

# C# THREADING

Systemy operacyjne pochodzące z firmy Microsoft, z założenia i od początku, ukierunkowane były raczej na rynek komputerów osobistych. Zatem celem nadrzędnym nie było osiągnięcie oszałamiających mocy obliczeniowych czy niezawodności, ale łatwości obsługi i konserwacji, przy równocześnie minimalnych kosztach.



Tak też skuteczne rozwiązania z zakresu współbieżności, które w obrębie *Portable Operating System Interface* wprowadzono już w

*POSIX.1b Real-time extensions (IEEE Std 1003.1b-1993)* odnośnie wieloprosesowości a wielowątkowości

*POSIX.1c Threads extensions (IEEE Std 1003.1c-1995)* nie wzbudziły - w tej, czy jakiegokolwiek innej formie - większego zainteresowania ze strony Microsoft. Nie można się temu dziwić, bowiem ukierunkowanie jest, a raczej było, tu całkowicie odmienne.

Co prawda pewien niewielki wyłom - w tym zakresie stanowiła rodzina systemów

WINDOWS SERVER NT | 2000 | 2003 | ...

nie mniej w dość ograniczonym zakresie.

**OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE**

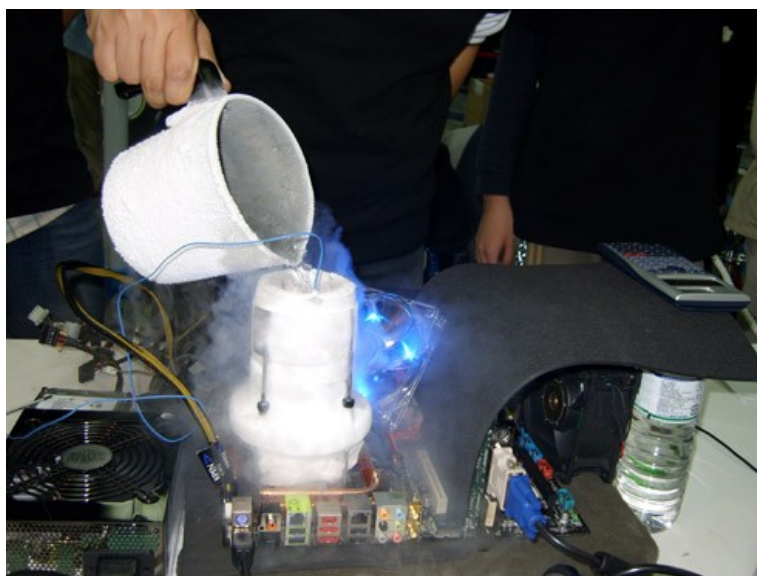
**KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL**



# C# THREADING

Z czasem jednak pojawiły się zupełnie inne i całkowicie nowe okoliczności wynikające z fizycznych barier podnoszenia wydajności platformy hardware'owej komputerów PC (w szczególności procesorów).

O ile osiągalne jest podniesienie wydajności procesora o tradycyjnej architekturze *single-core* do około 8.0 GHz (aktualny rekord 8.2 GHz), to jest to praktycznie trudne do zrealizowania. W szczególności wymagałoby chłodzenie przy pomocy ciekłego azotu.



Intel Pentium 4 Extreme Edition  
w 2008 rok 3.75 GHz

Dla praktycznych rozwiązań komercyjnych barierą jest około  
3.6 - 3.7 GHz  
Stąd i sięgnięto po architekturę typu *multi-core*.



Intel:  
od 2006 do dzisiaj.

OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING



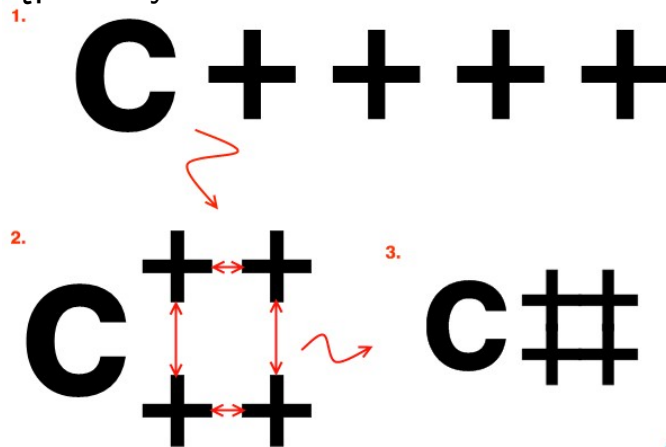
Ubočną ale, w gruncie rzeczy, naturalną konsekwencją zwrócenia się producentów komputerów osobistych ku architekturze *multi-core*, stało się zainteresowanie rozwiązaniami z zakresu oprogramowania współbieżnego.

Póki co dominujący na rynku komputerów PC system operacyjny *MsWindows* wspiera współbieżność od niedawna i w ograniczonym zakresie.

Aktualnie problem współbieżności rozwiązano skutecznie na poziomie wątków, aczkolwiek prowadzone są - wg doniesień *Microsoft* - bardzo intensywne prace rozwojowe w tym zakresie. Tak też należy się podziwiać bardzo dużego postępu w tym zakresie.

Niezależnie od wsparcia API systemowego kwestię otwartą stanowi wybór narzędzia, w postaci języka programowania. Aktualne koncepcje w zakresie rozwoju oprogramowania użytkowego dla systemu *MsWindows*, jakie prezentuje firma *Microsoft* uznają jako najbardziej perspektywiczne języki

*Common Intermediate Language (CIL)*  
a więc kodu wykonywanego w środowisku uruchomieniowym *.NET Framework*



OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

C# jest obiektywnym językiem programowania, opracowany został w firmie **Microsoft** pod koniec lat dziewięćdziesiątych, w zespole kierowanym przez Anders'Hejlsberg'a (wielce zasłużonego dla powstanie IDE i języka **Turbo Pascal** a później **Delphi** w firmie **Borland**). Samo oznaczenie '#' zaczerpnięte zostało z notacji muzycznej i symbolu krzyżyka '#' określającego ton o połowę wyższy – w tym przypadku 'C' byłby to dźwięk 'Cis'.

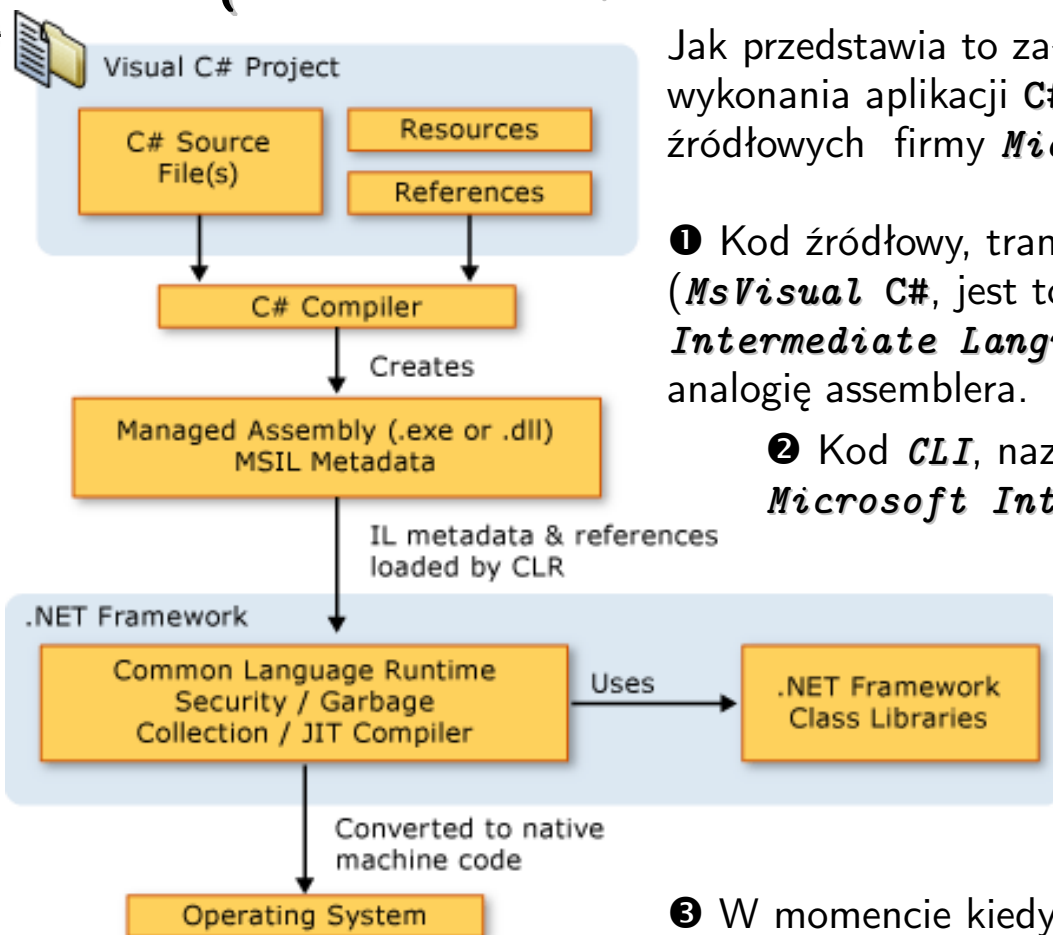


O ile wywodzi się, podobnie jak C++, stanowiąc jego nadzbiór to wprowadza bardzo wiele elementów i koncepcji języka **JAVA**. Wyraźnie daje się to zauważyć już w samej strukturze kodu a tym bardziej sposobu generowania wynikowego kodu uruchomieniowego oraz jego wykonania.

VER	DATA	.NET FRAMEWORK	VISUAL STUDIO
1.0	styczeń 2002	1.0	Rainier luty 2002
1.2	październik 2003	1.1	Everett kwiecień 2003
2.0	wrzesień 2005	2.0	Whidbey wrzesień 2005
3.0	sierpień 2007	3.5	Orcas listopad 2007
4.0	kwiecień 2010	4.0	Rosario kwiecień 2010

OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING



Jak przedstawia to załączony obok schemat powstawania i wykonania aplikacji C# (zaczepnięty w materiałów źródłowych firmy *Microsoft*), mamy tu fazy:

❶ Kod źródłowy, transformowany jest przez kompilator (*MsVisual C#*, jest to *csc.exe*), do postaci *Common Intermediate Language (CLI)* stanowiący swoistą analogię assemblera.

❷ Kod *CLI*, nazywany pierwotnie przez twórców C# *Microsoft Intermediate Language (MSIL)*, konsolidowany jest z zasobami resource jak chociażby ikony a także informacjami opisującymi konfigurację (jak wersja, język, bezpieczeństwo) ogólnie nazywane manifest – w końcu - zapisywany jest formie binarnej jako \*.EXE.

❸ W momencie kiedy pojawia się polecenie jego uruchomienia jest przekazywany maszynie wirtualnej *CLR*, która sprawdza nagłówek i jeżeli spełnione są określone w nim wymogi, przekazuje kompilacji *Just In Time (JIT)* kodu *CLI* aplikacji, tworząc w ten sposób natywny, wykonywany kod binarny (właściwy danej platformie sprzętowej i systemowej).

OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

Kod *CLI* uzyskany po kompilacji, podobnie jak kod źródłowy można swobodnie przeglądać. Umożliwia to program

**IL Disassembler (x64)**

dostępny pod zakładką

**Microsoft Visual Studio 2010>Microsoft Windows SDK Tools**

Weźmy najprostszy program C#, który wypisuje komunikat **Hello World !** na konsoli.

Sam program przedstawiać się może następująco

```
using System;
class Program
{
    static void Main()
    {
        Console.WriteLine("Hello World!");
    }
}
```

Kompilacja z konsoli (po uruchomieniu Visual Studio x64 Win64 Command Prompt (2010))

D:\.>csc.exe hello.cs

Microsoft (R) Visual C# 2010 Compiler version 4.0.30319.1

Copyright (C) Microsoft Corporation. All rights reserved.

OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

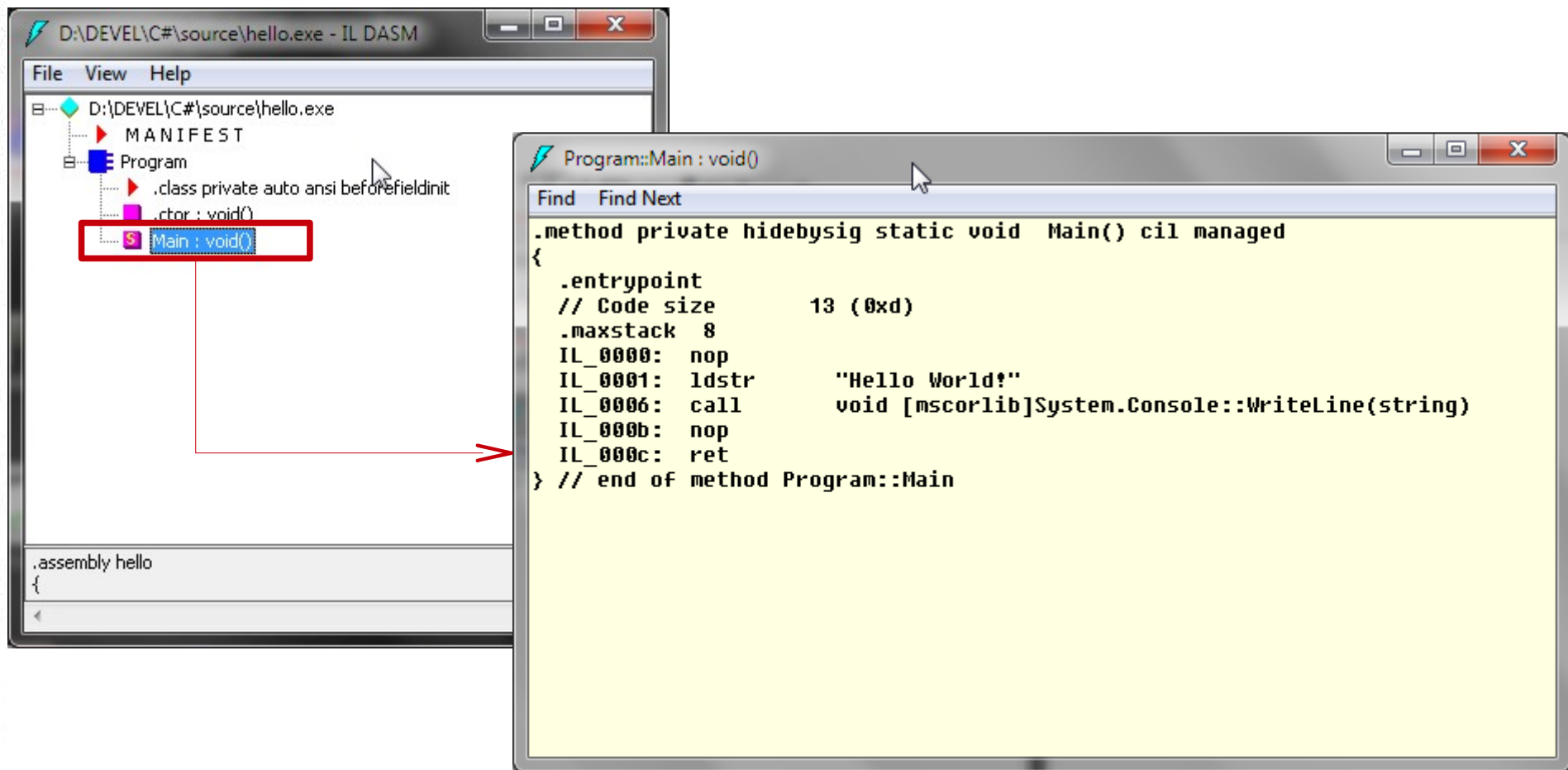
# C# THREADING

O ile nie wystąpił żaden błąd kompilacji, to uruchomienie - zatem przekazanie środowisku uruchomieniowemu maszyny wirtualnej .Net

D:\.>hello.exe

Hello World!

Gdyby otworzyć skompilowany kod CLI, w IL Disassembler

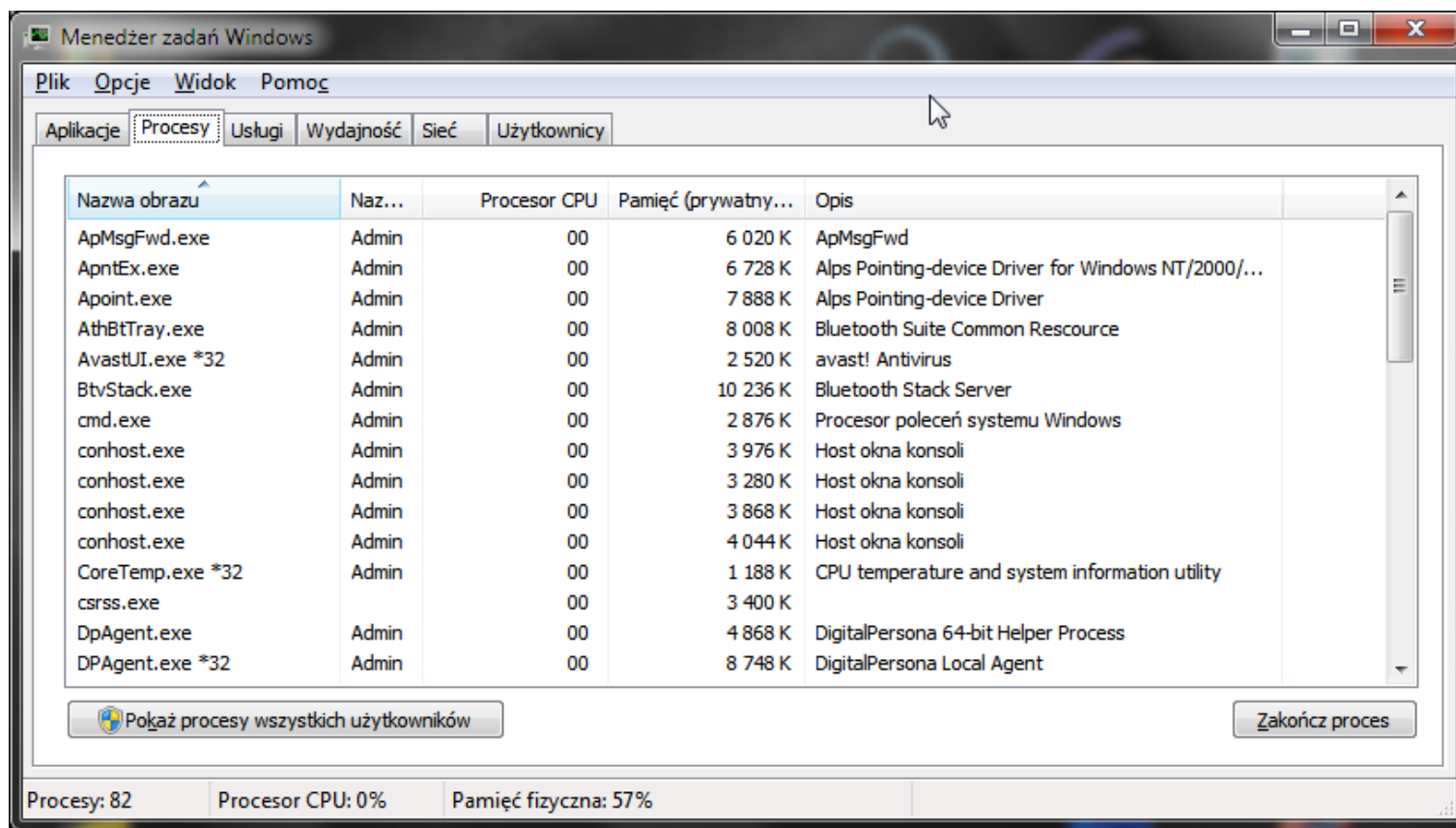


OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

Ilekoć uruchamiamy aplikację, to system dokonuje na jej rzecz przydziału zasobów w tym pamięci. Choć zwykle jest to jeden, to generalnie aplikacja może powołać do życia wiele procesów - każdy z nich jednak otrzyma własne zasoby, odrębne od innych procesów i chronione przez system, na jego rzecz, zasoby.

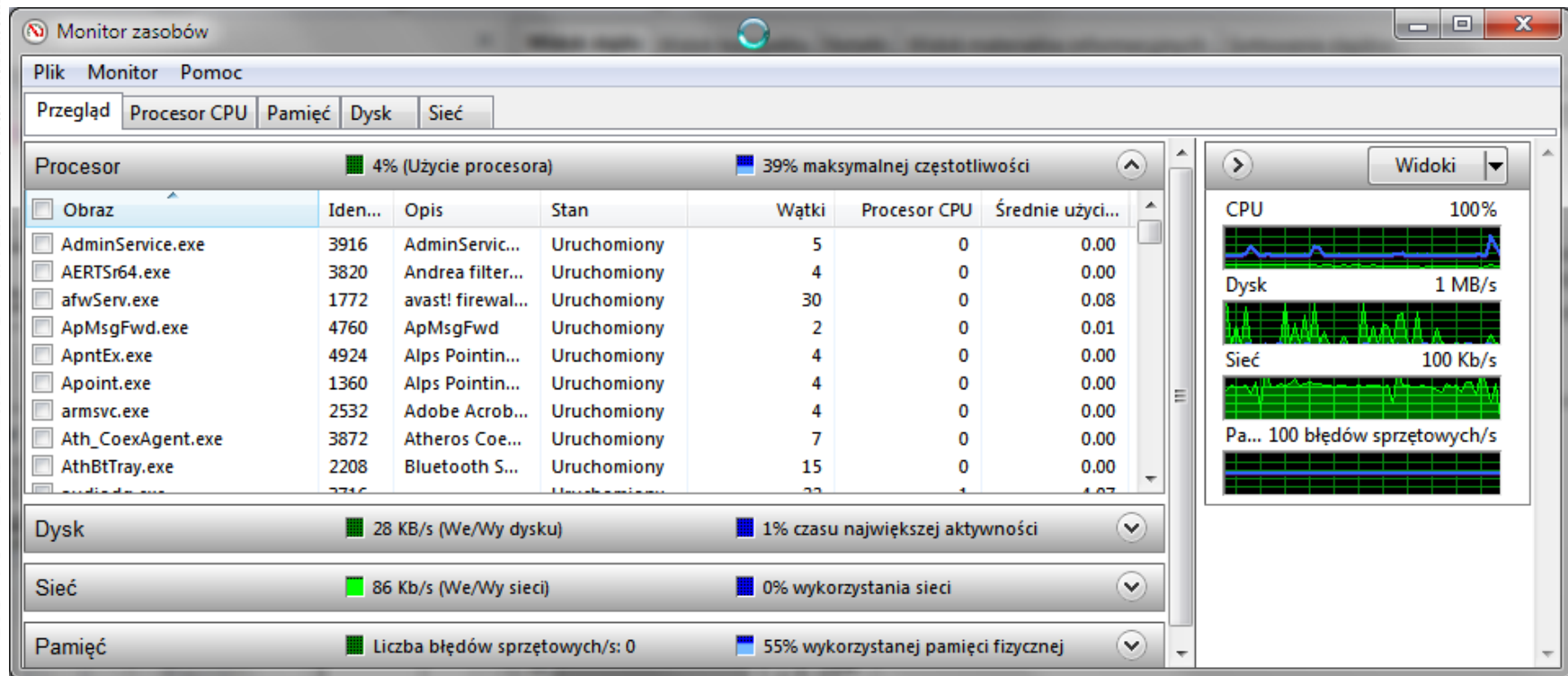
Listę aktywnych procesów i ich ogólną charakterystykę można łatwo pozyskać w systemie MsWindows uruchamiając **Task Manager**, czyli **Manager Zadań**.



OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

Dużo większe możliwości z zakresu monitorowania stanu procesów daje aplikacja menadżera zasobów (dostępna od pewnego czasu jego systemowe narzędzie administracyjne).



Zauważmy, że oprócz nazwy mamy tutaj także PID, informacje o stanie i wykorzystywanych zasobach.

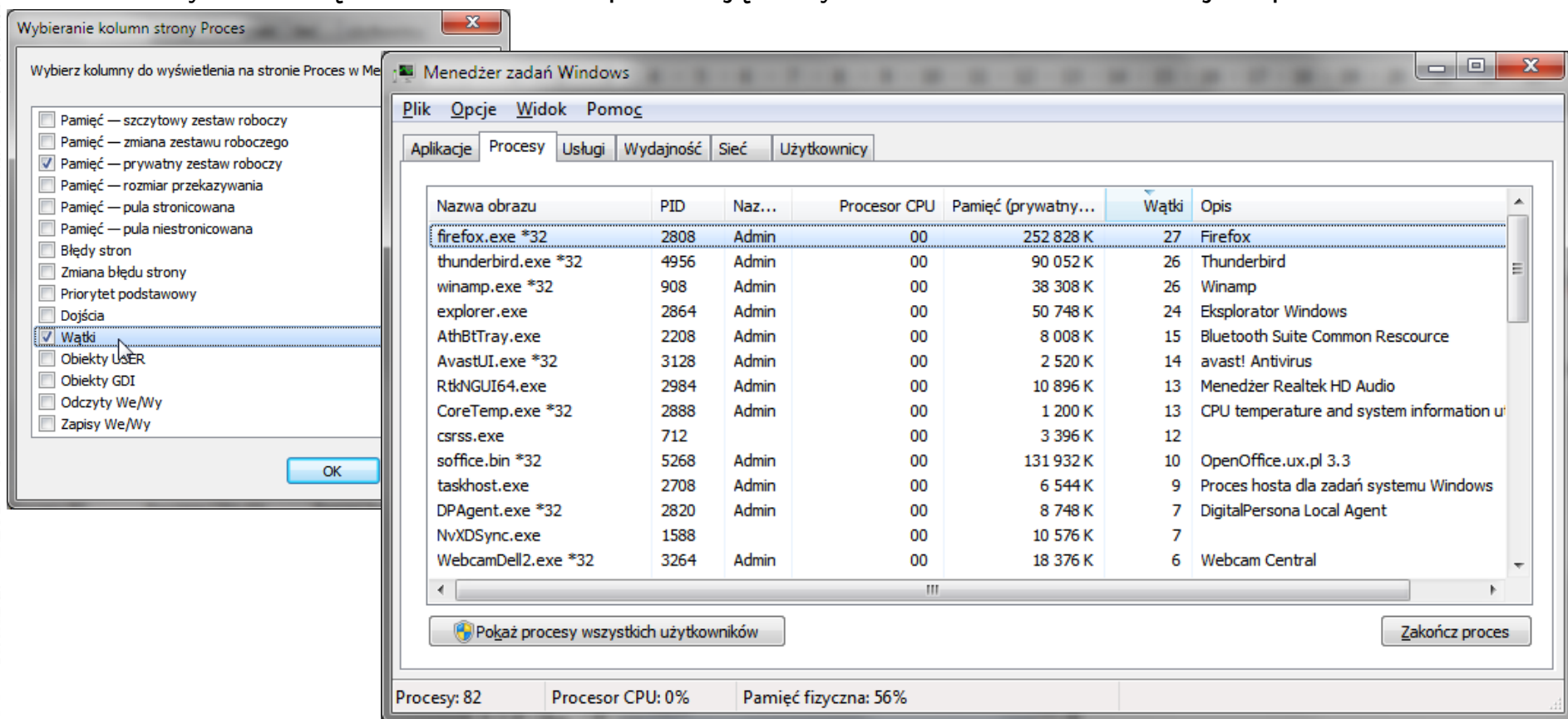
OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

Każdy proces, w obrębie przydzielonych mu przez system zasobów, może powołać wątek. Informację o wątkach związanych z każdym procesem można także wyświetlić przez Task Manager, wybierając z jego menu głównego

**Widok> Wybierz kolumny...**

wówczas wyświetli się dodatkowe okno pozwalające wybrać dodatkowe informacje o procesach.



Zobaczmy wówczas - przykładowo - że proces firefox.exe powołał 27 wątków, zaś występujący na 3 pozycji listy winamp.exe 26 wątków.

**OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE**  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

O ile możliwość użycia wątków przez proces nie jest czymś generalnie nowym dla tego środowiska, co jest spostrzeżeniem oczywistym, to C# i środowisko **.Net Framework** oferuje wobec nic całkowicie nową jakość.

Zawiera, czy powiela, standardowe elementy znane już znacznie wcześniej ze standardu **POSIX**, wprowadza jednak dla wątków reprezentację obiektową.

Przetwarzanie wielowątkowe zawsze będzie skuteczniejsze od sekwencyjnego, jeżeli dotyczyć będzie rozległych zestawów danych, stosunkowo jednorodnych. Dodatkowo, a może tym bardziej, że:

- ☐ wątki mogą być uruchamiane w tle, całkowicie bez angażowania użytkownika
- ☐ jeżeli proces nadrzędny będzie wymagał dostępu do danych, kiedy czas jest duży - wątek lub wątki działające w tle mogą znacznie ograniczyć wpływ tej niedogodności.

Co oczywiste dodatkowy wątki pochłaniają dodatkowe zasoby, zarządzanie nimi pochłania dodatkowy czas - jednak jak zaleca to aktualnie twórca systemu

*...that you use as few threads as possible in your applications.*





OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL


# C# THREADING


.Net Framework oferuje celem zarządzania wątkami udostępnia przestrzeń nazw  
`System.Threading`

obejmującą:

 **klasy:** `AbandonedMutexException` `AutoResetEvent` `Barrier` `BarrierPostPhaseException` `CancellationTokenSource` `CompressedStack` `CountdownEvent` `EventWaitHandle` `ExecutionContext` `HostExecutionContext` `HostExecutionContextManager` `Interlocked` `LazyInitializer` `LockRecursionException` `ManualResetEvent` `ManualResetEventSlim` `Monitor` `Mutex` `Overlapped` `ReaderWriterLock` `ReaderWriterLockSlim` `RegisteredWaitHandle` `Semaphore` `SemaphoreFullException` `SemaphoreSlim` `SynchronizationContext` `SynchronizationLockException` `Thread` `ThreadAbortException` `ThreadExceptionEventArgs` `ThreadInterruptedException` `ThreadLocal(Of T)` `ThreadPool` `ThreadStartException` `ThreadStateException` `Timeout` `Timer` `WaitHandle` `WaitHandleCannotBeOpenedException`

 **struktury:** `AsyncFlowControl` `CancellationToken` `CancellationTokenRegistration` `LockCookie` `NativeOverlapped` `SpinLock` `SpinWait`

 **delegacje:** `ContextCallback` `IOCompletionCallback` `ParameterizedThreadStart` `SendOrPostCallback` `ThreadExceptionHandler` `ThreadStart` `TimerCallback` `WaitCallback` `WaitOrTimerCallback`

 **wyliczenia:** `ApartmentState` `EventResetMode` `LazyThreadSafetyMode` `LockRecursionPolicy` `ThreadPriority` `ThreadState`

OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

W sposób bezpośredni, tworzeniu i zarządzaniu wątkami służy klasa **Thread**.

Za nim wątek będzie mógł być użyty należy najpierw zdefiniować stosowny obiekt tej klasy. Odbyna się to w tradycyjny sposób, zaś funkcja wątku podawana jest za pośrednictwem konstruktora **Thread()**.

Utwórzmy z wątku głównego procesu jeden wątek potomny o nazwie **thread**. Pierwszy będzie wypisywał na konsoli znaki 'x' zaś drugi (potomny) 'o'.

```
using System;
using System.Threading; //...konieczne włącznie przestrzeni nazw wątków

namespace Thread
{
    class Program
    {
        const int loops = 100;    //...krotność wykonania

        static void Main( string[] args )
        {
            //...definicja obiektu typu wątek (Thread), z funkcją wątku o()
            System.Threading.Thread thread = new System.Threading.Thread( o );
            //...uruchomienie wątku thread
            thread.Start();
            //...działanie własne wątku procesu głównego
            for (int i = 0; i < loops; i++) { Console.Write("x"); }
        }
    }
}
```

OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

Oczywiście pozostaje jeszcze zdefiniować funkcję wątku.

```
static void o()  
{  
    for (int i = 0; i < loops; i++) { Console.Write("o"); }  
}  
}
```

Kompilacja

D:\.>csc.exe oxo.cs

Microsoft (R) Visual C# 2010 Compiler version 4.0.30319.1

Copyright (C) Microsoft Corporation. All rights reserved.

i uruchomienie

D:\.>oxo.exe

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
```

Co oczywiste nie ma tu żadnej synchronizacji między obiema wątkami a chronologia czasowa ich koegzystencji jest mniej lub bardziej przypadkowa.

OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

Warto jeszcze, na wstępie zauważyć, że podobnie jak w *POSIX* wątek otrzymuje swój unikalny identyfikator ale - ponieważ jest to zmienna obiektowa - także i szeregu innych atrybutów. Przykładowo można mu nadać nazwę **Name**.

```
using System;
using System.Threading;

namespace Thread
{
    class Program
    {
        static void Main(string[] args)
        {
            //...nazwa dla wątku procesu głównego
            System.Threading.Thread.CurrentThread.Name = "MASTER";
            //...inicjowanie obiektu Thread
            System.Threading.Thread thread = new System.Threading.Thread( go );
            thread.Name = "SLAVE";
            thread.Start();
            //...i jeszcze działanie dla głównego - niech także wykona funkcję wątku
            go(); //...funkcję wątku wykonują oba, choć z różnym rezultatem.
        }

        static void go()
        { Console.WriteLine("Hello from {0}",System.Threading.Thread.CurrentThread.Name); }
    }
}
```

OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

Zapisując kod źródłowy pod nazwą **name.cs**, po kompilacji i uruchomieniu uzyskujemy na konsoli

```
D:\.>name.exe  
Hello from MASTER  
Hello from SLAVE
```

Tworząc nowy wątek bezpiecznie będzie sprawdzić, czy znajduje się już w stanie wykonania. Służy temu własność

```
public bool System.Threading.Thread.IsAl  IsAlive { get; }
```

Wykonanie dowolnego wątku można wstrzymać na czas określony w milisekundach, metodą

```
public static void System.Threading.Thread.Sleep( int millisecondsTimeout );
```

Oczywiście można napisać także - przykładowo

```
Thread.Sleep (TimeSpan.FromHours (1) );
```

co przyniesie wstrzymanie na 1 godzinę ☺

Wątkowi można polecić natychmiastowe zatrzymanie

```
public void System.Threading.Thread.Abort();
```

które generuje `ThreadAbortException`.

Przygotujemy teraz kolejny przykład, w którym:

- ☐ utworzymy wątek potomny, który na konsoli będzie sygnalizował aktywność wypisując '.'
- ☐ zatrzymamy wątek procesu głównego na 2000 milisekund
- ☐ po tym czasie nakážemy wątkowi przerwanie działania
- ☐ w funkcji wątku dodamy obsługę zdarzeń `try|catch|finalize`

OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE

KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

```
using System;
using System.Threading;
class Program
{
    static void Main(string[] args)
    {
        System.Threading.Thread thread = new System.Threading.Thread( work );
        thread.Start();
        while (!thread.IsAlive);
        Thread.Sleep( 2000 );
        thread.Abort();
    }

    static void work()
    {
        try
        {
            Console.Write( "[thread start]" );
            while( true ){ Console.Write( "." ); }
        }
        catch (Exception error)
        {
            Console.WriteLine( "[thread stop]" );
            //Console.WriteLine( "...caught exception {0}",error.ToString() );
        }
        finally
        {
            Console.WriteLine( "[work done]" );
        }
    }
}
```

OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

Jeżeli kod źródłowy zapiszemy pod nazwą **stop.cs**, to jego kompilacja

```
D:\.>csc.exe stop.cs
```

```
Microsoft (R) Visual C# 2010 Compiler version 4.0.30319.1
```

```
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
stop.cs(24,20): warning CS0168: The variable 'error' is declared but never used
```

tutaj oczywiście pojawia się ostrzeżenie odnośnie zmiennej **error**.

Wykonanie programu

```
D:\.>stop.exe
```

```
[thread
```

```
start].....
```

```
.....
```

```
.....
```

```
.....
```

```
.....
```

```
.....
```

```
.....
```

```
.....
```

```
.....
```

```
.....
```

```
.....
```

```
.....[thread stop]
```

```
[work done]
```

OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

Gdyby usunąć komentarz z metody `work`, czyli byłoby

```
static void work()
{
    try
    {
        Console.Write( "[thread start]" );
        while( true ){ Console.Write( "." ); }
    }
    catch (Exception error)
    {
        Console.WriteLine( "[thread stop]" );
        Console.WriteLine( "...caught exception {0}",error.ToString() );
    }
    finally{ Console.WriteLine( "[work done]" ); }
```

to otrzymalibyśmy w miarę precyzyjne informacje o przyczynie i warunkach zakończenia.

```
.....
.....[thread stop]
...caught exception System.Threading.ThreadAbortException: Thread was being aborted.
    at System.IO.__ConsoleStream.WriteFile(SafeFileHandle handle, Byte* bytes, Int32
numBytesToWrite, Int32& numBytesWrit
ten, IntPtr mustBeZero)
    at System.IO.__ConsoleStream.WriteFileNative(SafeFileHandle hFile, Byte[] bytes,
Int32 offset, Int32 count, Int32 mus
tBeZero, Int32& errorCode)
    at System.IO.__ConsoleStream.Write(Byte[] buffer, Int32 offset, Int32 count)
    at System.IO.StreamWriter.Flush(Boolean flushStream, Boolean flushEncoder)
    at System.IO.TextWriter.SyncTextWriter.Write(String value)
    at Program.work()
[work done]
```

OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

Prócz wymienionych a właściwie ponad nie, bo jest to zasadniczy sposób jaki należy użyć celem synchronizacji jest metoda

```
public void System.Threading.Thread.Join();
```

W kolejnym przykładzie wątek procesu głównego będzie tak długo kontynuował swe działanie do póki potomny nie zakończy.

```
using System;
using System.Threading;

class Program
{
    static void Main(string[] args)
    {
        System.Threading.Thread thread = new System.Threading.Thread( work );
        thread.Start();
        while( thread.IsAlive ){ Console.Write( "." ); }
        thread.Join();
    }
    static void work()
    {
        Console.Write( "[thread start]" );
        Thread.Sleep( 100 );
        Console.WriteLine( "[thread stop]" );
    }
}
```

OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

Efekt wykonania przedstawia się następująco

D:\>.join.exe

.....[thread start]

.[thread stop]

.

zwróćmy uwagę na  
fragmenty

Nic nie stoi na przeszkodzie aby Join() dotyczyło wielu wątków, czyli

```
static void Main(string[] args)
{
    Thread[] thread = new Thread[10];
    for (int ID = 0; ID < thread.Length; ID++)
    {
        thread[ID] = new System.Threading.Thread(work);
        thread[ID].Name = ID.ToString();
        thread[ID].Start();
    }
    for (int ID = 0; ID < thread.Length; ID++) { thread[ID].Join(); }
}

static void work()
{
    Console.WriteLine("{0}", Thread.CurrentThread.Name);
}
```

OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

Podstawowym narzędziem synchronizacyjnym C# jest obiekt `System.Threading.Monitor`;

Jego ogólny schemat użycia przedstawia się jako

```
Monitor.Enter( locked );  
try  
{  
    //...kod sekcji krytycznej  
}  
finally  
{  
    //...protokół wyjścia  
    Monitor.Exit( locked );  
}
```

Równoważne wobec tych wywołań jest użycie słowa kluczowego `lock`, jako

```
lock( locked )  
{  
    //...kod sekcji krytycznej  
}
```

OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

Weźmy jako przykład

```
using System;
using System.Threading;

class Job
{
    public Job() { /* czyli konstruktor domyślny */ }
    public void work()
    {
        for (int i = 0; i < 5; i++)
        { Console.WriteLine("{0}-{1} ", Thread.CurrentThread.Name, i); }
        Console.WriteLine();
    }
}

class Program
{
    static void Main(string[] args)
    {
        Thread[] thread = new Thread[10];
        Job job = new Job();
        for (int ID = 0; ID < thread.Length; ID++)
        {
            thread[ID] = new Thread( new ThreadStart( job.work ) );
            thread[ID].Name = ID.ToString();
        }
        foreach (Thread t in thread ) { t.Start(); }
        foreach (Thread t in thread) { t.Join(); }
    }
}
```

OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

Wprowadzając zaś wzajemne wykluczanie...

```
using System;
using System.Threading;
class Job
{
    public Job() { /* czyli konstruktor domyślny */ }
    public void work()
    {
        lock (this)
        {
            for (int i = 0; i < 5; i++)
            { Console.WriteLine("{0}-{1} ", Thread.CurrentThread.Name, i); }
            Console.WriteLine();
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        Thread[] thread = new Thread[10];
        Job job = new Job();
        for (int ID = 0; ID < thread.Length; ID++)
        {
            thread[ID] = new Thread( new ThreadStart( job.work ) );
            thread[ID].Name = ID.ToString();
        }
        foreach (Thread t in thread ) { t.Start(); }
        foreach (Thread t in thread) { t.Join(); }
    }
}
```

OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE

KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

Porównajmy teraz efekty wykonania obu programów.

W pierwszym przypadku (bez wykluczania)

```
0-0 0-1 0-2 0-3 0-4
3-0 3-1 1-0 1-1 4-0 2-0 2-1 2-2 2-3 2-4
4-1 4-2 4-3 4-4
5-0 5-1 5-2 5-3 5-4
1-2 1-3 1-4
7-0 7-1 7-2 7-3 7-4
8-0 8-1 8-2 8-3 8-4
3-2 3-3 3-4
9-0 9-1 9-2 9-3 9-4
6-0 6-1 6-2 6-3 6-4
```

W drugim przypadku (z wykluczaniem)

```
0-0 0-1 0-2 0-3 0-4
1-0 1-1 1-2 1-3 1-4
3-0 3-1 3-2 3-3 3-4
2-0 2-1 2-2 2-3 2-4
5-0 5-1 5-2 5-3 5-4
6-0 6-1 6-2 6-3 6-4
7-0 7-1 7-2 7-3 7-4
4-0 4-1 4-2 4-3 4-4
8-0 8-1 8-2 8-3 8-4
9-0 9-1 9-2 9-3 9-4
```

Zaznaczono sekwencję wykonania wątku 1.

Jak wspomniano wcześniej - identyczny efekt można uzyskać wprowadzając obiekt **Monitor**, czyli re-definiując metodą **work()** klasy **Job**

```
public void work()
{
    Monitor.Enter(this);
    try
    {
        for (int i = 0; i < 5; i++)
        { Console.WriteLine("{0}-{1} ", Thread.CurrentThread.Name, i); }
        Console.WriteLine();
    }
    finally { Monitor.Exit(this); }
}
```

OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

Innym sposobem ochrony zasobów współdzielonych jest użycie obiektu `System.Threading.Mutex`

Obiekt ten wyposażony jest w zestaw konstruktorów, z czego najczęściej używanym jest domyślny

```
public Mutex()
```

oraz dwie zgłaszające żądanie dostępu do zasobu współdzielonego oraz zwalniające ten zasób

```
public virtual bool WaitOne()
```

```
public void ReleaseMutex()
```

Ogólną zasadą użycia obiektu `Mutex` można przedstawić na przykładzie poniższego kodu.

```
using System;
```

```
using System.Threading;
```

```
class Program
```

```
{
```

```
    private static Mutex mutex = new Mutex(); //...definiujemy obiekt Mutex
```

```
    static void Main()
```

```
    {
```

```
        for (int ID = 0; ID < 5; ID++) //...tworzymy 5 wątków potomnych
```

```
        { // funkcję wątku będzie Go()
```

```
            Thread thread = new Thread( new ThreadStart( Go ) );
```

```
            thread.Name = ID.ToString();
```

```
            thread.Start();
```

```
        }
```

```
    }
```

OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE

KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

W funkcji wątku wykonamy 3 cykle dostępu do zasobu współdzielonego (chronionego obiektem `mutex`).

```
private static void Go()
{
    for (int cycle = 0; cycle < 3; cycle++)
    {
        Use();
    }
}
```

Działania wobec zasobu chronionego symuluje funkcja (metoda)

```
private static void Use()
{
    mutex.WaitOne();          //...czyli żądanie dostępu
    Console.WriteLine("thread {0}...use shared resource", Thread.CurrentThread.Name);
    //...sekcja krytyczna, początek
    Thread.Sleep(100);
    //...sekcja krytyczna, koniec
    mutex.ReleaseMutex(); //...zwolnienie zasobu
}
}
```

Kończy to kod programu.

Pozostaje skompilować i uruchomić.

OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

Efekt wykonanie przedstawia się następująco

```
thread 0...use shared resource
thread 0...use shared resource
thread 4...use shared resource
thread 4...use shared resource
thread 1...use shared resource
thread 3...use shared resource
thread 1...use shared resource
thread 0...use shared resource
thread 2...use shared resource
thread 4...use shared resource
thread 1...use shared resource
thread 3...use shared resource
thread 2...use shared resource
thread 2...use shared resource
thread 3...use shared resource
```

czyli tak jak należałoby oczekiwać.

Zwróćmy tu jeszcze raz uwagę na sekwencję wykonania - przykładowo zaznaczono wątek ostatni o ID 5.

OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

Weźmy teraz typowy przykład wyścigu w dostępie do zmiennej współdzielonej. Tutaj będzie to zmienna całkowita N, której nadamy wartość początkową 0 a każdy z wątków zwiększy ją o 1.

```
using System;
using System.Threading;

class Program
{
    private static Mutex mutex = new Mutex();
    private static int N = 0;

    static void Main()
    {
        Thread[] thread = new Thread[5];
        for (int ID = 0; ID < thread.Length; ID++)
        {
            thread[ID] = new Thread(new ThreadStart( Inc ));
            thread[ID].Name = ID.ToString();
        }
        foreach (Thread t in thread) { t.Start(); }
        foreach (Thread t in thread) { t.Join(); }
        Console.WriteLine("N = {0}", N);
    }

    //...tutaj powinna się znaleźć funkcja wątku Inc()
}
```

OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

Funkcja wątku, nich przedstawia się następująco (na wstępie pozostawmy komentarze przy wywołaniach mutex)

```
private static void Inc()
{
    int n;
    //mutex.WaitOne();
    n = N;
    Console.WriteLine("thread {0} take N={1}", Thread.CurrentThread.Name, n);
    n = n + 1;
    N = n;
    Console.WriteLine("thread {0} send N={1}", Thread.CurrentThread.Name, n);
    //mutex.ReleaseMutex();
}
```

Z racji istnienia wyścigu i braku synchronizacji między pobraniem wartości N a przypisaniem własnego wyniku, efekt

```
thread 0 take N=0
thread 0 send N=1
thread 2 take N=0
thread 4 take N=1
thread 4 send N=2
thread 2 send N=1
thread 1 take N=0
thread 3 take N=1
thread 3 send N=2
thread 1 send N=1
N = 2
```

... no i wynik całkowicie błędny (jak można by oczekiwać).

OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

Gdyby wprowadzić ochronę sekcji krytycznej obiektem `Mutex`, czyli

```
private static void Inc()
{
    int n;
    mutex.WaitOne();
    n = N;
    Console.WriteLine("thread {0} take N={1}", Thread.CurrentThread.Name, n);
    n = n + 1;
    N = n;
    Console.WriteLine("thread {0} send N={1}", Thread.CurrentThread.Name, n);
    mutex.ReleaseMutex();
}
```

to uzyskamy wynik w pełni poprawny

```
thread 0 take N=0
thread 0 send N=1
thread 4 take N=1
thread 4 send N=2
thread 1 take N=2
thread 1 send N=3
thread 3 take N=3
thread 3 send N=4
thread 2 take N=4
thread 2 send N=5
N = 5
```

OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL

# C# THREADING

Aktualne rozwiązanie problemu współbieżności w C# w obrębie

**System.Threading**

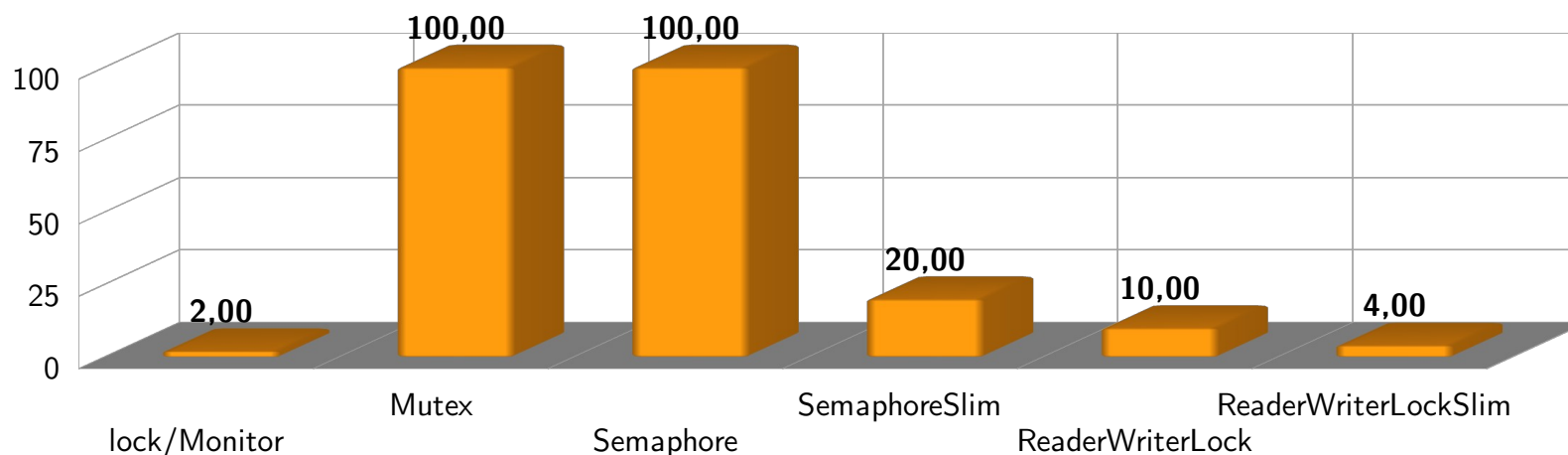
cehuje się dużą wszechstronnością i omówiono tylko niewielki jej fragment. Właściwie mógłby to być temat dla osobnych zajęć. Warto pamiętać o witrynie **MSDN**

<http://msdn.microsoft.com/en-us/library/system.threading.aspx>

Równocześnie charakteryzuje się aktualnie bardzo dużą dynamiką i ma jeszcze szereg cech znamienych wczesnego stadium.

Na koniec przedstawimy jeszcze podsumowanie odnośnie orientacyjnej efektywności poszczególnych metod ochrony sekcji krytycznej, czyli:

- ☐ **lock** (**Mutex.Enter()**/**Mutex.Exit()**) i **Mutex**, czyli semafore binarne
- ☐ **Semaphore**, tradycyjny semafor licznikowy
- ☐ **ReaderWriterLock** a więc semafor dedykowany obsłudze problemu czytelników i pisarzy, a dwa ostatnie mają jeszcze warianty **Slim**.



OBLICZENIA RÓWNOLEGŁE I SYSTEMY ROZPROSZONE  
KRYSPIŃ MIROTA, KMIROTA@ATH.BIELSKO.PL