Niti

.Net Framework obezbeđuje različite načine za postizanje konkurentnosti. Tri glavna načina su:

1. Direktnim pristupom nitima - pomoću Thread klase
2. Koristeći ThreadPool
3. Programiranjem baziranom na zadacima - pomoću Task klase

Thread:

Direktan pristup nitima predstavlja najniži nivo pristupa i usko je povezan sa operativnim sistemom, umesto sa CLR-om(eng. Common Language Runtime). Ovaj pristup zahteva opreznost prilikom rada, ali za uzvrat pruža najviše kontrole i opcija. Kako se radi o nitima na nivou operativnog sistema, svaka nit je opskrbljena svojim stekom i procesorskim resursima, zbog čega aplikacija može postati skupa(sa aspekta resursa), ukoliko se ne vodi dovoljno računa. Ove niti mogu biti uređene i kontrolisane postavljanjem veličine steka, kulture, kao i posmatranjem njihovog stanja. Svaka nit može biti zaustavljena, pauzirana ili nastavljena.

ThreadPool:

Kako bi se pojednostavilo uređivanje i kontrola niti, Microsoft je uveo koncept ThreadPool-a. ThreadPool je upravljan od strane CLR-a i predstavlja wrapper oko grupe niti. Kontrola nad ThreadPool-om je ograničena. Opciono se može izvršiti podešavanje veličine ThreadPool-a i zatim se na njemu obavlja željeni posao. CLR pronalazi slobodnu nit unutar ThreadPool-a, i njoj prosledjuje zadati posao na izvršavanje.
Korišćenjem ThreadPool-a izbegavaju se troškovi kreiranja previše niti. Međutim, ukoliko mu se prosledi previše velikih zadataka, mogu se zauzeti sve niti, čime se svaki sledeći posao odlaže dok neka od niti ne postane slobodna. Uz ovo, ThreadPool samo po sebi ne nagoveštava kada je neki od poslova završen, niti pruža način za preuzimanje rezultata. Zbog ovoga, ThreadPool je najbolje koristiti za kratke operacije, kod kojih korisniku nije potreban rezultat.

Task:

Kao još jedan korak unapred, Microsoft je kreirao Task Parallel biblioteku(eng. Task Parallel Library – TPL), koja kombinuje karakteristike Thread klase(notifikacije) i ThreadPool-a(automatsko upavljanje nitima). Zadaci(eng. Task) koriste TaskScheduler koji zakazuje poslove i radi na ThreadPool-u. Ovo predstavlja najnapredniji i najuredjeniji pristup za postizanje konkurentnosti i ima sledeće karakteristike:

- Opciju za nastavljane zadataka - ContinueWith metoda
- Task<T> - generički povratni tip za prosleđivanje rezultata
- Wait metodu za sinhrono izvršavanje zadataka i čekanje rezultata
- Mogućnost prosledjivanja velikih poslova na posebnu nit, umesto slanja na ThreadPool

Thread

Nit predstavlja nezavisnu putanju izvršavanja, koja je u stanju da bude pokrenuta istovremeno sa drugim nitima. C# klijentska aplikacija(Console, WPF, Windows Forms) se pokreće u okviru jedne niti, automatski kreirane od strane CLR-a i operativnog sistema, i pretvara se u višenitnu aplikaciju kreiranjem dodatnih niti. U nastavku je dat primer kreiranja jedne dodatne niti:

```
using System;
using System.Threading;

class ThreadTest
{
	static void Main()
	{
		Thread t = new Thread (WriteY);
		t.Start();
		for (int i = 0; i < 1000; i++) Console.Write ("x");
	}

	static void WriteY()
	{
		for (int i = 0; i < 1000; i++) Console.Write ("y");
	}
}
```

Main nit kreira novu nit t, unutar koje se izvršava metoda koja konstanto ispisuje karakter ‘y’. Istovremeno, main nit konstantno ispisuje karakter ‘x’. Rezultat ovoga je sledeći:

```
Output:	xxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
		xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyy
		yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxx
		xxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
		yyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
		...
```

Jednom kada se nit pokrene, njeno IsAlive svojstvo vraća true vrednost, do trenutka gašenja niti. Nit se gasi kada se završi sa izvršavanjem delegata prosleđenog konstruktoru klase Thread.

CLR svakoj niti dodeljuje sopstveni memorijski stek, kako bi se lokalne varijable čuvale zasebno. U sledećem primeru, definisana je metoda sa lokalnom varijablom, a zatim je ta metoda istovremeno pozvana u main niti, kao i našoj novokreiranoj niti.

```
static void Main()
{
	new Thread (Go).Start();
	Go();
}

static void Go()
{
	for (int cycles = 0; cycles < 5; cycles++) Console.Write ('?');
}
```

U svakom steku se pravi odvojena kopija promenljive cycle, tako da je ispisano 10 upitnika.

```
Output:	??????????
```

Niti dele podatke ukoliko imaju zajedničku referencu na istu instancu objekta. Na primer:

```
class ThreadTest
{
	bool done;

	static void Main()
	{
		ThreadTest tt = new ThreadTest();
		new Thread (tt.Go).Start();
		tt.Go();
	}


	void Go()
	{
		if (!done) { done = true; Console.WriteLine ("Done"); }
	}
}
```

S obzirom da obe niti pozivaju Go metodu sa istom ThreadTest instancom, između ovih niti će biti deljeno i done polje. Ovo rezultuje jednim ispisom “Done”, umesto dva.

```
Output:	Done
```

Statička polja predstavljaju još jedan način za deljenje podataka između niti, kao što je prikazano i u narednom primeru:

```
class ThreadTest
{
	static bool done;

	static void Main()
	{
		new Thread (Go).Start();
		Go();
	}

	static void Go()
	{
		if (!done) { done = true; Console.WriteLine ("Done"); }
	}
}
```

Ova dva primera ilustruju još jedan ključni koncept: sigurnost niti(ili tačnije, nedostatak iste). Rezultati/ispisi ovih primera su neodređeni - može se desiti da “Done” bude ispisano dva puta, iako je to malo verovatno. Međutim, ukoliko se zameni redosled operacija unutar Go metode, verovatnoća duplog ispisa naglo raste.

```
static void Go()
{
	if (!done) { Console.WriteLine ("Done"); done = true; }
}

Output:	Done
		Done
```

Problem nastaje kada jedna od niti proverava if uslov tačno kada druga nit izvršava WriteLine metodu - pre nego što uspe da promeni vrednost done promenljive u true.

Rešenje ovog problema je zaključavanje deljenih polja prilikom čitanja i pisanja. C# obezbeđuje lock naredbu isključivo za ovu namenu. Primer upotrebe lock naredbe je dat u nastavku:

```
class ThreadSafe
{
	static bool done;
	static readonly object locker = new object();

	static void Main()
	{
		new Thread (Go).Start();
		Go();
	}

	static void Go()
	{
		lock (locker)
		{
			if (!done) { Console.WriteLine ("Done"); done = true; }
		}
	}
}
```

Kada dve niti pristupe lock promenljivoj, u ovom slučaju locker, jedna od niti čeka, odnosno biva blokirana, dok se promenljiva ne oslobodi. Ovo osigurava da će u svakom trenutku samo jedna nit biti u stanju da pristupi kritičnom delu koda, i da će "Done" biti ispisano samo jednom. Kod koji je zaštićen na ovaj način, zove se thread-safe kod. Nit koja je blokirana ne koristi procesorske resurse.

Join i Sleep

Pozivajući Join metodu, možemo da sačekamo da se neka druga nit završi.

```
static void Main()
{
	Thread t = new Thread (Go);
	t.Start();
	t.Join();
	Console.WriteLine ("Thread t has ended!");
}

static void Go()
{
	for (int i = 0; i < 1000; i++) Console.Write ("y");
}
```

Prethodni kod ispisuje "y" 1000 puta, nakon čega odmah sledi "Thread t has ended!". Uz Join metodu može da se priključi i time-out, u milisekundama ili kao TimeSpan. Ovaj time-out vraća true vrednost, ukoliko se nit završila, ili false, ukoliko je isteklo zadato vreme.

Komanda Thread.Sleep pauzira/uspavljuje trenutnu nit na određen period.

```
Thread.Sleep (TimeSpan.FromHours (1));
Thread.Sleep (500);
```

Dok čeka na Sleep ili Join, nit je blokirana i ne troši procesorske resurse.

Kreiranje i pokretanje niti

Nit se kreira koristeći Thread konstruktor, prosleđujući ThreadStart delegat kao parametar. Ovaj delegat ukazuje na to gde počinje izvršavanje koda(od koje metode). ThreadStart delegat je definisan na sledeći način:

```
public delegate void ThreadStart();
```

Nakon kreiranja niti, potrebno je pozvati Start metodu, kako bi nit bila pokrenuta. Nit nastavlja sa radom dok njena metoda ne vrati vrednost, prilikom čega se nit završava.

```
class ThreadTest
{
	static void Main()
	{
		Thread t = new Thread (new ThreadStart (Go));
		t.Start(); // Run Go() on the new thread.
		Go(); // Simultaneously run Go() in the main thread.
	}

	static void Go()
	{
		Console.WriteLine ("hello!");
	}
}
```

U ovom primeru, nit t izvršava metodu Go - a u isto vreme main nit poziva tu istu metodu. Rezultat su dva gotovo trenutna “Hello!” ispisa.

Nit može biti kreirana i jednostavnije, specificirajući samo metodu i dozvoljavajući C#-u da zaključi da se radi o ThreadStart delegatu.

```
Thread t = new Thread (Go);
```

Jos jedna prečica je korišćenje lambda izraza ili anonimnih metoda, kao u narednom primeru:

```
static void Main()
{
	Thread t = new Thread ( () => Console.WriteLine ("Hello!") );
	t.Start();
}
```

Prosleđivanje podataka do niti

Najjednostavniji način za prosleđivanje argumenata metodi koja se pokreće u okviru neke niti jeste izvršavanjem lambda izraza koji pozivaju metodu sa željenim argumentima.

```
static void Main()
{
	Thread t = new Thread ( () => Print ("Hello from t!") );
	t.Start();
}

static void Print (string message)
{
	Console.WriteLine (message);
}
```

Ovim pristupom, metodi je moguće proslediti bilo koji broj parametara. Moguće je čak i uvrstiti celu implementaciju koda u telo lambde.

```
new Thread (() =>
{
	Console.WriteLine ("I'm running on another thread!");
	Console.WriteLine ("This is so easy!");
}).Start();
```

Drugi način za prosleđivanje parametara jeste njihovim prosleđivanjem Start metodi.

```
static void Main()
{
	Thread t = new Thread (Print);
	t.Start ("Hello from t!");
}

static void Print (object messageObj)
{
	string message = (string) messageObj; // We need to cast here
	Console.WriteLine (message);
}
```

Ovo funkcioniše jer Thread konstruktor prihvata jedan od dva tipa delegata:

```
public delegate void ThreadStart();
public delegate void ParameterizedThreadStart (object obj);
```

Ograničenje ParameterizedThreadStart delegata je prihvatanje samo jednog argumenta. Zbog toga što je ovaj argument tipa object, obično ga je neophodno kastovati. Sa ovim na umu, moguće je kreirati nove klase koje sadrže sve neophodne vrednosti za izvršavanje određene metode, i putem Start metode ih proslediti niti u vidu nove instance te klase, koristeci polimorfizam.

Imenovanje niti

Svaka nit ima Name svojstvo koje može biti postavljeno u svrhu debagovanja. Ovo je posebno korisno u Visual Studio-u, jer se ime niti prikazuje unutar Thread Window i Debug Location toolbar-a. Ime niti može biti postavljeno samo jednom, a svaki naredni pokušaj izmene imena izbacuje izuzetak.

Statičko svojstvo Thread.CurrentThread nam daje trenutno izvršavanu nit. U narednom primeru, podešava se ime niti.

```
class ThreadNaming
{
	static void Main()
	{
		Thread.CurrentThread.Name = "main";
		Thread worker = new Thread (Go);
		worker.Name = "worker";
		worker.Start();
		Go();
	}

	static void Go()
	{
		Console.WriteLine ("Hello from " + Thread.CurrentThread.Name);
	}
}
```

Pozadinske niti

Uobičajeno je da se niti koje kreiramo izvršavaju u prvom planu, odnosno da se kreiraju kao foreground niti. Ove niti održavaju aplikaciju pokrenutom dok god je bar jedna od njih pokrenuta. S druge strane, postoje i pozadinske, background niti, koje ovo ne rade. Kada se sve foreground niti izvrse, aplikacija se gasi, a sve pozadinske niti, koje su i dalje pokrenute, se naglo prekidaju.

Pozadinski status niti se može dobiti ili menjati putem IsBackground svojstva. Sledi primer ovoga:

```
class PriorityTest
{
	static void Main (string[] args)
	{
		Thread worker = new Thread ( () => Console.ReadLine() );
		worker.IsBackground = true;
		worker.Start();
	}
}
```

Prioritet niti

Svojstvo Priority određuje koliko izvršnog vremena će nit dobiti u odnosu na ostale aktivne niti u operativnom sistemu, sa sledećom skalom:

```
enum ThreadPriority { Lowest, BelowNormal, Normal, AboveNormal, Highest }
```

Ovo postaje bitno samo kada je istovremeno aktivno više niti. *Napomena: pozadinski status niti ne utiče na njen prioritet.*

Thread Pool

Postoji nekoliko načina za pristup thread pool-u:

- Pomoću Task Parallel biblioteke
- Pozivajući ThreadPool.QueueWorkItem
- Pomoću asinhronih delegata
- Pomoću BackgroundWorker-a

Pristup pomoću Task Parallel biblioteke

Thread pool-u je moguće pristupiti koristeći Task klasu iz Task Parallel biblioteke. Klasa Task, kao što će biti prikazano i u narednom poglavlju, ima dva oblika(generički i negenerički), tako da će biti prikazana dva primera upotrebe ove klase.

Kako bi se iskoristila negenerička Task klasa, potrebno je pozvati Task.Factory.StartNew metodu, prosleđujući delegat za željenu metodu:

```
using System.Threading.Tasks

static void Main()
{
	Task.Factory.StartNew (Go);
}

static void Go()
{
	Console.WriteLine ("Hello from the thread pool!");
}
```

Task.Factory.StartNew metoda vraća Task objekat, koji potom može biti upotrebljen za nadgledanje prosleđenog zadatka/metode - na primer, može se čekati završetak zadatka, koristeći Wait metodu.

Generička Task<TResult> klasa predstavlja podklasu negeneričke Task klase. Ova klasa nam dozvoljava vraćanje povratne vrednosti nakon završetka izvršavnja zadatka. U sledećem primeru se može videti primerna ove klase:

```
static void Main()
{
	Task<string> task = Task.Factory.StartNew<string>
	( () => DownloadString ("http://www.linqpad.net") );

	RunSomeOtherMethod();
	string result = task.Result;
}

static string DownloadString (string uri)
{
	using (var wc = new System.Net.WebClient())
	return wc.DownloadString (uri);
}
```

Pristup bez Task Parallel biblioteke

Task Parallel biblioteka ne može biti korišćena u ranijim verzijama .Net Framework-a. Umesto toga, moraju se upotrebiti neki od strijih načina za pristup thread pool-u: ThreadPool.QueueUserWorkItem i asinhroni delegati. Razlika između ovih načina je ta da asinhroni delegati dozvoljavaju vraćanje povratne vrednosti iz niti, ali oni zbog kompleksnosti neće biti obrađeni u okviru ovih materijala.

Kako bi se upotrebila metoda QueueUserWorkItem, potrebno je samo da se ona pozove, sa odgovarajućim delegatom kao parametrom:

```
static void Main()
{
      ThreadPool.QueueUserWorkItem (Go);
      ThreadPool.QueueUserWorkItem (Go, 123);
      Console.ReadLine();
}

static void Go (object data) // data will be null with the first call.
{
      Console.WriteLine ("Hello from the thread pool! " + data);
}

Output:      Hello from the thread pool!
             Hello from the thread pool! 123
```

Naša Go metoda mora da prihvati jedan object argument, kako bi bio zadovoljen WaitCallback delegat. Ovo pruža jednostavan način za prosleđivanje podataka do metode, kao i kod ParemetrizedThreadStart delegata. Za razliku od Task klase, metoda QueueUserWorkItem ne vraća objekat koji nam pomaže u nadgledanju i uređivanju izvršavanja koda.

Task

U verziji 5, C# je uveo mogućnost pisanja i pozivanja koda asinhrono. Asinhroni kod omogućava nastavak izvršavanja dugotrajnih operacija u odvojenim nitima, kako bi pozivajuća nit nastavila sa svojim zadacima. Kada pozivajuća nit predstavlja UI nit, aplikacija ima odziv i moguće je prikazati status bar ili indikator zauzetosti, ili omogućiti korisniku rad u drugom delu programa, dok je asinhroni proces pokrenut. Kada asinhroni proces vrati neku vrednost, moguće je interagovati sa korisnikom na neki način, ukoliko to ima smisla u našoj aplikaciji.

C# ima dve ključne reči koje podržavaju asinhrono programiranje: async i await. Opisivanje metode pomoću async modifikatora nam govori da ta metoda može da sadrži asinhroni kod. Ključna reč await se koristi nad klasom Task, kako bi se pokrenula asinhrona operacija.

```
using System.IO;
using System.Threading.Tasks;

public class Program
{
	static void Main()
	{
		Program.CreateFileAsync("test.txt").Wait();
	}

	static async Task CreateFileAsync(string fileName)
	{
		using (StreamWriter writer = File.CreateText(fileName))
		{
			await writer.WriteAsync("This is a test.");
		}
	}
}
```

U prethodnom primeru, na osnovu modifikatora async, vidimo da je metoda CreateFileAsync asinhrona. Klasa File i njena metoda CreateText vraćaju StreamWriter koji se koristi za upis u fajl.

Pravilan način za pozivanje asinhronih metoda je upotrebom ključne reči await uz instancu klase Task ili Task<T>, koju vraća ta metoda. Metoda WriteAsync vraća instancu klase Task, što znači da ona može da bude await-ovana, odnosno da na osnovu te instance možemo nadgledati prosleđeni zadatak i čekati na njegov završetak.

Naredba using zatvara fajl kada se završi ograđen blok koda. U ovom slučaju blok sadrži samo jednu liniju, tako da nisu neophonde vitičaste zagrade.

Deo asinhronog ugovora predstavlja očekivanje da će neki deo koda biti pokrenut unutar druge niti, kako bi se naša nit oslobodila za druge operacije. To je ono što radi i WriteAsync metoda. Iz tog razloga, nit se vraća na kod koji je pozvao asinhronu metodu. Pozivalac je u ovom slučaju Main metoda, koja poziva Wait metodu nad Task instancom vraćenom od strane CreateFileAsync metode, sprečavajući završavanje programa pre kraja pokrenutih asinhronih operacija.

Modifikator async je neophodan ukoliko se za metodu koristi ključna rec await. Ukoliko metoda ima async modifikator, a nema await, C# će izbaciti upozorenje i najaviti da će se ta metoda izvršiti sinhrono, odnosno da će pozivajuća nit biti blokirana do kraja izvršavanja te metode.

Povratni tipovi asinhronih metoda

Upotrebom modifikatora async, moguće je await-ovati bilo koju metodu koja je za to sposobna (eng. awaitable). Ugrađena biblioteka sadrži Task i Task<T> klase, koje su awaitable i koje bi trebalo koristiti u najvećem broju situacija. Vraćanjem Task instance, asinhrona metoda isključuje mogućnost povratka neke druge vrednosti, kao što je to bio slučaj kod prethodne CreateFileAsync metode.

Ukoliko je neophodno vratiti neku drugu vrednost, trebalo bi upotrebiti Task<T> klasu. Sledeći kod prikazuje primer ovoga.

```
public async Task<string> ReturnGreeting()
{
	await Task.Delay(1000);
	return "Hello";
}
```

Task.Delay predstavlja način za uspavljivanje niti na određen broj milisekundi. Prethodni primer prikazuje povratni tip Task<string>. Metoda vraća samo string "Hello" umesto instance Task<string>, jer se C# kompajler stara o tome umesto nas.

Asinhrona metoda može biti i void povratnog tipa. Ovo je prikazano u nastavku.

```
public async void SayGreeting()
{
	await Task.Delay(1000);
	Console.Writeline("Hello");
}
```

Ova metoda se izvršava asinhrono, ali asinhrone void metode imaju nekoliko mana: one nisu awaitable, i ne štite od izuzetaka, ali su neophodne za scenarije rukovanja događajima, gde metode moraju biti void tipa.

Pozivanje asinhrone void metode znači da kao odgovor ne dobijamo Task instancu, zbog čega se ne može čekati na njen završetak, niti ćemo uopšte znati kada se taj završetak dogodio.

Pozivanje asinhronih metoda

Kako bi asinhrone metode stvarno bile asinhrono izvršene, bez blokiranja pozivajuće niti, one moraju biti await-ovane. S obzirom da metoda može biti await-ovana samo iz druge asinhrone metode, a da u okviru konzolne aplikacije mora postojati sinhroni main, za pozivanje ovih metoda se moraju iskoristiti klasa Task i anonimne metode. Poziv jedne Task asinhrone metode je dat u nastavku.

```
public class Program
{
	static void Main()
	{
		Task t = Task.Run(async () => await AsyncMethod());
		//Izvršavanje ove metode može biti sačekano koristeći t.Wait komandu

		Console.WriteLine("Main done");
		Console.ReadLine();
	}
```

```
        public static async Task AsyncMethod()
        {
                await Task.Delay(5000);
                Console.WriteLine("Async method done");
        }
}
```

Pozivanje Task<T> metoda je nešto kompleksnije, jer ove metode vraćaju i određeni odgovor.

```
public class Program
{
        static void Main()
        {
                Task<string> t = Task.Run(async () => await AsyncMethod());

                Console.WriteLine("Main done");
                Console.WriteLine(t.Result);

                Console.ReadLine();
        }

        public static async Task<string> AsyncMethod()
        {
                await Task.Delay(5000);
                return "Async method done";
        }
}
```

Prilikom pozivanja ovih metoda treba voditi računa o redosledu izvršavanja zadataka, kako pozivajuća nit ne bi ostala blokirana. S obzirom da je za dobijanje t.Result vrednosti potrebno sačekati na ivršavanje asinhrone metode, pozivanje WriteLine(t.Result) komande pre WriteLine("Main done"), bi blokiralo main nit na 5 sekundi. Ovaj problem se javlja i prilikom korišćenja ThreadPool-a.

Asinhrone metode mogu biti pozvane i unutar asinhrone void metode, ali u ovom slučaju nikakav odgovor ne može biti prosleđen do pozivajuće niti. Primer ovog poziva je dat u nastavku.

```
public class Program
{
        static void Main()
        {
                AsyncVoidMethod();

                Console.WriteLine("Main done");
                Console.ReadLine();
        }

        public static async void AsyncVoidMethod()
        {
                await AsyncMethod();
        }

        public static async Task AsyncMethod()
        {
                await Task.Delay(5000);
                Console.WriteLine("Async method done");
        }
}
```