Threading using C# and .Net

cs795

Satish Lakkoju

- Outline :
  - Threads
  - System.Threading Namespace .
  - Thread Class – its methods and properties.
  - Thread Synchronization.
  - Monitors
  - C# Lock keyword.
  - Reader/Writer Locks
  - Conclusion

Threads :

- Thread is the fundamental unit of execution.
- More than one thread can be executing code inside the same process (application).
- On a single-processor machine, the operating system is switching rapidly between the threads, giving the appearance of simultaneous execution.

- With threads you can :
  - Maintain a responsive user interface while background tasks are executing
  - Distinguish tasks of varying priority
  - Perform operations that consume a large amount of time without stopping the rest of the application

- System.Threading Namespace
  - Provides classes and interfaces that enable multithreaded programming.
  - Consists of classes for synchronizing thread activities .
  - Chief among the namespace members is Thread class

- Thread Class
  - Implements various methods & properties that allows to manipulate concurrently running threads.

  - Some of them are :
    - CurrentThread
    - IsAlive
    - IsBackground
    - Name
    - Priority
    - ThreadState

- Starting a thread :

```
Thread thread = new Thread(new ThreadStart (ThreadFunc));
//Creates a thread object
// ThreadStart identifies the method that the thread executes when it
//starts

thread.Start();
//starts the thread running
```

Thread Priorities :

Controls the amount of CPU time that can be allotted to a thread.

```
ThreadPriority.Highest
ThreadPriority.AboveNormal
ThreadPriority.Normal
ThreadPriority.BelowNormal
ThreadPriority.Lowest
```

- Suspending and Resuming Threads
  - Thread.Suspend temporarily suspends a running thread.
  - Thread.Resume will get it running again
  - Sleep : A thread can suspend itself by calling Sleep.

  - Difference between Sleep and Suspend

- A thread can call sleep only on itself.

-Any thread can call Suspend on another thread.

- Terminating a thread
  - Thread.Abort() terminates a running thread.
  - In order to end the thread , Abort() throws a ThreadAbortException.

  - Suppose a thread using SQL Connection ends prematurely , we can close the the SQL connection by placing it in the finally block.

```
-    SqlConnection conn .........
     try{
           conn.open();
           ....
           .....
      }
      finally{
          conn.close();//this gets executed first before the thread ends.
      }
```

- A thread can prevent itself from being terminated with Thread.ResetAbort.

```
- try{
  …
  }
  catch(ThreadAbortException){
  Thread.ResetAbort();
 }
```

- Thread.Join()
  - When one thread terminates another, wait for the other thread to end.

- Thread Synchronization :
  - Threads must be coordinated to prevent data corruption.

- Monitors
  - Monitors allow us to obtain a lock on a particular object and use that lock to restrict access to critical section of code.
  - While a thread owns a lock for an object, no other thread can acquire that lock.
  - Monitor.Enter(object) claims the lock but blocks if another thread already owns it.
  - Monitor.Exit(object) releases the lock.

- Void Method1()

```
{
 ....
 Monitor.Enter(buffer);
 try
 {
       critical section;
 }
 finally
 {
       Monitor.Exit(buffer);
 }
}
```

Calls to Exit are enclosed in finally blocks to ensure that they're executed even when an exception arises.

- The C # Lock Keyword :

```
lock(buffer){
.......
}
```

is equivalent to

```
Monitor.Enter(buffer);
try
{
 critical section;
}
finally
{
 Monitor.Exit(buffer);
}
```

- Makes the code concise.
- Also ensures the presence of a finally block to make sure the lock is released.

- Reader/Writer locks :
  - Prevent concurrent threads from accessing a resource simultaneously.
  - Permit multiple threads to read concurrently.
  - Prevent overlapping reads and writes as well as
    overlapping writes.
  - Reader function uses :
    -AcquireReaderLock
    -ReleaseReaderLock
  - Writer funciotn uses :
    -AcquireReaderLock
    -ReleaseReaderLock

ReleaseLocks are enclosed in finally blocks to be absolutely certain that they

- Drawback :

  Threads that need writer locks while they hold reader locks will result in deadlocks.
  Solution is UpgradeToWriterLock and DowngradeFromWriterLock methods.

```
rwlock.AcquireReaderLock(Timeout.Infinite)
try{
  // read from the resource guarded by the lock
  .....
  //decide to do write to the resource

  LockCookie cookie = rwlock.UpgradeToWriteLock(Timeout.Infinite)

  try{
  // write to the resource guarded by the lock
  .....
  }
  finally{
            rwlock.DowngradeFromWriterLock(ref cookie);
  }
 }
finally{
  rwlock.ReleaseReaderLock();
}
```

- MethodImpl Attribute
  - For synchronizing access to entire methods.
  - To prevent a method from be executed by more than one thread at a time ,

```
[MehtodImpl] (MethodImplOptions.Synchronized)]
Byte[] TransformData(byte[] buffer)
{
……
}
```

Only one thread at a time can enter the method.

- Conclusion :
  - Using more than one thread, is the most powerful technique available to increase responsiveness to the user and process the data necessary to get the job done at almost the same time.

- References :
  - Programming Microsoft .NET – Jeff Prosise
  - http://msdn2.microsoft.com/
  - http://cs193n.stanford.edu/handouts/pdf/37%20Streams,%20Multithreading.pdf