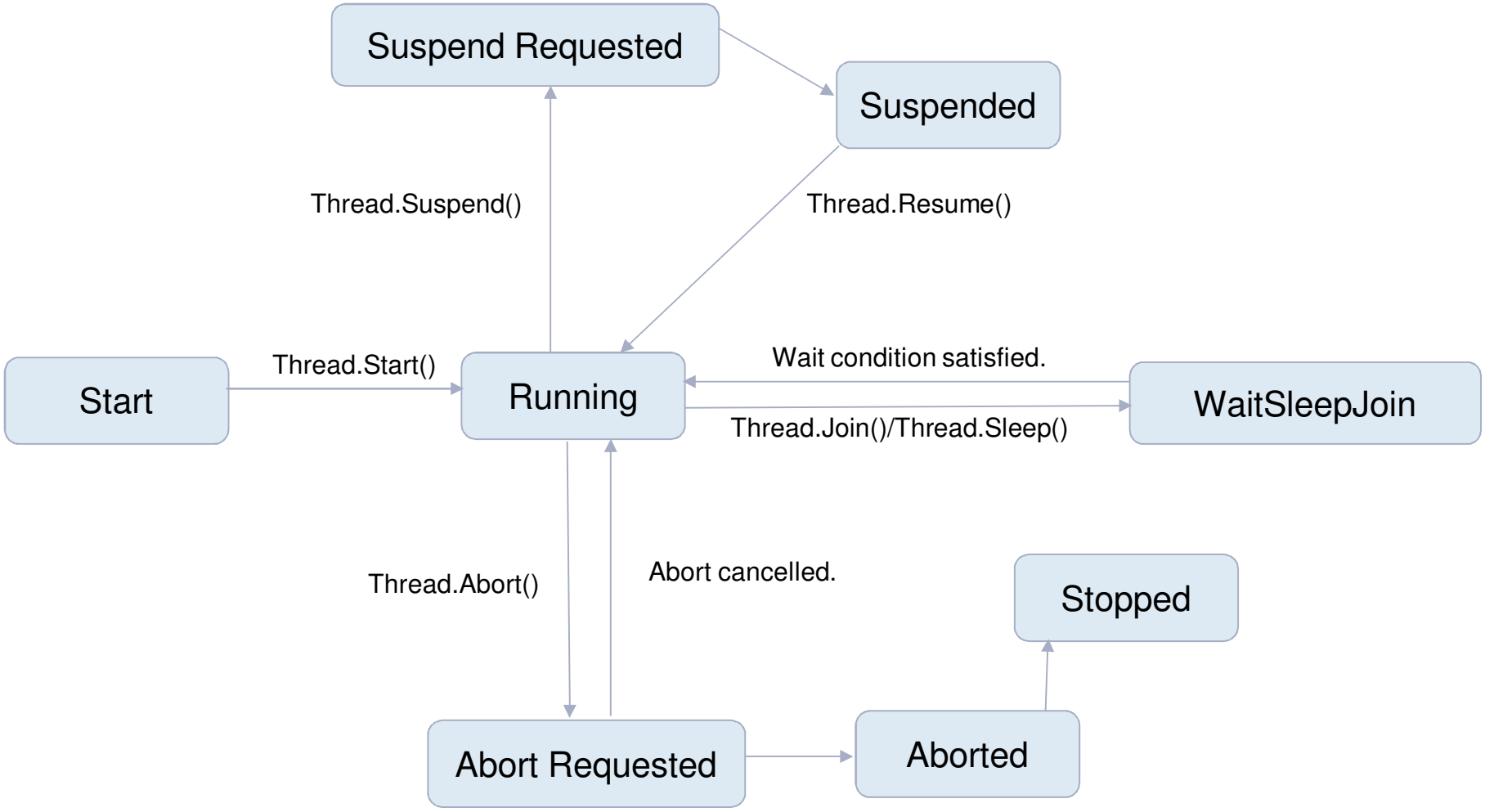
# Agenda

- Thread lifecycle
- Thread Implementation
- Multi-threading using `ThreadStart` delegate
- Multi-threading using `ThreadPool` class
- Challenges in multi-threading
- ThreadPriorities
- Race Condition
- Deadlock

# Threads and Multi-threading

- A thread or a thread of execution is a lightweight process or a context in which a program runs.
- Multiple threads within a process share some common resources and walk their own way to complete a process collectively.
- Multiple threads running simultaneously in a program is called multi-threading. For single processor environment, multiple threads are managed by means of time slices assigned to threads. When the time slice for a specific thread being executed gets over, the CPU switches to another thread for processing. This switching is called context switching.

# Thread Life Cycle



# Threading Implementation

- Threading in .NET is implemented using the classes under the System.Threading namespace.
- Threading can be implemented in two ways:
  - Using the `ThreadStart` delegate.
  - Using the `ThreadPool` class.
- For long running tasks it is preferable to create your own threads using the `ThreadStart` delegate.
- `ThreadPool` can be used for multithreading in two different ways:
  - Directly: Using `ThreadPool.QueueUserWorkItem`.
  - Indirectly: Using asynchronous methods like `Stream.BeginRead()` or the `BeginInvoke()` method of a delegate. These methods use the `ThreadPool` class internally.

# Threading Implementation (cont.)

- By default there are 25 threads in the thread pool. When all the 25 threads are in use then the request for a thread falls into a queue. Whenever any thread gets released, it takes up the item in the queue.
- Since the thread pool is being used by the .NET Framework libraries also, it should be used mainly for tasks running for a short time.
- Implementation of asynchronous delegate calls using BeginInvoke() has been explained in Module 5.4.

# Multithreading Using `ThreadStart` Delegate

```
using System;
using System.Threading;

namespace TestThreading
{
   class Program
   {
      static void Main()
      {
         ThreadStart task1 = new ThreadStart(ProcessTask1);
         ThreadStart task2 = new ThreadStart(ProcessTask2);
         Thread t1 = new Thread(task1);
         Thread t2 = new Thread(task2);
         t1.Start();
         t2.Start();

         Console.ReadLine();
      }

      static void ProcessTask1()
      {
         for (int i = 0; i < 5; i++)
         {
            Console.WriteLine("Thread t1 in action");
            Thread.Sleep(5);
         }
      }
      static void ProcessTask2()
      {
         for (int i = 0; i < 5; i++)
         {
            Console.WriteLine("Thread t2 in action");
            Thread.Sleep(10);
         }
      }
   }
}
```

Need to invoke these methods asynchronously.

Create 2 instances of ThreadStart delegate and associate the methods ProcessTask1() and ProcessTask2() to them.

Start executing methods ProcessTask1 and ProcessTask2 by starting threads t1 and t2 respectively.

Create 2 new threads and pass the delegates task1 and task2 to the constructors of the thread.

# Multithreading Using `ThreadStart` Delegate (cont.)

- Output 1

```
Thread t1 in action
Thread t2 in action
Thread t1 in action
Thread t1 in action
Thread t2 in action
Thread t1 in action
Thread t1 in action
Thread t2 in action
Thread t2 in action
Thread t2 in action
```

- Output 2

```
Thread t1 in action
Thread t1 in action
Thread t2 in action
Thread t1 in action
Thread t1 in action
Thread t2 in action
Thread t1 in action
Thread t2 in action
Thread t2 in action
Thread t2 in action
```

# Controlling Threads

- We need to maintain explicit control over a thread when creating and maintaining it.
- To switch between the different states of the thread lifecycle, the Thread class offers the following methods:
  - Thread.Join(): This method blocks the calling thread until the thread terminates.
  - Thread.Sleep(): This method blocks the thread for a specified amount of time in milliseconds.
  - Thread.Abort(): When the Abort() method is called, the thread terminates.
  - Thread.Start(): This method starts a thread and the state of the thread changes to running.
- Some Flags / properties which help in tracking a thread's state are:
  - IsAlive: Gets the execution status of the thread.
  - ThreadState: Gets the a value containing the states of the thread.
  - IsBackground: Determines whether a thread is a background thread or not.

# Multithreading Using `ThreadStart` Delegate

```
using System;
using System.Threading;
namespace TestThreading
{
   class Program
   {
      public static void Main()
      {
         ThreadStart task1 = new ThreadStart(ProcessTask1);
         Thread t1 = new Thread(task1);
         t1.Start();
         if (t1.IsAlive == true)
         {
            Console.WriteLine("Thread t1 is alive");
         }
         Console.WriteLine("Main thread in action");
         t1.Join();
         if (t1.IsAlive == false)
         {
            Console.WriteLine("Thread t1 is not alive");
         }
         Console.WriteLine("Main thread in action");
      }
```

```
      static void ProcessTask1()
      {
         for (int i = 0; i < 5; i++)
         {
            if (i == 4)
            {
               Thread.CurrentThread.Abort();
            }
            Console.WriteLine("Thread t1 in action");
            Thread.Sleep(5);
         }
      }
   }
}
```

```
Thread t1 is alive
Main thread in action
Thread t1 in action
Thread t1 in action
Thread t1 in action
Thread t1 in action
Thread t1 is not alive
Main Thread in action
```

# Multithreading Using the `ThreadPool` Class

```
using System;
using System.Threading;
namespace TestThreading
{
   class Program
   {
      public static void Main()
      {
         ThreadPool.QueueUserWorkItem(new WaitCallback(ProcessTask),null);
         Console.WriteLine("Main thread in action");
         Thread.Sleep(2000);
         Console.WriteLine("Main thread in action");
      }
      static void ProcessTask(object param)
      {
         for (int i = 0; i < 5; i++)
         {
            Console.WriteLine("Thread t1 in action");
          }
      }
  }
}
```

Invoking the method ProcessTask using ThreadPool class

This line waits for ProcessTask to complete

```
Main thread in action
Thread t1 in action
Thread t1 in action
Thread t1 in action
Thread t1 in action
Thread t1 in action
Main Thread in action
```

Control passes to ProcessTask() method

# ThreadPriorities

- ThreadPriorities decides the execution order of the objects, as in, the first position, the intermediate position and the last position.
- Thread priority is based on the importance of threads.
- The ThreadPriority enumeration defines the set of all possible values.
- Can get or set the threads priority with the priority property of the Thread Class.
- The following table lists the priorities that can be assigned to the thread.

| | |
|---|---|
| **Lowest** | The thread with this priority can be scheduled after threads with any other priority. |
| **BelowNormal** | The thread with this priority can be scheduled after threads with the Normal priority and before those with lowest priority. |
| **Normal** | The thread with this priority can be scheduled after threads with the AboveNormal priority and before those with BelowNormal priority. Threads have the normal priority by default. |
| **AboveNormal** | The thread with this priority can be scheduled after threads with Highest priority and before those with Normal priority. |
| **Highest** | The thread with this priority can be scheduled before threads with any other priority. |

# ThreadPriorities (cont.)

- Examples:

```
Public static void Main(string[] args)
{
    Thread ThreadOne=new Thread(FuncationOne);
    Thread ThreadTwo=new Thread(FuncationTwo);
    Thread ThreadThree=new Thread(FuncationThree);
    ThreadOne.Priority=ThreadPriority.Lowest;
    ThreadTwo.Priority=ThreadPriority.Highest;
    ThreadThree.Priority=ThreadPriority.Normal;
    ThreadOne.Start();
    ThreadTwo.Start();
    ThreadThree.Start();

}
```

# Challenges in Multi-threading

- Race conditions in multi-threading can lead to unpredictable and inconsistent results. This can be avoided using locks. But locks applied to long running code blocks can degrade performance. Also, locks are the main source of deadlocks.
- Deadlocks (due to multi-threading) can put a system to halt for an indefinite period of time.
- Threads normally share a common set of data.

# Race Condition

- It is normal for one thread to use the data of another thread. In such a case there will be a scenario when two or more threads will race to reach a code block first. As a result the data may get manipulated on its previous value producing unexpected results. This situation is called a Race Condition.
- Race conditions not only give unexpected results, but also provide different outputs at different points in time.
- Race conditions can be avoided by applying locks on the code block causing the race condition. Locks on long running code blocks result in performance degradation. Hence locks should be applied to code blocks running for a short period of time.

# Race Condition (cont.)

```
using System;
using System.Threading;
namespace TestThreading
{
   class Program
   {
      static int count = 0;
     static object padlock = new object();
      public static void Main()
      {
        ThreadStart td = new ThreadStart(ProcessTask);
         Thread t1 = new Thread(td);
         t1.Start();
         for (int i = 0; i < 5; i++)
         {
            int temp = count;
            Thread.Sleep(5);
            temp = temp + 1;
            count = temp;
         }
         t1.Join();
         Console.WriteLine(count);
      }
```

The two blocks of code that are manipulating the static variable **count** asynchronously.

```
      static void ProcessTask()
      {
         for (int i = 0; i < 5; i++)
         {
            int temp = count;
            Thread.Sleep(5);
            temp = temp + 1;
            count = temp;
         }
      }
   }
}
```

Applying locks

```
lock (padlock)
{
   int temp = count;
   Thread.Sleep(5);
   temp = temp + 1;
   count = temp;
}
```

Output 1

```
5
```

Output 2

```
8
```

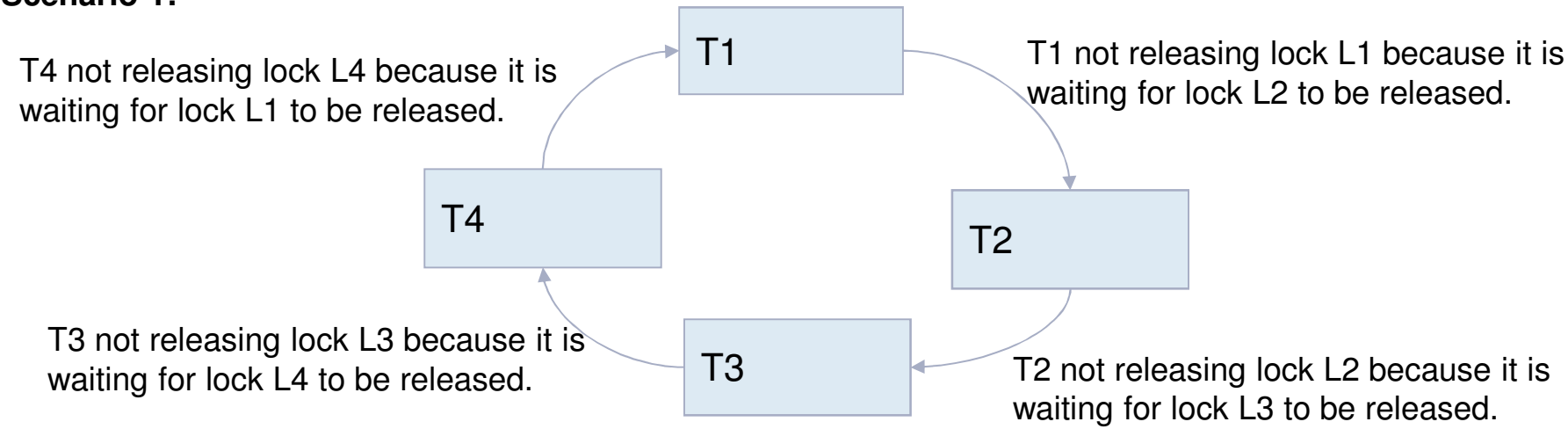
Corrected Output

```
10
```

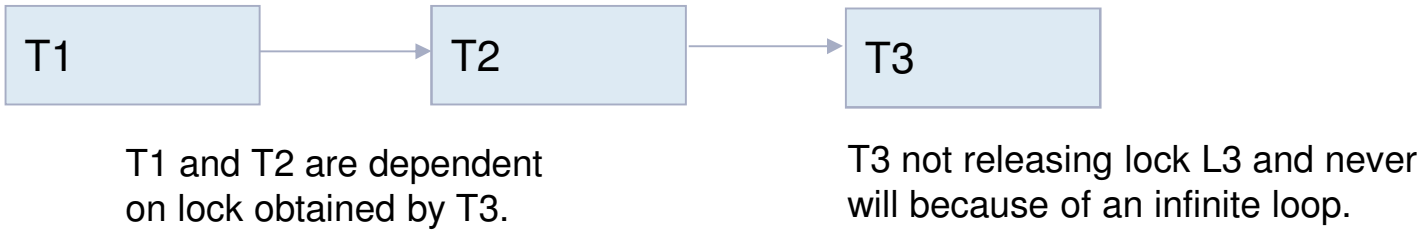
# Deadlocks

- When each of the threads in a program is waiting for a lock to get released, the situation is termed a Deadlock.

**Scenario 1:**




**Scenario 2:**



# Deadlocks (cont.)

```
using System;
using System.Threading;
namespace TestThreading
{
   class Program
   {
       static int count = 0;
      static object firstLock = new object();
      static object secondLock = new object();
      public static void Main()
      {
        ThreadStart td = new ThreadStart(ProcessTask);
         Thread t = new Thread(td);
         t.Start();
         Thread.Sleep(500);
         lock (secondLock)
        {
             Console.Writeline("Main: Second lock applied");
              lock (firstLock)
              {
                 Console.Writeline("Main: First lock applied");
              }
             Console.Writeline("Main: First lock released");
         }
         Console.WriteLine("Main: Second lock released");
      }
      static void ProcessTask()
      {
         lock (firstLock)
         {
             Console.Writeline("t: First lock applied");
             Thread.Sleep(1000);
             lock (secondLock)
             {
                 Console.Writeline("t: Second lock applied");
             }
             Console.Writeline("t: Second lock released");
         }
         Console.WriteLine("t: First lock released");
      }
   }
}
```

Thread t is invoked.

Main thread sleeps for 500ms to ensure that t acquires first lock.

T sleeps for 1000ms to ensure main thread grabs second lock.

```
t: First lock applied
Main: Second lock applied
```

Main trying to acquire firstLock but it is held by t.

T trying to acquire secondLock but it is held by Main