In this laboratory assignment you will learn how to implement threads in C#, and begin the development of an effective means of controlling thread execution and communication.

A program in execution is called a process or a thread. The primary difference between a process and a thread is that each process can access its own address space but cannot directly access the address space of another process. In contrast, a thread shares address space (memory) with other threads that have a common parent (or ancestor). For example, if a main program in C# creates (spawns) a thread, both the main program (parent) and the created thread share the address space of the main and therefore are both considered to be threads.

Threads execute concurrently, which means that, until they are forced to coordinate with each other by some explicit means, the relative order of execution of statements within two or more threads is arbitrary. Also, the relative speed at which the sequence of instructions in each of the threads executes is not specified, so that one thread may reach a particular state in its execution before or after another thread reaches any particular state in its own.

When we want two threads to work together we must set up some mechanism for synchronizing them, such as forcing one thread to wait until another thread is ready. Threads can be in a number of different states as shown in the state transition diagram below.

![State Transition Diagram](image)

A newly created thread enters the started state (i.e. placed in a ready list) and waits for its turn at the CPU. The CPU is allocated to kernel threads only by the dispatcher, a component of the operating system kernel. The dispatcher interrupts a running thread when its time quantum expires (usually a few milliseconds) sending it back to the started state. A thread can give up the CPU by requesting some operation such as an I/O (input/output) event, entering the blocked state. When the I/O event has been completed, the thread returns to the started state. When a thread is suspended it will remain in this state until some other thread calls its resume method. At any time a running thread can return to the started state by relinquishing the remainder of its current time quantum back to the dispatcher. Finally, when a thread completes its operation it terminates (enters the stopped state).
An application process can create user-level threads. These are not individually visible to the OS and so all user-level threads must share the time quantum allocated to the application that created them. Therefore, single-threaded application and a multi-threaded application running user-level threads would receive the same amount of CPU time from the OS dispatcher.

1. Start a New Project and create a Console Application, give the application an appropriate name such as Thread_Demo_1. For this example we will build a main program that spawns to other threads. Use the following source as a template for your program.

```csharp
using System;
using System.Threading;
namespace Thread_Demo_1
{
    public class Version_1
    {
        public static bool done = false;

        public static void T1()
        {
            System.Random rnd = new Random();
            while (!done)
            {
                Console.WriteLine("T1 in critical section");
                Thread.Sleep(rnd.Next(100));
                Console.WriteLine("T1 is leaving critical section");
            }
        }

        public static void T2()
        {
            System.Random rnd = new Random();
            while (!done)
            {
                Console.WriteLine("T2 in critical section");
                Thread.Sleep(rnd.Next(100));
                Console.WriteLine("T2 is leaving critical section");
            }
        }

        static void Main(string[] args)
        {
            Thread p = new Thread(new ThreadStart(T1));
            p.Start();
            Thread q = new Thread(new ThreadStart(T2));
            q.Start();
            Thread.Sleep(5000);
            done = true;
            Console.WriteLine("Run Finished");
            Console.ReadLine();
        }
    }
}
```

Note that done and threadnumber are declared as `public static void` making them accessible by the Main program and the two threads (T1 and T2) that are created. After the Main program creates and starts the two threads it puts itself to sleep for five seconds (5000 milliseconds). Following this five second delay the Main program sets done to true, which is a flag telling both threads to terminate. The `Console.ReadLine()` command in Main is a simple way to force the program to wait until the user presses the enter key. This gives us time to look at the output displayed in the Console window.
2. Compile, debug and execute this program, then make the modifications and answer the questions listed below:

a. Do you get the same results each time you run the program? ____________ Why or why not?
______________________________________________________________________________________
______________________________________________________________________________________

b. Can both threads be in their respective critical sections at the same time? ____________

c. Can one of the threads execute more often than the other? ____________.

3. Change the maximum Sleep time of one of the threads from 100 to 1000 and run the program.

a. How many executions of the while loop were completed for thread T1 _______, T2______?

b. Why do threads continue to execute after the Boolean done is set to true by the main method?
______________________________________________________________________________________
______________________________________________________________________________________

4. Modify your program to include a public integer value called threadnumber initialized to 1 and modify the threads in the manner shown below:

```csharp
public class Version_2
{
    public static bool done = false;
    public static int threadnumber = 1;

    public static void T1()
    {
        System.Random rnd = new Random();
        while (!done)
        {
            while (threadnumber == 2);
            Console.WriteLine("T1 in critical section");
            Thread.Sleep(rnd.Next(100));
            Console.WriteLine("T1 is leaving critical section");
            threadnumber = 2;
        }
    }

    public static void T2()
    {
        System.Random rnd = new Random();
        while (!done)
        {
            while (threadnumber == 1);
            Console.WriteLine("T2 in critical section");
            Thread.Sleep(rnd.Next(100));
            Console.WriteLine("T2 is leaving critical section");
            threadnumber = 1;
        }
    }
}
```
a. What is different in the execution of this version of the program compared to the previous version?
______________________________________________________________________________________

b. Can both threads be in their respective critical sections at the same time? ____________

5. Change the maximum Sleep time of one of the two threads from 100 to 1000 and run the program.

a. How many executions of the while loop were completed for thread T1 _______, T2______ ?

b. What is the maximum number of times each thread can execute after done is set to true? _____________

6. Modify your threads to include a Sleep time outside the critical section as shown below and run the program.

   public static void T1()
   {
      System.Random rnd = new Random();
      while (!done)
      {
         Thread.Sleep(rnd.Next(1000));
         while (threadnumber == 2) ;
         Console.WriteLine("T1 in critical section");
         Thread.Sleep(rnd.Next(1000));
         Console.WriteLine("T1 is leaving critical section");
         threadnumber = 2;
      }
   }

   public static void T2()
   {
      System.Random rnd = new Random();
      while (!done)
      {
         Thread.Sleep(rnd.Next(100));
         while (threadnumber == 1) ;
         Console.WriteLine("T2 in critical section");
         Thread.Sleep(rnd.Next(100));
         Console.WriteLine("T2 is leaving critical section");
         threadnumber = 1;
      }
   }

Discuss how you could change your program and threads so that one of the threads could execute more often than the other without both being in their respective critical sections at the same time. Hint: Consider including additional Boolean parameters T1_wants_to_run, and T2_wants_to_run.
______________________________________________________________________________________

Implement your approach and submit this modified source code. Be sure to include a comment block at the top of your program source code with your name, the course and project numbers.