Threads

App monotarea. Solo hace una tarea a la vez.

App multitarea. Realiza varias tareas de forma simultánea.

Método sleep admite parámetro que son milisegundos.

Para poder usar la clase threads hay que importar system.threading

Console.WriteLine("Hola bacterias desde thread 1");

Thread.Sleep(500);

Console.ReadKey();

class Program

{

static void Main(string[] args)

{

Thread t = new Thread(MetodoSaludo);

t.Start();

Console.WriteLine("Hola bacterias desde thread 1");

Thread.Sleep(500);

Console.WriteLine("Hola bacterias desde thread 1");

Thread.Sleep(500);

Console.ReadKey();

}

static void MetodoSaludo()

{

Thread.Sleep(150);

Console.WriteLine("Hola bacterias desde thread 2");

Thread.Sleep(500);

Console.WriteLine("Hola bacterias desde thread 2");

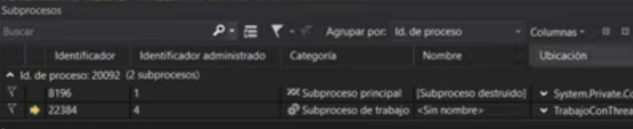
Thread.Sleep(500);

}

}

Thread t=new thread (función que se le pasa como segundo hilo).

T.start();



Sincronización y bloqueos de threads

Se sincroniza con: nombremétodo.join();

Thread t = new Thread(MetodoSaludo);

t.Start();

t.Join();

Eso hace que se ejecute esa parte del código y una vez terminado se pase a la siguiente parte del código.

Bloquear es que mientras un thread se ejecute no se pueda ejecutar otro.

Array de threads:

Thread[] hilosPersonas = new Thread[15];

Un mismo código:

class Program

{

static void Main(string[] args)

{

CuentaBancaria CuentaFamilia = new CuentaBancaria(2000);

Thread[] hilosPersonas = new Thread[15];

for (int i=0; i<15; i++)

{

Thread t = new Thread(CuentaFamilia.VamosRetirarEfectivo);

hilosPersonas[i] = t;

}

for (int i = 0; i < 15; i++)

{

hilosPersonas[i].Start();

}

Console.ReadKey();

}

}

class CuentaBancaria

{

double Saldo { get; set; }

public CuentaBancaria(double Saldo)

{

this.Saldo = Saldo;

}

public double RetirarEfectivo(double cantidad)

{

if (Saldo - cantidad < 0)

{

Console.WriteLine($"No dispone de tanto saldo, queda ${Saldo} en la cuenta");

return Saldo;

}

else

{

Console.WriteLine("Retirado {0} y queda {1} en la cuenta",cantidad, (Saldo-cantidad));

return Saldo -= cantidad;

}

}

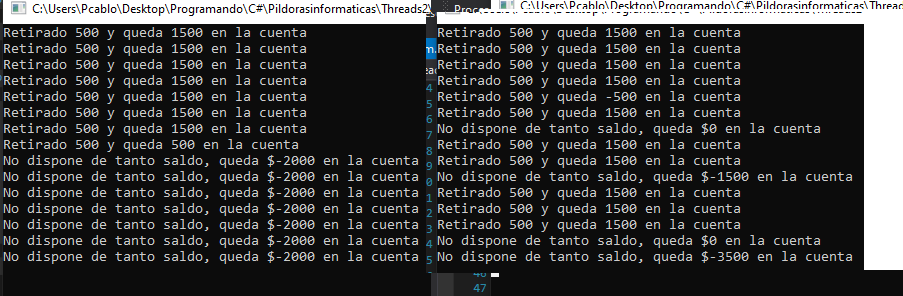
public void VamosRetirarEfectivo()

{

RetirarEfectivo(500);

}

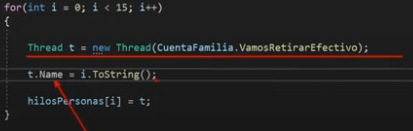
}



Puede ofrecer distintos resultados. Porque todos los threads pueden llegar a ejecutarse a la vez y no haber operaciones pasadas. Cuando cambia el resultado es porque si se ejecutó después.

Esto sucede porque no hay bloqueo de threads.

Dar nombre a Thread creado en un for:



CurrentThread devuelve el hilo actual.

public double RetirarEfectivo(double cantidad)

{

if (Saldo - cantidad < 0)

{

Console.WriteLine($"No dispone de tanto saldo, queda ${Saldo} en la cuenta. Hilo: {Thread.CurrentThread.Name}");

return Saldo;

}

else

{

Console.WriteLine("Retirado {0} y queda {1} en la cuenta. Hilo: {2}",cantidad, Saldo-cantidad, Thread.CurrentThread.Name);

return Saldo -= cantidad;

}

}

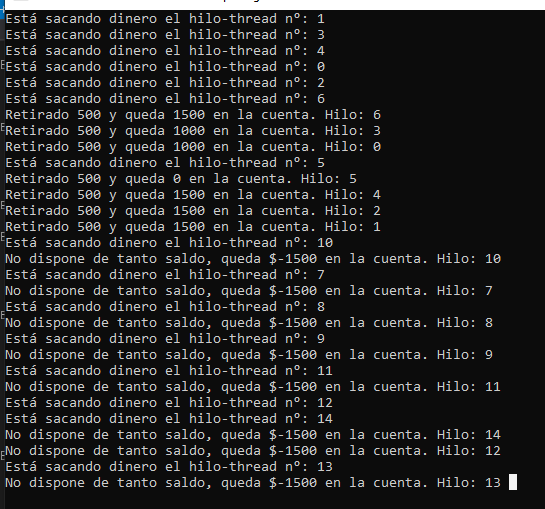
public void VamosRetirarEfectivo()

{

Console.WriteLine("Está sacando dinero el hilo-thread n°: {0}", Thread.CurrentThread.Name);

RetirarEfectivo(500);

}



Habla de un trozo de código clave que debiera ser bloqueado. En este caso, el método RetirarEfectivo. Mientras un thread realiza sus operaciones no debiera haber otro que pueda acceder ahí. Esto no es sinónimo de sincronizar, es indistinto el orden en que se ejecutan.

Se ejecutan con el método lock. Pide un parámetro de tipo object, que debe crearse.

class CuentaBancaria

{

private Object BloqueaSaldo = new Object();

double Saldo { get; set; }

public CuentaBancaria(double Saldo)

{

this.Saldo = Saldo;

}

public double RetirarEfectivo(double cantidad)

{

if (Saldo - cantidad < 0)

{

Console.WriteLine($"No dispone de tanto saldo, queda ${Saldo} en la cuenta. Hilo: {Thread.CurrentThread.Name}");

return Saldo;

}

lock (BloqueaSaldo) {

if (Saldo>=cantidad)

{

Console.WriteLine("Retirado {0} y queda {1} en la cuenta. Hilo: {2}", cantidad, Saldo - cantidad, Thread.CurrentThread.Name);

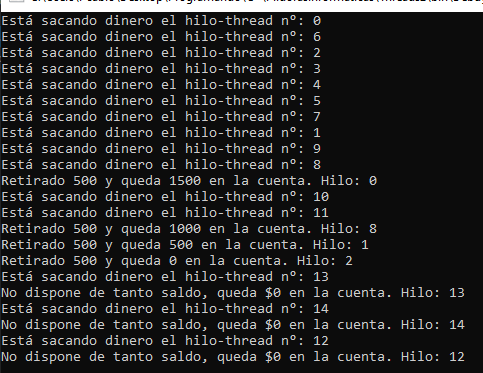
Saldo -= cantidad;

}

return Saldo;

}

}



Sigue siendo caótico y no hay sincronización. Pero llegado esa parte de código se ejecuta de uno a la vez. Por eso, tras la cuarta ejecución, no saca mas dinero. No siempre tira msj de que no dispone de tanto saldo. Se ve que entraron al método al mismo tiempo que los que si retiraron dinero y “esquivaron el if con el que informaría eso”.

Tareas completadas

Que una tarea comience su trabajo si otra tarea concluyó el suyo. No es una sincronización.

Usa la clase taskCompletionSource <genérico>.

trySetResult() marca cuando la tarea se ha terminado. Por eso se pone cuando un hilo termina su tarea.

Propiedad task (tarea) obtiene si se terminó o no.

class Program

{

static void Main(string[] args)

{

var tareaTerminada = new TaskCompletionSource<bool>();

//Sería lo mismo que:

// TaskCompletionSource<bool> tareaTerminada = new TaskCompletionSource<bool>();

var hilo1 = new Thread(() =>

{

//Aclara que lo de arriba es una expresión lambda

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

{

Console.WriteLine("Hilo 1");

Thread.Sleep(1000);

}

tareaTerminada.TrySetResult(true);

});

var hilo2 = new Thread(() =>

{

//Aclara que lo de arriba es una expresión lambda

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

{

Console.WriteLine("Hilo 2");

Thread.Sleep(1000);

}

});

hilo1.Start();

var resultado = tareaTerminada.Task.Result;

//Dime el resultado de la tarea que se llama tareaTerminada

hilo2.Start();

}

}

ThreadPool

Se usa para crear muchos hilos a ejecutar de manera concurrente.

En BBDD se usa mucho el pool de conexiones.

Pool de threads. Se reutilizan los threads de un pool cuando concluyen su primera tarea. Se reutilizan. Optimiza recursos que si se dedicara un thread a cada tarea. Aunque dice que es mas lento.



Sin establecerles un name, cada thread tiene un Id.

ThreadPool.QueueUserWorkItem

Con eso se crea el pool, recibe por parámetro un objeto

class Program

{

static void Main(string[] args)

{

for (int i = 0; i < 500; i++)

{

Thread t = new Thread(EjecutarTarea);

t.Start();

Console.ReadKey();

}

static void EjecutarTarea()

{

Console.WriteLine($"Thread n°:{ Thread.CurrentThread.ManagedThreadId} ha comenzado su tarea");

Thread.Sleep(1000);

Console.WriteLine($"Thread n°:{ Thread.CurrentThread.ManagedThreadId} ha terminado su tarea");

}

}

Esa es la forma normalita. Va a requerir que se creen 500 threads (un poco mas en realidad, no se por qué).

class Program

{

static void Main(string[] args)

{

for (int i = 0; i < 500; i++)

{

//Thread t = new Thread(EjecutarTarea);

// t.Start();

ThreadPool.QueueUserWorkItem(EjecutarTarea, i);

}

Console.ReadKey();

}

static void EjecutarTarea(Object o)

{

int tarea = (int)0;

Console.WriteLine($"Thread n°:{ Thread.CurrentThread.ManagedThreadId} ha comenzado la tarea n° {tarea}");

Thread.Sleep(1000);

Console.WriteLine($"Thread n°:{ Thread.CurrentThread.ManagedThreadId} ha terminado la tarea n° {tarea}");

}

}

De esta segunda forma con el threadPool se crearon 27 threads nada mas. Es bastante mas lento. Hay mucha reutilización de threads.

Se pasa como parámetro (EjecutarTarea, i). No tengo ni idea cómo es eso. Porque no se lo recibe explícitamente como argumento, pero aún así función.

Aclara que se puede determinar el numero máximo de threads en un pool. Si no, lo hace automáticamente.

Casteo o Casting P ej int tarea= (int)0;