Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

Szoftvertechnikák laborgyakorlat

4. mérés

Többszálú alkalmazások készítése

Thread, ThreadPool, szinkronizáció, ManualResetEvent,
WaitHandle, Invoke

A gyakorlatot kidolgozta: Szabó Zoltán, Benedek Zoltán, Simon Gábor, Kővári Bence
Utolsó módosítás ideje: 2020.04.04.

Tartalom

A gyakorlat célja

A kapcsolódó előadások: Konkurens (többszálú) alkalmazások fejlesztése

A gyakorlat célja, hogy megismertesse a hallgatókat a többszálas programozás során követendő alapelvekkel. A gyakorlat során a hallgatók megismerkednek a szálak indításával, leállításával, a szálbiztos osztályok készítésével, a ThreadPool szálak kezelésével, a szálak szinkronizálásával, és a Windows Forms szálkezelési sajátosságaival. Természetesen, mivel a témakör hatalmas, így csak alapszintű tudást fognak szerezni, de e tudás birtokában már képesek lesznek önállóan is elindulni a bonyolultabb feladatok megvalósításában.

Bevezető

A párhuzamosan futó szálak kezelése kiemelt fontosságú terület, melyet miden szoftverfejlesztőnek legalább alapszinten ismernie kell. A gyakorlat során alapszintű, de kiemelt fontosságú problémákat oldunk meg, ezért törekednünk kell arra, hogy ne csak a végeredményt, hanem az elvégzett módosítások értelmét és indokait is megértsük.

A feladat során egyszerű Windows Forms alkalmazást fogunk felruházni többszálas képességekkel, egyre komplexebb feladatokat megoldva. Az alapprobléma az, hogy van egy függvényünk, ami hosszú ideig fut, s mint látni fogjuk, ennek „direktben” történő hívása a felületről kellemetlen következményekkel jár.

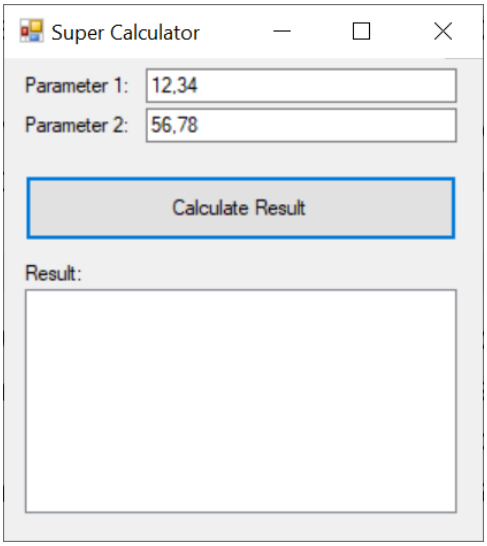
A megoldás során egy meglévő alkalmazást fogunk kiegészíteni saját kódrészletekkel. Az újonnan beszúrandó részeket az útmutatóban külön kiemeltük szürke háttérrel.

0. feladat – Ismerkedés a kiinduló alkalmazással, előkészítés

Töltsük le és csomagoljuk ki a 4. gyakorlathoz tartozó kiinduló alkalmazást a tárgyhonlapról a „Gyakorlat anyagok” szekcióból („Labor anyag (a labor idejére) - 4. Többszálú alkalmazások készítése”), és nyissuk meg solution-t Visual Studio-ban.

A feladatunk az, hogy egy bináris formában megkapott algoritmus futtatásához Windows Forms technológiával felhasználói felületet készítsünk. A bináris forma .NET esetében egy dll kiterjesztésű fájlt jelent, ami programozói szemmel egy osztálykönyvtár. A fájl neve esetünkben Algorithms.dll, megtalálható a letöltött zip állományban.

A kiinduló alkalmazásban a felhasználói felület elő is van készítve. Futtassuk az alkalmazást:



Az alkalmazás felületén meg tudjuk adni az algoritmus bemenő paramétereit (double számok halmaza): a példánkban mindig két double szám paraméterrel hívjuk az algoritmust, ezt a két felső szövegmezőben lehet megadni. A feladatunk az, hogy a „Calculate Result” gombra kattintás során futtassuk az algoritmust a megadott paraméterekkel, majd, ha végzett, akkor a Result alatti listázó mező új sorában jelenítsük meg a kapott eredményt a bemenő paraméterekkel együtt.

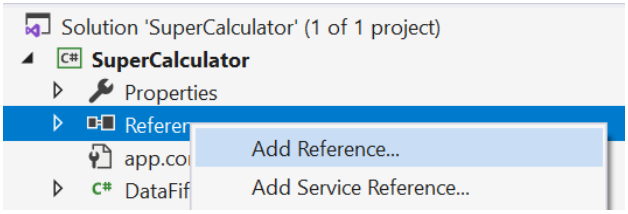
Következő lépésben ismerkedjünk meg a letöltött Visual Studio solutionnel:

1. Nézzük végig a MainForm osztályt. Az látjuk, hogy a felület alapvetően kész, csak az algoritmus futtatása hiányzik. Az eredmény és a paraméterei naplózásához is találunk egy ShowResult nevű segédfüggvényt.
2. A DataFifo-t egyelőre hagyjuk ki, csak a gyakorlat második felében fogjuk használni, majd később megismerkedünk vele.

A DLL-ben levő kód felhasználása

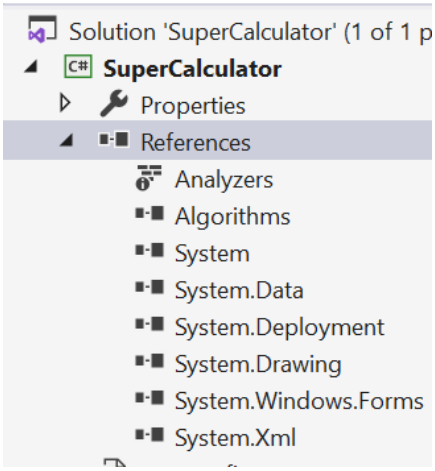
A mellékelt kiindulási solution mappájában megtaláljuk a *Algorithm.dll*-t. Ebben lefordított formában egy *Algorithms* névtérben levő *SuperAlgorithm* nevű osztály található, melynek egy *Calculate* nevű statikus művelete van. Ahhoz, hogy egy projektben fel tudjuk használni a DLL-ben levő osztályokat, a DLL-re a Visual Studio környezetünkben egy ún. **referenciát** kell tegyünk.

1. Solution Explorerben a projektünk „References” node-jára jobbklikkelve válasszuk az Add reference opciót!



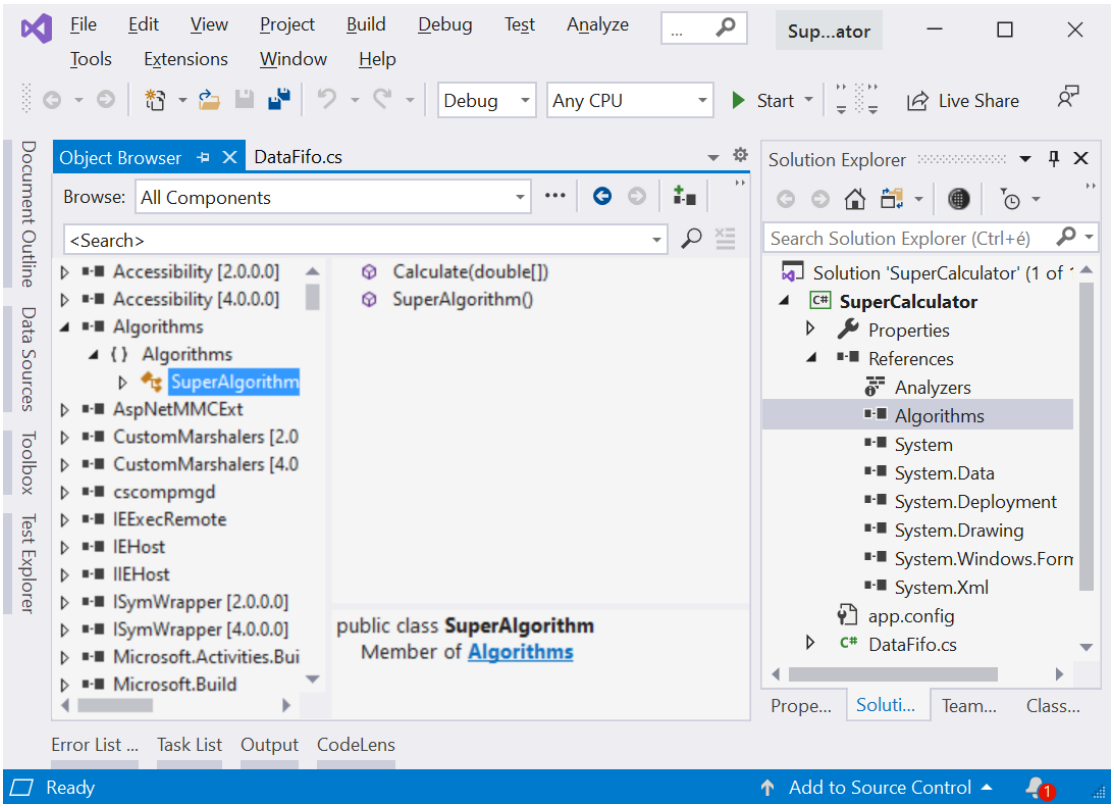
2. Az előugró ablak jobb alsó sarokban található „Browse” gomb segítségével keressük meg és válasszuk ki projekt mappájában található „Algorithms.dll” fájlt, majd hagyjuk jóvá a hozzáadást az OK gombbal!

A Solution Explorerben egy projekt alatti References csomópontot lenyitva láthatjuk a hivatkozott DLL-eket. Itt most már megjelenik az előbb felvett *Algortihms* referencia is. A System kezdetűek a .NET Framework beépített osztályainak implementációját tartalmazó DLL-ek hivatkozásai.



Kattintsunk duplán az *Algorithms* referencián a listában. Ekkor megnyílik az *Object Browser* tabfül, ahol megtekinthetjük, hogy az adott DLL-ben milyen névterek, osztályok találhatók, illetve ezeknek milyen tagjaik (tagváltozó, tagfüggvény, property, event) vannak. Ezeket a Visual Studio a DLL metaadataiból az ún. *reflection* mechanizmus segítségével olvassa ki (ilyen kódot akár mi is írhatunk).

Az alábbi ábrának megfelelően az Object Browserben baloldalt keressük ki az *Algorithms* csomópontot, nyissuk le, és láthatóvá válik, hogy egy *Algorithms* névtér van benne, abban pedig egy Super*Algorithm* osztály. Ezt kiválasztva középen megjelennek az osztály függvényei, itt egy függvényt kiválasztva pedig az adott függvény pontos szignatúrája:



1. Feladat – Művelet futtatása a főszálon

Most már rátérhetünk az algoritmus futtatására. Első lépésben ezt az alkalmazásunk fő szálán tesszük meg.

1. A főablakon lévő gomb Click eseménykezelőjében hívjuk meg a számoló függvényünket. Ehhez kattintsunk a Solution Explorerben duplán a MainForm.cs fájlra, majd a megjelenő Form Designer-ben a Calculate Result gombra. Egészítsük ki a kódot az újonnan behivatkozott algoritmus meghívásával.

```
private void buttonCalcResult_Click(object sender, EventArgs e)
{
  double p1, p2 = 0;
  if (Double.TryParse(textBoxParam1.Text, out p1) &&
    (Double.TryParse(textBoxParam2.Text, out p2)))
  {
      double[] parameters = new double[] { p1, p2 };
      double result = Algorithms.SuperAlgorithm.Calculate(parameters);
       ShowResult(parameters, result);
   }
   else MessageBox.Show(this, "Invalid parameter!", "Error");
}
```

2. Próbáljuk ki az alkalmazást, és vegyük észre, hogy az ablak a számolás ideje alatt nem reagál a mozgatásra, átméretezésre, a felület gyakorlatilag befagy.

Az alkalmazásunk eseményvezérelt, mint minden Windows alkalmazás. Az operációs rendszer a különböző interakciókról (pl. mozgatás, átméretezés) üzenetekben értesíti az alkalmazásunkat. Mivel a gombnyomást követően az alkalmazásunk egyetlen szála

a kalkulációval van elfoglalva, nem tudja azonnal feldolgozni a további felhasználói utasításokat. Amint a számítás lefutott (és az eredmények megjelennek a listában) a korábban kapott parancsok is végrehajtásra kerülnek.

2. Feladat – Végezzük a számítást külön szálban

Következő lépésben a számítás elvégzésére egy külön szálat fogunk indítani, hogy az ne blokkolja a felhasználói felületet.

1. Készítsünk egy új függvényt a MainForm osztályban, mely a feldolgozó szál belépési pontja lesz.

```
private void CalculatorThread(object arg)
{
      double[] parameters = (double[])arg;
      double result = Algorithms.SuperAlgorithm.Calculate(parameters);
      ShowResult(parameters, result);
}
```

2. Indítsuk el a szálat a gomb Click eseménykezelőjében. Ehhez cseréljük le a korábban hozzáadott kódot:

```
private void buttonCalcResult_Click(object sender, EventArgs e)
{
   double p1, p2 = 0;
   if (Double.TryParse(textBoxParam1.Text, out p1) &&
      (Double.TryParse(textBoxParam2.Text, out p2)))
   {
      double[] parameters = new double[] { p1, p2 };
      Thread th = new Thread(CalculatorThread);
      th.Start(parameters);
   }
   else MessageBox.Show(this, "Invalid parameter!", "Error");
}
```

A Thread objektum Start műveletében átadott paramétert kapja meg a CalculatorThread szálfüggvényünk.

3. Futtassuk az alkalmazást F5-tel! „InvalidOperationException, Cross-thread operation not valid” hibaüzenetet kapunk a ShowResult metódusban, ugyanis nem abból a szálból próbálunk hozzáférni a vezérlőhöz, amelyik létrehozta (a vezérlőt). A következő feladatban ezt a problémát oldjuk meg.

   A problémát a következő okozza. Windows Forms alkalmazásoknál él az alábbi szabály: az űrlapok/vezérlőelemek alapvetően nem szálvédett objektumok, így **egy űrlaphoz/vezérlőhöz csak abból a szálból szabad hozzáférni (pl. property-ét olvasni, állítani, műveletét meghívni), amelyik szál az adott űrlapot/vezérlőt létrehozta**. **Máskülönben kivételt kapunk.** Alkalmazásunkban azért kaptunk kivételt, mert a *listViewResult* vezérlőt a fő szálban hoztuk létre, a ShowResult metódusban az eredmény megjelenítésekor viszont egy másik szálból férünk hozzá (`listViewResult.Items.Add`). A fenti szabály alól van pár kivétel, ilyen pl. a *Control* osztályban definiált *InvokeRequired* property és *Invoke* metódusa bármely szálból biztonságosan elérhető. Az *InvokeRequired* tulajdonság értéke igaz, ha nem

a vezérlőelemet létrehozó szálból kérdezzük le az értékét, egyébként hamis. Az *Invoke* metódus pedig a vezérlőelemet létrehozó szálon futtatja le a paraméterként megadott metódust. Az InvokeRequired és a Invoke felhasználásával el tudjuk kerülni korábbi kivételünket, ezt fogjuk a következőkben megtenni.

3. Feladat – Tegyük szálbiztossá a ShowResult metódust (Invoke)

Módosítaniuk kell a ShowResult metódust, hogy mellékszálból történő hívás esetén se dobjon exceptiont.

1. Vegyünk fel egy új delegate típust MainForm.cs-ben, magában a MainForm osztályban.

```
private delegate void AddResultToList(double[] parameters, double result);
```

2. Egészítsük ki a ShowResult függvényt, majd futtassuk ismét

```
private void ShowResult(double[] parameters, double result)
{
   if (InvokeRequired)
   {
       Invoke(new AddResultToList(ShowResult),
                                       new object[] { parameters, result });
   }
   else if (!IsDisposed)
   {
       ListViewItem lvi = listViewResult.Items.Add(
                   parameters[0] + " # " + parameters[1] + " = " + result);
       listViewResult.EnsureVisible(lvi.Index);
       listViewResult.AutoResizeColumns(
                       ColumnHeaderAutoResizeStyle.ColumnContent);
   }
}
```

Ez a megoldás már működőképes. A Form osztály InvokeRequired metódusa igazat ad vissza, amennyiben nem az őt létrehozó szálból hívjuk meg. Ilyen esetekben a Form-ot az Invoke metódusán keresztül tudjuk megkérni, hogy egy adott műveletet a saját szálán (amelyik a Form-ot létrehozta, ez a legtöbb alkalmazásban a fő szál) hajtson végre. A fenti példában tulajdonképpen a ShowResult függvény önmagát hívja meg még egyszer, csak második esetben már a Form saját szálán. Ez egy bevett minta a redundáns kódok elkerülésére. Tegyünk töréspontot a ShowResult művelet első sorára, és az alkalmazást futtatva győződjünk meg, hogy a ShowResult művelet – különösen az Invoke tekintetében – a fentiekben ismertetetteknek megfelelően működik.

Vegyük ki a töréspontot, így futtassuk az alkalmazást: vegyük észre, hogy amíg egy számítás fut, újabbakat is indíthatunk, hiszen a felületünk végig reszponzív maradt.

4. feladat – Művelet végzése Threadpool szálon

Az előző megoldás egy jellemzője, hogy mindig új szálat hoz létre a művelethez. Esetünkben ennek nincs különösebb jelentősége, de ez a megközelítés egy olyan kiszolgáló alkalmazás esetében, amely nagyszámú kérést szolgál ki úgy, hogy minden kéréshez külön szálat indít, már problémás lehet. Két okból is:

- Ha a szálfüggvény gyorsan lefut (egy kliens kiszolgálása gyors), akkor a CPU nagy részét arra pazaroljuk, hogy szálakat indítsunk és állítsunk le, ezek ugyanis önmagukban is erőforrásigényesek.
- Túl nagy számú szál is létrejöhet, ennyit kell ütemeznie az operációs rendszernek, ami feleslegesen pazarolja az erőforrásokat.

Egy másik probléma jelen megoldásunkkal: mivel a számítás előtérszálon fut (az újonnan létrehozott szálak alapértelmezésben előtérszálak), hiába zárjuk be az alkalmazást, a program tovább fut a háttérben mindaddig, amíg végre nem hajtódik az utoljára indított számolás is: egy processz futása ugyanis csak akkor fejeződik csak be, ha már nincs futó előtérszála.

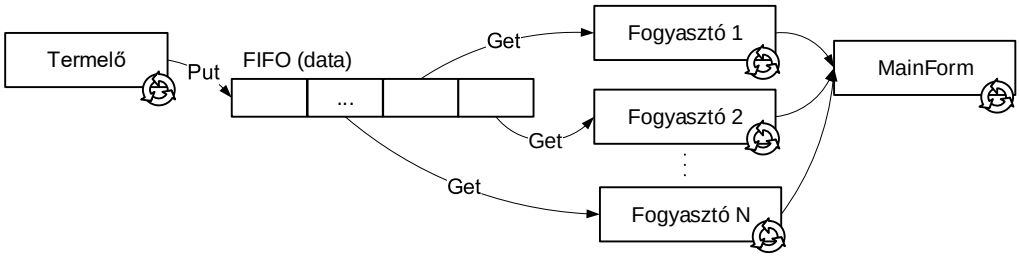
Módosítsuk a gomb eseménykezelőjét, hogy threadpool szálon futtassa a számítást. Ehhez csak a gombnyomás eseménykezelőjét kell ismét átírni.

```
private void buttonCalcResult_Click(object sender, EventArgs e)
{
  double p1, p2 = 0;
  if (Double.TryParse(textBoxParam1.Text, out p1) &&
     (Double.TryParse(textBoxParam2.Text, out p2)))
  {
     double[] parameters = new double[] { p1, p2 };
     ThreadPool.QueueUserWorkItem(CalculatorThread, parameters);
  }
  else MessageBox.Show(this, "Invalid parameter!", "Error");
}
```

Próbáljuk ki az alkalmazást, és vegyük észre, hogy az alkalmazás az ablak bezárásakor azonnal leáll, nem foglalkozik az esetlegesen még futó szálakkal (mert a threadpool szálak háttér szálak).

5. Feladat – Termelő-fogyasztó alapú megoldás

Az előző feladatok megoldása során önmagában egy jól működő komplett megoldását kaptuk az eredeti problémának, mely lehetővé teszi, hogy akár több munkaszál is párhuzamosan dolgozzon a háttérben a számításon, ha a gombot sokszor egymás után megnyomjuk. A következőkben úgy fogjuk módosítani az alkalmazásunkat, hogy a gombnyomásra ne mindig keletkezzen új szál, hanem a feladatok bekerüljenek egy feladatsorba, ahonnan több, a háttérben folyamatosan futó szál egymás után fogja kivenni őket és végrehajtani. Ez a feladat a klasszikus termelő-fogyasztó probléma, mely a gyakorlatban is sokszor előfordul, a működését az alábbi ábra szemlélteti:



A főszálunk a termelő, a „Calculate result” gombra kattintva hoz létre egy új feladatot. Fogyasztó/feldolgozó/munkaszálból azért indítunk majd többet, mert így több CPU magot is ki tudunk használni, valamint a feladatok végrehajtását párhuzamosítani tudjuk.

A feladatok ideiglenes tárolására a kiinduló solution-ünkben már némiképpen előkészített DataFifo osztályt tudjuk használni. Nézzük meg a forráskódját. Egy egyszerű FIFO sort valósít meg, melyben double[] elemeket tárol. A Put metódus hozzáfűzi a belső lista végéhez az új párokat, míg a TryGet metódus visszaadja (és eltávolítja) a belső lista első elemét. Amennyiben a lista üres, a függvény nem tud visszaadni elemet. Ilyenkor a false visszatérési értékkel jelzi ezt.

1. Hozzunk létre egy DataFifo példányt a Form-on belül:

```
private DataFifo fifo = new DataFifo();
```

2. Módosítsuk a gomb eseménykezelőjét az alábbiak szerint:

```
private void buttonCalcResult_Click(object sender, EventArgs e)
{
  double p1, p2 = 0;
  if (Double.TryParse(textBoxParam1.Text, out p1) &&
     (Double.TryParse(textBoxParam2.Text, out p2)))
  {
     double[] parameters = new double[] { p1, p2 };
     fifo.Put(parameters);
  }
  else MessageBox.Show(this, "Invalid parameter!", "Error");
}
```

3. Készítsük el az új szálkezelő függvényt naív implementációját az űrlap osztályunkban:

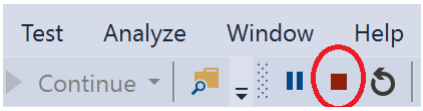
```
private void WorkerThread()
{
   double[] data = null;
   while (true)
   {
      if (fifo.TryGet(out data))
      {
         double result = Algorithms.SuperAlgorithm.Calculate(data);
         ShowResult(data, result);
      }
      Thread.Sleep(500);
   }
}
```

A Thread.Sleep bevezetésére azért van szükség, mert e nélkül a munkaszálak üres FIFO esetén folyamatosan feleslegesen pörögnének, semmi hasznos műveletet nem végezve is 100%-ban kiterhelnének egy-egy CPU magot. Megoldásunk nem ideális, később tovább fejlesztjük.

4. Hozzuk létre és indítsuk el a feldolgozó szálakat a konstruktorban:

```
public MainForm()
{
  ...
   new Thread(WorkerThread) { Name = "Szal1" }.Start();
   new Thread(WorkerThread) { Name = "Szal2" }.Start();
   new Thread(WorkerThread) { Name = "Szal3" }.Start();
}
```

5. Indítsuk el az alkalmazást, majd zárjuk is be azonnal anélkül, hogy a Calculate Result gombra kattintanánk. Az tapasztaljuk, hogy az ablakunk bezáródik ugyan, de a processzünk tovább fut, az alkalmazás bezárására csak a Visual Studioból, vagy a Task Manager-ből van lehetőség:



A feldolgozó szálak előtérszálak, kilépéskor megakadályozzák a processz megszűnését. Az egyik megoldás az lehetne, ha a szálak IsBackground tulajdonságát true-ba állítanánk a létrehozásukat követően. A másik megoldás, hogy kilépéskor gondoskodunk a feldolgozó szálak kiléptetéséről. Egyelőre tegyük félre ezt a problémát, később visszatérünk rá.

Indítsuk el az alkalmazást azt tapasztaljuk, hogy miután kattintunk a Calculate Result gombon nagy valószínűséggel kivételt fogunk kapni. A probléma az, hogy a DataFifo nem szálbiztos, inkonzisztensé vált. Két eredő ok is húzódik a háttérben:

Probléma 1

Nézzük a következő forgatókönyvet:

1. A sor üres. A feldolgozó szálak egy while ciklusban folyamatosan pollozzák a fifo-t, vagyis hívják a TryGet metódusát.
2. A felhasználó egy feladatot tesz a sorba.
3. Az egyik feldolgozó szál a TryGet metódusban azt látja, van adat a sorban, vagyis „if (innerList.Count > 0)” kódsor feltétele teljesül, és rálép a következő kódsorra. Tegyük fel, hogy ez a szál ebben a pillanatban elveszti a futási jogát, már nincs ideje kivenni az adatot a sorból.
4. Egy másik feldolgozó szál is éppen ekkor ejti meg az „if (innerList.Count > 0)” vizsgálatot, nála is teljesül a feltétel, és ez a szál ki is veszi az adatot a sorból.
5. Az első szálunk újra ütemezésre kerül, felébred, ő is megpróbálja kivenni az adatot a sorból: a sor viszont már üres, a másik szálunk kivette az egyetlen adatot a sorból az orra előtt. Így a innerList[0] hozzáférés kivételt eredményez.

Ezt a problémát csak úgy tudjuk elkerülni, ha a sor ürességének a vizsgálatát és az elem kivételét oszthatatlanná tesszük.

Az ürességvizsgálatot figyelő kódsort követő „Thread.Sleep(500);” kódsornak csak az a szerepe a példakódunkban, hogy a fenti peches forgatókönyv bekövetkezésének a valószínűségét megnövelje, s így a példát szemléletesebbé tegye. A későbbiekben ezt ki is fogjuk venni, egyelőre hagyjuk benne.

Probléma 2

A DataFifo osztály egyidőben több szálból is hozzáférhet a List<double[]> típusú innerList tagváltozóhoz. Ugyanakkor, ha megnézzük a **List<T>** dokumentációját, azt találjuk, hogy az **osztály nem szálbiztos (not thread safe)**. Ez esetben viszont ez nem tehetjük meg, nekünk kell zárakkal biztosítanunk, hogy a kódunk egyidőben csak egy metódusához/tulajdonságához/tagváltozójához fér hozzá (pontosabban inkonzisztencia csak egyidejű írás, illetve egyidejű írás és olvasás esetén léphet fel, de az írókat és az olvasókat a legtöbb esetben nem szoktuk megkülönböztetni, itt sem tesszük).

A következő lépésben a DataFifo osztályunkat szálbiztossá tesszük, amivel megakadályozzuk, hogy a fenti két probléma bekövetkezhessen.

6. feladat – Tegyük szábiztossá a DataFifo osztályt

A DataFifo osztály szálbiztossá tételéhez szükségünk van egy objektumra (ez bármilyen referencia típusú objektum lehet), amit kulcsként használhatunk a zárolásnál. Ezt követően a lock kulcsszó segítségével el tudjuk érni, hogy egyszerre mindig csak egy szál tartózkodjon az adott kulccsal védett blokkokban.

1. Vegyünk fel egy object típusú mezőt syncRoot néven a DataFifo osztályba.

```
private object syncRoot = new object();
```

2. Egészítsük ki a Put és a TryGet függvényeket a zárolással.

```
public void Put(double[] data)
{
  lock (syncRoot)
  {
     innerList.Add(data);
  }
}
```

```
public bool TryGet(out double[] data)
{
  data = null;
  lock (syncRoot)
  {
     if (innerList.Count > 0)
     {
        Thread.Sleep(500);
        data = innerList[0];
        innerList.RemoveAt(0);
        return true;
     }
     else return false;
  }
}
```

Most már nem szabad kivételt kapnunk. Ki is vehetjük a TryGet metódusból a mesterséges késleltetést (Thread.Sleep(500); sor).

7. feladat – Hatékony jelzés megvalósítása (ManualResetEvent használata)

A WorkerThread-ben folyamatosan futó while ciklus ún. aktív várakozást valósít meg, ami mindig kerülendő. Ha a Thread.Sleep-et nem tettük volna a ciklusmagba, akkor ezzel maximumra ki is terhelné a processzort. A Thread.Sleep megoldja ugyan a processzor terhelés problémát, de bevezet egy másikat: ha mindhárom munkaszálunk éppen alvó állapotba lépett, mikor beérkezik egy új adat, akkor feleslegesen várunk 500 ms-ot az adat feldolgozásának megkezdéséig.

A következőkben úgy fogjuk módosítani az alkalmazást, hogy blokkolva várakozzon, amíg adat nem kerül a fifo-ba. Annak jelzésére, hogy van-e adat a sorban egy ManualResetEvent-et fogunk használni.

1. Adjunk hozzá egy MaunalResetEvent példányt a DataFifo osztályunkhoz hasData néven.

```
private ManualResetEvent hasData = new ManualResetEvent(false);
```

2. A hasData alkalmazásunkban kapuként viselkedik. Amikor adat kerül a listába „kinyitjuk”, míg amikor kiürül a lista „bezárjuk”.

```
public void Put(double[] data)
{
  lock (syncRoot)
  {
     innerList.Add(data);
     hasData.Set();
  }
}
```

```
public bool TryGet(out double[] data)
{
  data = null;
  lock (syncRoot)
  {
     if (innerList.Count > 0)
     {
        data = innerList[0];
        innerList.RemoveAt(0);
        if (innerList.Count == 0)
           hasData.Reset();
        return true;
     }
     else return false;
  }
}
```

8. feladat – Jelzésre várakozás (blokkoló a Get)

Az előző pontban megoldottuk a jelzést, ám ez önmagában nem sokat ér, hiszen nem várakoznak rá. Ennek megvalósítása jön most.

1. Módosítsuk a metódust az alábbiak szerint (kidobjuk az üresség vizsgálatot és az eseményre való várakozással pótoljuk).

```
public bool TryGet(out double[] data)
{
  data = null;
  lock (syncRoot)
  {
     if (hasData.WaitOne())
     {
        data = innerList[0];
        innerList.RemoveAt(0);
        if (innerList.Count == 0)
           hasData.Reset();
        return true;
     }
     else return false;
  }
}
```

2. Ezzel a Thread.Sleep a WorkerThread-ben feleslegessé vált, kommentezzük ki!

A fenti megoldás futtatásakor azt tapasztaljuk, hogy az alkalmazásunk felülete az első gombnyomást követően befagy. Az előző megoldásunkban ugyanis egy amatőr hibát követtünk el. A lock-olt kódrészleten belül várakozunk a hasData jelzésére, így a főszálnak lehetősége sincs arra, hogy a Put műveletben (egy szintén lock-kal védet részen belül) jelzést küldjön hasData-val. Gyakorlatilag **egy holtpont (deadlock) helyzet alakult ki**.

Gyors hibajavításként megadhatunk egy időkorlátot a várakozásnál:

```
      if (hasData.WaitOne(100))
```

Teszteljük az alkalmazást! A megoldás ugyan fut, de az elegáns és követendő minta az, hogy lock-on belül kerüljük a blokkolva várakozást.

Valódi javításként cseréljük meg a lock-ot és a WaitOne-t:

```
public bool TryGet(out double[] data)
{
  data = null;
  if (hasData.WaitOne())
  {
     lock (syncRoot)
     {
        data = innerList[0];
        innerList.RemoveAt(0);
        if (innerList.Count == 0)
           hasData.Reset();
        return true;
     }
  }
  return false;
}
```

Így elkerüljük a deadlock-ot, **azonban a szálbiztosság sérült**, hiszen mire a lock-on belülre jutunk, nem biztos, hogy maradt elem a listában. Ugyanis lehet, több szál is várakozik a hasData.WaitOne() műveletnél arra, hogy elem kerüljön a sorba. Mikor ez bekövetkezik, a ManualResetEvent objektumunk mind átengedi (hacsak éppen gyorsan le nem csukja egy szál, de ez nem garantált).

3. Javításként tegyük vissza a lock-on belüli üresség-vizsgálatot.

```
data = null;
lock (syncRoot)
{
  if (innerList.Count > 0)
  {
     data = innerList[0];
     innerList.RemoveAt(0);
     if (innerList.Count == 0)
        hasData.Reset();
     return true;
  }
  else {return false;}
}
```

Ez már jól működik. Előfordulhat ugyan, hogy feleslegesen fordulunk a listához, de ezzel így most megelégszünk.

Teszteljük az alkalmazást.

9. Feladat – Kulturált leállás

Korábban félretettük azt a problémát, hogy az ablakunk bezárásakor a processzünk „beragad”, ugyanis a feldolgozó munkaszálak előtérszálak, kiléptetésüket eddig nem oldottuk meg. Célunk, hogy a végtelen while ciklust kiváltva a munkaszálaink az alkalmazás bezárásakor kulturált módon álljanak le.

1. Egy ManualResetEvent segítségével jelezzük a leállítást a fifo-nak. A fifo-ban vegyünk fel egy új eseményt, és vezessünk be egy SignalRelaseWorkers műveletet, mellyel az új eseményünk jelzett állapotba állítható.

```
private ManualResetEvent relaseWorkers = new ManualResetEvent(false);

public void SignalRelaseWorkers()
{
     relaseWorkers.Set();
}
```

2. A TryGet-ben erre az eseményre is várakozzunk. A WaitAny metódus akkor engedi tovább a futtatást, ha a paraméterként megadott WaitHandle típusú objektumok közül valamelyik jelzett állapotba kerül és visszaadja annak tömbbeli indexét.

```
public bool TryGet(out double[] data)
{
  data = null;
  if (WaitHandle.WaitAny(new WaitHandle[] { hasData, relaseWorkers }) == 0)
  {
     lock (syncRoot)
     ...
```

3. MainForm-ban vegyünk fel egy flag tagváltozót a bezárás jelzésére:

```
bool closing = false;
```

4. A form bezárásakor állítsuk jelzettre az új eseményt és billentsünk be be a flag-et is. (A Form osztály OnClosed metódusa mindig meghívódik bezáráskor.)

```
protected override void OnClosed(EventArgs e)
{
     base.OnClosed(e);
     closing = true;
     fifo.SignalRelaseWorkers();
}
```

5. Írjuk át a while ciklust az előző pontban felvett flag figyelésére.

```
private void WorkerThread()
{
  double[] data = null;
  while (!closing)
  {
     if (fifo.TryGet(out data))
     ...
```

6. Végül biztosítsuk, hogy a már bezáródó ablak esetében ne próbáljunk üzeneteket kiírni.

```
private void ShowResult(double[] parameters, double result)
{
  if (closing)
     return;

  ...
```

7. Futtassuk az alkalmazást, és ellenőrizzük, kilépéskor az processzünk valóban befejezi-e a futását.