**Multitasking**

* Multitasking is the concept of running multiple applications at the same time.
* For example, Windows operating system is the multitasking operating system.

**Process**

* A Process is a part of component of the operating with is responsible for executing the program or application. So, to execute each program or application, there will be a process.
* To see all the services which is available, you can go to run command and type:

**services.msc**

* So, under the operating system, we have processes that run our applications. So, under the process, an application runs. To run the code of an application the process will make use of a concept called Thread.

**What is Thread?**

**Threading**

* So, under the operating system, we have some process that runs our applications. So, under the process, an application runs. To run the code of an application the process will make use of a concept called Thread.
* Generally, when we talk about Thread, **Thread is a lightweight process**. A process has at least one thread which is commonly called as **main (current) thread** which executes the application code.
* A single process can have multiple threads.
* All the threading related classes are present in **System.Threading** namespace.

**Note:** All thread related function and properties are existed in below namespace:

**using System.Threading**

**Thread Class**

The Thread class helps us to perform tasks such as creating and setting the **priority** of a thread. We can use this class to control a thread and **obtain its status**. The thread class provides various properties that allow us to perform tasks such as obtaining the status of thread and specifying a name for the thread:

* **CurrentThread**: It retrieves the current thread which is running.
* **IsAlive:** It retrieves a value to indicate the current state of thread executing. The value of the IsAlive property is **true** if the thread has been **started** or has been **terminated**, otherwise the value is **false**.
* **Name:** It specifies a name for a thread.
* **Priority:** To obtain a value which indicates the scheduling priority of a thread. **By default**, the value of the Priority property is **Normal**. We can assign eighter **Highest**, **AboveNormal**, **Normal**, **BelowNormal** or **Lowest** value to the priority property.

**Methods of the Thread Class**

**Interrupt:** To interrupt the thread which is in the **WaitSleepJoin** state.

**Join:** To block the thread until another thread has **terminated**.

**Resume:** To resume the thread which has been **suspended** earlier.

**Sleep:** To block the thread for a specified time (period).

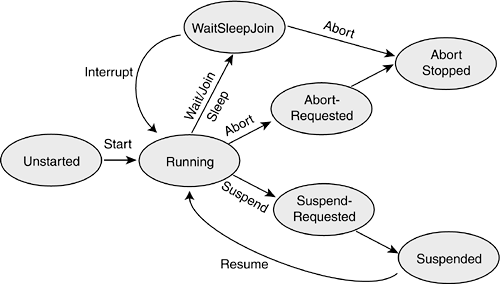
**SpinWait:** To make a thread wait the number of times specified in the iteration parameter.

**Start:** To start the thread.

**Suspend:** To suspend the thread.

Thread Lifecycle in C#

In C#, the tread lifecycle specifies the current status of the thread. The thread automatically switches from one state to another state. **Sometimes**, you can **switch its state** by **using the methods** offered by Thread Class.



**Ready / Unstarted**

* It is the initial state. The thread that has just been created.
* A thread enters the unstarted state when an object of thread class is created but the start method has not been called on it yet.
* In the unstarted state, a thread is **not** considered **alive** as it is not a thread of execution. Once the start method is called on the thread, it leaves the unstarted state and enters to the next state, but once it leaves unstarted state, **it is impossible** for a thread to return to this state in its lifecycle.
* The unstarted state could also be termed as the **new state**.

**Running State**

A thread enters the running state, when the thread scheduler has selected it to run (out of all the threads in the thread pool). In this state, a thread starts executing the Run method and it is alive and kicking. From the running state, a thread can enter to **not runnable** / **waiting** / **blocked** / **runnable** / **final** / **dead state**.

**Sleeping / Not Runnable State**

* The thread is temporarily paused. The .NET offers automatic switching between running and sleeping states when other threads are executed.
* A thread enters the **not runnable state** in one of these situations:
  + When a thread has called the **Wait** method on itself, and it is waiting for the other thread to notify it or wake it up.
  + When the **Sleep** method is called on the thread, asking in to sleep for a duration.
  + When a thread has called the **Join** method on another thread, which makes the first thread wait until another thread has finished its execution.
  + When a thread is waiting for an **input / output resource to be free**.
  + Apparently, some may also refer to this state as the **WaitSleepJoin** **state**.
* When a thread finds itself in any of the above-mentioned situations, such an event pushes the thread into the **not runnable state** and the thread is no longer eligible to run, but event in this state, the thread is still considered to be **alive**.
* **Note: When the thread gets out of this not runnable state, it re-enters into runnable state.**

**Suspended State**

* The thread is **suspended**. It will continue when you call **Resume** method.

**Dead State**

This is the **last state** in a thread’s lifecycle. A thread enters the dead state after it has successfully completed executing its entry point method i.e., **Run**, or when the **Abort** method has been called on it to abort its execution. In this state, the thread **is not alive** and hence if you try to call **Start** method on a dead thread, it raises the **ThreadStateException** exception.

Drawbacks of Single Thread Applications?

**How to implement multithreading in C#**

**Constructors of Thread Class in C#**

**ThreadStart Delegate**

**Learn 2**

**IsAlive and Join Method**

* In C#, Thread class provides the **Join()** method which allows the thread to wait until another thread completes its execution.
* If **‘t’** is a thread object whose is currently executing, the ‘**t.Join()**’ causes the current thread to pause its execution until another thread that joins, completes its execution.
* Join() method also has an overload where we can specify the timeout.
* If we do not specify the timeout, the current thread waits indefinitely, until the thread on which Join() is invoked completes.
* The overloaded ‘**Join(int milliseconds timeout)**’ method returns Boolean. ‘**true**’ if the thread has terminated otherwise **false**.
* Join is particularly useful when we need to wait and collect results from a thread execution or if we need to do some cleanup after the thread has completed.
* The **IsAlive** property returns Boolean. ‘**true**’ if the thread is still executing, otherwise ‘**false**’.

**Learn 3**

**Protecting shared resources from concurrent access in multithreading using locking**

We can protect shared resources from concurrent access in multithreading using below scenarios:

* **Interlocked.Increment()**
* **lock()** method
* **Monitor()** method

**Lock vs Monitor**

* The lock is the shortcut for **Monitor.Enter()** with try and finally.
* So, the lock provides the basic functionality to acquire an exclusive lock on a synchronized object.
* But, if you want more control to implement advanced multithreading solutions using **TryEnter()**, **Wait()**, **Pulse()**, and **PulseAll()** methods, then the **Monitor** class is the best option.

**Monitor Pulse(), Wait(), PulseAll() Methods**

* Monitor is also a locking mechanism that will ensure one thread is executing a piece of code at the same time.
* Monitor is no different from lock, but the monitor class provides more control over the synchronization of various threads trying to access the same lock of code.
* Monitor locks objects. While you can pass a value type to Enter() and Exit() methods, it is boxed separately for each call.
* Monitor has a signaling mechanism, like **Wait()**, **Pulse()** and **PulseAll()** methos for signaling / communication between threads.

**Monitor.Wait()**

A thread waits for other threads to notify.

**Monitor.Pulse()**

A thread notifies another thread.

**Monitor.PulseAll()**

A thread notifies all other threads within a process.

**Advantage of Synchronization**

* To avoid the deadlock
* To maintain consistency

**Manual Reset Event**