Before understanding the concept of Multithreading in C#, let us first understand multitasking.

Multitasking means performing multiple tasks simultaneously. Windows Operating System is a multitasking operating system. It means it has the ability to run multiple applications at the same time. For example, on my machine, I can open the Google Chrome Browser, Microsoft Word Document, Notepad, VLC Media Player, Visual Studio, etc., at the same time. This is possible because, on my machine, I have installed the Windows operating system, which is a multitasking operating system.

So, Multitasking refers to the concurrent execution of multiple tasks or processes on a computer system. It allows a computer to appear as though it is performing multiple tasks simultaneously, even though it typically has a single central processing unit (CPU). Multitasking is a fundamental feature of modern operating systems and plays an important role in improving computers’ efficiency, responsiveness, and usability.

**How Does the Operating System Execute Multiple Applications at a time?**

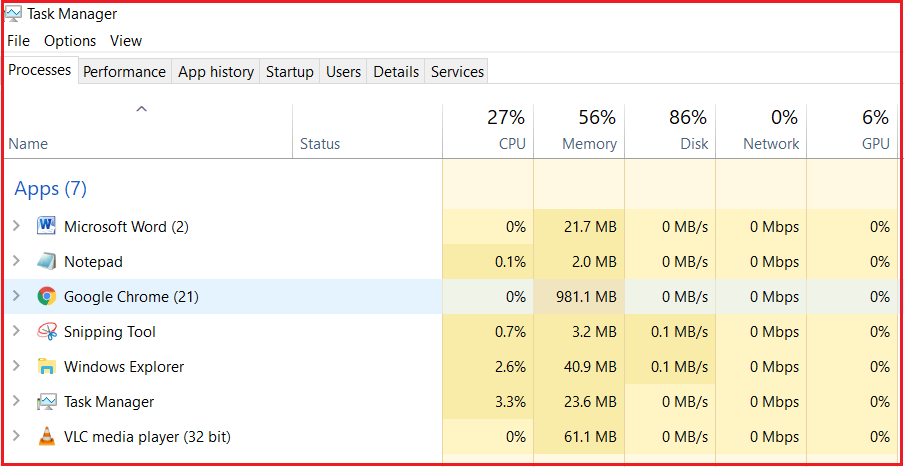
The operating system uses processes internally to execute multiple applications (Google Chrome Browser, Microsoft Word Document, Notepad, VLC Media Player, Visual Studio, etc.) simultaneously.

**What is a Process?**

A process is a part of the operating system (or a component under the operating system) responsible for executing the program or application. So, to execute each and every program or application, there will be a process.

In an operating system (OS), a process is a fundamental concept representing a program in execution. It is the smallest unit of work in a computer system’s execution. Each process has its own memory space, which includes the program code, data, and a stack for managing function calls and local variables. The OS manages processes and has its own resources, such as CPU time, file handles, and system state information.

To see the processes executing the programs or applications using the Task Manager, Just right-click on the Taskbar and click the Task Manager option to open the Task Manager window. From that window, click on the “Processes” button as shown below.



As you can see in the above image, each application is executed by one corresponding process. Along the same line, multiple processes are running in the background, known as the background processes. These background processes are known as Windows Services, and the Operating System runs many Windows services in the background.

So, we have an operating system, and under the operating system, we have processes that run our applications. So, the point that under the process, an application runs. To run the code of an application, the process internally uses a concept called Thread.

**What is Thread?**

Generally, a Thread is a lightweight process. In simple words, we can say that a Thread is a unit of a process that is responsible for executing the application code. So, every program or application has some logic or code, and to execute that logic or code, Thread comes into the picture.

In an operating system (OS), a thread is a smaller unit of execution within a process. Threads are sometimes called “lightweight processes” because they share the same memory space as the parent process, including its code, data, and resources. However, threads have their own execution context, including a program counter, registers, and stack. Threads within the same process can run concurrently, allowing for parallel execution of tasks.

By default, every process/application has at least one thread that is responsible for executing the application code, and that thread is called Main Thread. So, every application by default is a single-threaded application.

**Note:** All the threading-related classes in C# belong to **System.Threading** namespace.

**Example to Understand Threading in C#:**

Let us see an example to understand Threading in C#. The following is a simple program where we have a class called Program, and in that class, we have a method called Main, which simply prints a message on the Console window.

using System;

namespace ThreadingDemo

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Welcome to Dotnet world!");

Console.ReadKey();

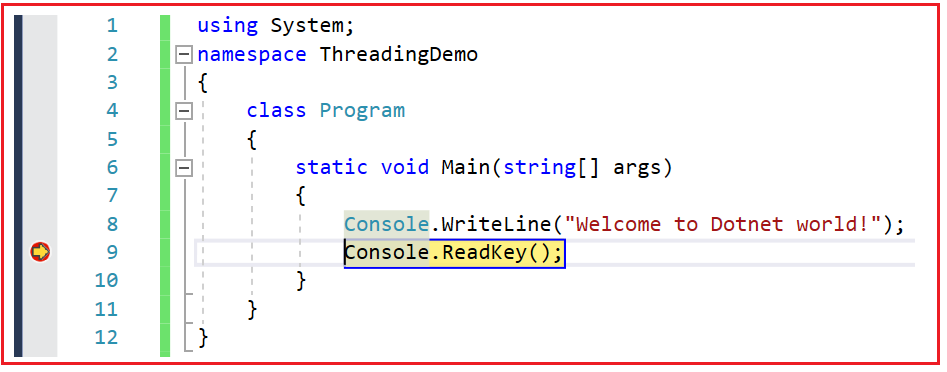
}

}

}

**Does the above Program Make use of Thread When Run?**

Yes, there is a Thread, and that thread will execute our application code, which is called the Main Thread. Now, let us prove this. Put a debugger point on your application code and then run the application. Let us assume you put the debugger point on line number 9, and when you run the application, you will see that the debugger will hit that point, as shown in the below image.



**Another Example**

using System;

namespace ThreadingDemo

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Welcome to Dotnet world!");

Thread t = Thread.CurrentThread;

t.Name= "Main Thread"; // Explicitly Given a name to Main Thread by default it has no name

Console.WriteLine("Current Executing Thread is : "+ Thread.CurrentThread.Name);

Console.WriteLine();

Console.ReadKey();

}

}

}

In single threaded application, the single thread is responsible for executing the particular method, if in case of multiple methods called from the main() then the main thread is responsible to execute all the methods one by one that are explicitly called. By default, every program is single threaded and executes one after the other.

Assume we have three methods Test1(), Test2(), and Test3(), that are getting executed one after another like Test1() executes first, Test2() executes second, and Test3() executes third.

**Drawback:**

If in case Test1() is performing some database action and taking time to load the data from the database. In that case the Test2() has to wait till the Test1() to complete the task because the methods are executing in sequential order and in same way Test3() also has to wait for the completion of Test2().

Example :

using System.Diagnostics;

namespace Multithreading

{

internal class MultiThreadDemo1

{

static Stopwatch sw = new Stopwatch();

static void Test1()

{

sw.Start();

for (int i = 0; i <= 100; i++)

{

Console.WriteLine("Test1 " + i);

if (i == 50)

{

Console.WriteLine("Thread Sleep");

Thread.Sleep(15000); //Puts the Current Thread To Sleep i.e. MainThread

Console.WriteLine("Thread woke-up");

}

}

sw.Stop();

Console.WriteLine("Ticks 1 : " + sw.ElapsedTicks + " ms : " + sw.ElapsedMilliseconds);

}

static void Test2()

{

sw.Start();

for (int i = 0; i <= 100; i++)

{

Console.WriteLine("Test2 " + i);

}

sw.Stop();

Console.WriteLine("Ticks 2 : " + sw.ElapsedTicks + " ms : " + sw.ElapsedMilliseconds);

}

static void Test3()

{

sw.Start();

for (int i = 0; i <= 100; i++)

{

Console.WriteLine("Test3 " + i);

}

sw.Stop();

Console.WriteLine("Ticks 3 : " + sw.ElapsedTicks + " ms : " + sw.ElapsedMilliseconds);

}

public static void Main()

{

Test1();

Test2();

Test3();

}

}

}

To overcome this problem the new concept is introduced and called as Mult-Threading:

Till now there was only one thread per process and which is executing every method one after another.

Process can have multiple threads and each thread is performing some actions in parallel with other.

Example: Test1() Test2() and Test3() if at any cost Test1() is performing some action Test2() and Test3() will be executed based on time sharing, and every thread will be given equal importance.

Output of above program wont change! it is fixed every time we execute it.

**Advantage of Multi-Threading is Maximum Utilization of CPU usage.**

Example of multi-threading:

using System.Diagnostics;

namespace Multithreading

{

internal class MultiThreadDemo2

{

static Stopwatch sw = new Stopwatch();

static void Test1()

{

sw.Start();

for (int i = 0; i <= 100; i++)

{

Console.WriteLine("Test1 " + i);

if (i == 50)

{

Console.WriteLine("Thread Sleep");

Thread.Sleep(15000);

Console.WriteLine("Thread woke-up");

}

}

sw.Stop();

Console.WriteLine("Ticks 1 : " + sw.ElapsedTicks + " ms : " + sw.ElapsedMilliseconds);

}

static void Test2()

{

sw.Start();

for (int i = 0; i <= 100; i++)

{

Console.WriteLine("Test2 " + i);

}

sw.Stop();

Console.WriteLine("Ticks 2 : " + sw.ElapsedTicks + " ms : " + sw.ElapsedMilliseconds);

}

static void Test3()

{

sw.Start();

for (int i = 0; i <= 100; i++)

{

Console.WriteLine("Test3 " + i);

}

sw.Stop();

Console.WriteLine("Ticks 3 : " + sw.ElapsedTicks + " ms : " + sw.ElapsedMilliseconds);

}

public static void Main()

{

Thread T1 = new Thread(Test1);

Thread T2 = new Thread(Test2);

Thread T3 = new Thread(Test3);

T1.Start();T2.Start();T3.Start();

}

}

}

We can see there are total 4 threads in above example MultiThreadDemo2 (T1,T2,T3 and main Thread).

CPU is allocating fixed time to execute each one of them. If in case thread T1 is waiting for action to be completed on another resources, other threads T2 and T3 will get executed, without waiting for the execution of current thread.

Above T1 is waiting for the completion of some task and for that reason T2 and T3 are executing in between, as if T1 gets the response, then it also starts its execution in between the T1 and T2.

But in job of main thread is to start the program, as it finishes early so exiting and thread whichever finishes the task will exit one by one.

Here CPU doesn’t wait for the task to complete, instead it gives chance to execute the other to get executed.

Output of above program changes every time we execute it.

**Advantage:**

So this will speed up the performance of the applications, and CPU utilization.

Note: Internally Thread T1 = new Thread(Test1); Thread Constructor will call ThreadStart Delegate and pass it as a parameter and that is done by the Framework CLR for us. All this happens implicitly.

Constructures of Thread Class:

ThreadStart obj = new ThreadStart(Test1); or ThreadStart obj = Test1;

Thread T = new Thread(obj);

T.Start();

Here we are calling ThreadStart explicitly.

Which is similar to

Thread T = new Thread(Test1); T.Start();

[A delegate is a type safe function pointer to method. Instantiation is the process of binding method with a delegate and can be done in following ways

//I //1. ThreadStart obj = new ThreadStart(Test1); Or

//2. ThreadStart obj = Test1; or

//3. ThreadStart obj = delegate { Test1(); }; or

//4. ThreadStart obj = () => Test1(); or

//5. ThreadStart obj = () => { // Method body };

//Thread T = new Thread(obj);

//T.Start();

//I or II

//II. Thread T = new Thread(Test1); T.Start();

]

To pass the values to method using delegates in multithreading use ParameterizedThreadStart delegate.

Thread.Join() : The Join method of the Thread class waits for the thread to complete its execution before continuing with the execution of the calling thread.

If at all we don’t want to exit the main thread/caller thread before the completion of its child / called threads then we should use Thread.Join().

Example 1 () Join without parameters:

namespace Multithreading

{

/// <summary>

/// In previous examples main thread gets exited befor completion of other 3 threads and

/// this should not happen. To avoid this use Thread.Join().

/// </summary>

internal class MultiThreadingDemo3\_Join

{

static void Test1()

{

Console.WriteLine("Test1 Thead is starting");

for (int i = 0; i < 10; i++)

{

Console.WriteLine("Test1() : " + i);

Thread.Sleep(5000); //delayed to test when main thread is exitin.i.e. at the end

}

Console.WriteLine("Test1 Thead is exiting");

}

static void Test2()

{

Console.WriteLine("Test2 Thead is starting");

for (int i = 0; i < 10; i++)

{

Console.WriteLine("Test2() : " + i);

}

Console.WriteLine("Test2 Thead is exiting");

}

static void Test3()

{

Console.WriteLine("Test3 Thead is starting");

for (int i = 0; i < 10; i++)

{

Console.WriteLine("Test3() : " + i);

}

Console.WriteLine("Test3 Thead is exiting");

}

static void Main()

{

Console.WriteLine("Main Thread Started");

Thread t1 = new Thread(Test1);

Thread t2 = new Thread(Test2);

Thread t3 = new Thread(Test3);

t1.Start(); t2.Start(); t3.Start();

t1.Join(); t2.Join(); t3.Join();//Without join random threads will call and/ exit any time

Console.WriteLine("Main Thread is Exiting");

}

}

}

Example 2 Join with parameters: Changes in main of example 1:

static void Main()

{

Console.WriteLine("Main Thread Started");

Thread t1 = new Thread(Test1);

Thread t2 = new Thread(Test2);

Thread t3 = new Thread(Test3);

t1.Start(); t2.Start(); t3.Start();

t1.Join(3000); t2.Join(); t3.Join();//Without join random threads will call and/ exit any time

Console.WriteLine("Main Thread is Exiting");

}

In above example 2 : t1.Join(3000); will wait for maximum 3000 milli-seconds, it doesn’t matter whether t1.Join(3000) finished its execution or not, main thread will exit after 3000 milli-seconds. If we don’t mention timebound then main thread will exit after all threads execution.

**Note:** Only parent thread will exit at the end **even** there is some delay in the execution of child thread and rest of the thread under the parent can exit at any time.

**Thread Locking: (How to call non static method using thread)**

Scenario / Issue : If in database one thread is writing a value and another is reading the same value then the context switching will create big issue.

Example 1:

namespace Multithreading

{

/// <summary>

/// Context Switching between thread is done bitween t1,t2,t3 see the output

/// </summary>

internal class MultiThreadingDemo4\_Lock

{

public void Display()

{

Console.Write("[C# is a ");

Thread.Sleep(5000);

Console.WriteLine("Object Oriented Language");

}

public static void Main()

{

MultiThreadingDemo4\_Lock obj = new MultiThreadingDemo4\_Lock();

Thread t1 = new Thread(obj.Display);

Thread t2 = new Thread(obj.Display);

Thread t3 = new Thread(obj.Display);

t1.Start(); t2.Start(); t3.Start();

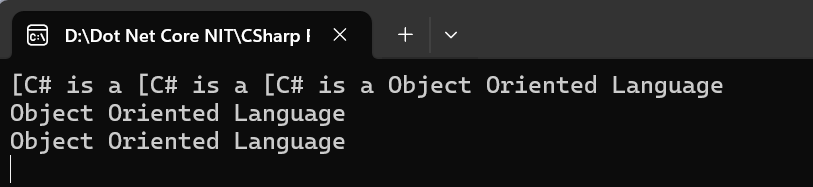
Console.ReadLine();

}

}

}

Output:



To resolve the problem of context switching we have provided the mechanism called locking.

It is a mechanism in which the sensitive code is allowed to execute by only one thread at a time i.e. and the thread gaining the control over the code is locked to avoid others to execute.

i.e. first thread is allowed to execute the Display method and after completion of first thread, second thread is allowed to execute the method, and after completion of the second thread, third thread is allowed to execute the method. No context switching between thread is done i.e. locked.

Example 2:

namespace Multithreading

{

internal class MultiThreadingDemo5\_Lock

{

/// <summary>

/// Every Thread is waiting in Wait zone to

/// finish the execution of previous thread

/// and executing one after other

/// </summary>

public void Display()

{

Console.WriteLine("Wait");

lock (this)

{

Console.Write("[C# is a ");

Thread.Sleep(5000);

Console.WriteLine("Object Oriented Language");

}

}

public static void Main()

{

MultiThreadingDemo5\_Lock obj = new MultiThreadingDemo5\_Lock();

Thread t1 = new Thread(obj.Display);

Thread t2 = new Thread(obj.Display);

Thread t3 = new Thread(obj.Display);

t1.Start(); t2.Start(); t3.Start();

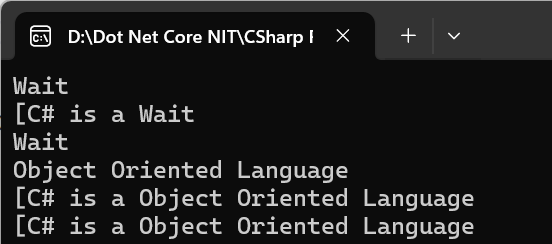
Console.ReadLine();

}

}

}

Output:



Thread Priority :

In a multi-threaded application if a method has to perform more work than the other then we can set the priorities accordingly.

We can set following priorities to the thread:

* Lowest: Least CPU Resources
* Below Normal
* Normal (Default: equal CPU utilization)
* Above Normal
* Highest: High CPU Resources

Example:

namespace Multithreading

{

internal class MultiThreadingDemo6\_Priorities

{

static long count1, count2;

public static void IncrementCount1()

{

while (true)

{

count1 += 1;

}

}

public static void IncrementCount2()

{

while (true)

{

count2 += 1;

}

}

public static void Main()

{

Thread t1 = new Thread(IncrementCount1);

Thread t2 = new Thread(IncrementCount2);

//to set priorities

t1.Priority = ThreadPriority.Highest;

t2.Priority = ThreadPriority.Lowest;

t1.Start(); t2.Start();

//lets make main thread sleep for 5000ms

Console.WriteLine("Main Thread Sleeping");

Thread.Sleep(5000);

Console.WriteLine("Main Thread Woke-up");

//to Terminate a thread use Abort()

t1.Abort(); t2.Abort();

//t1.Interrupt(); t2.Interrupt();

t1.Join(); t2.Join();

Console.WriteLine("count1 " + count2);

Console.WriteLine("count2 " + count2);

}

}

}