In this laboratory experiment you will implement a multi-threaded program to demonstrate the enforcement of mutual exclusion using software only. This means that your program will not use any special computer hardware features to support its operation.

In applications in which multiple processes (or threads) access a shared memory space, we need a mechanism to prevent two processes from writing to the same address simultaneously or one thread attempting to read an address that is being written to by another thread.

When a process or thread is accessing shared data, it is said to be in the critical section (also called critical region) of the program. Threads can run independently so long as neither is in its critical section. Ensuring that only one thread can be in its critical section at any given time is called enforcing mutual exclusion.

When multiple threads are running within a single program, the mechanism for enforcing mutual exclusion for the shared address space is an integral part of the computer language compiler and operating system. Special hardware features built into the CPU instruction set may be used to support the management of shared data access. However, in a distributed or networked application we need a way to manage shared data access that can be implemented in software only, on any platform running under any operating system.

In the previous laboratory experiment we studied some simple methods of enforcing mutual exclusion. One of these methods required that the threads take turns, which may seem fair but causes obvious problems when one of the threads needs to access shared data more often than the other. A fully functional implementation of mutual exclusion enforcement should have the following characteristics:

1. **Enforce Mutual Exclusion** - The most important characteristic, of course, is that two or more threads do not gain access to a shared resource to which exclusive access is required.

2. **Avoid Lockstep Synchronization** - In a typical application, threads or processes only occasionally need access to shared resources and not necessarily in any particular order. Lockstep synchronization is a condition in which threads must take turns accessing a shared resource. This condition may be acceptable for some applications such as producer-consumer or peer-to-peer communication, however we usually want to permit a thread to gain access to a resource multiple times when other threads do not need access to this resource (also called asynchronous concurrent execution). Avoiding lockstep synchronization is an important condition for mutual exclusion.

3. **Prevent Deadlock** - Deadlock occurs when two or more threads are waiting for a condition that will never occur. For example, there are two resources, both of which are needed to complete an operation. Two threads are each holding one of the two resources, while waiting for the other to become available. Any practical implementation of mutual exclusion must avoid deadlock.

4. **Avoid Starvation** - Starvation (also called indefinite postponement) is a situation in which one or more threads are delayed indefinitely while waiting to gain access to shared resources. This is different from deadlock in that access to the shared resources is possible but simply does not occur. Consider a fast thread that can gain access to a shared resource, return the resource and then gain access to it again before a waiting resource has an opportunity to get the shared resource.
(5) **Limit Resource Hoarding** - Resource hoarding is when a thread holds resources longer than is necessary to accomplish the task for which they are needed. Extreme examples of resource hoarding can lead to conditions similar to deadlock or starvation. Even if these conditions are avoided, holding resources longer than absolutely necessary leads to poor performance and is considered bad programming style.

(6) **Symmetric Implementation** - For some applications we can designate a particular thread as the controller or manager for all the other threads or processes (i.e. asymmetric). For most distributed applications it is necessary or desirable to share the task of enforcing mutual exclusion among all the threads. In this way, any subset of the threads can continue to execute properly with one or more of the threads offline.

Dekker's and Peterson's Algorithms are implementations of mutual exclusion enforcement for two threads that exhibit all the properties described above. In this experiment we will implement a version of Dekker's Algorithm in C#. The following code segments show sample public static variables for Dekker's and the event loop for one of the two threads (T1).

```csharp
public static int favored = 1;
public static bool t1wantsin = false;
public static bool t2wantsin = false;

public static void T1()
{
    System.Random rnd = new Random();
    while (!done)
    {
        Thread.Sleep(rnd.Next(100)); // time out of critical section
        t1wantsin = true;
        while (t2wantsin)
        {
            if (favored == 2)
            {
                t1wantsin = false;
                while (favored == 2);
                t1wantsin = true;
            }
        }
        Console.WriteLine("T1 in critical section");
        Thread.Sleep(rnd.Next(100)); // time in critical section
        Console.WriteLine("T1 is leaving critical section");
        favored = 2;
        t1wantsin = false;
    }
}
```

1. Build a multi-threaded console application that implements Dekker's Algorithm for two threads T1() and T2().

2. Vary the maximum time in and out of the critical section between the two threads to verify that the following properties are being implemented:

   a. **enforce mutual exclusion** - Describe how you verify that mutual exclusion is being enforced.
b. **prevent deadlock** - Describe the behavior that demonstrates that deadlock does not occur.

---

3. Modify your program to count the number of times each thread enters its critical section during a run of the simulation.

a. Give the counts for a sample run.

   T1() count = _______________     T2() count = _______________

b. Change the maximum amount of time one of the threads spends out of its critical section from 100 to 1000 and give the counts for a sample run.

   T1() count = _______________     T2() count = _______________

c. Now change the maximum amount of time the other thread spends in its critical section from 100 to 1000 and give the counts for a sample run.

   T1() count = _______________     T2() count = _______________

4. Change the max time both T1( ) and T2( ) are in their critical sections to 10. Leave the times each is out of the critical sections at 1000 and 100 and answer the following questions.

a. Compare the performances between these settings and the settings in 3-c. Describe the major differences you observe.

   ---

b. Which of the six properties from the list above IS NOT implemented in the 3-c version of this demonstration? ________ Explain:

   ---

5. Describe how you can change the settings (max times in and out of the critical sections) of the two threads to demonstrate that indefinite postponement (starvation) cannot occur. Note: It is important to set up the conditions for starvation in order to properly demonstrate that Dekker's Algorithm prevents it.

   ---

Submit a copy of your source code with this laboratory handout.