Tasks, Threads, Threading

<https://www.youtube.com/watch?v=gfkuD_eWM5Y>

**volatile**

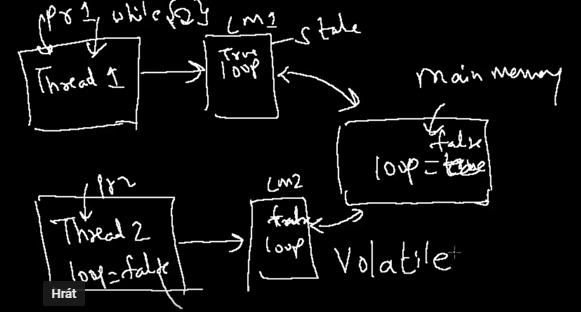
klíčové slovo označuje, že pole může být upraveno více podprocesy, které se provádějí současně.

Kompilátor, systém běhu a dokonce i hardware mohou uspořádat čtení a zápisy do paměti z důvodů výkonu.Pole, které jsou prohlášeny za volatile, nepodléhají těmto optimalizacím.

Přidáním volatilního modifikátoru se zajistí, že všechny vlákna budou sledovat volatilní zápisy provedené jiným vláknem v pořadí, v jakém byly provedeny. Neexistuje žádná záruka jediného úplného uspořádání volatilních zápisů, jak je vidět ze všech podprocesů.

Jinymi slovy: pokud je promenna volatile, vlakno, které ji chce cist ji nejdrive synchronizuje . (Každý procesor resp thread ma svoji local memory, kde se stav promenne muze lisit.)

<https://www.youtube.com/watch?v=DZUXDSEuqek&list=PLZ_kIs-AQuVa82FWRL43F56CMoVhY4TH6&index=6>



# **Tasks**

# Vytvoreni :

trida Task ma několik konstruktoru, jeden z parametru je Action delegat, ktereho muzeme napsat pomoci lambdy

Task t = new Task(()=> {

ConsoleWriteLine(“Task starts”);

Thread.Sleep(2000);

ConsoleWriteLine(“Task ends”);

});

**t.Start();** spusti task

**t.Stop()** zastavi task

**Task.Wait(t)** pocka v tomto miste nez Task t dokonci svoji praci

**Vytvoreni se spustenim**

Tast t = Task.**Run**(() =>{

ConsoleWriteLine(“Task starts”);

Thread.Sleep(2000);

ConsoleWriteLine(“Task ends”);

});

# Task.Factory

Pokud nechci vytvaret vice Tasku rucne (t1 = … , t1.Start() t2 = … t2.Start() atd), muzu pouzit Task.Factory.

Task.Factory.StartNew – vytvori a spusti task (stejne jako Task.Run())

Task.Factory.StartNew(() => DoSomething())

# ContinueWith

Pokud chceme, aby se po skonceni tasku pokracovalo jinym delegatem pridame na konec volani .ContinueWith(delegat) . Jde proste o zretezeni volani metod. Po skonceni provadeni prvni metody se zacne vykonavat dalsi.

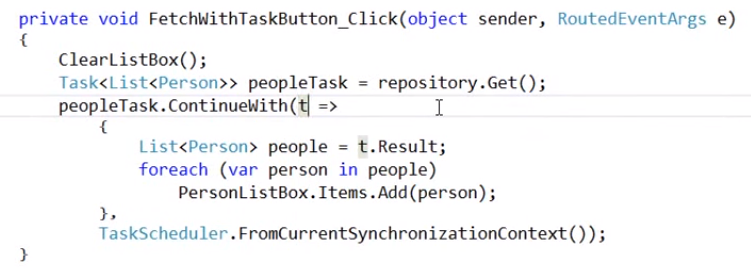
var t1 = Task.Factory.StartNew(() => DoSomething()).ContinueWith((prevTask) => DoSomethingElse());

# TaskScheduler.FromCurrentSynchronisationContext

<https://www.youtube.com/watch?v=0qiB3oW_nd8&list=PLdbkZkVDyKZWWjJ5yUdd_ooLORZgOSHSP.>

Mam Task po jehoz zkonceni chci zavolat novy task. V tomto nasl. Tasku chci predat UIThreadu vysledek ulohy. Dostanu vyjimku, ze nejsem na UIThreadu. K predani slouzi planovac uloh (TaskSheduler)

Parametrem FromCurrentSynchronisationContext rikam, ze mam pouzi stejny planovac uloh, jako je ten ze ktereho jsem vysel. V tomto prikladu je to UIThread protoze se zde reaguje na stlaceni tlacitka.



# Continuation option

Pouzitim tasku mame plnou kontrolu nad bezici operaci. V pripade, ze v bezicim tasku vznikne vyjimka a pouzili jsme ContinueWith, muzeme si pomoci continuation option zvolit za jakych okolnosti bude nasledujici task pokracovat.

Continuation option je enum ve kterem jsou moznosti jako NotOnCanceled, NotOnFaulted, OnlyOnCancelled atd..

Prvnimu tasku muzeme priradit hned nekolik .continueWith casti ve kterych definujeme co se ma stat pokud task projde, spadne, nebo proste co se ma stat vzdy za jakychkoli okolnosti. Kdyz chci tuto funkcionalitu pouzit v await-async muzu to udelat jedine pomoci bloku try-catch-finally

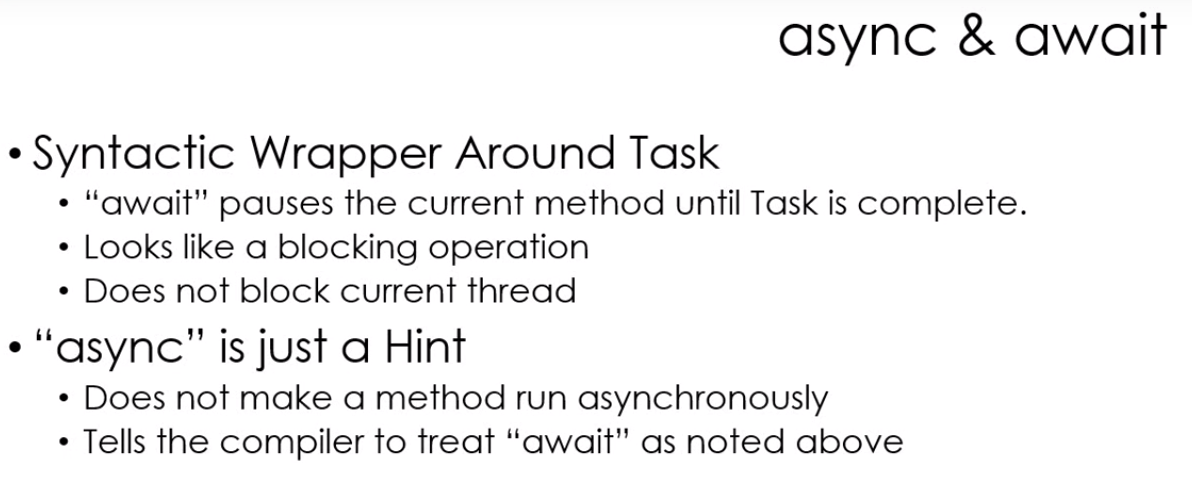
# Async await

Viz take: [Await Async\Await Async.docx](Await%20Async/Await%20Async.docx)

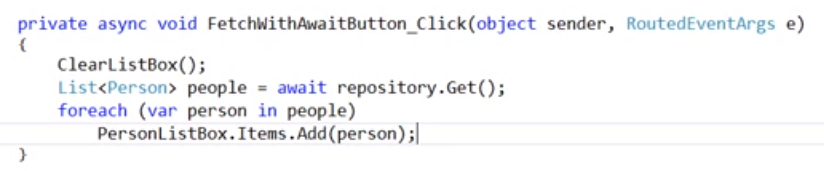
Je jakysi wrapper okolo tasku. “await” jen **zapauzuje** async metodu, dokud v ni bezici task nedobehne. Nezablokuje UI ani current threa. Nespousti metodu asynchronne. To dela az Task.Run(action)

“await” = pockej na Task

Je to nejjednodussi pouziti tasku, ale zaroven neposkytuje zadne moznosti kontroly tasku.



Nasledujici kod provadi to co bylo popsano v TaskScheduler.FromCurrentSynchronisationContext, pomoci async-await.



Pokud chci napodobit ovladani jako pri pouziti continuation option u tasku, musim pouzit try catch finally

# Parallel

Vytvori nekolik paralelnich vlaken naraz. Hlavni duvod pouziti tridy Parallel je, ze praci rozdeli rovnomerne na vsechny jadra procesoru.

var intList = new List<int> {1,2,3,4,5,6,7}

Parallel.ForEach(intLIst, (i)=> Console.WriteLine())

Vystup bude sprehazeny.

Parallel.For(0, 100 (i)=> Console.WriteLine(i))

Vypise cisla od nula do sto pomoci vice vlaken. Vsechny operace za tim, cekaji na dokonceni For loopu

# CancellationTokenSource

Pomuze zastavit pribihajici operaci na tasku. Tasku predam token na kterem muzu zavolat metodu Cancel . Pomoci Task.Factory spustim task v jinem threadu. V nasledujicim kodu muzu zavolat Cancel a probihajici task zastavit.

var source = new **CancellationTokenSource**();

try

{

var t1 = Task.Factory.StartNew(() => DoSomething(source.Token)).ContinueWith((prevTask) => DoSomethingElse(source.Token))

**source.Cancel(); // toto zastavi probihajici Task**

}

catch (Exception e){ cw ex.GetType()}

static void DoSomething(**CancelationToken token**)

{

If(token.IsCancellationRequested)

{

Console.WriteLine(“CancellationRequested ”);

token.throwIfCancellationRequested

}

// zbytek metody

}

# **Threads**

# Vytvoreni

Thread t = new Thread (threadStart);

Thread t = new Thread (parametrizedThreadStart);

# Spusteni

Thread.Start()

TaskFactory (vytvori a spusti vlakna)

# Lock

Pomoci Lockovani muzeme vlakna udelat threadsafe.

Lock neboli zamek, bere jako argument jakykoli objekt.

# AutoResetEvent a ManualResetEvent

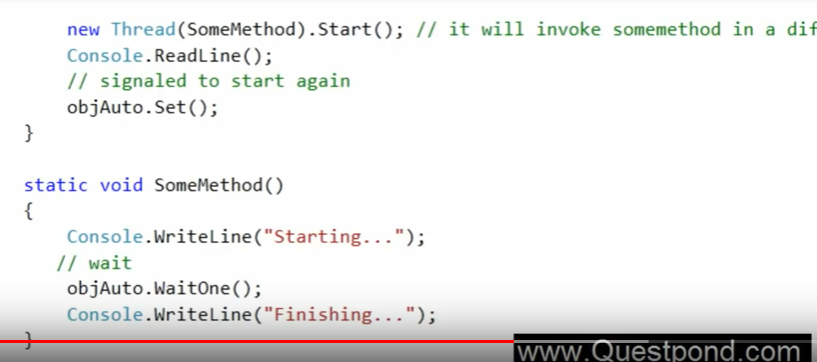
Pokud se vlakno dostane k mistu kde se vola na objektu AutoResetEvent metoda WaitOne(), prepne se do tzv. wait mode a ceka, az jine (typicky hlavni) vlakno na tomto objektu zavola Set();

<https://www.youtube.com/watch?v=xaaRBh07N34&t=302s>

AutoResetEvent objAuto = new AutoResetEvent (false);

objAuto.WaitOne() – vlakno dostane signal, ze si ma dat pauzu.

objAuto.Set(); - pauznute vlakno dostane signal, ze muze pokracovat.



Main thread vytvori vlakno. Vytvorene vlakno vleze do metody SomeMethod, tam se stopne. Main thread mezitim pokracuje a ceka na stisknuti klavesy na konzoli (Console.ReadLine) . Jakmile se klavesa stiskne, main thread zavola na objektu typu AutoResetEvent (na tom ktery stopnul vlakno), metodu Set(), cimz worker thread odblokuje a to muze pokracovat.

**Rozdil mezi AutoResetEvent a ManualResetEvent**

**AutoResetEvent** – (volame metody WaitOne a Set na objektu typu AutoResetEvent) Na vlakne muzeme volat WaitOne kolikrat chceme. Pokazde musime ale vlakno odblokovat pomoci metody Set()

**ManualResetEvent** – (volame metody WaitOne a Set na objektu typu ManualResetEvent) jakmile zavolame metodu Set() jednou, uz se vlakno nezastavi na dalsim volani WaitOne na vlakne.

# Stavy podprocesu

Stavy jsou vyctovym typem ThreadState. Kazdy rizeny podproces zacina ve stavu Unstarted. Jakmile zavolame metodu Start(), vstoupi do stavu Running. Ukoncene vlakno je ve stavu Finished.

# Thread.Abort

Jakmile je zavolano, vlakno vstoupi do stavu AbortRequested. Tento stav je urcitou formou stavu bezici, protoze podproces obdrzi vyjimku ThreadAbortedException, kterou musi obslouzit. Pokud vlakno, ktere vola Thread.Abort nezavola Thread.ResetAbort, prejde vlakno do stavu Aborted (predcasne ukoncene). Z tohoto stavu se muze vratit do stavu running pokud vlakno ktere zavolalo abort, zavola ResetAbort.

# Thread.Suspend, Resume

Pokud aktualni podproces zavola metodu Suspend, pak na danem miste blokuje a ceka na zavolani metody Resume Bezici vlakno prejde do stavu SuspendRequested (pozadovano pozastaveni) a nasledne do stavu Suspend. Vlakno se nezastavi okamzite, ale pozastavi az v bezpecnem bode (Safe point). **Bezpecny bod** je misto v kodu, kde lze bezpecne spustit uklid pameti. CLR pri uklidu pozastavi vsechny podprocesy. Jeli podproces uprostred multiinstrukcni operace ktera pristupuje k objektem z haldy, a pote prijde automaticka sprava pameti a presune objekty na jine misto v pameti nedopadne to dobre. Proto musi sprava pameti pockat, dokud vsechny podprocesy nedorazi do bezpecneho bodu, v nemz lze objekty presouvat.

# Thread.Interrupt

Probudi vlakno s tim, ze na nem vyvola vyjimku ThreadInterruptedException. Bud je vyjimka odchycena v bloku Try-Catch, nebo probubla na vrchol zasobniku a behove prostredi zastavi provadeni podprocesu kvuli neobslouzene vyjimce.

POKUD STEJNOU APLIKACI SPUSTITE NA VICEPROCESOROVEM SYSTEMU, PRAVDEPODOBNE UVIDITE VYSTUP V UPLNE JINEM PORADI.

# **Synchronizace cinnosti mezi podprocesy**

Potrebujeme pro komunikaci s podprocesy, pokud pristupuji soubezne k datum ke kterym soubezne pristupuji jine podprocesy. Muzou nastat dva problemy: Soubeh a uvaznuti podprocesu.

**Soubeh:**  Pri soubehu se dva podprocesy snazi pristupovat ke stejnemu mistu v pameti. Jeden podproces nedokonci zmenu bloku pameti, ztrati rizeni a druhy zacne tuto pamet cist s virou, ze je v platnem stavu.

**Uvaznuti:**  muze nastat pokud dva podprocesy navzajem cekaji az druhy uvolni nejaky prostredek. Oba tak cekaji a budou cekat vecne.

U vsech synchronizaci by se mel pouzivat tem nejlehci synchronizacni mechanizmus. Synchronizacni mechanizmy dovoli pristupovat k datum v jednom okamziku pouze jednomu podprocesu

# Synchronizacni mechanizmy (Mutex, Monitor, Interlocked)

# Mutex

# Monitor

# Interlocked

Se pouziva v pripade, kdy je mala pravdepodobnost delsiho cekani na ziskani zamku. Priklad : [c:\Users\Vladenka\OneDrive\Nielsen prace\LinqPad queries\Interlocked.linq](file:///c:\Users\Vladenka\OneDrive\Nielsen%20%20prace\LinqPad%20queries\Interlocked.linq)

Trida Interlocked ma nekolik metod, ktere zarucuji atomicke operace napr inkrementace.

Interlocked.Increment( ref …)

Interlocked.Decrement(ref …)

Interlocked.Exchange(ref arg1, arg2) = vymeni hodnotu arg1 za arg2

Interlocked.CompareExchange(ref arg1, arg2, arg3) vymeni hodnotu arg1 za arg2 jen pokud arg1 == arg3

# lock()

static Object \_lock = new Object (); // musí existovat jen jednou tj netvorit napr v bloku for

// pokud pouzijeme lock, jen jeden thread muze vstoupit do tohoto bloku a provadet kod uvnitr bloku

lock (\_lock)

{

Total++;

}

# spinlock()

(cyklicke uzamknuti) zamek spinlock je pro situace, kde se ocekava, ze trida, ktera ho ziskava bude jen zridkakdy cekat. Ziskal svuj nazev kvuli skutecnosti, ze pokud nemuze ziskat zamek, pak rotuje v tesnem cyklu, dokud jej neziska.

# AddOneMilionItems (3 vlakna, ukazka pouziti Interlocked a lock())

[c:\Users\Vladenka\Documents\Visual Studio 2017\Projects\ConsoleApp1\ConsoleApp1\bin\Debug\ConsoleApp1.exe.config](file:///c:\Users\Vladenka\Documents\Visual%20Studio%202017\Projects\ConsoleApp1\ConsoleApp1\bin\Debug\ConsoleApp1.exe.config)

namespace ConsoleApp1

{

// Musime zbuildit a spustit z cmd (bin\debug\ exe soubor) nekolikrat za sebou.

class Program

{

static void Main (string[ ] args)

{

var t1 = new Thread (() => AddOneMilion ());

var t2 = new Thread (() => AddOneMilion ());

var t3 = new Thread (() => AddOneMilion ());

t1.Start ();

t2.Start ();

t3.Start ();

t1.Join ();

t2.Join ();

t3.Join ();

Console.WriteLine ("Total: "+Total);

Console.ReadLine ();

}

private static int Total = 0;

static Object \_lock = new Object ();

static void AddOneMilion ()

{

for (int i = 0 ; i < 1000000 ; i++)

{

1)

Total++;

// Total bude v nekterych pripadech mensi napr. 2 800 000. Prvni thread dopocita promennou do milionu. Protoze vsechny thready pouzivaji

// sdilenou promennou, v te chvili se zastavi i ostatni thready i kdyz jeste nedopocitali do milionu.

2)

Interlocked.Increment (ref Total);

// Pouziti Interlocked zaruci, ze v prubehu inkrementace jsou ostatni thready zamcene

3)

// pokud pouzijeme lock, jen jeden thread muze vstoupit do tohoto bloku a provadet kod uvnitr bloku

lock (\_lock)

{

Total++;

}

}

}

}

}