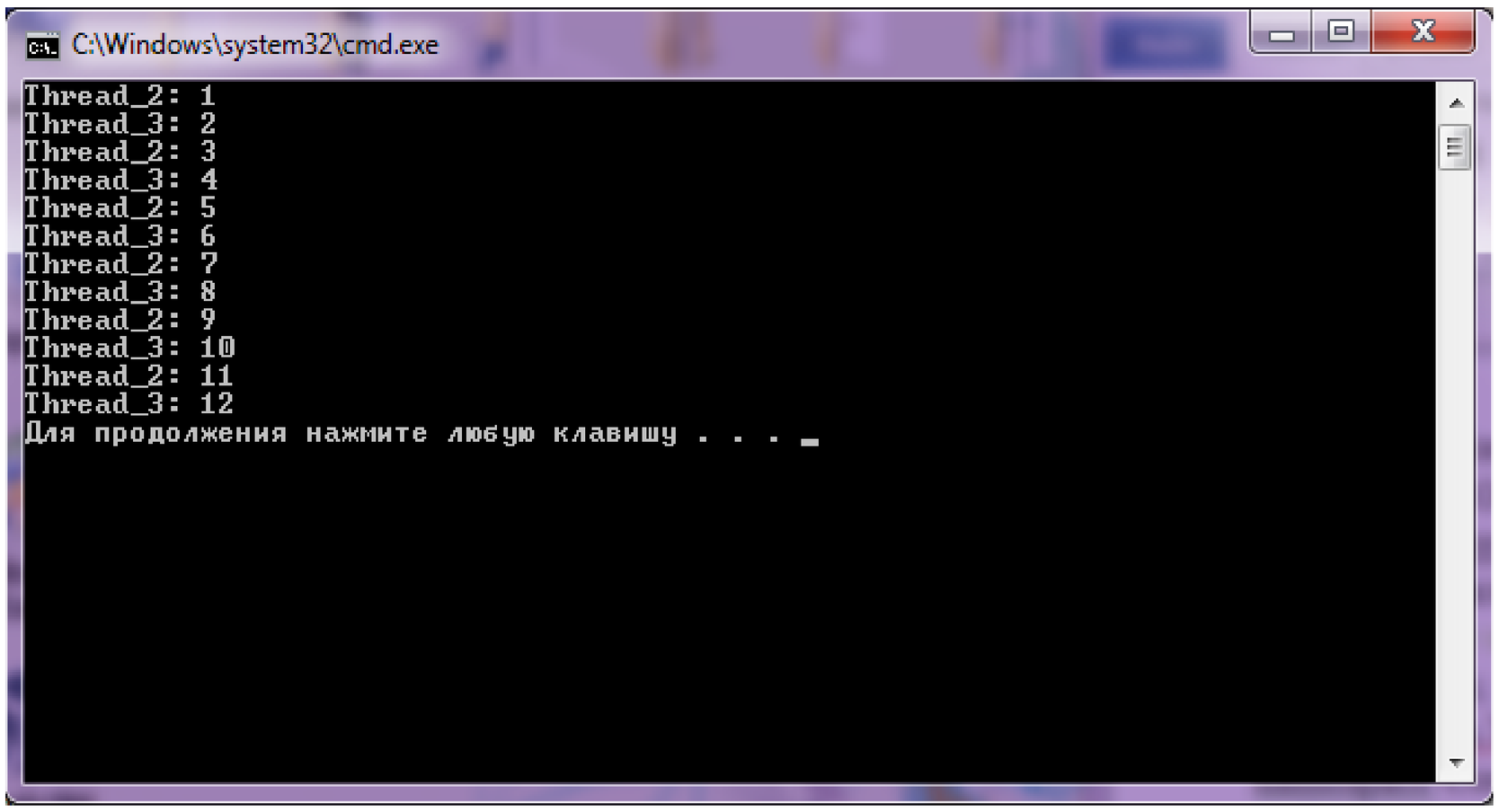
Решить проблему критической секции можно различными средствами. Например, использованием оператора lock. Образно говоря, оператор lock означает «не позволять другим потокам выполнять ту часть кода до тех пор, пока я ее выполняю». Оператор позволяет блокировать любой объект. В приведенном далее листинге оператор lock() блокирует код всего класса:

```
// Выделение критической секции с помощью lock
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;

namespace Use_lock
{
    class Program
```

```
{
    static int Runs = 0;                        // Общий счетчик
    static object obj = new object();           // Создаем объект для блокировки
    public static void CountUp()                // Метод CountUp увеличивает значение общего счетчика
    {
        while (Runs <= 10)
        {
            lock (obj)                          // Помещаем объект в метод lock (блокировка)
            {
                int Temp = Runs;
                Temp++;
                Console.WriteLine(Thread.CurrentThread.Name + " " + Temp);
                Thread.Sleep(1000);
                Runs = Temp;
            }
        }
    }
    static void Main()
    {
        // Создание и запуск двух потоков
        Thread t2 = new Thread(new ThreadStart(CountUp));
        t2.Name = "Thread_2:";
        Thread t3 = new Thread(new ThreadStart(CountUp));
        t3.Name = "Thread_3:";
        t2.Start();
        t3.Start();
    }
}
```

Использование оператора lock и объекта «obj» позволило обеспечить правильную работу программы, приведенной в предыдущем примере (рис. 2).



Монитор (Monitor)

Ключевое слово lock, рассмотренное в предыдущем примере, это на самом деле сокращенный способ использования класса Monitor. Класс Monitor предоставляет гибкие способы синхронизированного доступа к любому объекту кода программы. В следующем листинге предыдущий пример переписан с явным использованием класса Monitor:

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

namespace Use_Monitor
{
    class Program
    {
        static int Runs = 0;                        // Общий счетчик
static object obj = new object();  // Создание объекта в качестве параметра для класса Monitor
public static void CountUp()              // Метод CountUp увеличивает значение общего счетчика
{
    while (Runs <= 10)
    {
        Monitor.Enter(obj); // Активация класса Monitor через операцию доступа к методу Enter
        int Temp = Runs;
        Temp++;
        Console.WriteLine(Thread.CurrentThread.Name + " " + Temp);
        Thread.Sleep(1000);
        Runs = Temp;
        Monitor.Exit(obj);            // Конец работы класса Monitor
    }
}
static void Main()
        {
            // Создание и запуск двух потоков
            Thread t2 = new Thread(new ThreadStart(CountUp));
            t2.Name = "Thread_2 (M):";
            Thread t3 = new Thread(new ThreadStart(CountUp));
            t3.Name = "Thread_3 (M):";
            t2.Start();
            t3.Start();
        }
    }
}
```

Результат работы этой программы (рис. 3) в данном случае не отличается от предыдущего.

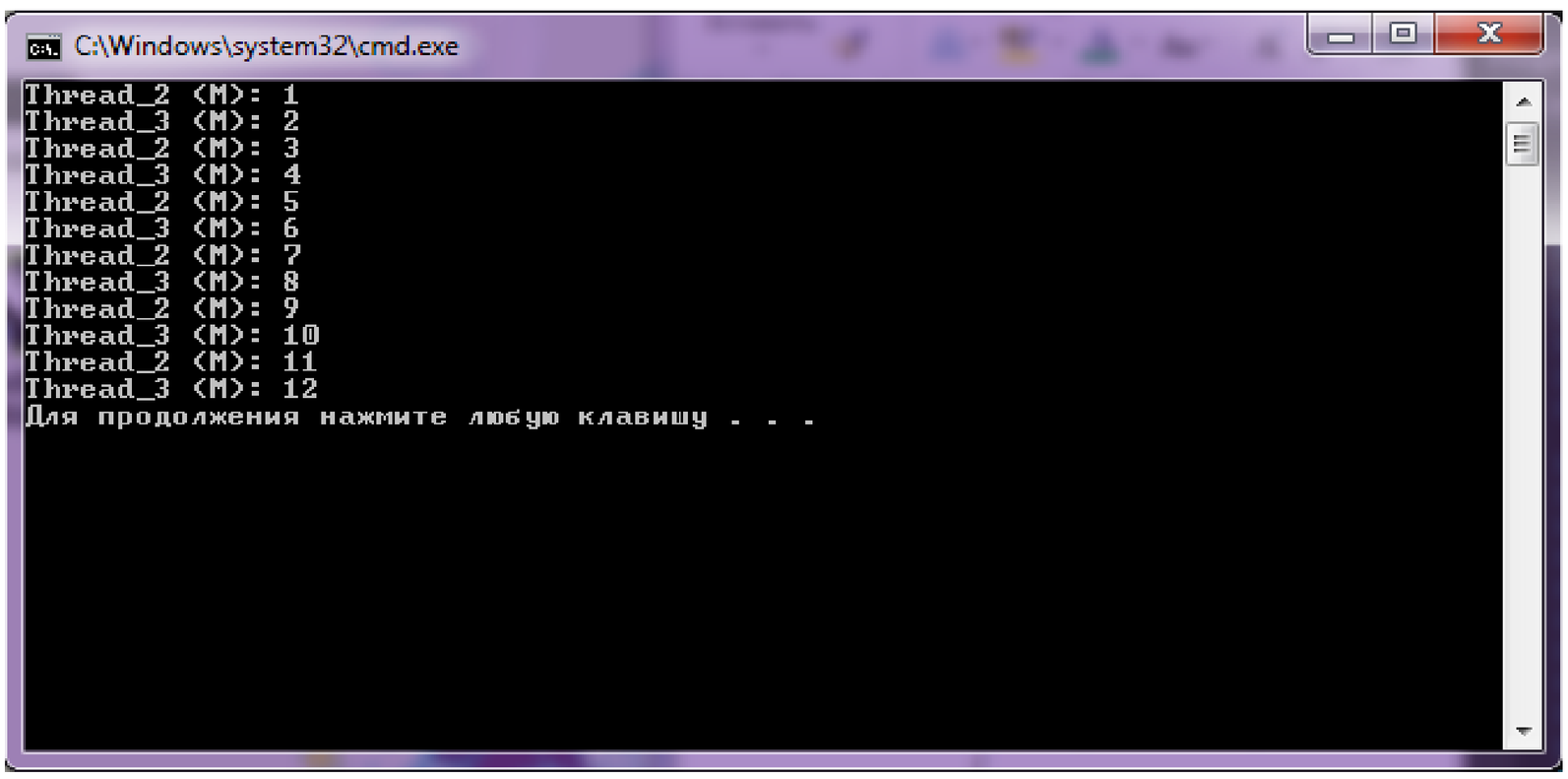


Рис. 3

Методы Monitor.Enter и Monitor.Exit работают точно так же, как и ключевое слово lock. Но класс Monitor имеет значительно больше возможностей, чем простая блокировка. Далее перечислены методы класса Monitor:

1. Enter() - блокирует объект, переданный классу Monitor. Если другой поток уже блокировал этот объект, то выполнение текущего потока будет приостановлено до тех пор, пока другой поток не освободит объект через команду Exit().
2. Pulse() - информирует следующий ждущий поток, что монитор временно закончил работу с объектом и поток может продолжить выполнение.
3. PulseAll() - сигнализирует всем потокам о том, что объект скоро будет освобожден.
4. TryEnter() - пытается заблокировать переданный объект. Если объект может быть заблокирован, возвращает «true», в противном случае возвращает «false».
5. Wait() - освобождает все блокировки и приостанавливает выполнение текущего потока до тех пор, пока другой поток не вызовет метод Pulse.

Семафоры

Другой вариант решения рассматриваемой проблемы связан с использованием объектов ядра ОС — семафоров. Семафоры используются для синхронизации потоков разных процессов. Это обусловлено тем, что семафор как объект реализован не в каком-либо приложении, а в ядре ОС Windows. Если два процесса создают объекты Mutex, передав конструкторам одинаковое имя, то они получают один и тот же семафор.

В определенный момент времени семафор может принадлежать только одному потоку. В данном варианте конструктор класса Mutex принимает два параметра (существуют также перегруженные версии конструктора). Первый параметр булевого типа определяет, должен ли объект изначально принадлежать потоку, создавшему его. Второй параметр представляет имя семафора.

Поток может вызвать метод WaitOne для получения семафора. Если никакой другой поток не использует семафор, тот его получает поток, вызвавший метод WaitOne. Если семафор используется другим потоком, то поток, вызвавший метод WaitOne, блокируется до тех пор, пока семафор не освободится. Когда работа, связанная с семафором, завершена, поток может вызвать метод ReleaseMutex, что приведет к освобождению семафора для другого ожидающего потока.

Класс Mutex также имеет метод WaitAll, ждущий до тех пор, пока все семафоры группы освободятся, и метод WaitAny, ожидающий до тех пор, пока освободится какой-либо семафор из группы:

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

namespace Use_Semaphor
{
    class Program
    {
        static int Runs = 0;                    // Общий счетчик
        static Mutex mtx;                       // Семафор
        public static void CountUp()    // Метод CountUp увеличивает значение общего счетчика
        {
            while (Runs <= 10)
            {
                mtx.WaitOne();                          // Получение семафора
                int Temp = Runs;
                Temp++;
                Console.WriteLine(Thread.CurrentThread.Name + " " + Temp);
```

```
            Runs = Temp;
            mtx.ReleaseMutex();                    // Освобождение семафора
            Thread.Sleep(1000);
        }
    }
    static void Main()
    {
        // Создание семафора (false – не принадлежит ни одному потоку на данный момент,
“RunsMutex” – имя семафора)
        mtx = new Mutex(false, "RunsMutex");
        // Создание и запуск двух потоков
        Thread t2 = new Thread(new ThreadStart(CountUp));   // Привязка потока к методу CountUp
        // Задаем имена потокам t2 и t3
        t2.Name = "Thread_2:";
        Thread t3 = new Thread(new ThreadStart(CountUp));   // Привязка потока к методу CountUp
        t3.Name = "Thread_3:";
        t3.Start();
        t2.Start();
    }
  }
}
```

Результат работы этой программы приведен на рис. 4. В консоль выводится по порядку по два числа от каждого потока. Дело в том, что после того, как поток выполнил свою работу, к примеру, «Thread_2», он освобождает семафор, «Thread_2» засыпает на 1 с., поток «Thread_3» мгновенно перехватывает семафор и выполняет свою работу, после чего также засыпает на 1 с.

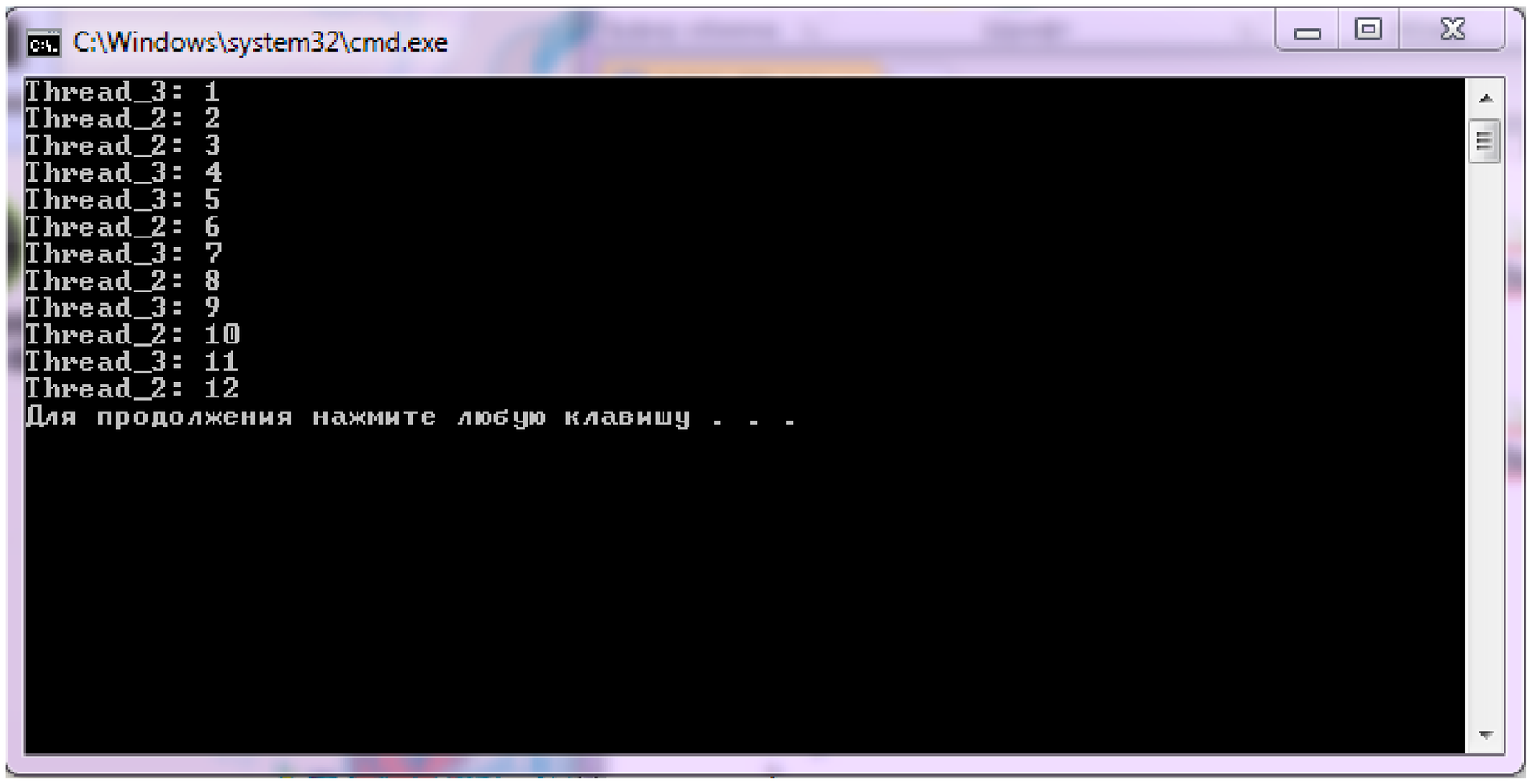


Рис. 4

Взаимоблокировка

Серьезной проблемой многопоточных программ является возможность взаимной блокировки потоков. Взаимная блокировка может произойти в том случае, если каждый из двух потоков ждет освобождения ресурса, заблокированного другим потоком. Предположим, в одном потоке выполняются следующие строки:

```
lock (A)              // Переменная A – переменная типа object
{
lock (B)              // Переменная B – переменная типа object
    {
        // Здесь что-то выполняется
```

```
        }
    }
        // В то же время в другом потоке выполняются такие строки
    lock (B)
    {
        lock (A)
        {
            // Здесь что-то выполняется
        }
    }
```

Если процессор переключится с первого потока на второй после того, как первый поток заблокировал объект A, то второй поток заблокирует объект B и будет ожидать освобождения объекта A. После того как управление перейдет к первому потоку, он будет ожидать освобождения объекта B. В результате ни один поток не сможет продолжить выполнение.

В программе, листинг которой следует далее, создается два объекта Section и Section1, которые используют общие данные (Runs и Runs1) и два метода для работы с ними:

```
// Взаимоблокировка потоков
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;

namespace CrossBlock_Threads
{
    class Program
    {
        private static string Runs_Numbers = "0123456789";   // Объявление полей — общие строки цифр от 0 до 9
        private static string Runs_Words = "абвгдежзик";     // строка букв от ‘а’ до ‘к’
        public static void CountUp()   // Метод CountUp увеличивает значение в строке Runs от 0 до 10
        {
            lock (Runs_Words)
            {
                for (int i = 1; i <= 10; i++)
                {
                    lock (Runs_Numbers)
                    {
                        string Res_Count = Runs_Numbers.Substring(i - 1, 1);  // Выделение определенного символа
                        Console.WriteLine(Thread.CurrentThread.Name + " " + Res_Count);
                        Thread.Sleep(1000);
                    }
                }
            }
        }
        // Метод Char выдает 10 букв в порядке, записанном в строке Runs1
        public static void Char()
        {
            lock (Runs_Numbers)
            {
                for (int i = 1; i <= 10; i++)
                {
                    lock (Runs_Words)
```

```
            {
                string Res_Char = Runs_Words.Substring(i - 1, 1);  // Выделение определенного символа
                Console.WriteLine(Thread.CurrentThread.Name + " " + Res_Char);
                Thread.Sleep(1000);
            }
        }
    }
}
static void Main()
{
    // Создание и запуск двух потоков, привязка к методам CountUp и Char
    Thread t2 = new Thread(new ThreadStart(CountUp));
    t2.Name = "Thread_2:";
            Thread t3 = new Thread(new ThreadStart(Char));
            t3.Name = "Thread_3:";
            t2.Start();
            t3.Start();
        }
    }
}
```

В этой программе поток «Thread_2» использует метод CountUp, в котором предварительно блокируется Runs_Words, и производится работа с Runs_Numbers, поток «Thread_3» использует метод Char, в котором блокируется Runs_Numbers, и производится работа с Runs_Words. Первым получает управление поток «Thread_2». Он блокирует Runs_Numbers, печатает цифру 0 и засыпает. Управление получает процесс «Thread_3». Он блокирует Runs_Numbers, но не печатает первую букву метода Runs_Words, поскольку поток «Thread_2» заблокировал Runs_Words. Управление переходит к потоку «Thread_2». Однако он не может выполняться, так как Runs_Numbers заблокирован потоком «Thread_3». Налицо тупиковая ситуация. Результат работы программы показан на рис. 5.

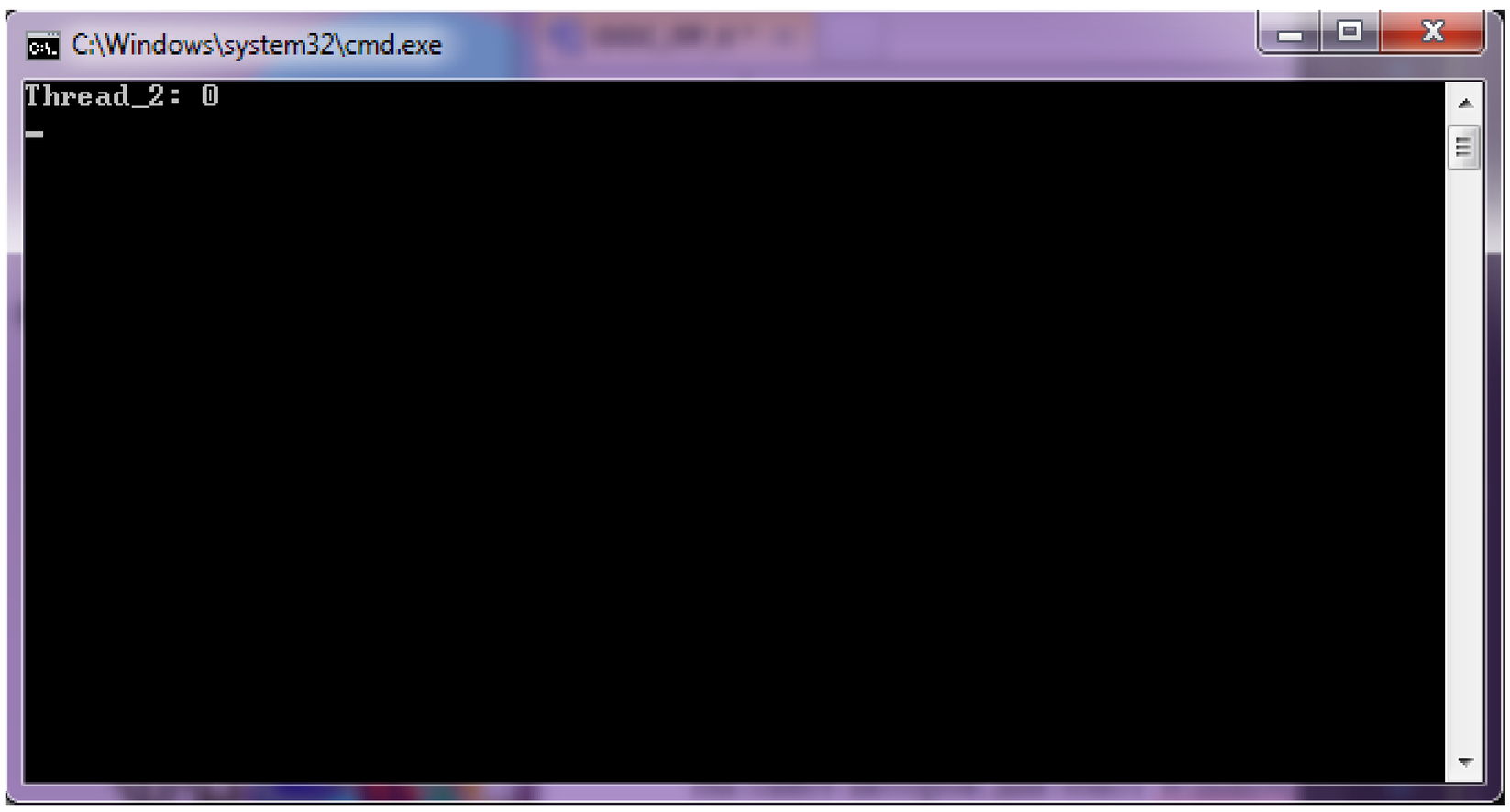


Рис. 5

Практическая часть

Изменить программу «CrossBlock_Threads» так, чтобы взаимоблокировка потоков не наступала.

Контрольные вопросы

1. Что делает команда Thread.Sleep(500) с потоком?
2. С помощью чего можно избежать попадания потоков в критическую секцию?
3. Какие параметры может включать в себя оператор lock?
4. Как решает проблему критической секции класс Monitor?
5. Что такое Семафор? Каким ключевым словом определяется Семафор в программе? Может ли Семафор быть привязан к потоку сразу с момента его создания?
6. Что такое взаимоблокировка потоков?

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.