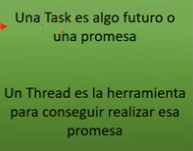
Task

Dice el muchacho:

Supongase que se quieren ejecutar 5 tareas. Si el procesador tiene 9 núcleos se está infrautilizando. Ahora, si tiene 4 nucleos no. Se lo está sobrecargando. Habrá un thread a la espera.

Para solventar esto surgen las task. Gestionan los pools de threads para optimizar el uso del procesador.

Están en un nivel de abstracción superior que los threads.



Se suelen usar tasks mas que threads.

Concurrencia es sinónimo de simultáneo.

class Program

{

static void Main(string[] args)

{

Task task = new Task(EjecutarTarea);

task.Start();

Task task2 = new Task(EjecutarTarea);

task2.Start();

Console.ReadKey();

}

static void EjecutarTarea()

{

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

{

var miThread = Thread.CurrentThread.ManagedThreadId;

Thread.Sleep(1);

Console.WriteLine("Esta vuelta de bucle corresponde al thread:"+ miThread);

}

}

}

Método run. Simplifica el código.

Task task = new Task(EjecutarTarea);

task.Start();

Task task2 = Task.Run(() => EjecutarTarea());

Es lo mismo que lo anterior, pero mas simple.

La primer tarea debe informar a la segunda que ha terminado, para que esta pueda comenzar.

La segunda tarea ya no usará Run, sino continueWith.

class Program

{

static void Main(string[] args)

{

Task task = Task.Run(() => EjecutarTarea());

Task task2 = task.ContinueWith(EjecutarOtraTarea);

Console.ReadKey();

}

static void EjecutarTarea()

{

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

{

var miThread = Thread.CurrentThread.ManagedThreadId;

Thread.Sleep(500);

Console.WriteLine("Esta vuelta de bucle corresponde al thread:" + miThread);

}

}

static void EjecutarOtraTarea(Task obj)

{

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

{

var miThread = Thread.CurrentThread.ManagedThreadId;

Thread.Sleep(500);

Console.WriteLine("Esto es otra tarea y esta vuelta de bucle corresponde al thread:" + miThread);

}

}

}

WaitAll WaitAny Wait

Se usa para dar prioridad a unas tareas por sobre otras.

Puede pasar que una tarea requiera los datos de otras 2 tareas. Por eso esas 2 tareas deben ejecutarse previamente. Ahí tiene sentido usar el wait.

Hay que guardar esas tareas previas en variables.

Con WaitAny, la tarea 3 empezará si la tarea 1 o la 2 han terminado.

Con Wait tiene que estar la tarea 1 completada para pasar a la 2.

class Program

{

static void Main(string[] args)

{

RealizarTodasTareas();

Console.ReadKey();

}

static void RealizarTodasTareas()

{

var tarea1=Task.Run(() =>

{

EjecutarTarea();

});

tarea1.Wait();

Task tarea2= Task.Run(() =>

{

EjecutarTarea2();

});

tarea2.Wait();

// Task.WaitAll(tarea1, tarea2);

// Task.WaitAny(tarea1, tarea2);

Task tarea3 =Task.Run(() =>

{

EjecutarTarea3();

});

}

static void EjecutarTarea()

{

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

{

var miThread = Thread.CurrentThread.ManagedThreadId;

Thread.Sleep(1000);

Console.WriteLine("Esta vuelta de bucle corresponde a la tarea 1");

}

}

static void EjecutarTarea2()

{

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

{

var miThread = Thread.CurrentThread.ManagedThreadId;

Thread.Sleep(500);

Console.WriteLine("Esta vuelta de bucle corresponde a la tarea 2");

}

}

static void EjecutarTarea3()

{

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

{

var miThread = Thread.CurrentThread.ManagedThreadId;

Thread.Sleep(300);

Console.WriteLine("Esta vuelta de bucle corresponde a la tarea 3");

}

}

}

Clase Parallel

La clase parallel tiene métodos for y foreach.

Hace que una tarea se ejecute concurrentemente (en paralelo) por varios threads.

class Program

{

private static int acumulador = 0;

static void Main(string[] args)

{

/\* for (int i = 0; i < 100; i++)

{

RealizarTareas(i);

Console.WriteLine($"Acumulador vale {acumulador}. Tarea realizada por el hilo {Thread.CurrentThread.ManagedThreadId}");

}

\*/

Parallel.For(0, 100, RealizarTareas);

//esto permite no utilizar varios task y sus run.

Console.ReadKey();

}

static void RealizarTareas(int i)

{

Console.WriteLine($"Acumulador vale {acumulador}. Tarea realizada por el hilo {Thread.CurrentThread.ManagedThreadId}");

if (acumulador % 2 == 0)

{

acumulador += i;

Thread.Sleep(300);

}

else

{

acumulador -= i;

Thread.Sleep(100);

}

}

}

Cancelación de tarea

2 clases: cancellationTokenSource y cancellationToken.

A veces se puede cancelar una tarea. Si eso sucede, hay que revertir y dejar todo como estaba.

class Program

{

static private int acumulador = 0;

static void Main(string[] args)

{

CancellationTokenSource MiToken = new CancellationTokenSource();

//este señala

CancellationToken cancelaToken = MiToken.Token;

//este cancela

Task tarea = Task.Run(()=> RealizarTarea(cancelaToken));

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

{

acumulador += 4;

Thread.Sleep(700);

if (acumulador >= 100)

{

MiToken.Cancel();

break;

}

}

Thread.Sleep(1000);

Console.WriteLine("Valor de acumulador:"+acumulador);

Console.ReadKey();

}

static void RealizarTarea(CancellationToken tokenDeAca)

{

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

{

acumulador++;

var MiThread = Thread.CurrentThread.ManagedThreadId;

Thread.Sleep(700);

Console.WriteLine("Ejecuta tarea el thread "+ MiThread);

Console.WriteLine(acumulador);

if (tokenDeAca.IsCancellationRequested)

{

acumulador = 0;

//Con esa linea se dejan las cosas como estaban inicialmente

return;

//Esto hace que salga del for y vuelva.

}

}

}

}