Haladó fejlesztési technikák

Task Parallel Library
async-await
Adatpárhuzamosság
Versenyhelyzet
Szinkronizáció
Holtpont

async-await

Aszinkronitás

„Asynchrony is essential for activities that are potentially blocking (…). Access to a web resource sometimes is slow or delayed. If such an activity is blocked within a synchronous process, the entire application must wait.

In an asynchronous process, the application can continue with other work that doesn't depend on the web resource until the potentially blocking task finishes.”

JavaScript

Az aszinkron kódolásra kiváló eszköz a JavaScript nyelv használata: az alapból aszinkron felépítésű kódban minden hosszadalmas függvényhívás, ahol erőforrásra kell várakozni, átlépésre kerül. Amikor a folyamat végzett, a visszatérésével egy úgynevezett callback-function foglalkozik.

Szinkron példa

```
Console.WriteLine("Megkezdem a letöltést...");
string tartalom = new
WebClient().DownloadString("http://users.nik.uni-
obuda.hu/prog3/_data/war_of_westeros.xml");
Console.WriteLine("Letöltve!");
Console.WriteLine(tartalom.Contains("Lannister") ? "Van benne
Lannister" : "Nincs benne Lannister");
```

```
Megkezdem a letöltést...
Letöltve!
Van benne Lannister
```

Klasszikus aszinkron megoldás esemény alapon

```
static void Main(string[] args)
{
    Console.WriteLine("Megkezdem a letöltést...");
    WebClient wc = new WebClient();
    wc.DownloadStringCompleted += Wc_DownloadStringCompleted;
    wc.DownloadStringAsync(new Uri("http://users.nik.uni-
obuda.hu/prog3/_data/war_of_westeros.xml"));

    Console.ReadLine();
}

private static void Wc_DownloadStringCompleted(object sender,
DownloadStringCompletedEventArgs e)
{
    Console.WriteLine("Letöltve!");
    Console.WriteLine(e.Result.Contains("Lannister") ? "Van benne
Lannister" : "Nincs benne Lannister");
}
```

Task alapú megoldás

```
Console.WriteLine("Megkezdem a letöltést...");
Task<string> t = new WebClient().DownloadStringTaskAsync("http://users.nik.uni-obuda.hu/prog3/_data/war_of_westeros.xml");
t.ContinueWith(x => {
    Console.WriteLine("Letöltve!");
    Console.WriteLine(x.Result.Contains("Lannister") ? "Van benne Lannister" : "Nincs benne Lannister");
});
```

async-await

```
async Task<string> AszinkronValasz()
{
    await Task.Delay(2000);
    return "kész";
}
```

- **await**
  - „bevárás”
  - A végrehajtás szüneteltetése, amíg a „bevárt” Task nem végez
  - **await**elni Task-ot lehet, akár visszatérési értékkel
  - Sok beépített nyelvi elem (*GetXXXAsync()*)
- **async**
  - Ahol **await**elsz valamit, azt a metódust az **async** kulcsszóval meg kell jelölni
  - Csak void, Task és Task <TResult> visszatérésű metódusok lehetnek **async**-ek

async-await alapú megoldás

```
static async void Elemez(string url)
{
    Console.WriteLine("Letöltés megkezdve!");
    string tartalom = await new
WebClient().DownloadStringTaskAsync(url);
    Console.WriteLine("Letöltve!");
    Console.WriteLine(url + ": " + (tartalom.Contains("Lannister") ?
"van benne Lannister" : "nincs benne Lannister"));
}
```

async-await alapú megoldás

```
static void Main(string[] args)
{
    Console.WriteLine("Megkezdem a letöltéseket...");
    string urls = "http://users.nik.uni-
```

```
Megkezdem a letöltéseket...
Letöltés megkezdve!
http://users.nik.uni-obuda.hu/prog3/_data/war_of_westeros.xml elemzése elindult
Letöltés megkezdve!
http://users.nik.uni-obuda.hu/prog3 elemzése elindult
Letöltés megkezdve!
http://nik.uni-obuda.hu/ elemzése elindult
Letöltés megkezdve!
http://uni-obuda.hu elemzése elindult
Letöltve!
http://users.nik.uni-obuda.hu/prog3/_data/war_of_westeros.xml: van benne
Lannister
Letöltve!
http://users.nik.uni-obuda.hu/prog3: nincs benne Lannister
Letöltve!
http://uni-obuda.hu: nincs benne Lannister
Letöltve!
http://nik.uni-obuda.hu/: nincs benne Lannister
```

async-await a Main()-ben

```
static async Task Main(string[] args)
{
    Console.WriteLine("Megkezdem a letöltést...");
    string tartalom = await new
WebClient().DownloadStringTaskAsync("http://users.nik.uni-
obuda.hu/prog3/_data/war_of_westeros.xml");
    Console.WriteLine("Letöltve!");
    Console.WriteLine(tartalom.Contains("Lannister") ? "Van benne
Lannister" : "Nincs benne Lannister");
    Console.ReadLine();
}
```

Vizuális alkalmazásoknál látványos

```
async Task<string> AszinkronValasz()
{
    await Task.Delay(2000);
    return "kész";
}

private async void button_Click(object sender, RoutedEventArgs e)
{
    textBox.Text = await AszinkronValasz();
}
```

Párhuzamos?

- **Valójában önmagában nincs külön szál**
- **Az async módosítószóval megjelölt metódus az await kulcsszóval megjelölt hívásig szinkron kerül végrehajtásra**
- **Ott azonban a metódus „suspended” állapotú lesz, amíg a Task nem végez**
  - A Task persze „dönthet úgy” hogy külön szálat alkalmaz
- **Amíg a metódus áll, a vezérlés visszatér a hívás helyére, és továbblép**
- **Amikor a Task végzett, a vezérlés visszakerül oda**

Adatpárhuzamos megoldások

Parallel class

- **Parallel.For(int, int, Action<>)**
- **Parallel.ForEach(IEnumerable<>, Action<>)**

- **Parallel.Invoke(Action[])**

Parallel.For

- Gyakorlatilag a for ciklus magjának kifejtése és `Action<int>` delegált meghívása minden ciklusváltozóra

```
int[] eredm = new int[10];
Parallel.For(0, eredm.Length,
    i => {
        eredm[i] = i * i;
        Console.Write(i + " kész! ");
    }
);

Console.WriteLine();
for (int i = 0; i < eredm.Length; i++)
    Console.Write(i + ": " + eredm[i] + ", ");
```

```
0 kész! 1 kész! 3 kész! 4 kész! 5 kész! 6 kész! 7 kész! 8 kész! 9
kész! 2 kész!
0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81,
```

Parallel.ForEach

- Hasonlóan a Parallel.For()-hoz, a gyűjtemény típusának megfelelő `Action<T>` delegáltat alkalmaz

```
Parallel.ForEach("hello world!", e =>
{
    Console.Write(char.ToUpper(e));
});
Console.ReadLine();
```

```
OHELWRD!LO L
```

Parallel.Invoke

- A Parallel.Invoke() egy `Action[]` bemenetet kap paraméterként, amely delegáltakat párhuzamosan igyekszik végrehajtani
- Az utasítás blokkol, addig nem lép tovább amíg minden Action végrehajtása véget nem ért

```
Parallel.Invoke(
    BasicAction,                // Param #0 - static method
    () =>                       // Param #1 - lambda expression
    {
        Console.WriteLine("Method=beta,Thread={0}", Thread.CurrentThread.ManagedThreadId);
    },
    delegate ()                 // Param #2 - in-line delegate
    {
        Console.WriteLine("Method=gamma,Thread={0}", Thread.CurrentThread.ManagedThreadId);
    }
);
```

VERSENYHELYZET

Utasítások végrehajtási sorrendje

- **Ha a szálak nem használnak közös változókat**
  - Egyszerű eset, nincs logikailag nehéz lépés
  - Ha ugyanazt csinálják a szálak, de különböző adatokon: adatpárhuzamosság
- **Ha a szálak közös változó(ka)t használnak**
  - Nem mindegy a végrehajtási sorrend
  - Átlapolás (interleaving) gondot okozhat

```
A = B = 0

R1 = A          R2 = B
B = 1           A = 2
```

Mi R1 és R2 értéke?

Versenyhelyzet példa

- Filmet szeretnél nézni a moziban
- A pénztárnál megkérdezed, hogy van-e szabad hely
- Azt a választ kapod, hogy igen, van
- Elmész a büfébe, néhány perc múlva visszatérsz, és jegyet szeretnél venni
- Azt a választ kapod, hogy nincs szabad hely

Versenyhelyzet példa

```
if (SzabadHely() > 0)
    JegyetVesz()
```

```
if (SzabadHely() > 0)
    JegyetVesz()
```

Az ellenőrzés és a cselekvés között eltelt időben a szabad helyek száma változhatott egy másik szál hatására

Versenyhelyzet - példa

```
static int sum = 0;

static void Main(string[] args)
{
    Thread th01 = new Thread(Szamol);
    Thread th02 = new Thread(Szamol);
    th01.Start();
    th02.Start();
    th01.Join();
    th02.Join();
    Console.WriteLine("Sum: " + sum);
    Console.ReadLine();
}

static void Szamol()
{
    for (int i = 0; i < 50000; i++)
    {
        sum = sum + 1;
    }
}
```

```
Sum: 54942
```

```
Sum: 56444
```

```
Sum: 82556
```

```
Sum: 100000
```

```
Sum: 66394
```

Versenyhelyzet - példa

```
static int sum = 0;

static void Main(string[] args)
{
    Task th01 = new Task(Szamol, TaskCreationOptions.LongRunning);
    Task th02 = new Task(Szamol, TaskCreationOptions.LongRunning);
    th01.Start();
    th02.Start();
    Task.WhenAll(th01, th02).ContinueWith(
                                x => Console.WriteLine("Sum: " + sum));
    Console.ReadLine();
}

static void Szamol()
{
    for (int i = 0; i < 50000; i++)
    {
        sum = sum + 1;
    }
}
```

```
Sum: 56444


Sum: 82556
```

Szinkronizáció

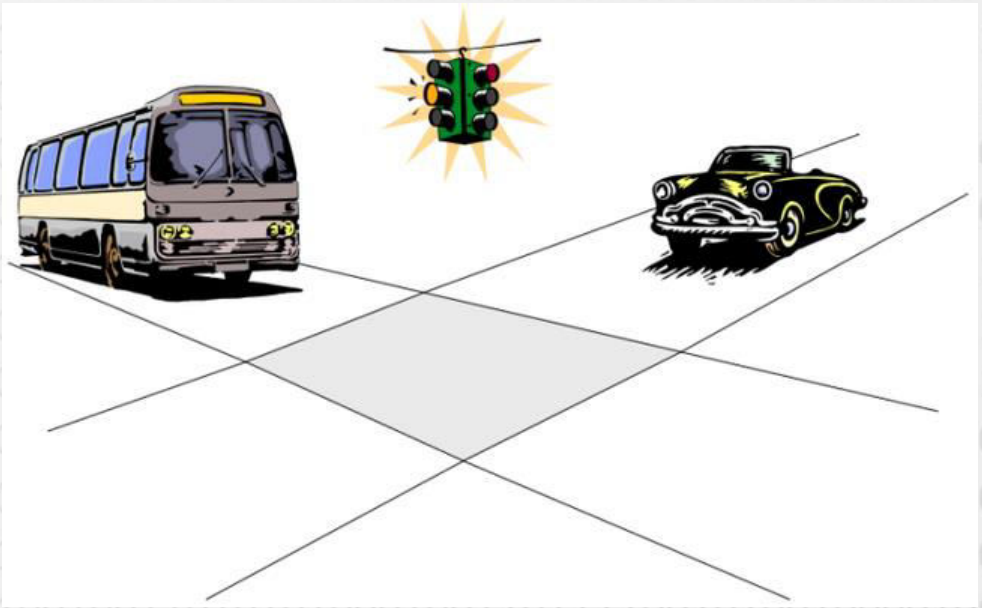
- **A szinkronizáció a feladatok végrehajtási sorrendjének érvényre juttatása, a relatív sorrend meghatározása**
- **Gyakran a közösen használt adatok köré összpontosul**
- **Két fő típus**
  - Kölcsönös kizárás (mutual exclusion, „mutex”)
  - Feltételes szinkronizáció

Kritikus szakasz

- A kód azon részét, ahol a közös függést okozó változók vannak, kritikus szakasznak nevezzük
- Többféle szinkronizációs megoldás létezik a kritikus szakaszok biztonságossá tételére, melyek célja, hogy a kritikus szakaszban csak egy szál legyen egy időben
- A fő kihívás olyan megoldás választása, amely amellett hogy biztonságos, hatékony is

Kritikus szakasz

- **A kritikus szakaszt szokás szinkronizációs blokknak is nevezni: belépési és kilépési pontja van**



Megoldások kölcsönös kizárásra

- **Tegyük fel, hogy két szál szeretné ugyanazt a közös változót használni:**
  - Előbb engedélyt kell kérjen a kritikus szakaszba lépésre – ha nem kapja meg, várakozik
  - A kritikus szakaszban elvégzi a szükséges műveleteket
  - Végül kilépés után engedélyt ad más várakozó szálaknak a belépésre

SZINKRONIZÁCIÓ

Lock

lock (object o) { … } struktúra C#-ban

- **Kölcsönös kizárás megvalósítása**
- **object o: szinkronizációs objektum**
  - Bármely közös változó lehet, referencia típusúnak kell lennie
  - Egyszerre csak egy szál szerezheti meg
- **Amelyik szálnál az objektum, az tud belépni ({) a kritikus szakaszba**
- **Kilépéskor (}) a lockot elengedi**
- **FIFO várósor: first-come first-serve**

Lock - példa

```
static int sum = 0;
static object lockObject = new object();

static void Main(string[] args)
{
    Task th01 = new Task(Szamol, TaskCreationOptions.LongRunning);
    Task th02 = new Task(Szamol, TaskCreationOptions.LongRunning);
    th01.Start();
    th02.Start();
    Task.WhenAll(th01, th02).ContinueWith(
                                x => Console.WriteLine("Sum: " + sum));
    Console.ReadLine();
}

static void Szamol()
{
    for (int i = 0; i < 50000; i++)
    {
        lock (lockObject) { sum = sum + 1; }
    }
}
```

```
Sum: 100000
Sum: 100000
   Sum: 100000
      Sum: 100000
```

Lock – példa #2

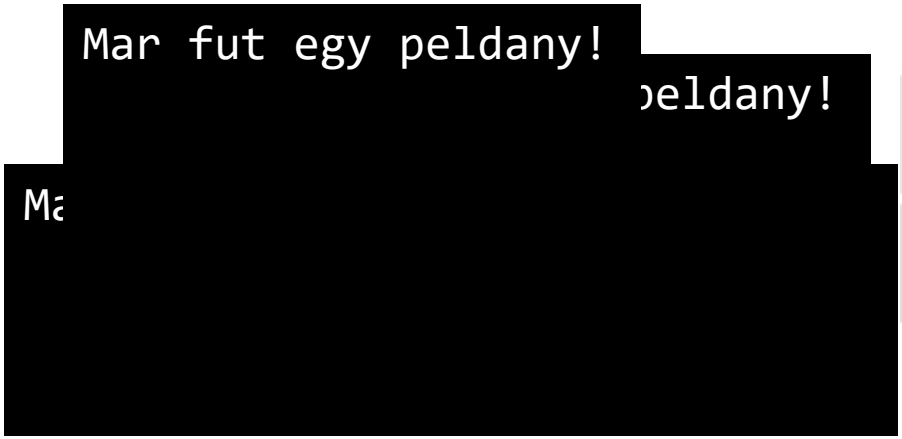
```
static int sum = 0;
static object lockObject = new object();

static void Main(string[] args)
{
    Task th01 = new Task(Szamol, TaskCreationOptions.LongRunning);
    Task th02 = new Task(Szamol, TaskCreationOptions.LongRunning);
    th01.Start();
    th02.Start();
    Task.WhenAll(th01, th02).ContinueWith(
                                    x => Console.WriteLine("Sum: " + sum));
    Console.ReadLine();
}

static void Szamol()
{
    lock (lockObject)
    {
        for (int i = 0; i < 50000; i++)
        {
            sum = sum + 1;
        }
    }
}
```

```
Sum: 100000
Sum: 100000
   Sum: 100000
      Sum: 100000
```

Szinkronizáció folyamatok között

- **Mutex**
  - Folyamatok közötti kölcsönös kizárásra
  - Mutex class, elnevezett példány

```
static void Main()
{
    using (var mutex = new Mutex(false, "csakegylehet"))
    {
        if (!mutex.WaitOne(TimeSpan.FromSeconds(3), false))
        {
            Console.WriteLine("Mar fut egy peldany!");
            return;
        }
        RunProgram();
    }
}
static void RunProgram()
{
    Console.WriteLine("asd");
    Console.ReadLine();
}
```



Atomicitás

- Atomi az a műveletsor, amely tovább nem osztható, nem megszakítható
- A rendszer további részeinek azonnaliként hat elvégzése
- Atomi az a művelet, amely végrehajtása nem megszakítható konkurrens műveletek által

```
Interlocked.Add(ref var, 1);
Interlocked.Increment(ref var);
Interlocked.Decrement(ref var);
Interlocked.CompareExchange(ref var, compare, change);
```

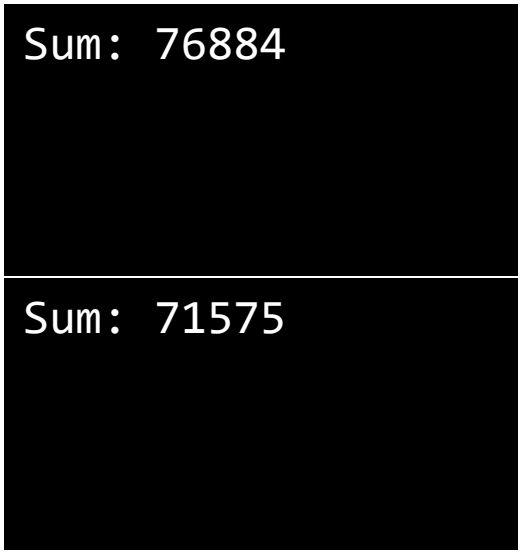
Atomi művelet - példa

```
static int sum = 0;

static void Main(string[] args)
{
    Task th01 = new Task(Szamol, TaskCreationOptions.LongRunning);
    Task th02 = new Task(Szamol, TaskCreationOptions.LongRunning);
    th01.Start();
    th02.Start();
    Task.WhenAll(th01, th02).ContinueWith(
                                x => Console.WriteLine("Sum: " + sum));
    Console.ReadLine();
}

static void Szamol()
{
    for (int i = 0; i < 50000; i++)
    {
        Interlocked.Increment(ref sum);
    }
}
```

```
Sum: 100000
Sum: 100000
Sum: 100000
Sum: 100000
```

Parallel.For() és versenyhelyzet - példa

```
static int sum = 0;

static void Main(string[] args)
{
    Parallel.For(0, 100000, i => sum += 1);
    Console.WriteLine("Sum: " + sum);
    Console.ReadLine();
}
```



Parallel.For() és lock - példa

```
static int sum = 0;

static void Main(string[] args)
{
    Parallel.For(0, 100000, i => { lock (lockObject) sum += 1; });
    Console.WriteLine("Sum: " + sum);
    Console.ReadLine();
}
```

```
Sum: 100000
```

```
Sum: 100000
```

Parallel.For() és atomi művelet - példa

```
static int sum = 0;

static void Main(string[] args)
{
    Parallel.For(0, 100000, i => Interlocked.Increment(ref sum));
    Console.WriteLine("Sum: " + sum);
    Console.ReadLine();
}
```

```
Sum: 100000
```

```
Sum: 100000
```

Szemafor

- **Edsger Dijkstra, 1964**
- **A vasúti terminológia alapján**
  - A sínszakasz a kritikus szakasz
  - A vonat akkor léphet be, ha a jelzőkar nyitott állapotú
  - Amikor a vonat belép, a jelzőkart zártra állítja
  - Kilépéskor a jelzőkart átbillenti nyitottra
- **Dijkstra P() és V() atomi műveletekkel írta ezt le**
  - P(): belépéskor
  - V(): kilépéskor

Szemafor

- **Egy egész értéken végzett atomi inkrementálás / dekrementálás**
  - Belépéskor az érték csökkentése, és értékvizsgálata
    - Ha az érték kisebb mint zéró, akkor a szál blokkolódik
  - Kilépéskor inkrementálás
    - Ha van várakozó szál, akkor a *következő* értesítése
- **Alkalmazható kölcsönös kizárásra is**
  - Ezt bináris szemafornak nevezik
- **Ugyanakkor alkalmas arra is, hogy egyidejűleg egynél több szálat engedjen be a kritikus szabaszba**
  - Felső korlát adható

Szemafor a C#ban

- **Semaphore és SemaphoreSlim (>= .NET 4.0)**
  - A Semaphore folyamatok között is alkalmazható, de lassabb

```
static SemaphoreSlim _sem = new SemaphoreSlim(3);
static void Main()
{
    for (int i = 1; i <= 5; i++)
        new Thread(Enter).Start(i);
}
static void Enter(object id)
{
    Console.WriteLine(id + " wants to enter");
    _sem.Wait();
    Console.WriteLine(id + " is in!");
    Thread.Sleep(1000 * (int)id);
    Console.WriteLine(id + " is leaving");
    _sem.Release();
}
```

```
1 wants to enter
1 is in!
2 wants to enter
2 is in!
3 wants to enter
3 is in!
4 wants to enter
5 wants to enter
1 is leaving
4 is in!
2 is leaving
5 is in!
```

HOLTPONT

Holtpont

- **Holtpont akkor következik be, ha egy szál blokkolva van, és a blokkolás sosem szűnik meg**
- **Azaz, ha egy szál kritikus szakaszhoz ér, de sem tud belépni**
- **Körülményektől függően többféle holtpont definiálható**
  - Ön-holtpont
    - A szál olyan lock-ra vár, amely már az övé
  - Rekurzív holtpont
    - A kritikus szakaszban levő szál olyan funkciót próbál elérni, amely visszahivatkozik a lockkal védett részre
  - Zársorrend holtpont (lock-order deadlock)
    - Két szál, két lock: egyiknél egyik, a másiknál másik, és egymásra várnak


11455

Holtpont

- **Klasszikus holtpont példa:**
  - Erőforrásokat lockkal próbáltunk védeni. Mindkét szál már belépett az első kritikus szakaszba, ott viszont várakoznia kell, hogy hozzájusson a másik lehetséges erőforráshoz.

1. szál

```
1  lock (a)
2  {
3    // feldolgozás
4    lock (b)
5    {
6    // feldolgozás
7    }
8  }
```

2. szál

```
1  lock (b)
2  {
3    // feldolgozás
4    lock (a)
5    {
6    // feldolgozás
7    }
8  }
```

Holtpont

```
static class DemoStaticClass
{
    static int idx;
    static DemoStaticClass()
    {
        Task.Run(() => Valami()).Wait();
    }

    public static void Valami()
    { idx++; Console.WriteLine("valami"); }
}

//...
DemoStaticClass.Valami();
```

A statikus konstruktor az osztály első elérése előtt fut le.

1. A DemoStaticClass.Valami() előtt lefutna a static konstruktor
2. Ami ismét a Valami() metódusra hivatkozik, és vár az eredményére: ez blokkol, hisz a statikus konstruktor még nem végzett

Az eredmény holtpont.

Holtpont

```
Process p = new Process()
{
    StartInfo = new ProcessStartInfo("soksokkimenet.exe")
    {
        RedirectStandardOutput = true,
        UseShellExecute = false,
        CreateNoWindow = true
    }
};

p.Start();
p.WaitForExit();
Console.WriteLine(p.StandardOutput.ReadToEnd());
```

A folyamatok átirányított kimenete véges méretű bufferrel rendelkezik.

1. Ha a buffer megtelik, és a folyamat írna rá, várakozni kényszerül arra, hogy a buffer ürüljön
2. A gazdafolyamat azonban arra vár hogy a folyamat végezzen, és csak utána olvassa ki a buffer tartalmát

Az eredmény holtpont.

Versenyhelyzet és holtpont kivédése

- **Versenyhelyzet akkor áll fenn, amikor több szál próbál egyszerre azonos adathoz hozzáférni**
  - Ha megoldható, ne használjunk a szálak számára közös objektumokat (referenciatípusok esetén különösen figyelni!) – Bizonyos feladatok/adatok esetén értéktípusok megfontolandók lehetnek
  - Ha ez nem megoldható, akkor szálszinkronizáció
  - De: OK a szálak számára közös adat, HA az elérés minden szál részéről read-only
- **A holtpont kivédésére nincs megoldás (alapos tervezés!)**

Források

- Szabó-Resch Miklós Zsolt és Cseri Orsolya Eszter Haladó Programozás előadásfóliái
- Kertész Gábor Párhuzamos és Elosztott Rendszerek Programozása előadásfóliái
- MSDN