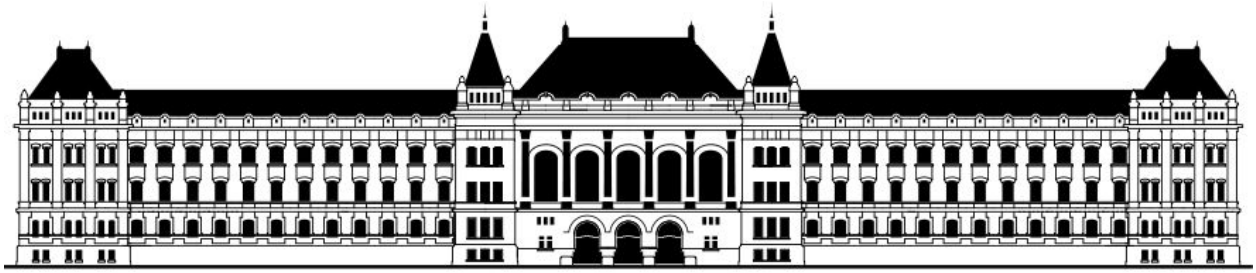
Szálkezelés a C# nyelvben

oktatási segédanyag az Operációsrendszerek tárgyhoz

Méréstechnika és
Információs Rendszerek
Tanszék



Készítette:
Gazder Bence

Tartalomjegyzék

- Bevezető
- Mik azok a szálak és mire jók?
- C# szálak
  - Példa szál létrehozására
  - Példa kimenetének elemzése
  - Mi is történt?
  - További szálindítási lehetőségek
  - Szálak állapotai
  - Példa
  - Szálak lehetséges állapotai
  - Szálak szinkronizálása
  - Thread pool

Bevezető

Ez a segédanyag a BME-VIK Mérnökinformatikus BSc képzésének Operációs rendszerek tantárgyához készült, és a szálkezelést szeretné bemutatni C# környezetben.
A dokumentum alap szintű C# tudást feltételez, tehát a szintaktika és a nyelvi elemek nem lesznek részletesen magyarázva, csak dokumentációk és egyéb információforrások lesznek róluk linkelve.
Minden témában linkek találhatók a példák mellett, akiket mélyebben érdekel a téma további információkat találnak a linkek mögött.

Esetleges hibákat kérlek ide írjátok meg: bencegazder@gmail.com vagy khazy@mit.bme.hu

Mik azok a szálak és mire jók?

Többször szembesülhetünk olyan problémával, hogy az alkalmazásunk több egymástól független dolgot szeretne csinálni és szeretnénk ezeket a feladatokat különválasztani egymástól.

Példák párhuzamosíthatóságra:

- Ray tracer alapú képalkotás, ahol minden pixel színe a másiktól függetlenül meghatározható.
- Egy asztali alkalmazás, ami rendelkezik GUI-val és üzleti logikával. A jobb felhasználó élmény szempontjából előnyös ha a kezelőfelület külön szálon fut, hiszen akkor a háttérszámítások nem akaszthatják meg a az alkalmazást látványosan.

Amennyiben jól terveztük meg a programunkat, akkor mindezek a feladatok egymással párhuzamosan végezhetőek el. Ebben nyújtanak segítséget nekünk a szálak (thread).

Mielőtt továbbhaladnánk, néhány fogalmat vezessünk be, példának pedig a Google Chrome-ot használjuk:

- **Alkalmazás (application):** A futtatható bináris maga. Pl.: chrome.exe
- **Folyamat (process):** A bináris futó példánya. Egy alkalmazáshoz több folyamat is tartozhat. Pl.: Minden böngésző fül egy külön folyamat.
    - Tulajdonságok: Virtuális cpu-n és memórián futnak, nincs tudomásuk a többi folyamatról. PID (process ID) alapján azonosíthatóak.
    - Tulajdonságok megjelenítése Windows feladatkezelőben:

Image Name	PID	CPU	CPU Time	Base Pri	Threads
chrome.exe	1888	00	0:00:03	Normal	9
chrome.exe	2292	00	0:00:04	Normal	11
chrome.exe	3616	01	1:04:12	Normal	69

- **Szál (thread):** A CPU használat alapvető egysége, a szekvenciális kód. Egy folyamathoz több szál is tartozhat (legalább egy). Pl.: Az adott laphoz tartozó adatok betöltését a böngésző több szál segítségével végzi el (lásd fenti példa képet).
    - Tulajdonságok: A folyamat memóriaterületét látja. Saját stackje és program counterje van. Egy folyamat legalább 1 threadből áll. Prioritással rendelkezik, amit az ütemező használ fel az ütemezés gyakoriságának eldöntésére.

Ütemezés: Windows rendszerekben az ütemező szálakat ütemez!
A modern windows rendszerek pre-emptív ütemezőt használnak, tehát az operációs rendszer bizonyos időközönként megszakítja a szál futását, és egy másik szálat kezd el futtatni.

Minden szálhoz tartozik egy prioritás, ami egy 0-31-ig terjedő szám.
Prioritási szintek:
- 0: Zero page thread, memórialapok kinullázására használt kernelszál prioritása
- 1-15: Felhasználó által változtatható prioritás.
- 16-31: Valós idejű “realtime” prioritási szint. Nem garantál igazi valósidejú működést. Adminisztrátor állíthatja csak be.

Minél magasabb prioritású egy szál, annál többször kerül futtatásra egy bizonyos időintervallumban, melyet **quantumnak** nevezünk. Ezt az időintervallumot semmilyen körülmények között sem szabad konstansnak tekinteni!

Miért használjunk szálakat?

A modern operációs rendszerek natívan támogatják a szálak létrehozását és kezelését. Továbbá a mai asztali és mobil processzorok már kivétel nélkül több maggal rendelkeznek, tehát egyszerre több szálat tudnak párhuzamosan futtatni.

Vigyázat!

Túl sok szál létrehozása esetén már nem fog nőni az alkalmazásunk teljesítménye, mivel a szálaknak is van overheadje. Kizárólag abban az esetben használjunk szálakat, amennyiben biztosak vagyunk, hogy igazak az alábbi megállapítások:
- Nem fogja bonyolítani a problémát
- Biztosan párhuzamosítható a feladat

A továbbiakban a szálak használatáról lesz szó C# környezetben, egyszerűbb példákkal illusztrálva.

C# szálak

A C# nyelv a szálak létrehozását nyelvi szinten támogatja, a szálakhoz kapcsolódó funkciókat pedig a System.Threading használatával lehet elérni. (lehet mit csinálni?)

A következő példa a legegyszerűbb szál létrehozást mutatja be.

Példa szál létrehozására

```
using System;
using System.Threading;

namespace OprePelda
{
  public class HivandoObjektum
  {
    public void hivandoFuggveny()
    {
      Console.WriteLine("Ez most ezen a threaden fut:"
        + Thread.CurrentThread.ManagedThreadId);
    }
  }

  public class Program
  {
    private static void Main(string[] args)
    {
      Console.WriteLine("Ez most ezen a threaden fut:"
        + Thread.CurrentThread.ManagedThreadId);

      HivandoObjektum peldanyositottObjektum = new HivandoObjektum();
      Thread szal = new Thread(
        new ThreadStart(peldanyositottObjektum.hivandoFuggveny));
      szal.Start();

      Console.WriteLine("Ez most ezen a threaden fut:"
        + Thread.CurrentThread.ManagedThreadId);
    }
  }
}
```

Kimenet:

Ez most ezen a threaden fut:1
Ez most ezen a threaden fut:1
Ez most ezen a threaden fut:3

Példa kimenetének elemzése

A fenti program két szálon fut. Ezt úgy lehet bizonyítani, hogy a kiírásnál láthajuk, hogy hiába indítottuk el a threadet a `szal.Start()` hívással, a következő konzolra kikerülő sor az 1-es ID-val rendelkező threadből kerül ki, ahonnan az első kiírás is érkezett. Tehát a szál elindítása időt vesz igénybe, és ez alatt az idő alatt az első szál tovább futott.

Mi is történt?

Ahhoz, hogy egy új szálat tudjunk indítani az alábbi lépéseket kell elvégezni:

- Példányosítjuk azt az osztályt, melyet használni szeretnénk
- Létrehozunk egy Thread objektumot, melynek paraméterül adunk egy ThreadStart delegate-et, amelynek megadjuk a példányosított osztály használni kívánt függvényét
- Elindítjuk a szálat s Start() metódussal.

Természetesen ezek a lépések kiegészülhetnek további lépésekkel, hiszen a nyelv rengeteg további lehetőséget biztosít a szálak használatára.

További szálindítási lehetőségek

Az előző példa a legegyszerűbb szálindítást mutatta be. A fenti lépések közül elhagyható a ThreadStart delegate elkészítése, mert amennyiben csak a függvényt adjuk át a Thread objektum konstruktorának, akkor a függvény automatikusan elkészíti a megfelelő delegate-et, tehát a megadott függvény alapján ThreadStart vagy ParameterizedThreadStart delegate-et készít.

Amennyiben a hívandó függvény statikus, akkor az objektum példányosítása is elhagyható, és elég csak `HivandoObjektum.hivandoFuggveny` formában hivatkozni rá.

Függvények kaphatnak paramétereket, nincs ez másként most sem. A szál konstruktorában megadott függvény kétféleképpen kaphatja meg ezeket:

1. A hívandó függvény egy object típusú paramétert vár, amit a Thread.Start(param) metódus paramétttereként kap meg. Az objektum megfelelő típusra kasztolásával lehet a paramétert használni.
   Függvény: `hivandoParameteresFuggveny(object i){ ... }`
   Thread indítás:

```
Thread szalParammal = new
  Thread(peldanyositottObjektum.hivandoParameteresFuggveny);
szalParammal.Start(10);
```

Ennek több hátránya is van:

a. kasztolni kell
b. csak egy paramétere lehet a függvénynek, tehát ha több bemenetet szeretnénk használni, akkor egy listában kell átadni ezeket
c. kényelmetlen

2. Lambda kifejezéssel készítjük el a delegate-et, amit a Thread konstruktorának átadunk. Itt a függvénynek több paramétere is lehet, nem kell kasztolnunk a függvényen belül, hogy az argomentumokat használni tudjuk.
   Függvény: `hivandoParameteresFuggveny(int i){ ... }`
   Thread indítás:

```
Thread szalLambdaParammal = new Thread( () =>
  peldanyositottObjektum.hivandoParameteresFuggveny(10));
szalLambdaParammal.Start();
```

Példa:

```
using System;
using System.Threading;

namespace OprePelda
{
  public class HivandoObjektum
  {
    public static void hivandoFuggveny()
    {
      Console.WriteLine("Nem kaptam paramétert");
    }

    public void hivandoParameteresFuggveny(int i)
    {
      Console.WriteLine("Ezt a paramétert kaptam: " + i);
    }

    public void hivandoParameteresFuggveny(object i)
    {
      Console.WriteLine("Ezt a paramétert Objectként kaptam: " + i);
    }
  }

  public class Program
  {
    private static void Main(string[] args)
    {
      //Statikus metódus hívása
      Thread szalParamNelkul = new Thread(
        HivandoObjektum.hivandoFuggveny);
```

```
      //Valojaban:
      //Thread szalParamNelkul = new Thread(
      //  new ThreadStart(HivandoObjektum.hivandoFuggveny));
      szalParamNelkul.Start();


      //Példányosított objektum paraméteres függvényének hívása
      HivandoObjektum peldanyositottObjektum = new HivandoObjektum();
      Thread szalParammal = new Thread(
        peldanyositottObjektum.hivandoParameteresFuggveny);
      //Valojaban:
      //Thread szalParammal = new Thread(
      //  new ParameterizedThreadStart(peldanyositottObjektum.hivandoParameteresFuggveny));
      szalParammal.Start(20);


      //Paraméterek átadása lambda függvénnyel
      Thread szalLambdaParammal = new Thread(
        () => peldanyositottObjektum.hivandoParameteresFuggveny(10));
      szalLambdaParammal.Start();
    }
  }
}
```

Kimenet:

Nem kaptam paramétert
Ezt a paramétert Objectként kaptam: 20
Ezt a paramétert kaptam: 10

Hivatkozások:

- Thread [osztály]
- ThreadStart [delegate]
- ParameterizedThreadStart [delegate]
- => [lambda operator]

A delegate-ekről röviden: http://stackoverflow.com/questions/1735203/delegates-in-c-sharp

Szálak állapotai

Egy összetett program esetében minden szálnak valamilyen dedikált feladata van. Ezeket a feladatok össze kell hangolni.
A több szál használata esetén elengedhetetlen, hogy meg tudjuk állapítani az egyes szálakról, hogy elvégezték-e már a feladatukat.

Példa

```
using System;
using System.Threading;

namespace OprePelda
{
  public class HivandoObjektum
  {
    public void hivandoFuggveny()
    {
      Console.WriteLine("Futok már futok...");
      Thread.Sleep(1000); //Jó munkához idő kell
    }
  }

  public class Program
  {
    private static void Main(string[] args)
    {
      HivandoObjektum peldanyositottObjektum = new HivandoObjektum();
      Thread szal = new Thread(
        new ThreadStart(peldanyositottObjektum.hivandoFuggveny));
      Console.WriteLine("Az indított szál állapota: " + szal.ThreadState);
      szal.Start();
      Console.WriteLine("Az indított szál állapota: " + szal.ThreadState);
      szal.Join(); //Megvárjuk amíg az indított szál végez
      Console.WriteLine("Az indított szál állapota: " + szal.ThreadState);
    }
  }
}
```

Kimenet:

Az indított szál állapota: Unstarted
Az indított szál állapota: Running
Futok már futok...
Az indított szál állapota: Stopped

Szálak lehetséges állapotai

Állapot	Leírás
Aborted	A szál abortálva lett (Thread.Abort) és a szál már véget ért, de még nem váltott át Stopped állapotba.
AbortRequested	A szálra Thread.Abort hívás érkezett, a szálban egy ThreadAbortException keletkezett, amit elkaphatunk és biztonságosan felszabadíthatjuk a foglalt erőforrásokat. Az abortálást megszakítani a Thread.ResetAbort() hívással tudjuk.
Background	A szál háttérszálként van futtatva, tehát a Thread.IsBackground property igazra van állítva. Ez a gyakorlatban azt jelenti, hogy a szál nem akadályozza a folyamat befejeződését. Egy folyamat akkor ér véget, ha az összes nem háttérszál befejezte a futását.
Running	A szálat elindították a Thread.Start() metódussal, nincs blokkolva és nem abortálták.
Stopped	A szál megállt, már nem lehet újraindítani.
StopRequested	A szál hamarosan meg fog állni.
Suspended	A szál szüneteltetve van.
SuspendRequested	A szálra szüneteltetés lett kérve.
Unstarted	A szál elkészült, de még nem indították el a Thread.Start() metódussal.
WaitSleepJoin	A szál blokkolva van. Addig van ebben az állapotban, amíg az a szál amire meghívták véget nem ér. Esetünkben: szal.join() sornál a fő szál WaitSleepJoin állapotba kerül, amíg a hivandoFuggveny le nem fut.

Hivatkozások:

- ThreadState
- Thread.Join
- Thread abortálásról szóló dokumentáció

Szálak szinkronizálása

Előző példákban már megismerkedhettünk a szálak állapotaival. Sajnos azt gyorsan beláthatjuk, hogy komplex szinkronizálási feladatokra nem fogjuk használni az állapotokat, hanem valamilyen másik eszközhöz kell nyúlnunk.

Mielőtt további példákba kezdenénk bele ismét be kell vezetni néhány fogalmat a szál szinkronizáció témakörében. Az újonnan bevezetett fogalmak: Szemafor (Semaphore), Mutex, lock.
Továbbá meg fogunk ismerkedni C# specifikus megoldásokkal is.

Kritikus szakasz: Egy szekvenciális kódrészlet, melyben kölcsönös kizárást kell biztosítanunk bizonyos erőforrásokra. A kritikus szakaszhoz tartozó műveletet atomi, tehát nem megszakítható műveletként kell végrehajtani.

Például: konzolra kiírást atomi műveletként kell végrehajtani mert ha egyszerre két szál is ír akkor a betűk összekeverednek.

Lock: A közös erőforrás “lezárása” legegyszerűbb megvalósítási, hogy egy lock bitet alkalmazunk aminek értéke 1 ha a közös erőforrást éppen használják (pl.: írnak a konzolra) és 0 ha nem használja senki, viszont amint használni kezdjük 1-es értéket vesz fel.

Példa:

```
public class HivandoObjektum
{
  static object lockObjektum = new Object(); //erre fogunk lockolni
  static int szamlalo = 0;
  public void hivandoFuggveny()
  {
    lock(lockObjektum){
      szamlalo++;
      }
}
```

A számláló növelése atomi művelet, mivel előfordulhat az, hogy két szál egymás után kiolvassa a számot, és ekkor még mindkét kiolvasott érték ugyan az lesz. Mind a két szál megnöveli 1-el az értéket és visszaírja a memóriába a megnövelt értéket.

Tehát például ha a számláló értéke 0-volt akkor két növelés után az elvárt kimenet 2 lenne, de ha nem alkalmazunk lock-ot akkor előfordulhat, hogy 1 lesz.

Szemafor (Semaphore): Egy továbbfejlesztett lock amin két fő művelet értelmezett: belépés és kilépés. Mikor a szemafort létrehozzuk megadhatjuk mennyien tartózkodnak benne jelenleg és mennyi a maximum.

Példa:

```
using System;
using System.Threading;
using System;
using System.Threading;

namespace OprePelda
{
  public class Program
  {
    private static Semaphore szemafor;

    private static void Main(string[] args)
    {
      //Két szál tartózkodhat a szemaforban és jelenleg 2 szabad hely van
      szemafor = new Semaphore(2, 2);

      for (int i = 0; i < 4; ++i) {
        Thread szal = new Thread(() => hivandoFuggveny(i));
        szal.Start();
        Thread.Sleep(400); //várunk egy kicsit hogy a szál elinduljon
      }

    }

    public static void hivandoFuggveny(int szalSzama)
    {
      Console.WriteLine("Thread {0} belépett a függvénybe.", szalSzama);
      szemafor.WaitOne();
      Console.WriteLine("Thread {0} belépett a szemaforba.", szalSzama);
      Thread.Sleep(1000);
      szemafor.Release();
      Console.WriteLine("Thread {0} kilépett a szemaforból.", szalSzama);
    }
  }
}
```

Kimenet:

Thread 0 belépett a függvénybe.
Thread 0 belépett a szemaforba.
Thread 1 belépett a függvénybe.
Thread 1 belépett a szemaforba.

Thread 2 belépett a függvénybe.
Thread 0 kilépett a szemaforból.
Thread 2 belépett a szemaforba.
Thread 3 belépett a függvénybe.
Thread 1 kilépett a szemaforból.
Thread 3 belépett a szemaforba.
Thread 2 kilépett a szemaforból.
Thread 3 kilépett a szemaforból.
Látszik, hogy először a 0 és az 1-es thread belépnek a szemaforba (tehát tényleg 2 hely van) és a 2-es szál várakozik és csak akkor lép be miután a 0-ás szál elhagyta a szemafort.

Mutex: Hasonlóan működik mint a szemafor annyi eltéréssel, hogy a szemafor csak az alkalmazáson belül érvényesül, ezzel ellentétben a mutex folyamatok között is érvényesül.

WaitHandle: Az eseményre várakozó és szinkronizáló osztályok ősosztálya. A statikus metódusait használjuk. Várakozásra és szinkronizálásra a leszármazott osztályokat használjuk.

Például: AutoResetEvent vagy ManualResetEvent

Statikus metódusai:
WaitAll(WaitHandle[]): Addig vár az adott szálon, míg a paraméterként megadott WaitHandle tömb minden eleme Set()-elve nem lesz. Timeout opcionálisan megadható.

WaitAny(WaitHandle[]): Addig vár az adott szálon, míg a paraméterként megadott WaitHandle tömb valamely eleme Set()-elve nem lesz. Timeout opcionálisan megadható.

SignalAndWait(WaitHandle, WaitHandle): Két megadott WaitHandle-t kezel az első paraméterként megadott Handle-t Set()-eli és a második paraméterként megadott Handle-re pedig WaitOne() függvényt hív.

Tehát a SignalAndWait(a,b) és a SignalAndWait(b,a) nem ekvivalens.
A SignalAndWait művelet nem garantáltan atomi művelet!

AutoResetEvent: Használata esetén a Reset() művelet hívása automatikus, tehát amint egy szál belép az általa védett területre, akkor garantált hogy nem lép be egy másik ugyan arra a kódrészletre.

ManualResetEvent: Használata esetén amint egy szál belép az általa védett kódrészletre, akkor meg kell hívnia a Reset() függvényt hogy több szál ne léphessen be a ManualResetEvent-el védett kódrészletre.

AutoResetEvent és ManualResetEvent közötti különbség gyorsan elmagyarázva:

“A ManualResetEvent egy ajtó amit manuálisan kell becsuknunk (reset) míg az AutoResetEvent egy sorompós jegyszedő állomás ahova egyszerre csak egy autó mehet be “ - Dan Goldstein [forrás]

A sorompó “felengedése” a Set() a “leengedése” pedig a Reset().

Példa:

A példában a két hivandoFuggveny ugyan az, azzal a különbséggel hogy az egyikük AutoResetEvent-et, a másik pedig ManualResetEventet használ!

Létrehozunk 3-3 szálat mind a két esetnek és megfigyeljük, hogy milyen sorrendbenvégzik ela feladataikat.

```
using System;
using System.Threading;

namespace OprePelda
{
  public class Program
  {
    public static AutoResetEvent autoResetEvent;
    public static ManualResetEvent manualResetEvent;

    private static void Main(string[] args)
    {
      Thread[] autoResetThreadek = new Thread[3];
      autoResetEvent = new AutoResetEvent(true); //először az AutoResetEvent-et
teszteljük!

      for (int i = 0; i < 3; i++ ){
        autoResetThreadek[i] = new Thread(() => hivandoFuggvenyAuto());
        autoResetThreadek[i].Name = "Auto" + i;
      }
      Thread.Sleep(1000);
      for (int i = 0; i < 3; i++){
        autoResetThreadek[i].Start();
      }
      for (int i = 0; i < 3; i++){
        autoResetThreadek[i].Join();
      }

      Console.WriteLine();

      Thread[] manualResetThreadek = new Thread[3];
      manualResetEvent = new ManualResetEvent(true);
```

```
      //Az ManualResetEvent szálai
      for (int i = 0; i < 3; i++)
      {
        autoResetThreadek[i] = new Thread(() => hivandoFuggvenyManual());
        autoResetThreadek[i].Name = "Manual" + i;
      }
      Thread.Sleep(1000);
      for (int i = 0; i < 3; i++)
      {
        autoResetThreadek[i].Start();
      }
      for (int i = 0; i < 3; i++)
      {
        autoResetThreadek[i].Join();
      }

    }

    public static void hivandoFuggvenyAuto() {
      Console.WriteLine("{0} szál belépett. VÁR", Thread.CurrentThread.Name);
      autoResetEvent.WaitOne();
      Console.WriteLine("{0} szál belépett. FUT", Thread.CurrentThread.Name);
      Thread.Sleep(400);
      Console.WriteLine("{0} szál belépett. KILÉPETT", Thread.CurrentThread.Name);
      autoResetEvent.Set();
    }

    public static void hivandoFuggvenyManual()
    {
      Console.WriteLine("{0} szál belépett. VÁR", Thread.CurrentThread.Name);
      manualResetEvent.WaitOne();
      Console.WriteLine("{0} szál belépett. FUT", Thread.CurrentThread.Name);
      Thread.Sleep(400);
      Console.WriteLine("{0} szál belépett. KILÉPETT", Thread.CurrentThread.Name);
      manualResetEvent.Set();
    }
  }
}
```

Kimenet:

Auto0 szál belépett. VÁR
Auto0 szál belépett. FUT
Auto2 szál belépett. VÁR
Auto1 szál belépett. VÁR
Auto0 szál belépett. KILÉPETT
Auto1 szál belépett. FUT
Auto1 szál belépett. KILÉPETT
Auto2 szál belépett. FUT
Auto2 szál belépett. KILÉPETT

Manual0 szál belépett. VÁR
Manual0 szál belépett. FUT
Manual2 szál belépett. VÁR
Manual2 szál belépett. FUT
Manual1 szál belépett. VÁR
Manual1 szál belépett. FUT
Manual2 szál belépett. KILÉPETT
Manual0 szál belépett. KILÉPETT
Manual1 szál belépett. KILÉPETT

A kimenetből látszik, hogy az AutoResetEvent esetén mindig egy szál tartózkodik a védett szekcióban, míg a ManualResetEvent esetén az event nem“csukja vissza maga után a sorompót” és azért egyszerre mind a 3 szál beléphet a védett szekcióba.

Amenyiben a második (manual) esetben a WaitOne után Reset()-et hívánk (lockolva a két utasítást) akkor megegyező kimenetet kapnánk.

Hivatkozások:

- Semaphore [osztály]
- Lock [keyword]
- WaitHandle [osztály]
- AutoResetEvent [osztály]
- ManualResetEvent [osztály]
- Versenyhelyzetekről bővebben

Thread pool

Korábban már olvashattuk, hogy a szálak létrehozása hiába gyorsabb, mint a folyamatoké még mindig óvatosan kell bánni mennyiségükkel.

Képzeljünk el egy webserver-t. Maximum 20 felhasználó szokta használni a hosztolt szolgáltatást, és hogy a kérések ne okozhassanak fennakadást, minden kérést külön szálon kezelünk. Korábban láthattuk, hogy a szálak létrehozása is erőforrásokat igényel, és a szálak nagy mennyiségben való kezelése bonyolult kódhoz vezet, melynek továbbfejlesztésiköltségei magasak. Az ideális az lenne, ha minden kérés külön szálon futna, de meg tudnánk spórolni a szál létrehozási költségeket.

A ThreadPool osztály erre nyújt megoldást. Az osztály előre elkészíti az általunk megadott mennyiségű szálat, és ezeknek a szálaknak adhatunk feladatokat. Megadhatunk egy minimum és egy maximum értéket a szálak mennyiségére.

A kiosztott feladatokat az osztály ütemezi, és osztja el a szálak között, megspórolva nekünk a szálak menedzselésének feladatát.

Példa:

```
using System;
using System.Threading;

namespace OprePelda
{
  public class HivandoObjektum
  {
    public static int threadCount = 0;
    public static object lockObjektum = new object();
    ManualResetEvent resetEvent;

    public HivandoObjektum(ManualResetEvent resetEvent_) {
      resetEvent = resetEvent_;
    }

    public void hivandoFuggveny(object i)
    {
      //a kiírás helyességéhez lockolni kell mert több szál írhatja egyszerre
      lock (lockObjektum) {
        threadCount++;
        Console.WriteLine("START Szama: {0}   Össz munka: {1}",
          (int)i, threadCount);
      }
      Thread.Sleep(1000); //jó munkához idő kell
      lock (lockObjektum) {
        threadCount--;
        Console.WriteLine("STOP   Szama: {0}   Össz munka: {1}",
          (int)i, threadCount);
      }
      resetEvent.Set(); //Munka készre állítása
    }
  }

  public class Program
  {
    private static void Main(string[] args)
    {
      const int munkakSzama = 9;
      ManualResetEvent[] keszEsemenyek = new ManualResetEvent[munkakSzama];

      for (int i = 0; i < munkakSzama; i++) {
        keszEsemenyek[i] = new ManualResetEvent(false);
        HivandoObjektum munkaObjektum = new HivandoObjektum(keszEsemenyek[i]);
        ThreadPool.QueueUserWorkItem(munkaObjektum.hivandoFuggveny, i);
      }
```

```
      //A threadpoolban lévő szálak háttérszálak tehát meg kell várnunk hogy az
      //összes lefusson
      WaitHandle.WaitAll(keszEsemenyek);
    }
  }
}
```

Kimenet:

START Szama: 0 Össz munka: 1
START Szama: 3 Össz munka: 2
START Szama: 6 Össz munka: 3
START Szama: 2 Össz munka: 4
START Szama: 4 Össz munka: 5
START Szama: 5 Össz munka: 6
START Szama: 1 Össz munka: 7
START Szama: 7 Össz munka: 8
STOP Szama: 0 Össz munka: 7
START Szama: 8 Össz munka: 8
STOP Szama: 4 Össz munka: 7
STOP Szama: 3 Össz munka: 6
STOP Szama: 1 Össz munka: 5
STOP Szama: 5 Össz munka: 4
STOP Szama: 6 Össz munka: 3
STOP Szama: 7 Össz munka: 2
STOP Szama: 2 Össz munka: 1
STOP Szama: 8 Össz munka: 0

A példából látszik, hogy a szálak ilyen módon kezelése rendkívül egyszerű. A legtöbb kód a kiírás és a hozzá tartozó segédváltozók kezelése miatt kellett megírni.

Vigyázat!

A kezelt szálak háttérszálak, tehát ha a nem háttérszálak véget értek, a ThreadPoolban lévő szálak nem fogják az alkalmazás bezárását megakadályozni.

A minimumszálak mennyiségét tilos magas értékre beállítani, mert amennyiben ezek nincsenek kihasználva, akkor a szálak csak az erőforrásokat foglalják.

Hivatkozások:

- ThreadPool osztály
  - Szálak maximum és minimum mennyiségének beállítása