Poboljšanje propusnosti

Uvod u rad sa nitima

- C# podržava paralelno izvršavanje koda kroz concept niti – engl. ***multithreading***.
- Jedna nit je nezavisna izvršna putanja (nit) koja se može izvršavati istovremeno sa drugim.
- C# program se startuje u jednoj niti koja se kreira automatski od strane CLR i operativnog sistema. (the "**main**" thread)
- Prostori imena za rad sa nitima
- `using System;`
  `using System.Threading;`

Primer 1.

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

t.Start();

while (true)

     Console.Write ("x");

....

static void WriteY()

{
    while (true)
    Console.Write ("y");

}
```

Lokalne promenljive

- CLR pridružuje svakoj niti sopstveni stek za skladištenje lokalnih promenljivih.

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

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

Zajedničke promenljive

- Niti dele podatke ako imaju zajedničke reference na iste objekte:

```
static bool done;
Program tt = new Program();
new Thread (tt.Go).Start();
tt.Go();

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

Statička polja i nejasna kontrola koda

```
// st. polja se dele izmedju svih niti
static bool done;
static void Main(){
    new Thread (Go).Start();
    Go();
}
static void Go() {
    if (!done) { done = true; Console.WriteLine
    ("Done"); }
}
```

```
Done
Done  (usually!)
```

Zaključavanje koda

```
static bool done;
static object locker = new object();
static void Main() {
     new Thread (Go).Start();
     Go();
}
static void Go() {
     lock (locker) {
          if (!done) { Console.WriteLine
     ("Done");
                       done = true; }
     }
 }
```

- "*Done*" će biti prikazano samo jednom i to UVEK.

- Kada dve niti istovremeno sadrže blok komandu **lock** , jedna nit je na čekanju tj. **blokirana** je, dok zaključana promenljiva ne postane slobodna. Tako se garantuje da samo jedna nit bude dostupna jednom delu koda.
- Takav kod koji je zaštićen od nedređenog višenitnog konteksta naziva se thread-safe.

Privremeno zaustavljanje niti

- Sleep

- `Thread.Sleep ( 30000 );`
- `Thread.Sleep (TimeSpan.FromSeconds (30));`

Sinhronizacija niti: Join

- Jedna nit može čekati drugu da završi sa radom pozivom metode **Join**:

```
Thread t = new Thread (Go);
t.Start();
t.Join();
```

Threads vs. Processes

- Sve niti u jednoj aplikaciji su logički sadržani u jednom procesu – tj. jedinici operativnog sistema u kome se aplikacija izvršava.
- Niti imaju neke sličnosti sa procesima – tipično procesi dele procesorsko vreme na isti način kao niti unutar procesa. **Ključna razlika je što su procesi potpuno izolovani jedan od drugog; nit deli (heap) memoriju sa drugim nitima iste aplikacije.**
- Na primer, jedna nit može pribavljati podatke dok ih druga može prikazivati.

Kada koristiti niti?

Kada ne koristiti niti

- Najveći nedostatak upotrebe niti je što može dovesti do veoma kompleksnih programa.
- Više niti zahteva i gubitak vremena na prebacivanje sa jedne niti na drugu što može izazvati usporenje.

Niti i asinhrone metode

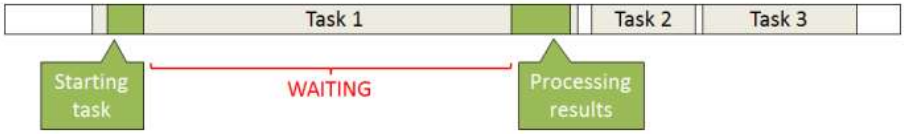
C#

Uvod

- Mada su procesori sve brži i brži, očekivanja i zahtevi za neprikidnim vezama sa podacima preko Interneta, učinili su da se značajniji dobitak jedino može ostvariti paralelnim radom i raspodelom procesorskog vremena i ostalih resursa na više zadataka
- Interakcija sa računarom je inicirana od korisnika i uglavnom vođena događajima koji su inicirani od njega. U slučaju poziva metoda koje se mogu izvršavati neprihvatljivo dugo aplikacija sa kojom se radi ne sme da ostane u stanju čekanja tj. blokiranja. Na primer, pri pozivu neke metoda koja obavlja preuzimanje sadržaja sa Interneta, vreme od svega nekoliko sekundi može učiniti korisnika nezadovoljnim u pogledu rada. U tom periodu svi događaji od operativnog sistema se ne obrađuju pa samim tim i od korisnika. Dakle, nastupa vremenski period kada nema odziva. Ovo se nastoji izbeći kada god je to moguća. A moguće je primenom asinhronih poziva.

Sinhroni model

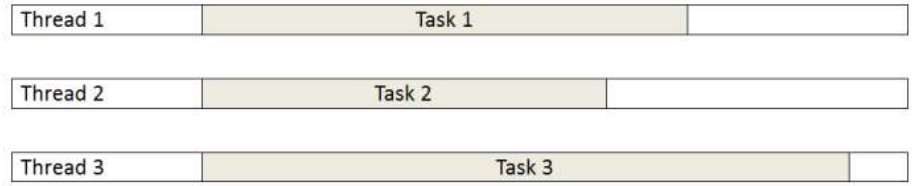
- Svi programi se mogu predstaviti zadacima koji se odnose na neke poslove koji se izvršavaju. Zadaci mogu biti i same metode u nekom programskom modelu. U slučaju tradicionalnog, sinhronog programskom modela, jedan zadatak sledi nakon završetka prethodnog.



- Pošto se zadaci obavljaju sekvencijalno, jedan posle drugog, jedan zadatak se može osloniti na ishod prethodnog. Problemi nastaju kada neki zadatak traje predugo, na primer čitanje ogromnog fajla sa diska, preuzimanje sadržaja preko Interneta i slično. Podrazumevano, operativni sistem dodeljuje jednu nit za svaki proces, a pošto ta nit postane blokirana onda i rad sa UI elementima takođe postaje blokiran.

Paralelne niti

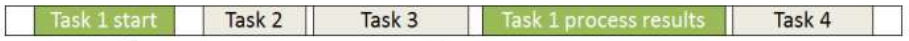
- Prirodno rešenje pomenutog problema je dodavanje više niti u isti proces.



- Na ovaj način, svaki zadatak se izvršava u zasebnoj niti, a operativni sistem vodi računa o tome da svaka nit dobije odgovarajuće resurse odnosno da se zadaci izvršavaju istovremeno. Pomoću ovog modela moguće je da se UI kontrole i događaji održe u jednoj niti a da zadaci koji mogu da utiču na dugotrajnost ili blokiranost aplikacije budu izmeštene u druge niti. Prebacivanje izvršavanja iz niti u nit ili odvojeno izvršavanje niti na više procesora je pod kontrolom operativnog sistema.

Asinhrone metode

- Međutim i prethodno rešenje ima i svoje nedostatke. Pošto zadaci koji se paralelno izvršavaju treba i da sarađuju u okviru jedne aplikacije, veliki problem je rad sa podacima tj kontrola pristupa podacima iz više niti, a to po nekada može da učiniti rad programera jako težak sa ovakvim modelom. Primenom tzv. asinhronog modela mogu se dobiti gotovo identična funkcionalnost ali uz mnogo manje problema koji postoje kod rada sa više niti.



- Na ovaj način se programiranje čini znatno jednostavnijim uz velike prednosti u pogledu izvršavanja dugotrajnih zadataka. Na prethodnoj slici, dugotrajni zadatak 1 je iniciran, naravno u asinhronom modu. Zatim se nastavlja sa izvršavanje zadataka 2 odnosno 3 da bi se nastavilo sa zadatkom 1, obradom rezultata, pošto se isti završio.

Task

- `Thread` – Klasa za obavezno kreiranje niti (low-level concept).
- `Task` – Viši nivo apstrakcije. Više se tumači kao akcija koja će se izvršti u budućnosti.
- Podrazumevani Task upućuje na metodu bez argumenata:
- `// može se proslediti metoda bez objekta Action`
- `Task task = new Task(velikiObracun);`
- `task.Start();`

```
void velikiObracun()
{
    System.Threading.Thread.Sleep(2000);
}
```

- Task koji upućuje na akciju sa argumentom (uvek se koristi object tip):
- `Action<object> a;`
- `a = velikiObracun;`
- `Task task = new Task(a, 2000);`
- `task.Start();`

```
void velikiObracun(object ms)
{
    System.Threading.Thread.Sleep((int)ms);
}
```

Produžavanje niti

```
private void button1_Click(object sender,
EventArgs e){
    Task task = new Task(velikiObracun);
    task.Start();
    Task noviTask =
    task.ContinueWith(korektivniObracun);
}
void velikiObracun(){
    button1.BackColor = Color.Green;
    System.Threading.Thread.Sleep(2000);
}
void korektivniObracun(Task task){
    button1.BackColor = Color.Red;
    System.Threading.Thread.Sleep(2000);
}
```

- Ako se prvi zadatak ne prekida i nema izuzetaka onda se može produžiti i na sledeći način:

```
Task task = new Task(velikiObracun);
task.ContinueWith(korektivniObracun);
task.Start();
```

Sinhronizacija niti

- Obustavlja izvršavanje niti dok se zadatak task ne završi:
- `task.Wait();`

- Obustavlja izvršavanje niti dok se oba zadatka ne završe:
- `Task.WaitAll(task, task2);`

- Obustavlja izvršavanje niti dok se bar jedan zadatak ne završi:
- `Task.WaitAny(task, task2);`

Konkurentnost

- Istovremeni pristup podacima je veoma značajan, ali za programere obično, problematičan deo vezan za rad sa istovremenim zadacima odnosno nitima.
- Kao i kod niti, zabrana pritupa se može obaviti naredbom **lock** tj zaključavanjem bilo kog objekta na nekom bloku naredbi oivičenom vitičastim zagradama na koju se lock odnosi. Na tom bloku važi da samo jedna nit može pristupiti u jednom trenutku tom bloku. Ako bi druga nit stigla do tog bloka pre nego što prva napusti, ona mora da sačeka. Na ovaj način je sigurno samo jedna nit na tom bloku moguća.
- ```
  object myObject = new object();
  lock (myObject)
  {
        //Acces to critical ressources
  }
  ```
  ```

Paralelizam

Task u realizaciji paralelizma

- Primer 1a: Početni problem sa sporim sinhronim odzivom.

```
private void button1_Click(object sender, EventArgs e)
{
    velikiRacun();
}
void velikiRacun()
{
    System.Threading.Thread.Sleep(5000);
    this.BackColor = Color.Green;
}
```

- Primer 1b. Primer jedne metode koja se asinhrono izvršava, ali i dalje sporo!

```
private void button1_Click(object sender, EventArgs e)
{
    Task t1 = new Task(velikiRacun);
    t1.Start();
    this.BackColor = Color.Red;
}
void velikiRacun()
{
    System.Threading.Thread.Sleep(5000);
    this.BackColor = Color.Green;
}
```

- Primer 2a. Podela jedne spore metode na više manjih metoda

```
void velikiRacun()
{
    for (int i = 1; i < 5; i++)
    {
        velikiRacunModifikovan(i);
    }
    this.BackColor = Color.Green;
}
void velikiRacunModifikovan(object i)
{
    System.Threading.Thread.Sleep((int)1000);
}
```

Primer 2b. Poziv manjih metoda paralelno. Ubrzavanje!

```
private void button1_Click(object sender, EventArgs e){
    velikiRacun();
}
void velikiRacunModifikovan(object i){
    System.Threading.Thread.Sleep((int)1000);
}
```

```
void velikiRacun(){
    Task[] t = new Task[5];
    for (int i = 0; i < 5; i++)    {
        t[i] = new Task(velikiRacunModifikovan, i);
        t[i].Start();
    }
    Task.WaitAll(t);
    this.BackColor = Color.Green;
}
```

Klasa *Parallel*

- Omogućava da paralelizuju neke uobičajene programske aktivnosti bez redizajna aplikacije. Ona kreira sopstveni skup Task objekata i ima nekoliko statičkih metoda koje vam stoje na raspolaganju.

- Parallel.For
- Definiše petlju u kojoj se svaka iteracija može izvršavati paralelno u posebnom zadatku. Metoda ima početnu, krajnju vrednost i metodu koja prima celobrojnu vrednost.
- `Parallel.For(0, 4, velikiRacunModifikovan);`

```
Mala izmena u argumentu metode velikiRacunModifikovan

void velikiRacunModifikovan(int i)
{
    System.Threading.Thread.Sleep((int)1000);
}
```

- Parallel.ForEach<T>
- Slično iskazu foreach, definiše petlju u kojoj se sve iteracije mogu paralelno izvršavati. Iskaz prima argument tipa lenumerable<T> i upućuje se na metodu koja prima argument tipa T
- `int[] i = { 0,1,2,3,4 };`
- `Parallel.ForEach<int>(i, velikiRacunModifikovan);`

- Parallel.Invoke
- Izvršavanje skupa metoda bez parametara kao paralelnih zadataka. Na primer:

```
Paralell.Invoke(metod1, metod2, trecametoda);
```

Obustavljanje zadataka

- Neophodno je imati mogućnost prekida dugih zadataka.
- Prekid bi trebao da se završi u nekom od mogućih ili prihvatljivih stanja.
- Klasa Task implementira upotrebu jedne strukture za kontrolisani prekid koja se naziva CancellationToken.
- Ako se namerava obustaviti zadatak, tada se podešava svojstvo IsCancellationRequested na true. Metoda koja se izvršava može da proverava vrednost ovog svojstva u različitim tačkama izvršavanja i da zahteva prekid. Metoda može i da zanemari ovaj zahtev i nastavi sa radom.
- Aplikacija kreira CancellationToken kreirajući objekat CancellationTokenSource, koji sama koristi za kontrolu prekida.

Primer

```
CancellationTokenSource cancelS = new
    CancellationTokenSource();

private void button1_Click(object sender,
EventArgs e)
{
    Action<object> a = velikiRacun;
    Task t = new Task(a, cancelS.Token);
    t.Start();
}

private void button2_Click(object sender,
EventArgs e)
{
    cancelS.Cancel();
}
```

```
void velikiRacun(object t)
{
    CancellationToken token =
(CancellationToken)t;
    System.Threading.Thread.Sleep((int)5000);
    if (token.IsCancellationRequested)
    {
        this.BackColor = Color.Green;
        return;
    }
    System.Threading.Thread.Sleep((int)15000);
    this.BackColor = Color.Green;
}
```

Rad sa više operacija i sinhronizacija rada

- Pogledajmo kod sa bez i sa primenom klase Task:

```
private void button1_Click(object sender, EventArgs
e)
{
    dugotrajnaMetoda1();
    dugotrajnaMetoda2();

    label1.Text = "KRAJ";
}

private void dugotrajnaMetoda1()
{
    System.Threading.Thread.Sleep(5000);
}
private void dugotrajnaMetoda2()
{
    System.Threading.Thread.Sleep(5000);
}
```

```
private void button1_Click(object sender, EventArgs e)
{
    Task t = new Task(dugotrajnaMetoda1);
    t.ContinueWith(dugotrajnaMetoda2);
    t.Start();

    label1.Text = "KRAJ";
}

private void dugotrajnaMetoda1()
{
    System.Threading.Thread.Sleep(5000);
}
private void dugotrajnaMetoda2(Task t)
{
    System.Threading.Thread.Sleep(5000);
}
```

- Napomena: *Za prvi kod poruka se pojavljuje nakon 10sek a za drugi odmah. Za prvi slučaj imamo blokiranje, a u drugom netačno prikazivanje poruke o završetku posla.*

- Koje su opcije?
- 1. Sačekati da se završe zadaci pa onda pokazati poruku

```
private void button1_Click(object sender, EventArgs e)
{
    Task t = new Task(dugotrajnaMetoda1);
    t.ContinueWith(dugotrajnaMetoda2);
    t.Start();
    t.Wait();
    label1.Text = "KRAJ";
}
```

- Ostaje problem blokiranja tekuće niti.

- 2. Dodati novu metodu u red za izvršavanje zadataka koja bi prikazivala poruku.

```
private void button1_Click(object sender, EventArgs e)
{
    Task t = new Task(dugotrajnaMetoda1);
    t.ContinueWith(dugotrajnaMetoda2);
    t.ContinueWith(poruka);
    t.Start();
    t.Wait();
}
private void poruka(Task t)
{
    label1.Text = "KRAJ";
}
```

- Međutim pojavljuje se

Koncept asinhronih operacija

- .NET 4.0 uvodi nov pristup, nazvan asinhroni šablon baziran na zadacima. Pre uvođenja ovog šablona kada su programeri želeli da rešavaju neki problem asinhrono, morali su da definišu redosled izvršavanja. Ovaj šablon im je omogućio da rade sa kodom kao da je sinhron.
- Instanca Task klase ukazuje na vrednost koja će biti prosleđena u određenom trenutku u budućnosti. Ne treba da brinete kako i u kojoj niti će se izračunavati ova vrednost. Treba samo da znate da je neophodno izvesno vreme za određivanje rezultata, a vi možete da koristite Task objekat za sprečavanje blokiranja trenutnog izvršavanja dok čekate rezultate.
- Pomoću Task objekta možete da proveravate status asinhronog izvršavanja, da čekate da se izvršavanje okonča, odnosno da dobijete rezultujuću vrednost. Postoji čak i generička vrednost ove klase Task<TResult> koju možete da koristite prilikom pristupanja rezultatima različiih asinhronih operacija u kojima se primenjuje jaka tipiziacija.

Objašnjenje problema i novo rešenje

- Samo nit korisničkog interfejsa može da upravlja kontrolama interfejsa!
- Rešenje je u korišćenju metode Invoke

```
private void button1_Click(object sender, EventArgs e)
{
    Task t = new Task(dugotrajnaMetoda1);
    t.ContinueWith(dugotrajnaMetoda2);
    t.ContinueWith(poruka);
    t.Start();
}

private void poruka(Task t)
{
    zaUImetode objMetode = new zaUImetode(porukaUI);
    label1.Invoke(objMetode);
}

delegate void zaUImetode();
private void porukaUI()
{
    label1.Text = "KRAJ";
}
```

Da li ovo može biti jednostavnije urađeno?

Primer: Upravljanje progressbar kontrolom iz druge niti

- Sve što čini UI nalazi se u jednoj niti.
- Pristup iz druge niti može izazvati grešku tipa "**Cross-thread operation not valid: Control 'textBox1' accessed from a thread other than the thread it was created on„.**
- Pristup se obavlje pomoću metode Invoke. Toj metodi se prosleđuje objekat delegata koji se unapred priprema. Na primer:

```
delegate void MetodaPromeneIndikatora(object br);
MetodaPromeneIndikatora delPromeniIndikator;
public Form1()
{
    InitializeComponent();
    delPromeniIndikator = new MetodaPromeneIndikatora(promeniIndikator);
}
```

```
public void dugotrajnaOperacija(){
    for(int br=0; br<100; br++){
        System.Threading.Thread.Sleep(500);
        br++;
        this.progressBar1.Invoke(delPromeniIndikator, br);
    }
}
```

```
Action action = new Action(dugotrajnaOperacija);
Task t = new Task(action);
t.Start();
```

```
void promeniIndikator(object br)
{
    progressBar1.Value = (int)br;
}
```

Nove ključne reči

- Nove ključne reči async modifikator i await operator. Pomoću njih prebacujete na programskog prevodioca "fizički posao" pretvaranja sinhronog koda u asinhroni.
- Async koristimo da bi ukazali prevodiocu na to koji blok koda treba da se izvršava asinhrono, a await koristimo za operaciju koja može da se izvršava duže od ostalih da bi se sprečilo blokiranje niti koja se izvršava. Ranije, do pojave ovog modela, čekali smo da se pojavi završni događaj pomoću rukovaoca događajem, nakon cega smo vraćali upravljanje na pozivni metod, operator await upravo to omogućava.
- Ukoliko primenite await u funkciji koja se dugo izvršava u nekom asinhronom metodu, daju se instrukcije programskom prevodiocu da tretira ostatak izvršavanja ovog koda pomoću posebnog rukovaoca za taj zadatak, a zatim da se vrati u pozivni metod. Nakon što se izvrši zadatak, inicira se nastavak izvrsavanja, što omogućava da se nastavi u sinhronom radu.
- Asinhroni metodi ne startuju nove niti, umesto toga oni koriste dobro poznate pozivne metode, ali ste vi pošteđeni od primene celokupnog postupka za njihovo kreiranje. Zapravo programski prevodilac obavlja teži deo posla, tako sto generiše kod koji u potpunosti upravlja tokom izvršavanja. Da bi se obili željeni rezultati očekivana operacija mora da vrati void, Task ili Task<T> instance.

Rešenje 1.

```
private async void button1_Click(object sender,
EventArgs e)
{
    await dugotrajnaMetodaA();
    await dugotrajnaMetodaB();

    label1.Text = "KRAJ";
}

private Task dugotrajnaMetodaA()
{
    Task t = new Task(dugotrajnaMetoda1);
    t.Start();
    return t;
}
private Task dugotrajnaMetodaB()
{
    Task t = new Task(dugotrajnaMetoda2);
    t.Start();
    return t;
}

private void dugotrajnaMetoda1()
{
    System.Threading.Thread.Sleep(5000);
}
private void dugotrajnaMetoda2()
{
    System.Threading.Thread.Sleep(5000);
}
```

Rešenje 2.

```
private async void button1_Click(object sender, EventArgs e)
{
    Task t1 = Task.Run(new Action(dugotrajnaMetoda1));
    Task t2 = Task.Run(new Action(dugotrajnaMetoda2));

    await t1;
    await t2;

    label1.Text = "KRAJ";
}
```

```
private async void button1_Click(object sender, EventArgs e)
{
    Task t1 = new Task(dugotrajnaMetoda1);
    t1.Start();
    Task t2 = new Task(dugotrajnaMetoda2);
    t2.Start();
    await t1;
    await t2;


    label1.Text = "KRAJ";
}
```

```
private async void button1_Click(object sender, EventArgs e)
{
    Task t1 = Task.Run(()=> { System.Threading.Thread.Sleep(5000); });
    Task t2 = Task.Run(() => { System.Threading.Thread.Sleep(5000); });
    await t1;
    await t2;

    label1.Text = "KRAJ";
}
```

Asinhrone metode koje vraćaju vrednost

- Primeri do sada su imali asinhrone metode koje ne vraćaju vrednost, tj. Koriste objekat tipa Task za izvršavanje asinhronih operacija. U ovom slučaju radi se sa tipom: Task<T>
- Neka naša metoda vraća neki rezultat:

```
private void button1_Click(object sender, EventArgs e)
{
    Task<bool> task = new Task<bool>(velikiObracun);
    task.Start();

    MessageBox.Show("Dobijeni rezultat: " + task.Result);
}
```

```
private void button1_Click(object sender, EventArgs e)
{
    Task<bool> task = Task.Run(()=>velikiObracun());
    task.Start();

    MessageBox.Show("Dobijeni rezultat: " + task.Result);
}
```

```
bool velikiObracun()
{
    System.Threading.Thread.Sleep(5000);
    return true;
}
```

- A realizacija asinhrone metode je slična ranije definisanim:

```
private async void button1_Click(object sender, EventArgs e)
{
    Task<bool> task = new Task<bool>(velikiObracun);
    task.Start();
    await task;
    MessageBox.Show("Dobijeni rezultat: " + task.Result);
}
```

Zadatak

- Napisati program za prikaz nivoa tečnosti u rezevoaru. Rezervoar može imati proizvoljan broj ulaznih tokova i proizvoljan broj izlaznih tokova. Svaki tok može biti uključen ili isključen i imati specifičan protok.
- Svaki od tokova treba da nezavisno radi koristeći niti.
- Program treba da ima sledeće mogućnosti:
  - da se definišu tokovi.
  - da prikazuje trenutni nivo tečnosti u rezervoaru
  - pritiskom na dugme se uključuje/isključuje određeni tok