Threads :- It is a virtual CPU. It is the basic unit to which an OS allocates processor time. It is an independent execution within a program.

A process :- it is the running instance of the program.

Two types of threaded applications

Single threaded :- can perform only one task at a time.

Multithreaded applications (browser applications)

srolling

MulitiThreading

Thread based multitasking: - small unit of task in single application, are called Light weight applications.

Example (Word 🡪 keyboard buffering, storing of data, checking of spelling and grammar

(Browser 🡪 Loading the images , Scrolling of page)

Process based multitasking :- When more than one processes (applications) are executed parallely in a main thread.

Example (OS :- word, browser, calculator, Timer)

Console.WriteLine("Creation of Our Main Thread");

Thread th = Thread.CurrentThread;

th.Name = "Main Thread";

Console.WriteLine(th.Name);

Working with Threads

Createing, managing and destroying , we have to understand the life cycle of a thread

class Program

{

public static void childthreadcall()

{

Console.WriteLine("Child thread called");

}

static void Main(string[] args)

{

Console.WriteLine("=======Creation of Our Main Thread=========");

Console.WriteLine("Calling child thread");

ThreadStart ts = new ThreadStart(childthreadcall);

Thread th = new Thread(ts); // it is unstarted / ready mode

th.Start(); // runnable mode

Console.WriteLine("===========Main Ends here======");

}

}

}

using System;

using System.Threading;

namespace ThreadedDemo

{

class Program

{

public static void FunctionCall1()

{

Console.WriteLine("=========Child begins here======");

for (int i = 1; i <= 10; i++)

{

Console.WriteLine(i);

}

Console.WriteLine("============ Child Ends here=======");

}

public static void FunctionCall2()

{

Console.WriteLine("=========Child begins here======");

for (int i = 11; i <= 20; i++)

{

Console.WriteLine(i);

}

Console.WriteLine("============ Child Ends here=======");

}

static void Main(string[] args)

{

Console.WriteLine("======== Main begins here========");

Thread th = Thread.CurrentThread;

th.Name = "Main Thread";

Console.WriteLine("Current Thread {0}", th.Name);

// object of thread

Console.WriteLine("====Main calling child thread=====");

ThreadStart ts1 = new ThreadStart(FunctionCall1);

Thread t1 = new Thread(ts1);

t1.Start(); // runnable mode

ThreadStart ts2 = new ThreadStart(FunctionCall2);

Thread t2 = new Thread(ts2);

t2.Start();

Console.WriteLine("======== Main Loaded completely====");

}

}

}

To run any program under OS is called as process

Timer- asycn

browser

word

OS behind the scene is running multiple processes called as Windows Services.

Thread is a Virtual CPU , a unit which executes the code inside application. Every application has one primary thread by default ie Main thread.

Every program by default is single threaded model.

Main()

Thread t= Thread.CurrentThread;

t.Name=”My Thread”;

C.WL(Thread.CurrentThread.Name);

// call all 3 static methods

class ThreadDemo{

//create 3 static methods (printing numbers from 1 to 20

Tes1() Test2() Test3()

In Test2()

For(){ if(i==10) Thread.Sleep(5000);

}

As test2 is on sleep it will not allow another method to execute.

(It is a drawback in single threaded model).

To overcome this drawback we have MultiThreaded concept.

Process :- with multiple threads (actions or set of activities)

Previous example can be executed simaltenously by implementing Multiple threads.

T1 🡺 Test1() t2 🡪 Test2() t3🡪 Test3()

OS is going to allocate some time period to execute or share the time slice between each thread.

Advantage :- maximum utilization of CPU resources.

Main()

//creating child threads

Thread t1=new Thread(Test1);

Thread t2=new Thread(Test2);

Thread t3=new Thread(Test3);

T1.start(); t2.start(); t3.start();

Managing Threads

We need to manage the life or the activity of a thread. This can be achieved by using various methods of thread class.

We want the running thread to pause for a period of time, so as to allow another thread to be in running mode.

**Thread.Sleep(int ms);** :- This takes single argument represents time in milliseconds for which you want the thread to pause. It is a static method which is called with the class name.

Thread.Sleep(0) , **Thread.Sleep(Timeout.Infinite)** :- putting the current thread to sleep infinitely.

Process based multitasking :- Windows OS (System level)

Threaded based multitasking :- MSWord (document 🡪 spelling and grammar

Browser :- Loading of Images and Scrolling of page

Application level

Running / Resume

Non Runnable mode - Sleep()

Interrupt() :- state-> runnable mode

Paused

The sleeping thread can be resumed by calling **Interrupt()** of the thread class.

Sleeping – suspended --- suspended --- Resume() --- sleeping() -🡪 Runnable🡪 Dead state (abort)

class Program

{

public static void FunctionCall1()

{

try

{

Thread.Sleep(Timeout.Infinite);

}

catch (ThreadInterruptedException ex)

{

Console.WriteLine("cannot sleep, interrupted by main()");

}

finally

{

for (int i = 1; i <= 10; i++)

{

Console.WriteLine(i);

}

}

}

public static void FunctionCall2()

{

Console.WriteLine("=========Child begins here======");

for (int i = 11; i <= 20; i++)

{

Console.WriteLine(i);

}

Console.WriteLine("============ Child Ends here=======");

}

static void Main(string[] args)

{

Console.WriteLine("======== Main begins here========");

Thread th = Thread.CurrentThread;

th.Name = "Main Thread";

Console.WriteLine("Current Thread {0}", th.Name);

// object of thread

Console.WriteLine("====Main calling child thread=====");

ThreadStart ts1 = new ThreadStart(FunctionCall1);

Thread t1 = new Thread(ts1);

t1.Start(); // runnable mode

t1.Interrupt(); //main() is interrupting the thread which is sleeping infinitely

ThreadStart ts2 = new ThreadStart(FunctionCall2);

Thread t2 = new Thread(ts2);

t2.Start();

Console.WriteLine("======== Main Loaded completely====");

}

}

}

There is another method, **Thread.Suspend()** , which is used to suspend the execution of the thread.

**Thread.Resume()** :- this will cause the thread to resume its execution.

public class TestingThread

{

public void functioncall()

{

for(int x = 0; x <= 5; x++)

{

Console.WriteLine(x);

}

}

}

class Program

{

static void Main(string[] args)

{

Console.WriteLine("=== main calling child thread===");

TestingThread tt = new TestingThread();

ThreadStart ts = new ThreadStart(tt.functioncall);

Thread t1 = new Thread(ts);

Console.WriteLine("Thread State {0}", t1.ThreadState);

// runnable mode

t1.Start();

Console.WriteLine("Thread State {0}", t1.ThreadState);

try

{

t1.Suspend();

}

catch (PlatformNotSupportedException ex)

{

Console.WriteLine("Not supported");

}

Console.WriteLine("Thread State {0}", t1.ThreadState);

try

{

t1.Resume();

}

catch (PlatformNotSupportedException ex1)

{

Console.WriteLine(ex1.Message);

}

finally

{

Console.WriteLine("Thread State {0}", t1.ThreadState);

}

}

}

Destroying Thread :- The thread can be destroyed by calling Thread.Abort(). When the thread is destroyed it throws ThreadAbortException(). This exception cannot be caught.

Joining Threads :- **Join()** allows one thread to wait until another thread completes its execution.

class Program

{

public static void childthreadcall1()

{

try

{

Console.WriteLine("Child thread called");

Thread.Sleep(5000);

for (int i = 1; i <= 5; i++)

{

Console.WriteLine(i);

}

Console.WriteLine("Waking up");

}

catch(ThreadAbortException ex)

{

Console.WriteLine(ex.Message);

}

finally

{

Console.WriteLine("Unable to handle");

}

}

public static void childthreadcall2()

{

try

{

Console.WriteLine("Child thread called");

Thread.Sleep(5000);

for (int i = 11; i <= 15; i++)

{

Console.WriteLine(i);

}

Console.WriteLine("Waking up");

}

catch (ThreadAbortException ex)

{

Console.WriteLine(ex.Message);

}

finally

{

Console.WriteLine("Unable to handle");

}

}

static void Main(string[] args)

{

Console.WriteLine("=======Creation of Our Main Thread=========");

Console.WriteLine("Calling child thread");

//it is a reference to the method

ThreadStart ts1 = new ThreadStart(childthreadcall1);

Thread th1 = new Thread(ts1); // it is unstarted / ready mode

th1.Start(); // runnable mode

th1.Join();

ThreadStart ts2 = new ThreadStart(childthreadcall2);

Thread th2 = new Thread(ts2);

th2.Start();

th2.Join();

Console.WriteLine("===========Main Ends here======");

}

}

}

States :- In life cycle of a Thread

Unstarted state or Ready mode :- when the thread object is created

Runnable :- by calling start()

Non-Runnable :- Sleeping , Waiting, Blocked (Sleep(), Suspend(), Join())

dead :- when application terminates 0r Abort()

**Implementing Multithreading**

Helps us to perform various operations at same time.

Process based | Threaded based :- processor load is reduced, because of switching between multiple threads.

**Advantages of Multithreading**

Increase performance

System resource usage is minimized.

Access multiple applications at same time

Simplified program structure

**Limitations**

Race Condition :- Lack of synchronization .

Deadlock condition :- where two threads are watiing for each other to complete their execution.

Lock Starvation :- Execution of a thread is postponed because of its low priority.

Setting Priorities

class Program

{

public static void childthread1()

{

Console.WriteLine("Child thread1 started");

for (int i = 1; i <= 5; i++)

{

Console.WriteLine(i);

}

Console.WriteLine("Child thread1 finished");

}

public static void childthread2()

{

Console.WriteLine("Child thread2 started");

for (int i = 11; i <= 15; i++)

{

Console.WriteLine(i);

}

Console.WriteLine("Child thread2 finished");

}

static void Main(string[] args)

{

ThreadStart Child1 = new ThreadStart(childthread1);

ThreadStart Child2 = new ThreadStart(childthread2);

Console.WriteLine("Main Begins here");

Thread th1 = new Thread(Child1);

Thread th2 = new Thread(Child2);

th1.Priority = ThreadPriority.Highest;

th2.Priority = ThreadPriority.Lowest;

th1.Start();

th2.Start();

}

}

}

Synchronization of threads:- ensures that when two or more threads need to access the shared resource , then that resource is used by only one thread at a time.

Monitor class :- it ensures that only one thread has access to that resource at given time.

Enter() :- it grants the lock on that resource

Exit() :- it releases the lock from that resource

public void Updatesalary(){

lock(this); | Monitor.Enter(this); | Monitor.Exit(this);

}

ReadSalary()

Task

Write a program using Monitor.Enter(this) and lock(this)

class FileAccess

{

public void WriteData(string s)

{

//Monitor.Enter(this);

lock (this)

{

Console.Write(s);

Console.WriteLine("Writing data completed");

//Monitor.Exit(this);

}

}

}

class Program

{

static FileAccess obj = new FileAccess();

public static void ChildThread1()

{

Console.WriteLine("Child1 started writing data");

obj.WriteData("child 1 -- my new data");

}

public static void ChildThread2()

{

Console.WriteLine("Child2 started writing data");

obj.WriteData("child2 -- my new data");

}

static void Main(string[] args)

{

ThreadStart ts1 = new ThreadStart(ChildThread1);

ThreadStart ts2 = new ThreadStart(ChildThread2);

Thread t1 = new Thread(ts1);

Thread t2 = new Thread(ts2);

t1.Start();

t2.Start();

}

}

}

**Async and Await**

Asynchronous programming is very popular with the help of the async and await keywords in C#. When we are dealing with UI, and on button click, we use a long-running method like reading a large file or something else which will take a long time, in that case, the entire application must wait to complete the whole task. In other words, if any process is blocked in a synchronous application, the whole application gets blocked, and our application stops responding until the whole task completes.

Asynchronous programming is very helpful in this condition. By using Asynchronous programming, the Application can continue with the other work that does not depend on the completion of the entire task.

Suppose we are using two methods as Method1 and Method2 respectively, and both the methods are not dependent on each other, and Method1 takes a long time to complete its task. In Synchronous programming, it will execute the first Method1 and it will wait for the completion of this method, and then it will execute Method2. Thus, it will be a time-intensive process even though both methods are not depending on each other.

We can run all the methods parallelly by using simple thread programming, but it will block UI and wait to complete all the tasks. To come out of this problem, we have to write too many codes in traditional programming, but if we use the async and await keywords, we will get the solutions in much less code.

Also, we are going to see more examples, and if any third Method, as Method3 has a dependency of method1, then it will wait for the completion of Method1 with the help of await keyword.

Async and await in C# are the code markers, which marks code positions from where the control should resume after a task completes.

**Async** and **Await** keywords were introduced in C# 5.0 by Microsoft. When you use the “**Async**” keyword, you can write code the same way you wrote synchronous code. The compiler takes care of all the complexity and frees you to do what you do best: writing the logic. There are some rules for writing the Async method:

* The method signature must have the async keyword.
* The method name should end with Async (this is not enforced, but it is a best practice).
* The method should return Task, Task<T>, or void.

**Async and Await Demo**

**C# simple synchronous example**

In the next example, we execute three methods synchronously.

**Program.cs**

using System;

using System.Diagnostics;

using System.Threading;

var sw = new Stopwatch();

sw.Start();

f1();

f2();

f3();

sw.Stop();

var elapsed = sw.ElapsedMilliseconds;

Console.WriteLine($"elapsed: {elapsed} ms");

void f1()

{

Console.WriteLine("f1 called");

Thread.Sleep(4000);

}

void f2()

{

Console.WriteLine("f2 called");

Thread.Sleep(7000);

}

void f3()

{

Console.WriteLine("f3 called");

Thread.Sleep(2000);

}

With Thread.Sleep, we emulate some longer computations.

var sw = new Stopwatch();

sw.Start();

We measure the execution time of the methods with Stopwatch.

f1();

f2();

f3();

The methods are called consecutively.

$ dotnet run

f1 called

f2 called

f3 called

elapsed: 13034 ms

**C# simple asynchronous example**

Now, the example is rewritten using async/await keywords.

**Program.cs**

using System;

using System.Diagnostics;

using System.Threading;

using System.Threading.Tasks;

var sw = new Stopwatch();

sw.Start();

Task.WaitAll(f1(), f2(), f3());

sw.Stop();

var elapsed = sw.ElapsedMilliseconds;

Console.WriteLine($"elapsed: {elapsed} ms");

async Task f1()

{

await Task.Delay(4000);

Console.WriteLine("f1 finished");

}

async Task f2()

{

await Task.Delay(7000);

Console.WriteLine("f2 finished");

}

async Task f3()

{

await Task.Delay(2000);

Console.WriteLine("f3 finished");

}

We measure the execution time of three asynchronous methods.

Task.WaitAll(f1(), f2(), f3());

The Task.WaitAll waits for all of the provided tasks to complete execution.

async Task f1()

{

await Task.Delay(4000);

Console.WriteLine("f1 finished");

}

The f1 method uses the async modifier and returns a Task. Inside the body of the method, we use the await operator on the Task.Delay.

$ dotnet run

f3 finished

f1 finished

f2 finished

elapsed: 7006 ms

Now the execution took 7 s. Also note that the order in which the tasks finished is different.

**C# async Main method**

When we are using the await operator inside the Main method, we have to mark it with the async modifier.

**Program.cs**

using System;

using System.Diagnostics;

using System.Threading.Tasks;

namespace AsyncMain

{

class Program

{

static async Task Main(string[] args)

{

var sw = new Stopwatch();

sw.Start();

Console.WriteLine("task 1");

Task task1 = doWork();

Console.WriteLine("task 2");

Task task2 = doWork();

Console.WriteLine("task 3");

Task task3 = doWork();

await Task.WhenAll(task1, task2, task3);

Console.WriteLine("Tasks finished");

sw.Stop();

var elapsed = sw.ElapsedMilliseconds;

Console.WriteLine($"elapsed: {elapsed} ms");

}

static async Task doWork()

{

await Task.Delay(1500);

}

}

}

Inside the Main method, we call the doWork three times.

$ dotnet run

task 1

task 2

task 3

Tasks finished

elapsed: 1550 ms

### **Synchronous execution: doing things one after the other**

The user clicks a button and waits for the application to finish handling the click event. Since only one thing can happen at a time, the UI stops responding until the event has been completely handled. In the same way, the application can’t do anything in the background while UI is available for user input.

### **Concurrent: doing multiple things at the same time**

The user clicks a button, and the application triggers a separate thread in the background to execute the task needed to satisfy user’s request concurrently. The thread responsible for handling UI events becomes available again immediately after starting the background thread, keeping the UI responsive.

### **Parallel: doing multiple copies of something at the same time**

The user instructs the application to process all the files in a folder. The application triggers a number of threads with the processing logic and distributes the files among these threads.

### **Asynchronous: not having to wait for one task to finish before starting another**

The application starts a database query asynchronously. While the query is in progress, it also starts reading a file asynchronously. While both tasks are in progress, it does some calculation.  
When all these tasks are finished, it uses the results of all these three operations to update the UI.

The [Task asynchronous programming model (TAP)](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/task-asynchronous-programming-model) provides an abstraction over asynchronous code. You write code as a sequence of statements, just like always. You can read that code as though each statement completes before the next begins. The compiler performs many transformations because some of those statements may start work and return a [Task](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task) that represents the ongoing work.

That's the goal of this syntax: enable code that reads like a sequence of statements, but executes in a much more complicated order based on external resource allocation and when tasks complete. It's analogous to how people give instructions for processes that include asynchronous tasks. Throughout this article, you'll use an example of instructions for making a breakfast to see how the async and await keywords make it easier to reason about code, that includes a series of asynchronous instructions. You'd write the instructions something like the following list to explain how to make a breakfast:

1. Pour a cup of coffee.
2. Heat up a pan, then fry two eggs.
3. Fry three slices of bacon.
4. Toast two pieces of bread.
5. Add butter and jam to the toast.
6. Pour a glass of orange juice.

If you have experience with cooking, you'd execute those instructions **asynchronously**. You'd start warming the pan for eggs, then start the bacon. You'd put the bread in the toaster, then start the eggs. At each step of the process, you'd start a task, then turn your attention to tasks that are ready for your attention.

Cooking breakfast is a good example of asynchronous work that isn't parallel. One person (or thread) can handle all these tasks. Continuing the breakfast analogy, one person can make breakfast asynchronously by starting the next task before the first completes. The cooking progresses whether or not someone is watching it. As soon as you start warming the pan for the eggs, you can begin frying the bacon. Once the bacon starts, you can put the bread into the toaster.

For a parallel algorithm, you'd need multiple cooks (or threads). One would make the eggs, one the bacon, and so on. Each one would be focused on just that one task. Each cook (or thread) would be blocked synchronously waiting for bacon to be ready to flip, or the toast to pop.

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/