

Szoftvertechnikák

Konkurens alkalmazások fejlesztése

Benedek Zoltán



Automatizálási és
Alkalmazott
Informatikai Tanszék

Ez az oktatási segédanyag a Budapesti Műszaki és Gazdaságtudományi Egyetem oktatója által kidolgozott szerzői mű. Kifejezetten felhasználási engedély nélküli felhasználása szerzői jogi jogosertésnek minősül

Tartalom

Alapok

- > Folyamat (Process)
- > Szál (Thread)

Szálkezelés

- > Szálak indítása
- > Szálak leállítása
- > Egyebek

Szálak/folyamatok szinkronizálása

- > Problémafelvetés
- > Kölcsönös kizárás
- > Jelzés

Tartalom

Folyamat (process)

Többszálú alkalmazások



A szálkezelés alapjai .NET felügyelt környezetben

Szálak szinkronizációja

Mire kell odafigyelni a szálak kommunikációja során?

Többszálú WinUI alkalmazások

Thread-pool

Egyéb szálkezeléshez kapcsolódó technikák

Holppont (deadlock)

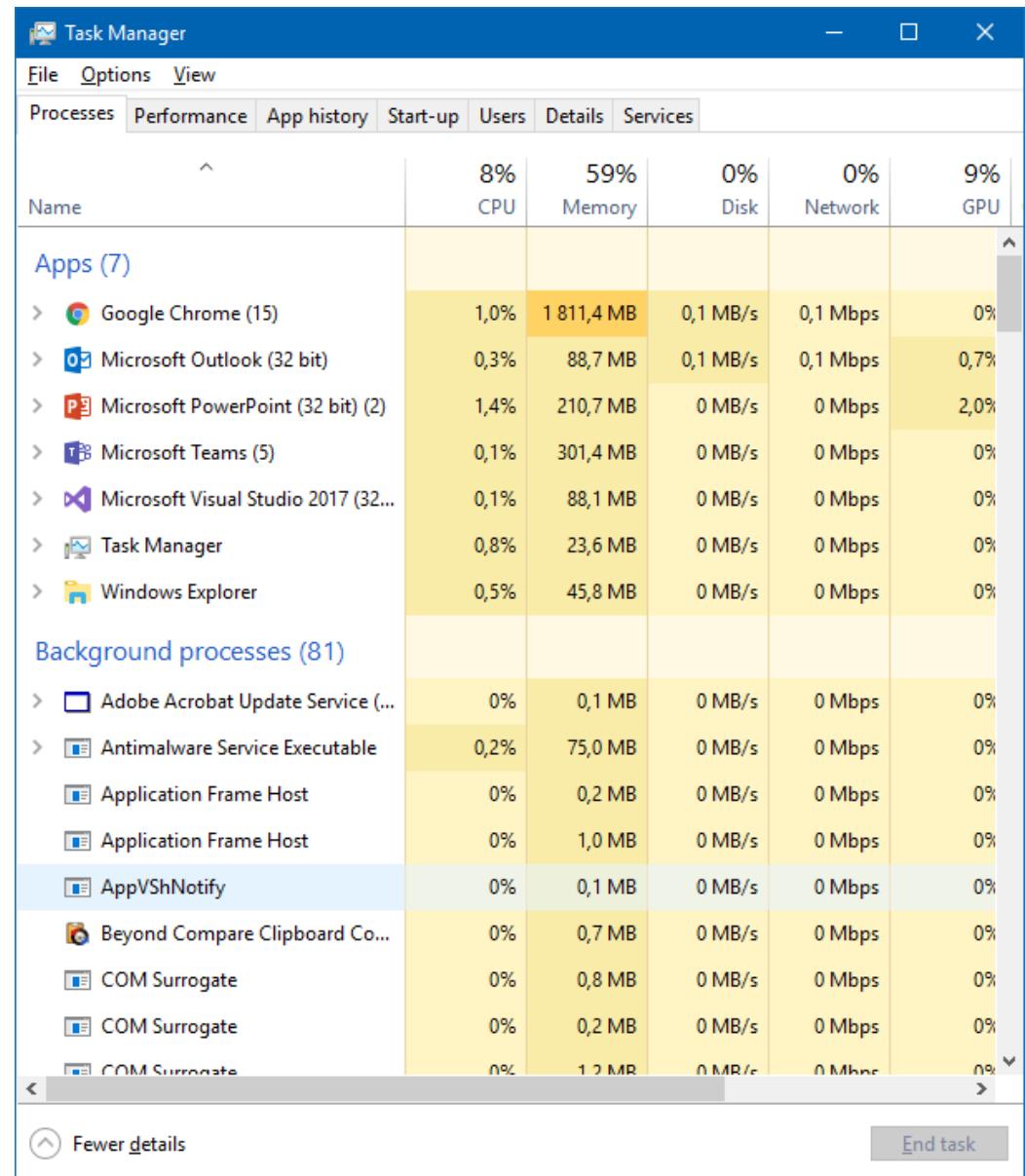
ÖSSZEFOGALÁS

Thread safe interface minta

Folyamat (process)

Folyamat (process, processz)

- Alkalmazás (program) != folyamat
 - > A folyamat a program egy betöltött példánya
- Mi tartozik a folyamathoz
 - > Egy saját memória címtartomány (private address space)
 - > Rendszererőforrások
 - > Legalább egy szál (thread) fut
- Minek az egysége a folyamat?



The screenshot shows the Windows Task Manager window with the 'Processes' tab selected. The table displays various running processes across five columns: Name, CPU, Memory, Disk, Network, and GPU. The 'CPU' column is currently sorted in descending order.

Name	CPU	Memory	Disk	Network	GPU
Apps (7)					
Google Chrome (15)	1,0%	1 811,4 MB	0,1 MB/s	0,1 Mbps	0%
Microsoft Outlook (32 bit)	0,3%	88,7 MB	0,1 MB/s	0,1 Mbps	0,7%
Microsoft PowerPoint (32 bit) (2)	1,4%	210,7 MB	0 MB/s	0 Mbps	2,0%
Microsoft Teams (5)	0,1%	301,4 MB	0 MB/s	0 Mbps	0%
Microsoft Visual Studio 2017 (32...	0,1%	88,1 MB	0 MB/s	0 Mbps	0%
Task Manager	0,8%	23,6 MB	0 MB/s	0 Mbps	0%
Windows Explorer	0,5%	45,8 MB	0 MB/s	0 Mbps	0%
Background processes (81)					
Adobe Acrobat Update Service (...)	0%	0,1 MB	0 MB/s	0 Mbps	0%
Antimalware Service Executable	0,2%	75,0 MB	0 MB/s	0 Mbps	0%
Application Frame Host	0%	0,2 MB	0 MB/s	0 Mbps	0%
Application Frame Host	0%	1,0 MB	0 MB/s	0 Mbps	0%
AppVShNotify	0%	0,1 MB	0 MB/s	0 Mbps	0%
Beyond Compare Clipboard Co...	0%	0,7 MB	0 MB/s	0 Mbps	0%
COM Surrogate	0%	0,8 MB	0 MB/s	0 Mbps	0%
COM Surrogate	0%	0,2 MB	0 MB/s	0 Mbps	0%
COM Surrogate	0%	1,2 MB	0 MB/s	0 Mbps	0%

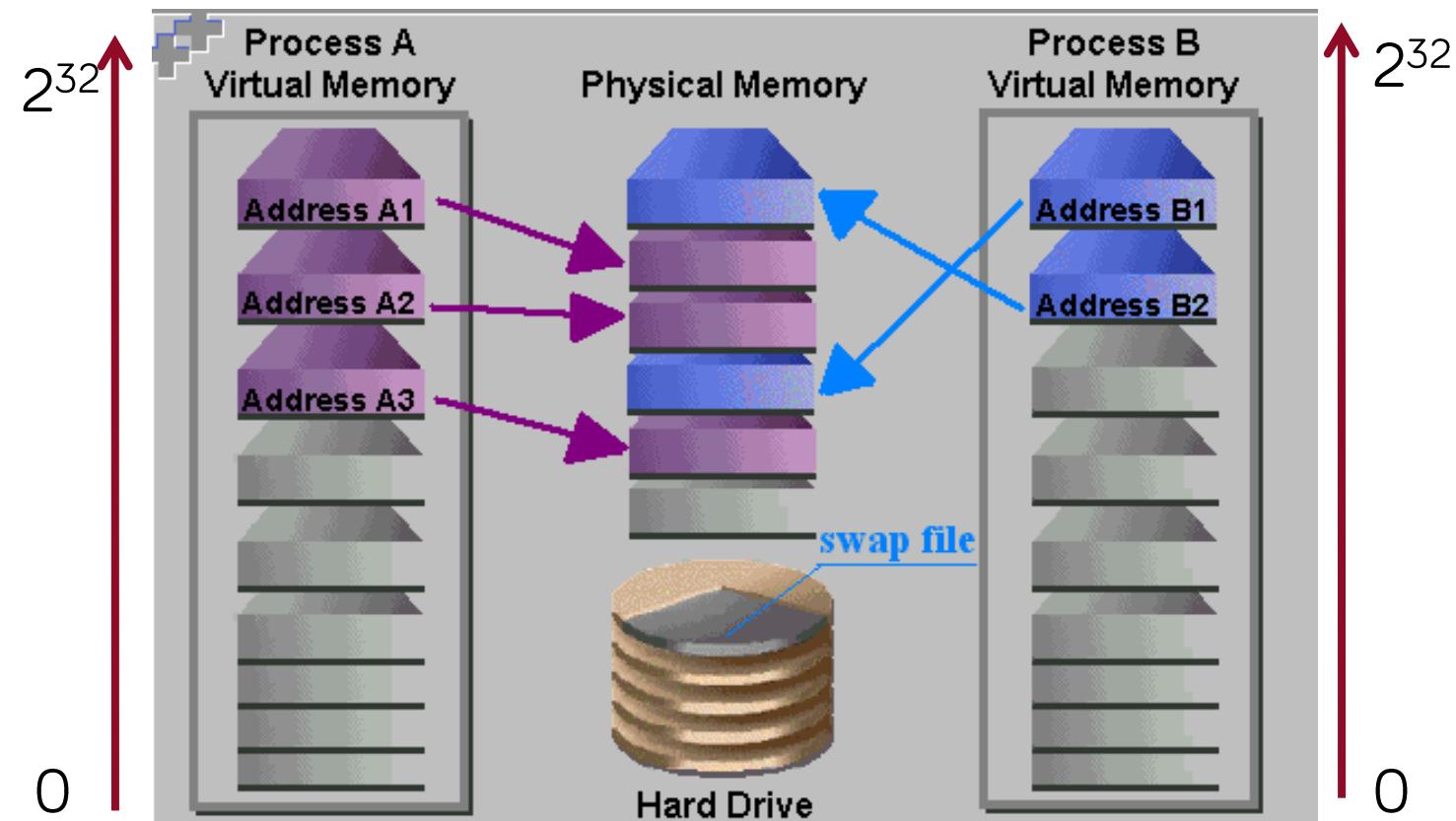
Folyamat

- A folyamat alapvetően **VÉDELMI** egység!
 - > A futási/ütemezési egység a szál
- A folyamatok egymástól **elszigeteltek**
 - > Védettek egymástól (ha az egyik rossz címre ír, nem ronthatja el a másik memóriáját)
 - > Az OS is védett a felhasználói folyamatoktól
 - > Egy folyamat (kényszerített) megszüntetésének nincs hatása a többire/OS-re
- Mi védett? A memória!
 - > minden **folyamat** saját kb. 2 GB virtuális címtartományt kap (32 bites OS esetén)
 - A teljes címtartomány egy 32 bites OS alatt 2^{32} (4 GB), de az OS kb. 2 GB-ot fenntart magának
 - Ma már persze az OS általában 64 bites, 2^{64} a címtartomány, de ezt a nagyságrendet fejben már nehéz elképzelni
 - > A virtuális címeket a memóriamenedzser fizikai címekre képezi le (RAM)

Memóriakezelés

- A leképezés során a fizikai memóriában más fizikai címekre képződnek le a folyamatok virtuális címei -> ez biztosítja a védelmet

32 bites OS példa



Memóriakezelés

- 32 bites OS alatt minden folyamat kb. 2 GB virtuális címtartománnyal rendelkezik
- Nem áll automatikusan rendelkezésre: **használat előtt foglalni kell!**
 - > C/C++ globális/statikus változók – A folyamat indulásakor allokálódik hely
 - > Lokális változók – A stacken allokálódik hely
 - > malloc, new – Dinamikus allokálásra
- A színpad mögött az **OS szolgáltatja a memóriát, a .NET ezzel gondolkodik.**
- Ha érvénytelen címre hivatkozunk:
 - > „Access violation at address ...” üzenet

Ez értelmetlen, nem lehet „csak úgy” egy tetszőleges címre írni (foglalni kellene):

```
int* p = 0xfa2343c2;  
*p = 21;
```

Folyamatok „programozása”

- .NET-ben
 - > [System.Diagnostics.Process](#) osztály
 - Start() – elindít egy folyamatot
 - Kill() – terminálja a folyamatot
 - Process.GetCurrentProcess() – aktuális process elérése
 - > Mit csinál a következő kódrészlet?

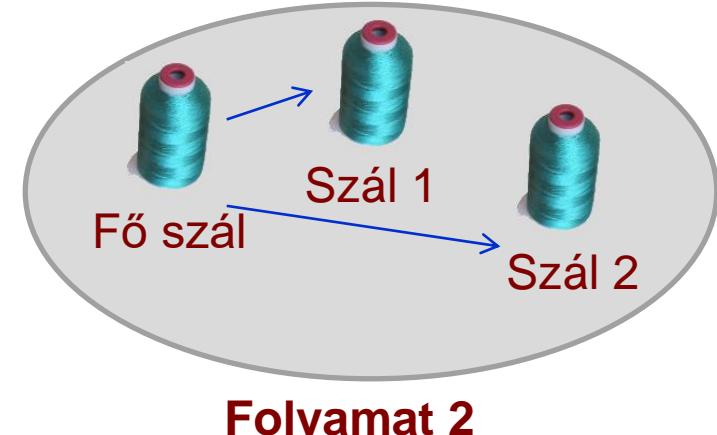
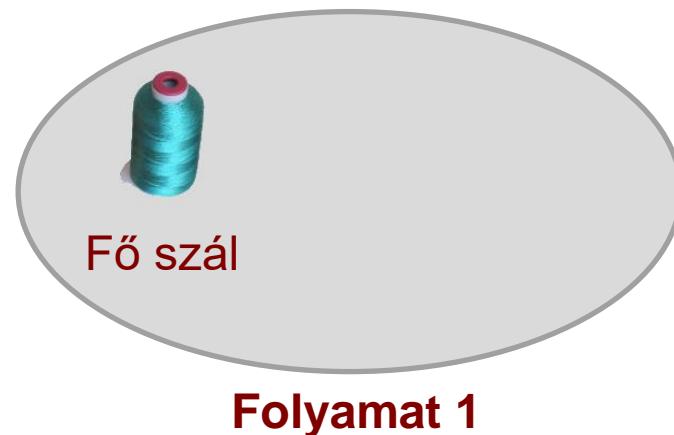
```
Process.GetCurrentProcess().Kill();
```



Többszálú alkalmazások

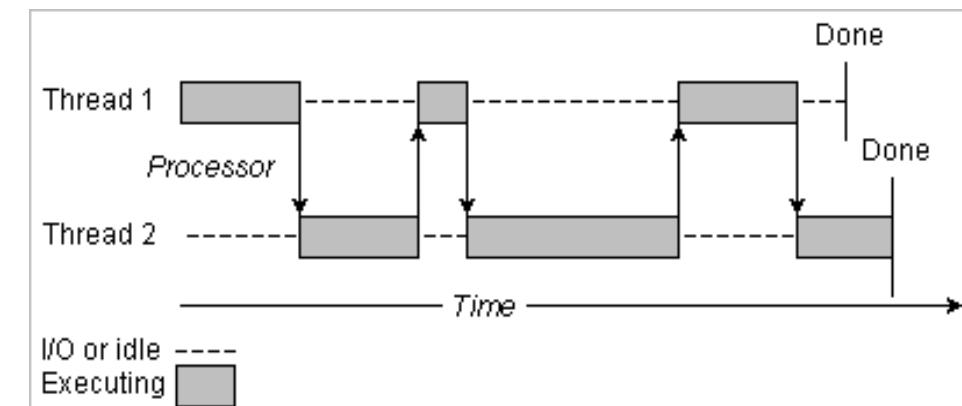
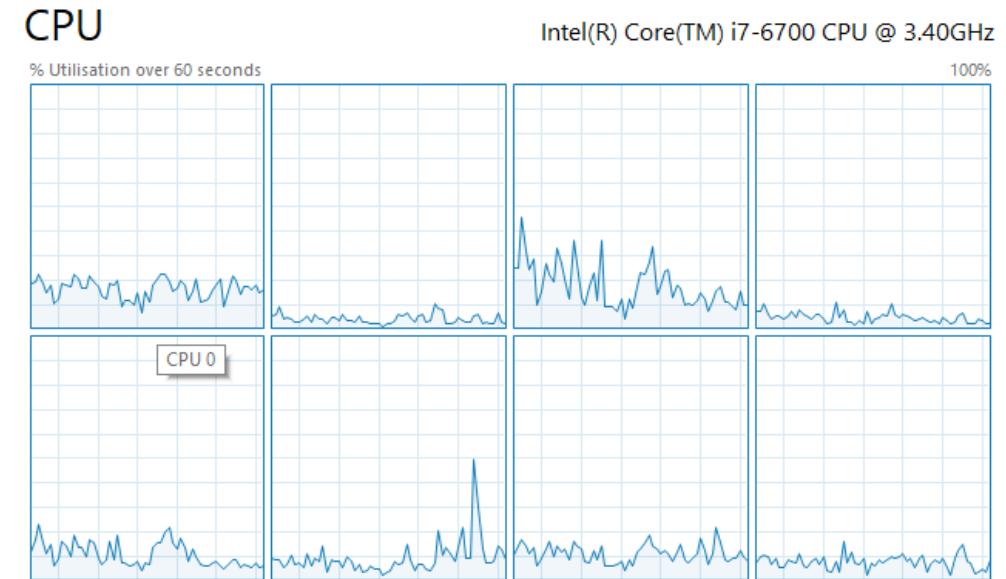
A szál fogalma

- A szál (thread) a futási/ütemezési egység
- Az eddigi alkalmazásaink **egyszálúak** voltak
 - > A folyamat elindulásakor létrejött a folyamat **fő szála** (main thread)
 - > Ebből indíthatunk újabb szálakat, ekkor az alkalmazásunk **többszálú** (multithreaded) lesz.
 - > A szál egy folyamaton belüli feladatként fogható fel



Szálak ütemezése

- Az OS a szálakat ütemezi
 - > Kiválaszt egyet a futásra kész szálak közül, és a processzort hozzárendeli
 - > Egy CPU-n egyidőben egy szál futhat (a processzormagok száma számít)
- Ütemezési típusok
 - > Nem preemptív: egy szál csak akkor kaphat futási jogot, ha más szál önként lemond futási jogáról. Ilyen pl. a Windows 3.1. Kiéheztetés!
 - > Preemptív: az ütemező egy idő után elveszi a futási jogot a száltól, ha nagyobb prioritású szál futásra kész, vagy lejárt a szál időszelete ($n \times 10$ millisec). A legtöbb modern OS ilyen. Egy CPU esetén is látszólag párhuzamosan futnak a szálak.
- Szálak állapota
 - > Futó
 - > Felfüggesztett
 - > I/O műveletre vár
- Szálak prioritása
 - > Visszatérünk ...



Többszálú alkalmazások előnyei

Demo

- **1. Átlagosan jobb CPU kihasználás elérése**

- > Amíg egy szál IO műveletre vár
 - (pl. adat olvasása diszkről), addig más szál futhat
- > Több CPU (CPU mag) kihasználása egy alkalmazáson belül
 - Egy szál csak egy CPU-n tud futni. Ha egy alkalmazáson belül szeretnénk több CPU magot is kihasználni, az csak akkor lehetséges, ha az alkalmazás több szálat indít.
 - Pl. egy adathalmaz feldolgozását több szálra bízzuk, mindenki szál egy adott rész feldolgozásáért felel.

Többszálú alkalmazások előnyei

Demo

- **2. Hosszú blokkoló művelet GUI alkalmazásokban**
 - > Ne futtassuk a fő szálban, mert a GUI befagy
 - > A művelet háttérszálban futtatásával a főszál reaktív marad. A főszál feladata ez esetben a felhasználói események kezelése.
 - > Megjegyzés: ezt ma már Task-okkal és async/await-tel elegánsabban is meg tudjuk oldani, félév végén lesz ezekről szó.

Többszálú alkalmazások előnyei

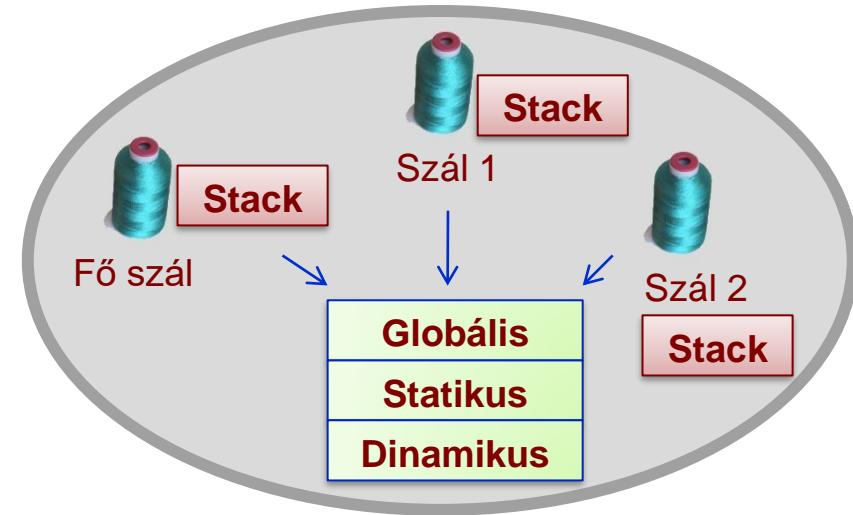
- **3. Időzítésérzékeny feladatok**
 - > Pl. folyamatvezérlés külön, nagyprioritású szálban futtatása.
- **4. Kiszolgáló (szerver) alkalmazások
(pl. webszerver) esetében kisebb átlagos válaszidő**
 - > Ha a kiszolgáló alkalmazás egyszálú, akkor egy olyan kliens kiszolgálása, mely sokáig tart, hosszú ideig feltartja az ezzel egyidőben beérkező, de elvileg gyorsan kiszolgálható kliensek kéréseit (mert a kérések kiszolgálása az egyszálúság miatt sorban történik).
 - > Ha a kiszolgáló többszálú (az egyes klienseket más-más szál szolgálja ki), az időszeletes ütemezés miatt még egy CPU mag esetén sem áll fent az előző pontban vázolt „kiéheztetés” esete.

Szál ↔ Folyamat

- A szál sokkal kisebb erőforrásigényű, mint a folyamat
 - > Különösen Windows környezetben, Linux/Unix alatt ez kevésbé igaz
- A **folyamatok** elszigeteltek: a folyamatok közötti kommunikáció nehéz
- A **szálak** egy adott folyamaton belül egy címtartományban vannak, így „könnyen” kommunikálnak
 - > A C++ a globális, statikus, dinamikus változók közösek
 - > **Minden szál saját stackkel rendelkezik**
 - A szál lokális változóit csak az adott szál látja



Folyamat 1



Folyamat 2

Szinkron/aszinkron végrehajtás

- Definíciók
 - > Szinkron végrehajtás: a hívó a művelet befejezéséig várakozik (blokkolt)
 - > Aszinkron végrehajtás: a hívó a művelet befejezését nem várja meg. A rendszer a hívót valamilyen módon értesíti a művelet befejezéséről (eredmény)
- Aszinkron végrehajtás hogyan valósítható meg?
 - > I/O műveletek (fájlkezelés, hálózatkezelés stb.) esetén **aszinkron I/O végrehajtással** (ha az OS támogatja). Az OS belső megszakításkezelésére épül, nem igényel külön szálat.
 - > Megoldható a **művelet külön szálban futtatásával**. Ez költségesebb, külön szálat igényel, de nem csak I/O esetén használható.

A szálkezelés alapjai .NET felügyelt környezetben

Szál indítása

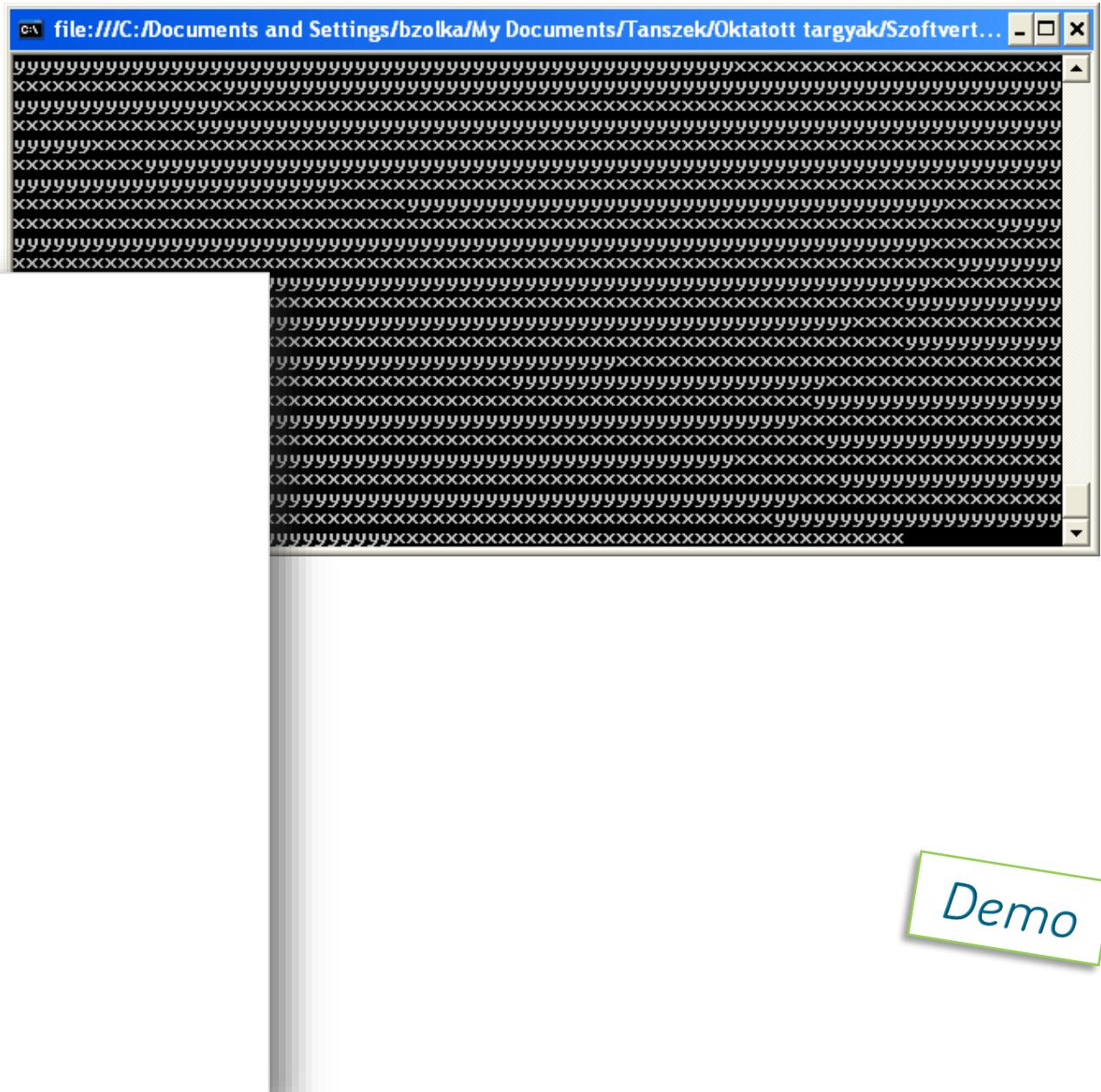
- System.Threading névtér,
- Thread osztály
- Indítsunk szálat →

Szál indítása

- A WriteY az ún. szálfüggvény.

```
class Program
{
    static void Main(string[] args)
    {
        Thread t = new Thread(WriteY);
        t.Start();
        while (true)
            Console.Write("x");
    }

    static void WriteY()
    {
        while (true)
            Console.Write("y");
    }
}
```



Demo

Szál indítása

- A Thread konstruktor **ThreadStart** típusú delegate-et vár
 - > Ez void visszatérésű **paraméter nélküli** delegate

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

- A Start() művelet indítja a szálat
 - > Hívásáig a szálat az OS nem ütemezi
- Ha kilép a szálfüggvény: a szál befejezi futását
- **Egy object típusú paraméter** is átadható a szálfüggvénynek
 - > A Thread osztály **ParameterizedThreadStart** delegate paraméterezősű konstruktora támogatja ezt →

Paraméterezett szálindítás

- A Start()-nak kell megadni a szálfüggvény paraméterét

```
class Program
{
    static void Main(string[] args)
    {
        Thread t = new Thread(WriteAny);
        t.Start("Z"); // paraméter átadása a szálfüggvénynek
        while (true)
            Console.Write("X");
    }
    static void WriteAny(object param) // "Z"-t kap
    {
        while (true)
            Console.Write(param);
    }
}
```

Adat átadása szálnak objektum metódusreferenciával

- Objektum (nem statikus) metódusreferencia átadásával

```
class ThreadClass
{
    private string text;

    public ThreadClass(string text)
    {
        this.text = text;
    }

    public void WriteAny()
    {
        while (true)
            Console.WriteLine(text);
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        ThreadClass ts = new ThreadClass("Z");
        Thread t = new Thread(ts.WriteAny);
        t.Start();
        while (true)
            Console.WriteLine("X");
    }
}
```

A ts objektumhoz tartozó WriteAny a szálfüggvény. Így a WriteAny eléri a ts objektum állapotát, pl. a text tagváltozót.

Előtér- és háttérszálak

Demo

- A létrehozott szál alapértelmezésben előtér szál
- Egy processz csak akkor lép ki, ha minden előtér szál befejezte a futását.
- **Háttérszál** indítása példa

Ha a példában beállítjuk az `IsBackground` tulajdonságot `true`-ra, a program azonnal kilép, amint a `Main` függvény kilép (mert a `Main` az alkalmazás főszála, mely előtér szál, és ez volt az utolsó futó előtér szál).

```
static void Main(string[] args)
{
    Thread t = new Thread(ThreadFunc);
    t.Start();
    t.IsBackground = true;
    Console.WriteLine("Én vagyok a főszál és végeztem...");
}

static void ThreadFunc()
{
    for (int i = 0; i < 5; i++)
    {
        Thread.Sleep(1000);
        Console.WriteLine("Én vagyok a háttérszál: " + i);
    }
}
```

Néhány hasznos művelet/tulajdonság

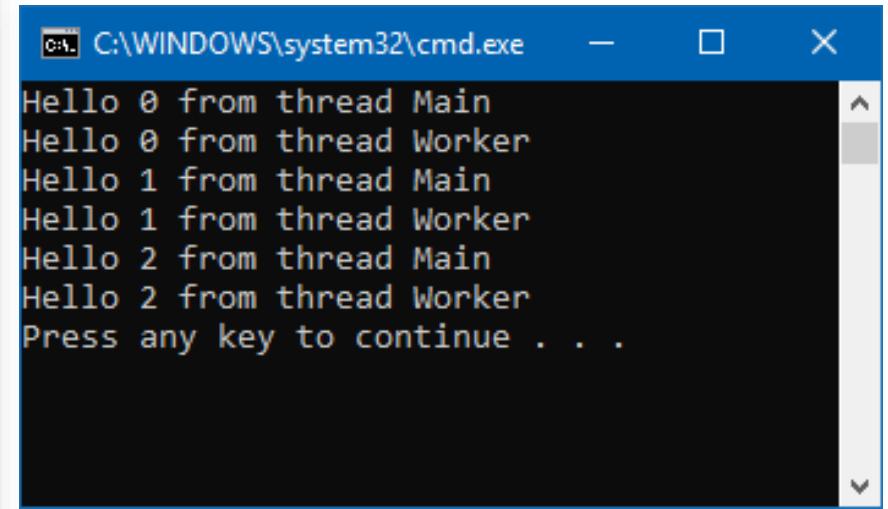
- Thread.CurrentThread statikus property
 - > Aktuális szálat adja vissza
- Name tulajdonság
 - > Név adható a szálnak (nyomkövetést segíti)
- Thread.Sleep(int millisec) és Thread.Sleep(TimeSpan ts) statikus művelet
 - > Elaltatja a hívó szálat
 - > A Sleep passzívan várakozik (nem használ CPU időt)

Példa

Demo

```
class Program
{
    static void Main()
    {
        Thread.CurrentThread.Name = "Main";
        Thread worker = new Thread(SayHello);
        worker.Name = "Worker";
        worker.Start();
        SayHello();
    }

    static void SayHello()
    {
        for (int i = 0; i < 3; i++)
        {
            Console.WriteLine("Hello {0} from thread {1}",
                i, Thread.CurrentThread.Name);
            Thread.Sleep(1000);
        }
    }
}
```

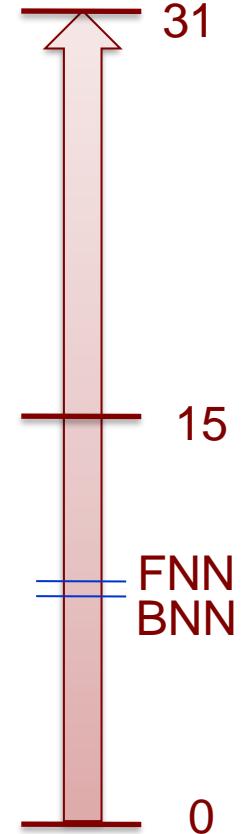


A `SayHello` függvényt két szál is hívja: a fő szál és a „worker” szál. A `SayHello` függvényben a `Thread.CurrentThread.Name` más lesz a két esetben, mert más nevet adtunk a két szálnak.

Szálak prioritása

- Az eredő prioritást (0..31 közötti szám) a folyamat prioritási osztálya és a szál prioritása együttesen határozza meg
- **Folyamatprioritás**
 - > enum ProcessPriorityClass { Idle, BelowNormal, **Normal**, AboveNormal, High, RealTime}
- **Szálprioritás**
 - > enum ThreadPriority { Lowest, BelowNormal, **Normal**, AboveNormal, Highest }
- Az alapértelmezett minden kettőre a Normal
- OS függő a pontos működés
- GUI alkalmazásoknál az aktív ablak szála kap(hat) egy kis boostot

```
Process.GetCurrentProcess().PriorityClass = ProcessPriorityClass.High;  
Thread.CurrentThread.Priority = ThreadPriority.Highest;
```



Szálak ütemezése, realtime kitérő – nem csak .NET*

- RealTime folyamatprioritás
 - > Ezzel arra kérjük az OS-t, hogy soha ne vegye el a futási jogot.
 - > Veszélyes játék, kizárhatjuk vele az OS-t!
 - Ha végtelen ciklusba kerülünk, csak a reset gomb segít!
 - Ha nem adunk futást az OS-nek, „befagy” az egér, a harddisk nem tud írni stb.
 - > Az OS egy alacsonyabb szál prioritását megnövelheti, ha egy nagyobb prioritású szál vár az eredményére.
- Ha „garantált” válaszidő szükséges
 - Pl. folyamatvezérlés, szimuláció, hangátvitel (Skype), multimédia lejátszás stb.: célszerű különválasztani a következőket
 - > Folyamatvezérlés: nagy /realtime prioritású szálba/folyamatba
 - > GUI megjelenítés: kisebb prioritású szálba/folyamatba
 - A GUI újrafestése így nem használja el a CPU erőforrás nagy részét
 - Ha a beavatkozás kritikus (pl. nyomógombra leállítás), akkor ez nem tehető meg, vagy tegyük a beavatkozást lehetővé tevő felületeket egy harmadik nagyprioritású szálba/folyamatba.

„Realtime” kitérő *

- Valósidejű (realtime) operációs rendszerek
 - > Hard realtime
 - Garantált válaszidő/végrehajtásidő adott t időn belül. Nem feltétlenül jelenti azt, hogy gyorsan, t lehet „nagy”!
 - Pl. ha emberélet függhet rajta
 - > Soft realtime
 - A válaszidő nagy valószínűséggel biztosított
- Megjegyzések
 - > A szemétgyűjtést alkalmazó környezetek általánosságában kevésbé determinisztikusak

Kivételkezelés

- Ha kezeletlen kivétel van bármilyen szálban → az alkalmazás kilép!
- A kivételt a szálban kell kezelní, az indító nem kapja meg (kivétel az async delegate)!

```
public static void Main()
{
    // Eköré hiába tennénk try-catch blokkot, az csak a szálindítás
    // esetleges kivételét kapná el, a szál futásához tartozókat,
    // vagyis a szálfüggvény által dobottakat NEM.
    new Thread(ThreadFunc).Start();
}

static void ThreadFunc()
{
    try { throw new Exception("Test error"); }
    catch (Exception ex)
    {
        // Pl. naplózzuk a kivételt, jelezzük a hibát az indítónak stb.
    }
}
```

Kivételkezelés WinUI alkalmazásokban

- Az eseménykezelők kivételei központilag kezelhetők
 - > Application.UnhandledException esemény (ha eseménykezelőkben nem kezelt kivétel van).
 - > Csak a fő szál (eseménykezelők) kivételeit kezeli: a pluszban indított szálakét nem!

```
public partial class App : Application
{
    public App()
    {
        this.InitializeComponent();
        this.UnhandledException += App_UnhandledException;
    }

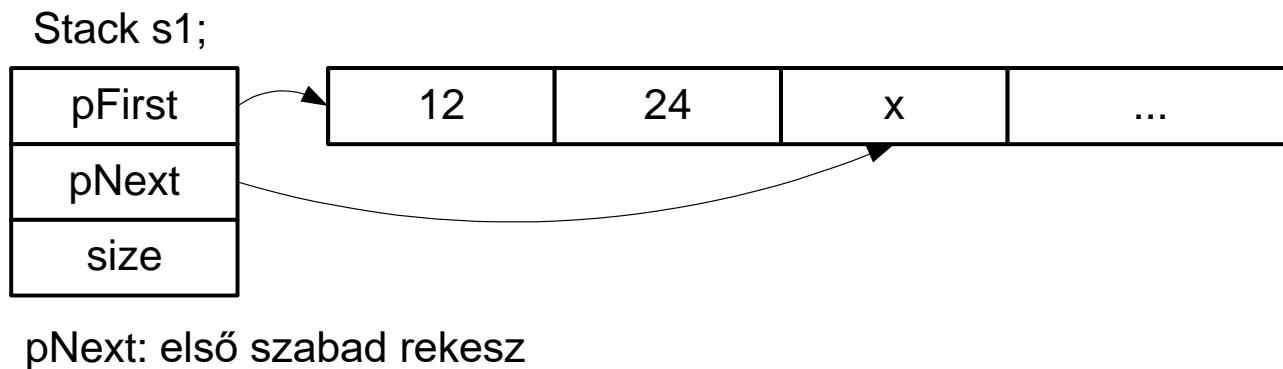
    private void App_UnhandledException(object sender,
                                         Microsoft.UI.Xaml.UnhandledExceptionEventArgs e)
    {
        // Log exception, then either exit the app or continue...
    }
}
```

Szálak szinkronizációja

Mire kell odafigyelni a szálak kommunikációja során?

A kölcsönös kizárási problémája

- Környezet:
 - > Megosztott erőforráshoz több szál fér hozzá
 - > Megosztott erőforrás: memória (változók, objektumok), fájl stb.
 - > Még egy CPU/mag esetén sem tudjuk, mikor veszi el az ütemező a szál futási jogát, mikor jár le a szál időszelete!
- Meg kell őrizni a konzisztenciát ebben a környezetben is!
- Stack példa
 - > A megosztott erőforrás egy **Stack s1** objektum, két szál használja.
 - > Az **s1** objektumban a **pFirst** az elemeket tároló memória elejére mutat, a **pNext** ebben az első szabad helyre (ahova a Push-nak írnia kell). Memóriakép:



magyarázat →

A kölcsönös kizárás problémája – stack példa magyarázat

- A Push két lépésből áll (**a két lépés nem atomi, más közbeékelődhet!**):
 - > 1. Ahova pNext mutat, az új érték beírása
 - > 2. pNext megnövelése
- Problémás forgatókönyv (ritka, de bekövetkezhet!)
 - > Szál1 Push(10)-et hív. Ahova pNext mutat (ábrán x rekesz) beíródik a 10. Ekkor a szál elveszti a futási jogát, pl. lejár az időszelete (még mielőtt pNext megnövelődne).
 - > Szál2 Push(20)-at hív. Ahova pNext mutat, beíródik a 20. Ezzel felülcsapta a Szál 1 által beírt 10-értéket! Majd megtörténik pNext megnövelése.
 - > Szál1 visszakapja a futási jogát. Most már ő is megnöveli pNext-et.
- Az s1 objektum inkonzisztens lett. Ahelyett, hogy egymás mellé beíródott volna a 10 és a 20: beíródott 10, ez felülíródott 20-szal, és az ezt követő memóriarekeszben (ahova 20-nak kellett volna íródnia) egy korábbi, ki tudja milyen érték maradt.
- A problémát az okozza, hogy a Push két lépése nem atomi, be tudott más is ékelődni ide, pl. egy másik Push hívás. Gondoljuk át, hogy két „egymásra futó” Pop is hasonlóan problémás!
 - > **A megoldás az lesz, hogy a két lépést „oszthatatlanná” tessük:** ha beékelődne ide egy másik szál hívása, várnia kell, amíg minden lépés lefut.

A kölcsönös kizárási problémája

- Példa2: Person objektumok adatainak fájlba/memóriába írása több szálból
 - > Akkor konzisztens, ha egy adott személy objektum minden adata egymás után íródik ki a fájlba (erre építünk beolvasáskor)
 - > Problémás forgatókönyv (ritka, de bekövetkezhet)
 - „Szál A” elkezdi kiírni az o1 Person objektum adatait fájlba. De mielőtt minden adatát kiírná, elveszti a futási jogát.
 - „Szál B” kiírja az o2 Person objektumot.
 - „Szál A” visszakapja a futási jogát, befejezi az o1 adatainak kiírását.
 - > A fájlunk inkonzisztens lett: o1 részei közé beékelődött egy o2 objektum:



A kölcsönös kizárási problémája

- Hogyan őrizzük meg a konzisztenciát? Kölcsönös kizárással
 - > Biztosítanunk kell, hogy a megosztott erőforráshoz egy időben csak egy szál férjen hozzá.
- Próbálkozzunk: vezessünk be egy **bool flag** változót, mely jelzi, hogy foglalt-e az erőforrás (ha foglalt, várni kell, pl. egy while ciklusban).

```
while (flag); // Várunk, ha foglalt  
flag = 1; // Először foglaljuk  
// Itt használjuk az erőforrást  
flag = 0; // Ha már nem használjuk
```

Szál A

```
while (flag);  
flag = 1;  
// Itt használjuk az erőforrást  
flag = 0;
```

Szál B

- Ha pont a while(flag); és a flag = 1; között veszti el „Szál A” a futási jogát: baj van →
 - > „Szál B” ekkor szabadnak látja a flag-et, és használja az erőforrást
 - > De amikor „Szál A” visszakapja a futási jogát, már túl van a feltételvizsgálaton, ő is használja az erőforrást.
 - > Vagyis „Szál A” és „Szál B” egyszerre használja az erőforrást, pont ez szerettük volna elkerülni ☹ ☹ ☹.

Kölcsönös kizáráás

- A megoldás: ún. oszthatatlan `test_and_set` utasításra van szükség.
 - > Ezt a hardver architektúrának kell támogatnia
 - > Erre építve megoldható a kölcsönös kizáráás problémája, **kritikus szakasz** alakítható ki, és további szinkronizációs konstrukciók készíthetők
 - > Az előző flag-es példánk is `test_and_set` volt, de nem volt oszthatatlan!
- **Kritikus szakasz**
 - > Az a kódrészlet, amelyből a megosztott erőforráshoz férünk hozzá, vagyis amelyre garantálni kell az atomi/oszthatatlan hozzáférést
 - > Az OS és a .NET támogatja kialakítását, rövidesen látjuk majd
- További probléma volt
 - > A `while(flag);` teljesen feleslegesen használja egy CPU magot (akár 100%-osan is!)
 - Ezt hívjuk **aktív várakozásnak**, kerülni kell!

.NET szinkronizációs konstrukciók I.

- A kölcsönös kizárás megvalósítására zárolási konstrukciók:

Név	Cél	Folyamatok között is?	Sebesség
lock C# utasítás (Monitor.Enter/Monitor.Leave)	Biztosítja, hogy egy adott erőforráshoz/kódrészlethez egy időben csak egy szál férhet hozzá.	nem	gyors
Mutex	Mint a lock, de folyamatok között is . Pl. annak megoldására, hogy egy alkalmazásból csak egy példány indulhasson.	igen	közepes
Semaphore	Mint a Mutex, de nem egy, hanem max. N hozzáférést engedélyez.	igen	közepes
SemaphoreSlim	Működése alapvetően megegyezik a Semaphore-ral, annak „lightweight” változata.	Nem	gyors
ReaderWriterLock és ReaderWriteLockSlim	Sok olvasóra optimalizált megoldás. Egyszerre több olvasó is hozzáférhet az erőforráshoz, de íróból csak egy (illetve az író kizára az olvasókat is). Pl. ritkán módosított cache megvalósítása.	nem	közepes

A lock használata

- Feladat: Két szál versenyez, aki előbb ér „célba”, írja ki, hogy „Winner”. Csak egy győztes lehet, ezt garantálni kell!

```
class ThreadSafeClass
{
    static bool done; // Közös erőforrás

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

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

!

A lock használata

- Az előző dia megoldásában csak egy egyszerű „bool done” flag-gel próbáljuk megvalósítani a kölcsönös kizárást. Ez nem jó megoldás (korábban se működött ez a technika)!
- A problémás forgatókönyv:
 - > Szál1 megvizsgálja az if-ben a done értékét, de true-ba már nem tudja állítani, mert pont itt elveszti a futási jogát (lejár az időszak). Az előző dián egy piros nyíl jelöli ezt a pontot.
 - > Szál2 is megvizsgálja a done értékét, false-nak látja, és kiírja a „Winner szöveget”.
 - > Szál1 visszakapja a futási jogát: már túl van a done feltételvizsgálaton, nem „néz vissza”, hanem ő is kiírja a „Winner” szöveget. Ezt el akartuk kerülni!
- Kis valószínűséggel ugyan, de hibásan működik a logika.

A lock használata

- Helyes kölcsönös kizárási megvalósítása a lock utasítással alkalmazásával

```
class ThreadSafeClass
{
    static bool done; // Közös erőforrás
    static object syncObject = new object();
    static void Main()
    {
        new Thread(Run).Start();
        new Thread(Run).Start();
    }

    static void Run()
    {
        lock (syncObject)
        {
            if (!done) {
                done = true;
                Console.WriteLine("Winner");
            }
        }
    }
}
```

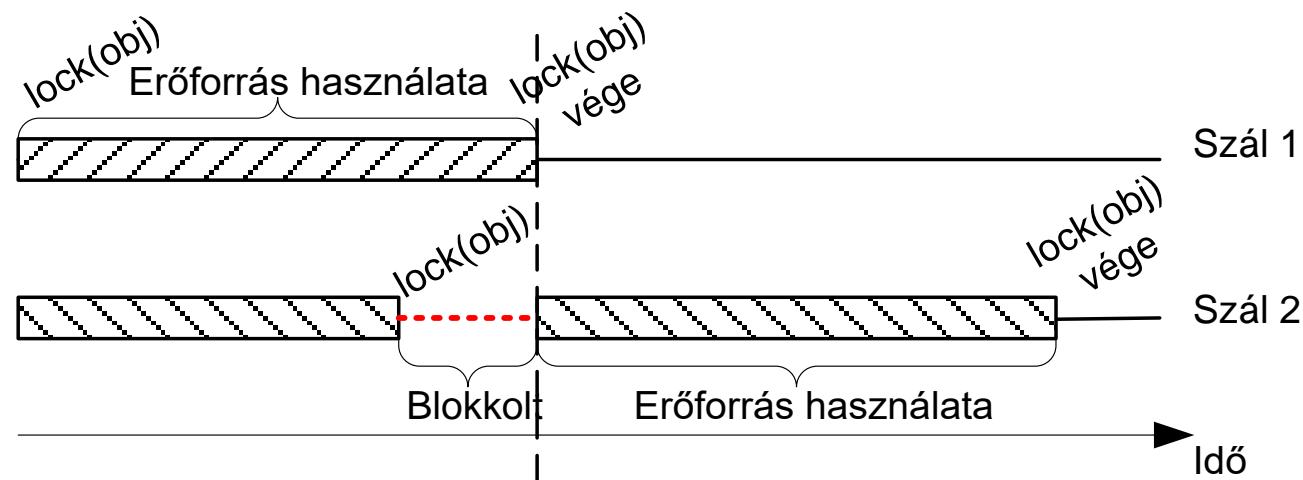
Magyarázat (fontos !!!)

- A lock blokkon belül egyszerre csak egy szál tartózkodhat (ha ugyanaz a lock paramétere, ami itt fennáll)
- Ha egy szál a lock blokkon belül van, a többinek a lock blokk elején várni kell
- Így hiába veszti el a szál1 a futási jogát az előző dia forgatókönyve szerint: a szál2 nem tud a lock blokkba belépni, meg kell várnia, míg a szál1 visszakapja a futási jogát, az beállítja a done-t true-ra, és elhagyja a lock blokkot.
- Így viszont garantáltan egy győztes lesz ☺.
- Hasonlít a Java synchronized-hoz

A lock használata

- A lock működése (részletesen):
 - > Egy objektum paramétert kell neki adni
 - > Megvizsgálja, hogy zárolva van-e az objektum
 - Ha nincs, atomi módon zárolja, majd tovább fut (a lock tulajdonképpen egy oszthatatlan test_and_set)
 - Ha igen, vár (blokkolja a hívó szálat), amíg fel nem szabadul
 - > A várakozás nem használ CPU időt
 - > A várakozó szálak egy sorba kerülnek, „érkezési sorrendben” kapnak hozzáférési jogot
 - > A lock blokkból kilépéskor oldja az objektumon levő zárat

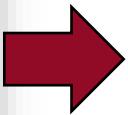
```
while (flag) ;  
flag = 1;  
// Itt használjuk az erőforrást  
flag = 0;
```



A lock és a Monitor osztály

- A lock utasításból a fordító ezt generálja

```
lock (syncObject)
{
    // ...
}
```



```
try
{
    Monitor.Enter(syncObject);
    // ...
}
finally
{
    Monitor.Exit(syncObject);
}
```

- Monitor.TryEnter – hasonló, de időkorlát is megadható

Zárak egymásba ágyazása

- A korábban zároló szál ismételten megszerezheti ugyanazt lock-ot, nem kerül blokkolásra.
- A zár elengedéséhez a zárolással megegyező számú zárfeloldás szükséges

```
static object syncObject = new object();

static void Main()
{
    lock (syncObject)
    {
        Console.WriteLine("locked");
        Nest();
        Console.WriteLine("still locked");
    } // Lock elengedése
}

static void Nest()
{
    lock (syncObject)
    {
        // ...
    } // Marad a lock
}
```

Mi lehet szinkronizációs objektum?

- Vagyis mi lehet lock paraméter? Csak referencia típus (osztály)!
- Hogyan védjük a tagváltozókat?
 - > A nem statikusakat nem statikus tagváltozóval (objektumszintű zár)
 - > A statikus tagváltozókat statikus tagváltozóval (osztályszintű zár)

Objektumszintű:

```
class ThreadSafeClass
{
    long var1 = 0;
    object syncObject = new object();

    void Increment()
    {
        lock (syncObject) { var1++; }
    }
}
```

Osztályszintű:

```
class ThreadSafeClass
{
    static long var1 = 0;
    static object syncObject
        = new object();

    static void IncrementStatic()
    {
        lock (syncObject) { var1++; }
    }
}
```

Magyarázat a következő két dián



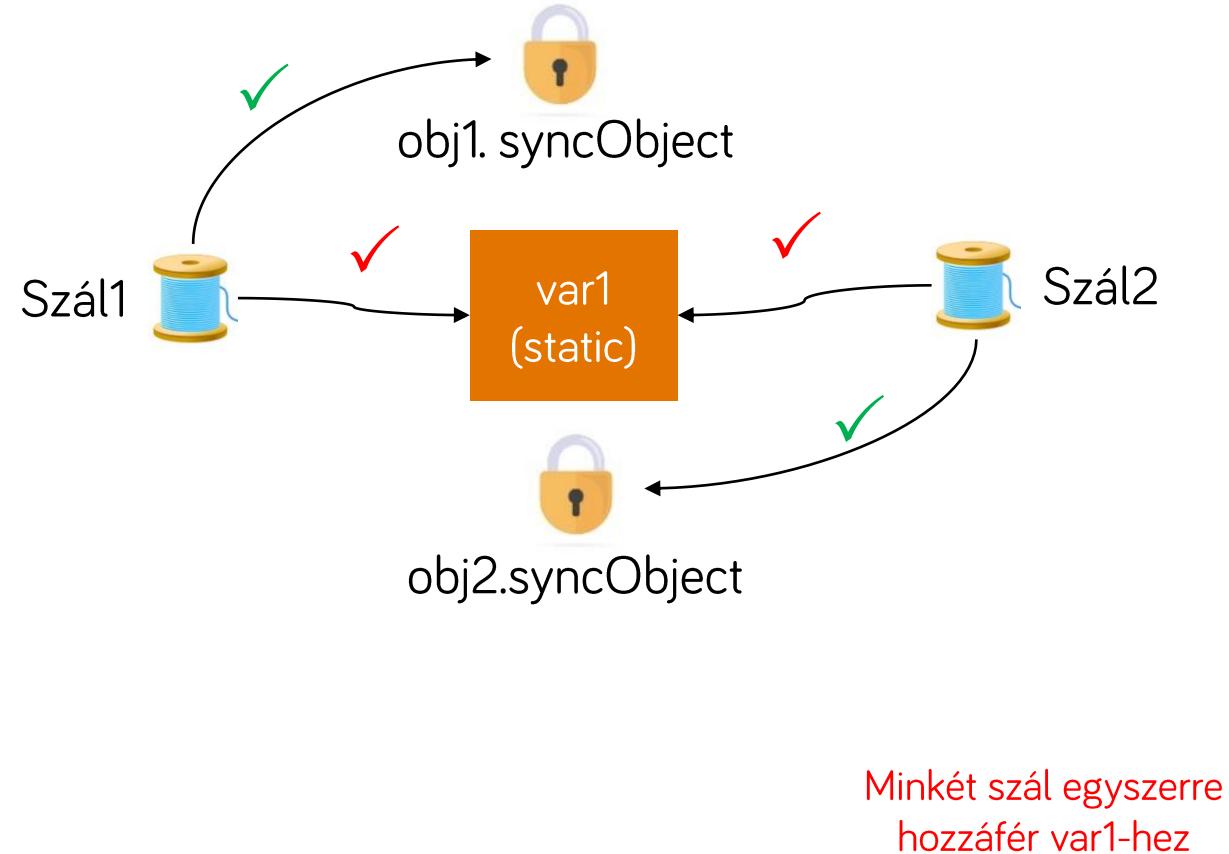
Nem statikus és statikus változók védelme

Magyarázat az előző példához: mi történik, ha nem követjük a szabályt:

- A) Ha statikus tag változót nem statikus objektummal (zárral) védünk

Probléma, hogy a zárak objektumonként vannak, míg védendő változó egy van (mert statikus):

Ha egy szál megszerzi a zárat egy objektumon (pl. obj1.syncObject), attól még más szál más objektumon (pl. obj2.syncObject) vele párhuzamosan meg tudja szerezni -> **EGYSZERRE** férhetnek hozzá a közös védett változóhoz, nem valósult meg a kölcsönös kizárás! ☹

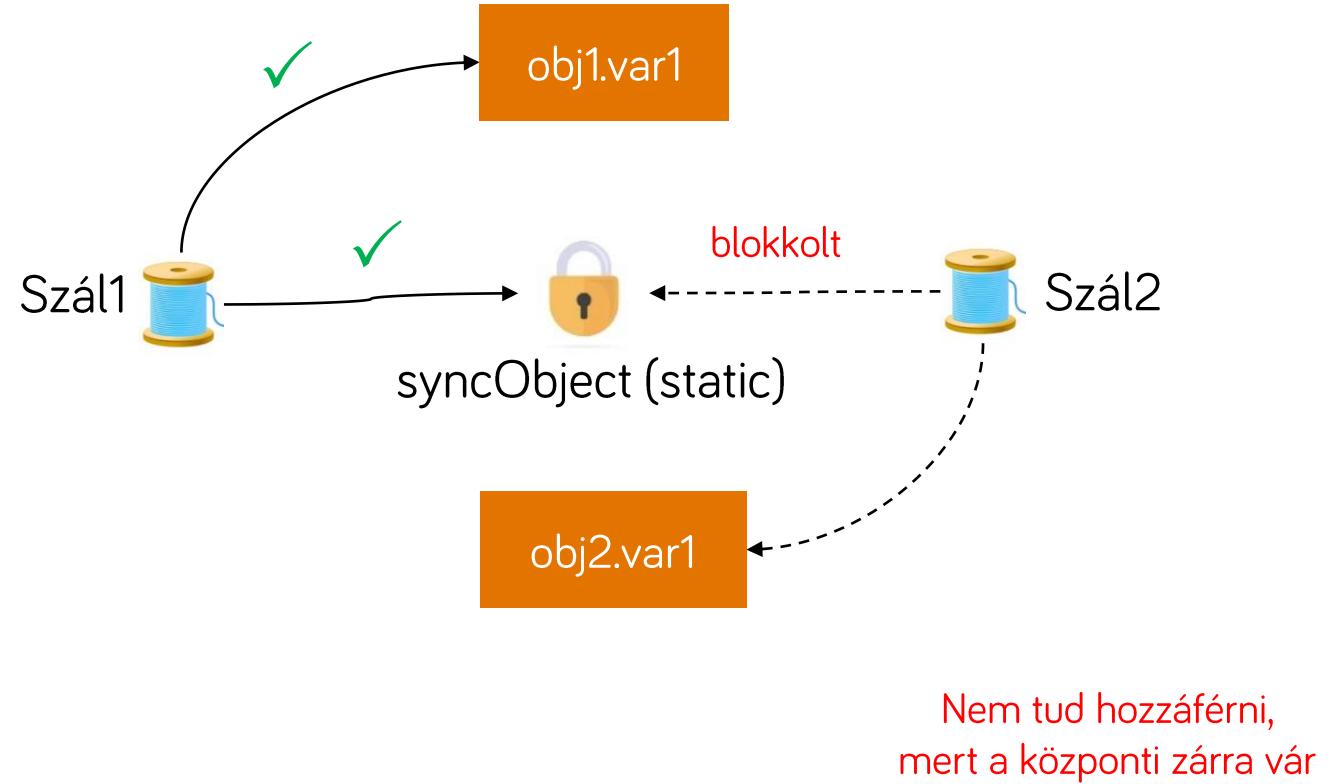


Nem statikus és statikus változók védelme

> B) Ha nem statikus tagváltozót statikus objektummal védünk

Probléma, hogy egy központi zárral (mert statikus) védünk különböző objektumokban helyet foglaló változókat:

Ha egy szál megszerzi a központi zárat és használ egy objektumot, akkor más szál más független objektumhoz sem tud hozzáférni → felesleges szűk keresztmetszetet vezetünk be ezzel 😊 (cél: különböző objektumokhoz LEHESSEN egyidőben hozzáférni).



Szálbiztos (thread-safe) osztályok

- Egy osztály szálbiztos, ha úgy lett megírva, hogy többszálú környezetben is biztonságosan használható.
 - > Maga garantálja a konzisztenciát
 - > A felhasználás során már nem kell zárolni...
- Szálbiztos Stack osztály →
 - > (üres/tele feltétel ellenőrzések nélkül)

```
public class Stack<T>
{
    readonly int size;
    int current = 0;
    T[] items;
    object syncObject = new object();

    public void Push(T item)
    {
        lock (syncObject)
        {
            items[current++] = item;
        }
    }

    public T Pop()
    {
        lock (syncObject)
        {
            return items[--current];
        }
    }
}
```

Szálbiztos osztályok

- Tegyünk minden osztályt szálbiztossá?
 - > A zárolás viszonylag időigényes ...
 - > Ezért ha az osztályt az esetek többségében egyszálú környezetben használják, akkor nem célszerű.
 - Ekkor is lehet köré egy szálbiztos csomagolóosztályt (wrapper) írni, aminek az interfésze ráadásul megegyezhet az eredeti osztállyal.
- .NET keretrendszer osztályok
 - > Tipikusan csak a statikus tagváltozók védettek, így csak a statikus műveletek/tulajdonságok szálbiztosak (nézzük meg a dokumentációt)
- Kitérő - C környezetben (pl. a printf függvény szálbiztos?)
 - > A globális és a statikus lokális változók a problémásak
 - > Pl. a C Runtime Librarynek van szálbiztos és nem szálbiztos változata, linker beállítás függő melyiket használjuk

Mi szálbiztos és mi nem?

- Mit kell védeni?

- > A C# nyelven csak a 32 bites és az annál kisebb változók olvasása, írása atomi (bool, byte, char, short, ushort, int, uint), valamint a referenciajának olvasása/írása is atomi.
- > Ha 64 bites a CPU és az OS/.NET is 64 bites, akkor a 64 bites változók olvasása/írása is atomi.

```
static int x, y;
static long z;

class A {...}

static void Test()
{
    long myLocal;
    x = 3; // Atomi minden esetben
    z = 3; // Nem atomi 32 bites OS alatt (z 64 bites)
    myLocal = z; // Nem atomi 32 bites OS alatt

    A a, b;
    a = b; // Referencia állítása atomi

    y += x; // Nem atomi: két művelet (olvasás + írás)
    x++; // Nem atomi: két művelet (olvasás + írás)
}
```

Miért nem szálbiztos a ++ operátor

- Pl. miért nem atomi/szálbiztos a „++”
 - > A ++ nem atomi, három lépésből áll: változó kiolvasása, megnövelése, majd növelt érték visszaírása a változóba
 - > T. f. h. két szál (T1 és T2) kb. egyszerre növeli egy x változó értékét „x++” -szal, x értéke kezdetben 2. Egy szerencsétlen forgatókönyvre példa:
 - T1 kiolvassa x értékét (2), és ezután elveszti a futási jogát
 - T2 kiolvassa x értékét (2), megnöveli és visszaírja ezt, így x értéke 3.
 - T1 visszakapja a futási jogot: a korábban kiolvasott értéket (2) megnöveli, visszaírja, így x értéke 3.
 - A probléma az, hogy kétszer is lefutott x++, de x értéke 4 helyett 3.
- Hogyan tehetők ezek atomivá?
 - > lock-kal/Mutexsel vagy:
 - > Nemblokkoló atomi utasítással: **InterlockedXXX, nagyon hatékony** ➔

Interlocked atomi utasítások

- Mit kell tudni? Egy példát mutatni (pl. kiemelt rész).

```
class Program
{
    static long sum;

    static void Main()
    {
        Interlocked.Increment(ref sum);
        Interlocked.Decrement(ref sum);
        Interlocked.Add(ref sum, 3); // Érték hozzáadása
        // 64 bites érték olvasása
        Console.WriteLine(Interlocked.Read(ref sum));
        // Érték írása és az előző olvasása atomi módon
        // Ez 3-at ír ki, sum 10 lesz
        Console.WriteLine(Interlocked.Exchange(ref sum, 10));
        // Ha sum 10, akkor 123-at ír bele atomi módon
        Interlocked.CompareExchange(ref sum, 123, 10);
    }
}
```

Volatile mezők

```
class Foo
{
    public int x;
    public int y;
}
```

```
// szál 1
foo.y = 5;
foo.x = 1;
```

Nem biztos,
hogy 5-öt ír ki!

```
// szál 2
if (foo.x == 1)
{
    Console.WriteLine(foo.y);
}
```

- A „szál 2” 5-öt kellene mindig kiírjon: elvileg, amikor x értéke 1 lesz, akkor y már 5 kellene legyen (az írásuk sorrendjéből adódik). De:
 - > A compiler számára megengedett, hogy a nem-volatile mezők írási és olvasási sorrendjét felcserélje, ha azon a szálon nem látszik a különbség (a többi szálat nem veszi figyelembe). A példánkban y és x írása így lehet felcserélődik!
 - > Előfordulhat, hogy az y változó értéke regiszterben cache-elt, ezt a „szál 2” nem látja

Volatile mezők

- Megoldás
 - > A volatile írás nem lehető későbbre.
 - > A volatile olvasás nem lehető előbbie.
 - > „Azonnal” megtörténik a memóriába írás (nem cache-elt regiszterben)

```
class Foo
{
    public volatile int x;
    public volatile int y;
}
```

- Megjegyzés: **lock esetén nincs szükség a volatile alkalmazására**
 - > lock blokkba belépéskor minden megtörténik a változók memóriából frissítése
 - > lock blokkból kilépéskor minden megtörténik a változók memóriába kiírása

Szál kiléptetése – Alternatíva 1

- Egy bool változóval
 - > A szálnak rendszeres időközönként rá kell nézni!
 - > A bool hozzáférést NEM kell lock-olni

```
class ThreadClass
{
    static volatile bool exit = false;

    static void Main()
    {
        Thread thread = new Thread(Run);
        thread.Start();
        Thread.Sleep(2000);
        Console.WriteLine(
            "Signaling to worker thread.");
        exit = true;
        Console.WriteLine("Waiting for
            worker thread to exit.");
        thread.Join();
        Console.WriteLine("Worker thread
            definitely has exited.");
    }
}
```

- Szál kilépésének bevárása:
`Thread.Join()`

```
static void Run()
{
    int counter = 0;
    while (!exit)
    {
        Thread.Sleep(300);
        Console.WriteLine(
            counter++.ToString());
    }

    Console.WriteLine(
        "Exiting from worker thread.");
}
```

.NET szinkronizációs konstrukciók II.

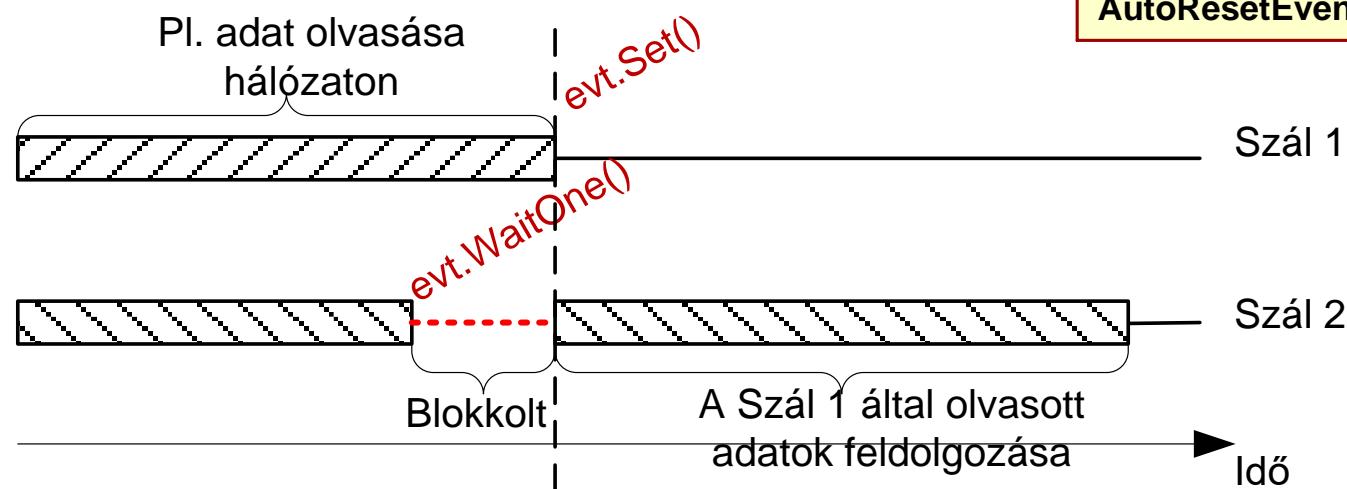
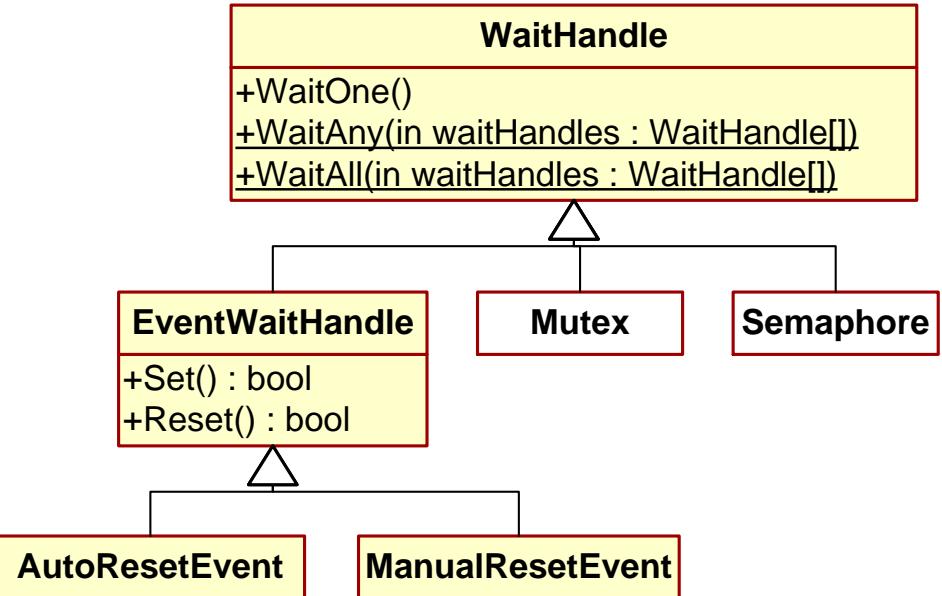
- A lock megoldja a kölcsönös kizárás problémáját, ezt láttuk
- Más szálnak jelzésre (pl. munka kezdhető, lépj ki stb.) nem alkalmas
- Jelzésre alkalmas szinkronizációs konstrukciók:

Név	Cél	Folyamatok között is?	Sebesség
AutoResetEvent és ManualResetEvent	Lehetővé teszi hogy egy szál hatékony módon várakozzon egy más szál által küldött jelzésre.	igen	közepes
Monitor.Wait and Monitor.Pulse	Lehetővé teszi hogy egy szál hatékony módon várakozzon tetszőleges blokkoló feltételre.	nem	közepes

A ManualResetEvent-nek van ManualResetEventSlim változata, mely kisebb erőforrásigényű, de csak egy folyamaton belül használható

AutoResetEvent és ManualResetEvent

- **WaitHandle**: Olyan osztályok őse amelyek objektumaira várakozni lehet
- Az **AutoResetEvent** és **ManualResetEvent** (két WaitHandle leszármazott) jelzésre/eseményre való várakozást tesz lehetővé



`AutoResetEvent evt = new AutoResetEvent(false);`

AutoResetEvent és ManualResetEvent

- **Set** művelet:
 - > Jelzettbe állítja az event objektumot, így elenged egy várakozó objektumot.
 - > Azt nem tartja nyilván, hányszor volt Set hívás (a több is egynek számít).
- **WaitOne** művelet:
 - > Ha az event objektum nem jelzett, várakozik (a CPU-t nem terheli)
 - > Ha az event objektum jelzett:
 - Tovább engedi a szálat, visszatér
 - **AutoResetEvent** esetén automatikusan nem jelzettbe állítja az event objektumot, így csak egy várakozó szál jut át egy Set után (olyan, mint egy forgókapus beléptető, egy „jelzésre” csak egyet enged be)
 - **ManualResetEvent** esetén: ez nem állítja az esemény állapotát (nem csukja le maga mögött a „sorompót”)
- **Reset** művelet:
 - > Reseteli (nem jelzett állapotba állítja) az event objektumot

AutoResetEvent példa

```
class SimpleEventDemo
{
    // A konstruktorparaméterben az event kezdeti állapotát
    // adjuk meg (false - nem jelzett).
    static AutoResetEvent wh = new AutoResetEvent(false);

    static void Main()
    {
        new Thread(Run).Start();
        Thread.Sleep(1000); // Várunk egy kicsit
        wh.Set(); // Ébresztő
    }

    static void Run()
    {
        Console.WriteLine("Várunk értesítésre ...");
        wh.WaitOne(); // Várakozás
        Console.WriteLine("Az értesítés megérkezett.");
    }
}
```

Megjegyzés: a WaitOne-nak timeout is megadható.

AutoResetEvent – példák*

- AutoResetEvent – jóváhagyás példa
 - > Lásd ThreadDemo solution...
- Termelő/fogyasztó sor példa
 - > Az életben gyakran van szükség hasonlóra. Pl.:
 - Egy háttérszál olvas a soros porton/hálózatról adatokat, üzenetekké alakítja és beteszí egy sorba további feldolgozásra.
 - Terheléskiegyenlítés támogatása (burst-ösen érkeznek a feladatok, ne kelljen eldobni akkor se, ha nem tudjuk azonnal feldolgozni)
 - Skálázhatóság: több fogyasztó szál lehet, párhuzamosan (több CPU esetén) dolgozhatják fel a feladatokat
 - > Lásd ThreadDemo solution...

További WaitHandle leszármazottak

- [ManualResetEvent](#)
 - > „Kapuként” funkcionál: ha jelzett, mindenki mehet, ha nem jelzett, mindenki vár.
 - > Vagyis az AutoResetEvent-tel szemben nem csak egy várakozót enged át.
- [Mutex](#)
 - > Mint a lock, kölcsönös kizáráusra
 - > Eltérő folyamatok szálai között is, használható, de a lock-nál néhány százszor lassabb.
 - > A WaitOne-nal lehet lefoglalni (ami atomi módon zárolja, illetve blokkol, ha foglalt), elengedni a ReleaseMutex-szel lehet.
- [Semaphore](#)
 - > Mint a mutex, de egynél több (N, megadható) hozzáférést enged.
Ha max. N hozzáférést szeretnénk.
 - > A WaitOne-nal lehet rá várakozni. Ha még szabad a szemafor, nem blokkol, eggyel csökkenti a szemafor számlálót.
Ha nem szabad, blokkol. Elengedni a ReleaseSemaphore-ral lehet (növeli a szemafor számlálót).

WaitHandle

- WaitHandle leszármazott
 - > implementálja az IDisposable-t (egy natív leírót csomagol)
- **WaitHandle.WaitAny** statikus művelet
 - > Egynél több leíróra vár.
Ha bármelyik jelzett/szabad, akkor nem blokkol.
 - > Példa: lásd gyakorlat példája
- **WaitHandle.WaitAll** statikus művelet
 - > Egynél több leíróra vár.
 - > Ha minden jelzett/szabad, csak akkor nem blokkol.
 - > Ritkán használt.

Szinkronizációs problémakörök összefoglalása

- **Kölcsönös kizárásra:** zárolási konstrukciók (lock, Mutex, Semaphore, ReaderWriterLock, ReaderWriterLockSlim)
 - > A problémát el is tudjuk kerülni, ha ún. **immutable osztályokat** használunk. Ezek nem megváltoztathatók, minden módosító művelet új objektumot hoz létre. Ilyen pl. a .NET string osztálya is
 - pl. a string Replace művelete nem azt a string objektumot módosítja, melyre meghívtuk a műveletet, hanem egy új objektumot ad vissza, melyben a karakterek már le vannak cserélve
 - > Interlocked osztály használata
- **Jelzés és jelzésre várakozásra:** ManualResetEvent, AutoResetEvent
- **Egyebek:** Sleep, Join stb.
- Természetesen más lehetőségek is vannak, a tárgy keretében nem nézzük

Többszálú WinUI alkalmazások

Architektúra

- A WinUI GUI elemek (vezérlőelemek/ablakok) csak abból a szálból érhetők el, mely létrehozta őket
 - > A GUI elemeket szinte mindenkor a fő szálban hozzuk létre
 - > Munkaszálból ne hívunk rájuk műveletet/property-t (kivételt kapunk)
 - > A GUI elemekhez hozzáérni más szálból a `DispatcherQueue` segítségével lehet
 - > A `DispatcherQueue.TryEnqueue` a paraméterben megadott delegate-et a kontroll létrehozó szálból (tipikusan a fő szálból) hívja meg.
- A `DispatcherQueue` használatára laboron látunk példát, abból érhető meg a használata!

Thread-pool

Thread-pool

- Kiszolgáló alkalmazást írunk. Cél több kérés párhuzamos kiszolgálása
- Megoldás 1. - minden beérkező kérésnek új szálat indítunk. Problémák:
 - > Nagyszámú párhuzamos kérés esetén **túl sok szál fut**
 - Folyamatonként ~100-nál többet nem célszerű, a szálak közötti sok váltás miatt
 - > A szál indítása viszonylag költséges (**akkor jelentős a relatív veszteség, ha a szálfüggvény gyorsan lefut !!!**)
- Thread-pool alkalmazása az igazi megoldás
 - > Előre elindítunk néhány szálat
 - kiszolgálás során ezekből allokálunk
 - > A kiszolgálás végeztével a szál visszakerül a poolba, új kérést szolgálhat ki
 - > Ha nincs szabad szál a poolban:
 - Új szálat indítunk és teszünk a poolba,
 - Ha már túl sok szál van (néhányszor tíz) - blokkoljuk a hívót míg nem szabadul fel szál
 - > **Így megspóroljuk a sok szálindítással/leállítással járó költséget**

Thread-pool

- ThreadPool .NET osztály
- minden folyamattal létrejön egy
- Mikor célszerű a ThreadPool használata?
 - > Csak rövid műveleteket futtatunk.
- Ne blokkoljuk a ThreadPool szálakat (máskülönben hamar kimerítjük a ThreadPoolt).
- A Thread.Interrupt nem igazán használható kiléptetésre
 - > Flaggel
 - > Esetleg ManualResetEvent-tel kombinálva

A thread-pool programozása

```
class Test
{
    static ManualResetEvent doneEvent = new ManualResetEvent(false);

    public static void Main()
    {
        ThreadPool.QueueUserWorkItem(Run); // Delegate parameter
        Console.WriteLine("Waiting for threads to complete...");
        doneEvent.WaitOne();
        Console.WriteLine("Worker has ended!");
        Console.ReadLine();
    }

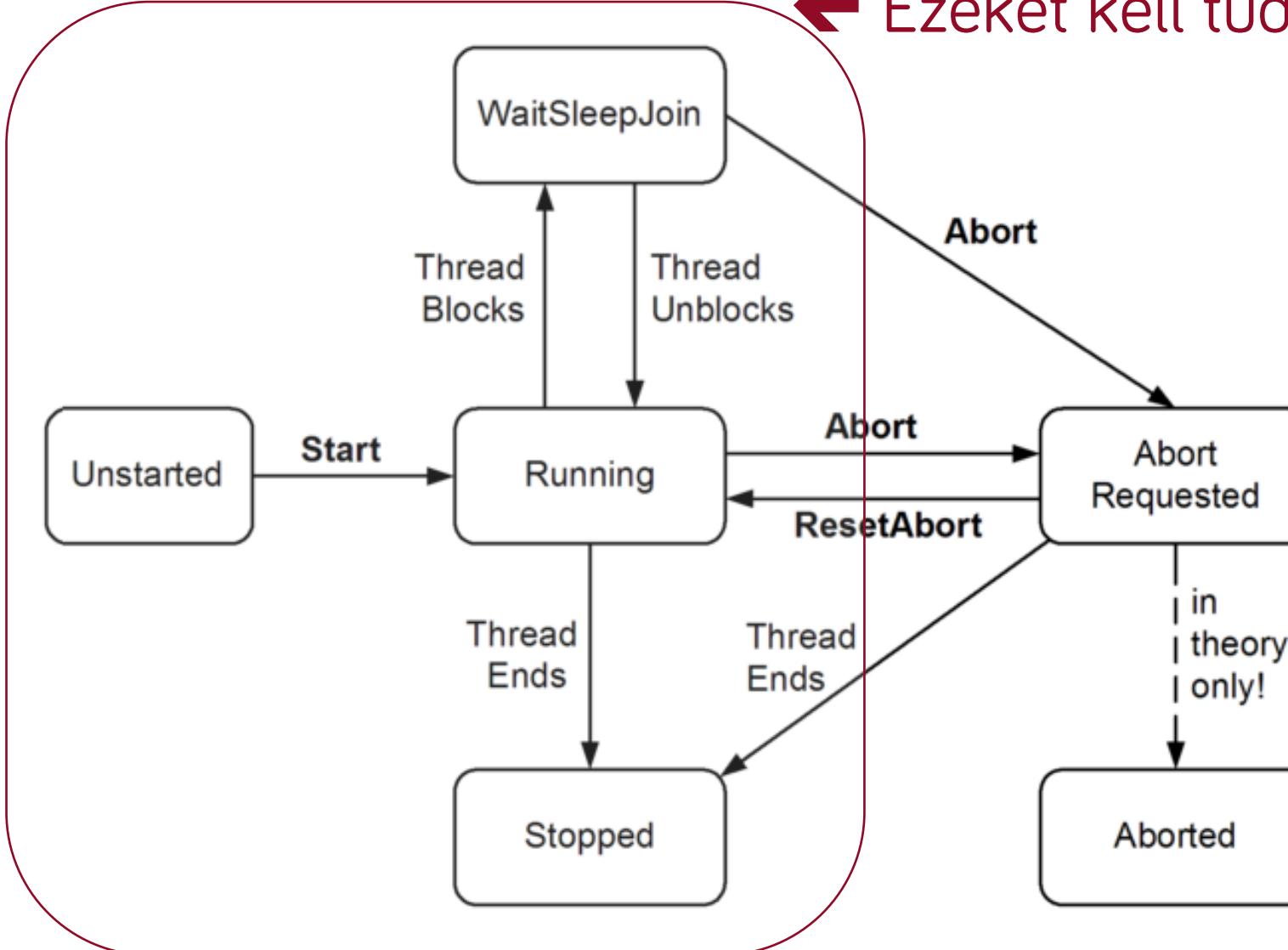
    public static void Run(object instance)
    {
        Console.WriteLine("Started...");
        Thread.Sleep(1000);
        Console.WriteLine("Ended");
        doneEvent.Set();
    }
}
```

ThreadPool
osztály
QueueUserWorkItem
művelete

Egyéb szálkezeléshez kapcsolódó technikák

Szál állapotok

Ezeket kell tudni



Szálállapotok

- **WaitSleepJoin állapot**
 - > Blokkolt állapot
 - > `Sleep`, `Join`, `lock`, `WaitOne`, `WaitAny`, `WaitAll` utasítások hatására
 - > Mind blokkol. Meddig? Négy módon léphet ki blokkolt állapotból:
 - A várakozási feltétel teljesül
 - Timeout lejár (ha adtunk meg)
 - A várakozás a `Thread.Interrupt` hívásával megszakításra kerül
 - A várakozás a `Thread.Abort` hívásával megszakításra kerül

Szál kiléptetése – alternatíva 2

Demo

- Eddig korábban bool flag tagváltozóval. De mi van, ha osztályunk más osztályok metódusát hívja, és a blokkolás ezekben történik? Ekkor az osztályunk nem tudja a bool flaget ellenőrizni.
- A megoldás a **ThreadInterrupt** használata
 - > A szál, amire meghívjuk, kap egy **ThreadInterruptedException-t**, de csak ha **WaitSleepJoin** állapotban volt
 - Vagyis a while(true); -ből nem fogja kiléptetni a szálat
 - Ha az Interrupt hívásakor nem volt WaitSleepJoin állapotban, akkor a ThreadInterruptedException akkor keletkezik a szálban, mikor az legközelebb WaitSleepJoin állapotba lép.

Thread.Interrupt

```
Thread thread = new Thread(Run);
thread.Start();
Thread.Sleep(2000);
thread.Interrupt(); // Signaling to worker thread.
thread.Join(); // Waiting for worker thread to exit.
Console.WriteLine("Worker thread definitely has exited.");
// ...
static void Run()
{
    try
    {
        string task;
        while (true)
        {
            task = queue.GetTask();
        }
    }
    catch (ThreadInterruptedException)
    {
    }
    Console.WriteLine("Exiting from worker thread.");
}
```

```
class OneItemQueue
{
    public string GetTask()
    {
        // WaitSleepJoin állapotban
        // blokkolt.
        autoResetEvent.WaitOne();
        return task;
    }
    ...
}
```

Szál kiléptetése – alternatíva 3

- `Thread.Abort()` művelet - .NET-ből kivezetik – már ezért se használjuk!
- Hasonlít a `Thread.Interrupt`-ra, de
 - > A szál `ThreadAbortException`-t kap
 - > Nem csak `WaitSleepJoin` állapotban, hanem bármilyen állapotban lehet a szál
 - Vagyis a `while(true);` -ból is kilépteti!
 - A `finally` blokkok azért lefutnak (persze ebben lehet még egy végtelen ciklus ☺)
- Ne használjuk, mert
 - > Nem tudjuk mit csinál éppen a szál! Lezáratlan dolgok maradhatnak mögöttünk...
- Korábban csak egy esetben:
 - > az alkalmazásból való kilépés esetén
 - > ekkor a magunk mögött hagyott leírók úgyis felszabadulnak.



```
StreamWriter w = File.CreateText("myfile.txt");
// Ha itt fut be az Abort, baj van!
try
{
    w.WriteLine("Abort-Safe");
}
finally { w.Dispose(); }
```

Thread safe interface minta

Thread safe interface minta

- A cél: minimális zárolási overhead
- Probléma
 - > Egy osztály műveletei tetszőleges módon és számban hívhatják egymást. Előfordulhat, hogy egy művelet zárol, majd egy általa hívott másik művelet is zárol (már feleslegesen, hiszen a szál már megszerezte a zárat).
- Megoldás:
 - > Publikus műveleteknél zárolás
 - > Privát műveleteknél: megbízunk abban, hogy a hívó művelet már zárolt, ha szükséges.

Thread safe interface minta - probléma

```
public class ThreadSafeClass2
{
    private List<int> list = new List<int>();
    private object syncObject = new object();

    public void FunctA()
    {
        lock (syncObject)
        {
            list.Clear();
            FunctB();
        }
    }

    public void FunctB()
    {
        // Ha FunctA hívja, ez már felesleges, mert a hívó már zárol.
        lock (syncObject)
        {
            list.Add(1);
        }
    }
}
```

Thread safe interface minta - megoldás

```
public class ThreadSafeClass2
{
    private List<int> list =
        new List<int>();
    private object syncObject =
        new object();

    public void FunctA()
    {
        lock (syncObject)
        {
            functA();
        }
    }

    public void FunctB()
    {
        lock (syncObject)
        {
            functB();
        }
    }

    private void functA()
    {
        list.Clear();
        functB();
    }

    private void functB()
    {
        list.Add(1);
    }
}
```

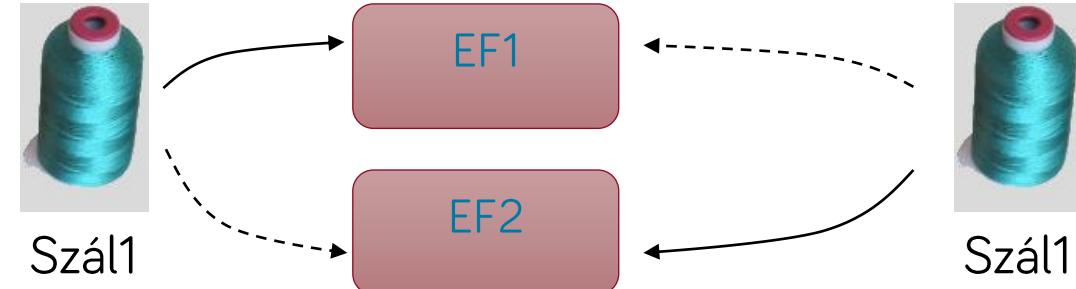
Holtlpont (deadlock)

Holtpont (deadlock)

Demo

- **Holtpont**

- > Kettő (vagy több) szál kölcsönösen egymásra vár.
- > A probléma oka: foglalva várakozás!



- **Detektálható**

- > Megfelelő irányított gráfot kell detektálni.
- > Nekünk kell(ene) megírni.
- > Nem szoktuk.
- > Inkább megpróbáljuk elkerülni.

- **Elkerülése**

- > Az erőforrásokat minden szálban ugyanabban a sorrendben foglaljuk le
- > Nem mindig tehető meg ☹

- **Alternatív megoldás**

- > Adott időkorlátig várakozunk az erőforrásra (timeout alkalmazása).

Deadlock példa (lock alkalmazásakor)

- Ha RunA és RunB szálfüggvények kb. egyszerre indulnak, holtpontot eredményez →
- RunA megszerzi syncObjectA zárat, RunB syncObjectB-t, majd a belső lock-ban kölcsönösen egymásra várak.
- Megoldás: RunB is olyan sorrendben szerezze meg a zárákat, mint A, ekkor nem lehet holtpont (ehhez át kell rendezni RunB kódját).

```
static void RunA()
{
    Console.WriteLine(
        "RunA: indulok..."); 
    lock (syncObjectA)
    {
        Thread.Sleep(1000);
        lock (syncObjectB)
        {
            Console
                .WriteLine("RunA:
hahó!");
        }
    }
}

static void RunB()
{
    Console.WriteLine(
        "RunB: indulok..."); 
    lock (syncObjectB)
    {
        Thread.Sleep(1000);
        lock (syncObjectA)
        {
            Console
                .WriteLine("RunB:
hahó!");
        }
    }
}
```

ÖSSZEFOGALÁS

Többszálú alkalmazások hátrányai

- Növeli az alkalmazás komplexitását
 - > Meg kell oldani a szálak szinkronizációját. Pl. kölcsönös kizárási problémája.
- Nagyon-nagyon-nagyon-nagyon nehezen kinyomozható problémákat okozhat a nem megfelelő szinkronizáció
 - Pl. a kölcsönös kizárási elmaradása miatt havonta egyszer elszáll az alkalmazás
 - Nem lehet kidebuggolni, hiszen nem reprodukálható
- A túl sok szál alkalmazása terheli a rendszert
 - > Ütemezni kell, váltáskor context switch – CPU igény
 - > Szálanként saját stack – memóriaigény
- Holtpont (deadlock) kialakulásának veszélye
- Tanulság: a konkurens programozást nem lehet csak félre megtanulni.

További téma körök (nem voltak)

- Magasabb szintű szálkezelés (félév végén lesz)
 - > Task, Task<T>
- Aszinkronitás támogatása nyelvi szinten (félév végén lesz)
 - > async, await kulcsszavak, aszinkron metódusok
- Egyebek
 - > Mutex, Semaphore használata
 - Hasonlóképpen használható, mint az AutoResetEvent
 - > Nevesített Mutex, Event, Semaphore
 - Ha folyamatok közötti szinkronizációra van szükség
 - > Monitor.Wait, Monitor.Pulse és Monitor.PulseAll használata
 - > Thread Local Storage
 - Szálhoz rendelt információ
 - > Stb.

Irodalom

- Threading in C#
 - > <http://www.albahari.com/threading/>
- Managed threading basics
 - > <https://learn.microsoft.com/en-us/dotnet/standard/threading/managed-threading-basics>