Programowanie Współbieżne

C#

Wątki

Kiedy przydają nam się wątki?

Gdy chcemy by nasz program reagował gdy wykonuje w tle jakieś „ciężkie” zadanie

Różnego rodzaju procesy – serwery. Podczas oczekiwania na dane na jednym wątku, program może coś wykonywać na innym.

Gdy mamy program, który wykonuje sporo obliczeń (np. kompresja plików multimedialnych) i chcemy je w jakiś sposób zrównoleglić. Efekt będzie odczuwalny gdy fizycznie będziemy dysponować wieloma rdzeniami. Liczbę tę można sprawdzić za pomocą **Environment.ProcessorCount**

Wątki

Kiedy wątki mogą nam szkodzić?

•Gdy będzie ich za dużo. Czas przełączania i alokacji zbyt kosztowny,

•Gdy zadanie wykonywane przez wątek będzie krócej trwało niż powołanie danego wątku.

•Gdy w pełni nie przewidzimy interakcji pomiędzy wątkami, debugowanie jest bardzo kłopotliwe.

•Gdy używamy dużo dysku, nie powinniśmy powoływać wiele wątków, a raczej jeden, dwa i szeregować zadania odczytu i zapisu. (ktoś próbował skopiować z płyty CD/DVD kilka plików naraz?)

Wątki

- Gdy operujemy na wątkach musimy dodać do programu

```
using System.Threading;
```

- Każdy program ma przynajmniej jeden wątek, zwany wątkiem głównym
- Każdy wątek ma swój oddzielny stos więc zmienne lokalne są modyfikowane niezależnie.
- Zmienne globalne są współdzielone przez wątki (często wymagana synchronizacja)

Wątki

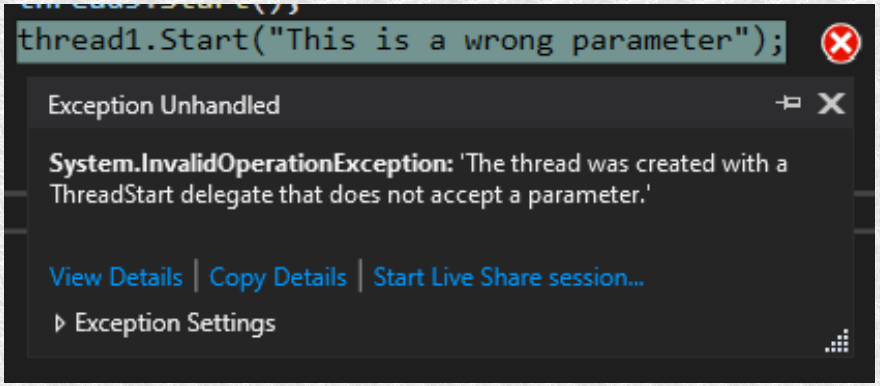
Przykład uruchomienia metod działających jako wątki.

```
static void OurThreadWithoutParameters()
{
    Console.WriteLine("Thread without parameters");
}

static void OurThread()
{
    Console.WriteLine($"Overloaded method Thread without parameeter");
}

static void OurThread(object o)
{
    string message = o as string;
    Console.WriteLine($"Overloaded method Thread with parameeter got message:
{message}");
}

private void OurThreadWithParameter(object o)
{
 if (o == null)
    {
    Console.WriteLine("I got null as a parameter");
    return;
    }
Console.WriteLine("Thread with parameter got message: " + (string)o);
}
```

Wątki

Tworzenie i uruchamianie wątków bez parametrów

```
Thread thread1 = new Thread(new ThreadStart(OurThreadWithoutParameters));
Thread thread2 = new Thread(OurThreadWithoutParameters); //compilator adds
automaticly new(ThreadStart
Thread thread3 = new Thread(new ThreadStart(OurThread));
//Thread thread3 = new Thread(OurThread); //When we have overloaded methods (without
and with parameters, the compiler does not know whether to use ThreadStart or
ParameterizedThreadStart
thread1.Start();
thread2.Start();
thread3.Start();
//thread1.Start("This is a wrong parameter");//Invalid operation exception
```



Wątki

Przykład tworzenia i uruchomienia wątków z parametrem

```
Thread thread4 = new Thread(new
ParameterizedThreadStart(OurThreadWithParameter));
Thread thread5 = new Thread(OurThreadWithParameter);
Thread thread6 = new Thread(new ParameterizedThreadStart(OurThread));

thread4.Start(); //Possible such a start, but there will be no parameters o
= null
thread5.Start("Message for thread 5");
thread6.Start("Message for thread 6");
```

Wątki

Wywołanie anonimowe

```
static void OurThreadWithTypedParameter(string message)
{
    Console.WriteLine("Thread with typed parameter got message: " + message);
}

string changingMessage;
changingMessage = "Message befor thread created";
Thread thread7 = new Thread(delegate ()
{ OurThreadWithTypedParameter(changingMessage); }); //using an anonymous
method, we do not have to specify the object parameter, but we can specify a
specific type, e.g. string
changingMessage = "Message after thread chreated";
Thread thread8 = new Thread(delegate()
    {
     Console.WriteLine(changingMessage);
     });
thread7.Start();
thread8.Start();
```

Wątki

Przykład wywołania z notacją lambda

```
Thread thread9 = new Thread((parameter) =>
        {  //thread code
        Console.WriteLine("Lambda thread show message: " + (parameter as
string));
        });
thread9.Start("Some message for lambda created thread");

Thread thread10 = new Thread((parameter) =>
        {  //thread code
        OurThreadWithTypedParameter(parameter as string);
        });
thread10.Start("Some message for lambda created thread with method");
```

Wątki

Przykład wywołania z metod wątków z obiektu

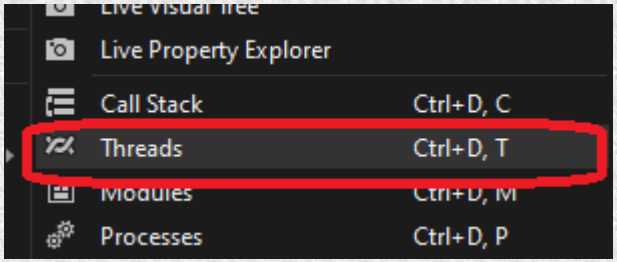
```
public class VariousThreads
        {
            public void thread1()
             {
             MessageBox.Show("I'm thread 1");
             }
            public void thread2()
             {
             MessageBox.Show("I'm thread 2");
             }
        }

VariousThreads varTh = new VariousThreads();
  Thread th1 = new Thread(varTh.thread1);
  Thread th2 = new Thread(varTh.thread2);
  th1.Start();
  th2.Start();
```

Wątki

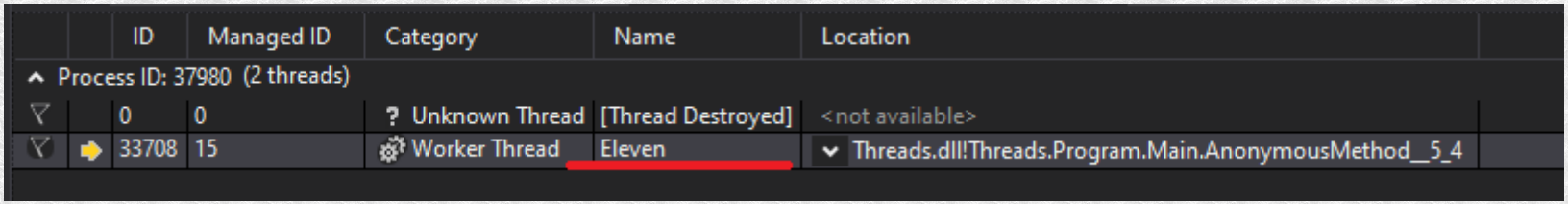
Nazywanie wątków - pomoc przy debugowaniu

```
Thread thread11 = new Thread(() =>
Console.WriteLine($"Hello. I'm {Thread.CurrentThread.Name} thread.")
            );
thread11.Name = "Eleven";
thread11.Start();
```

Listę wątków do debugowania można pokazać wybierając: Debug->Windows->Threads



W tym oknie widzimy listę wątków danej aplikacji wraz ich nazwami



Wątki

Wątki pierwszoplanowe i drugoplanowe

Domyślnie **`IsBackground = false`** dlatego po zamknięciu głównej aplikacji nadal widzimy wątki z niej powstałe. Gdy ustawimy na **`true`** wyjście z wątka głównego powoduje natychmiastowe zakończenie wątków potomnych. Blok **`finaly`** jest pomijany. Jest to sytuacja niepożądana, dlatego powinniśmy poczekać na koniec wątków potomnych.

Zmiana pracy z tła do pierwszoplanowej i odwrotnie nie wpływa na priorytet.

Wątki

Wątki pierwszoplanowe i drugoplanowe

```
Thread parentThread = new Thread(() =>
        {
        Console.WriteLine($"Thread {Thread.CurrentThread.Name} started");
        Thread childThread1 = new Thread(() =>
            {
            Console.WriteLine($"Thread {Thread.CurrentThread.Name} started");
            Thread.Sleep(5000);
            Console.WriteLine($"Thread {Thread.CurrentThread.Name} finished");
            })
            { Name = "ChildThread1"}; //backgroud = false as default
            //{ Name = "ChildThread1",IsBackground = true };
         Thread childThread2 = new Thread(() =>
            {
            Console.WriteLine($"Thread {Thread.CurrentThread.Name} started");
            Thread.Sleep(5000);
            Console.WriteLine($"Thread {Thread.CurrentThread.Name} finished");
            })
            { Name = "ChildThread2"};//backgroud = false as default
            //{ Name = "ChildThread2",IsBackground=true };

            childThread1.Start();
            childThread2.Start();
            Console.WriteLine($"Thread {Thread.CurrentThread.Name} finished");
        })
        { Name = "Parent" };
parentThread.Start();
```

Wątki

Priorytetowość

```
enum ThreadPriority { Lowest, BelowNormal, Normal,
  AboveNormal, Highest }
```

- Oznacza jak dużo czasu procesora przyznane jest dla danego wątka w grupie wątków jednego procesu.
- Ustawienie na Highest wcale nie oznacza, że będzie to wątek czasu rzeczywistego. Trzeba by było również ustawić priorytet dla procesu.

```
Process.GetCurrentProcess().PriorityClass =
    ProcessPriorityClass.High;
```

- Jest jeszcze wyższy priorytet **Realtime**, wtedy nasz proces będzie działał nieprzerwanie, jednak gdy wejdzie w pętle nieskonczoną nie odzyskamy kontroli nad systemem.
- W przypadku gdy nasza aplikacja posiada (zwłaszcza skomplikowany) interfejs graficzny, też nie powinniśmy podnosić priorytetu, gdyż odświerzanie spowoduje zpowolnienie całego systemu.

Wątki

```
static void Main(string[] args)
{
    Process process = Process.Start("notepad.exe");
     Thread.Sleep(3000);
     process.Kill();

     var e = Process.Start(@"C:\Program Files (x86)\Microsoft\Edge\Application\
msedge.exe");

     var processes = Process.GetProcesses();
     foreach (var item in processes)
        {
         try
            {
            Console.WriteLine(item.ProcessName + item.MainModule.FileName);
            }
        catch (Exception ex)
            {
            Console.WriteLine(ex.Message);
            }
        }
    var edges = Process.GetProcessesByName("msedge");
    Thread.Sleep(3000);
    edges[0].Kill(true);
}
```

Wątki

Wyjątki

```
Try
   {
    thread.Start();
   }
catch
    {
    Console.WriteLine("Error during starting thread");
    }
```

- Takie uruchomienie zwróci nam jedynie wyjątek przy uruchamianiu, nie przechwycimy wyjątku "rzuconego" w wątku.
- Wyjątki z wątków mogą zakończyć aplikację, trzeba je przechwytywać na poziomie wątków.

Czasomierze

- System.Timers.Timer

```
static void Main(string[] args)
        {
            System.Timers.Timer timerSystem = new System.Timers.Timer(1500);
            timerSystem.Elapsed += TimerSystem_Elapsed;
            //  timerSystem.AutoReset = false; //If we want one shot
            timerSystem.Start();
            Console.ReadLine();
            timerSystem.Stop();
         }
private static void TimerSystem_Elapsed(object sender,
System.Timers.ElapsedEventArgs e)
  {
  Console.WriteLine("System timer tick tok...");
  }
```

Czasomierze

- System.Threading.Timer

```
   static void Main(string[] args)
        {TimerState ts = new TimerState() { Counter = 0 };
            System.Threading.Timer timerThreading = new
System.Threading.Timer(new System.Threading.TimerCallback(TimerTick), ts, 3000,
1000);
            Console.ReadLine();
            timerThreading.Change(System.Threading.Timeout.Infinite, 0);
            Console.WriteLine("The End");
         }        private static void TimerTick(object timerState)
        {
            TimerState tState = (TimerState)timerState;
            System.Threading.Interlocked.Increment(ref tState.Counter);
            Console.WriteLine($"Timer tick set counter to: {tState.Counter}");
        }
        class TimerState
        {
            public int Counter;
        }
```

Synchronizacja

Blokowanie

- Procesy zablokowane z powodu oczekiwania na jakieś zdarzenie, np. **`Sleep, Join, lock, Semaphore`** itp.
  Natychmiastowo zrzekają się czasu procesora, dodają **`WaitSleepJoin`** do właściwości **`ThreadState`** i nie kolejkują się do czasu odblokowania.
- Odblokowanie może nastąpić z 4 przyczyn:
  - Warunek odblokowania został spełniony
  - Minął timeout
  - Został przerwany przez **`Thread.Interrupt`**
  - Zosatł przerwany przez **`Thread.Abort`** `(.net<5.0 nie .net core)`

Synchronizacja

Oczekiwanie **Sleep** i **SpinWait**

```
Thread.Sleep(0); // resign of the assigned time quantum
Thread.Sleep(1000); // sleep for 1000 ms
Thread.Sleep(TimeSpan.FromHours(1)); // sleep for 1 hour
Thread.Sleep(Timeout.Infinite); // sleep forever until break.
```

- Ogólnie Sleep powoduje rezygnację wątku z czasu procesora. Wątek taki nie jest kolejkowany przez podany czas.

```
Thread.SpinWait (100); // nic nie rób przez 100 cykli
```

- Wątek nie rezygnuje z procesora, jednak wykonuje na nim puste operacje. Nie jest w stanie **WaitSleepJoin** i nie może być przerwany przez **Interrupt**. Można zastosować gdy chcemy czekać bardzo krótko.
- Podobnie zachowuje się wątek podczas aktywnego czekania

Synchronizacja

Oczekiwanie na potomka - **Join**

```
Thread thread1 = new Thread(() =>
    {
     Console.WriteLine("Thread started and going to sleep for 3 s...");
     Thread.Sleep(3000);
     Console.WriteLine("Thread woke up and finished");
     });
thread1.Start();
Console.WriteLine("Main thread has started child thread and going to
wait for it...");
thread1.Join();
Console.WriteLine("Main thread lived to see");
```

- Czekamy na zakończenie watka. Mechanizm, zbierania komunikatów nie jest zatrzymany, więc jak klikniemy jakiś guzik w wątku głównym to ostatecznie doczekamy się reakcji.

Sekcja krytyczna

- Kiedy blokować
  - Wszędzie tam, gdzie wiele wątków może mieć dostęp do wspólnych zmiennych
  - Wszędzie tam, gdzie chcemy mieć niepodzielność operacji, np. sprawdzenie warunku i wykonanie czegoś
- Na co uważać
  - Nie powinniśmy zbyt dużo blokować bo ciężko analizować taki kod a łatwo spowodować *DeadLock*
  - Zbyt duże fragmenty kodu wykonywane przez pojedynczy proces powodują zubożenie współbieżności.

Sekcja krytyczna

Wiele wątków (n_threads) wykonuje to samo zadanie na wspólnej zmiennej:

```
for (int ii = 0; ii < 1000000; ii++)
   {
        {
            counter++;
        }
   }
```

Bez zabezpieczeń wynik będzie wynosił <= n_threads*1000000

Sekcja krytyczna

lock

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

...

for (int ii = 0; ii < 1000000; ii++)
      {
      lock (locker)
         {
         counter++;
         }
      }
```

- W danym momencie tylko jeden wątek, może przebywać w chronionym obszarze inne będą czekały w kolejce *FIFO*
- Wątki czekające są w stanie **`WaitSleepJoin`**.
- Wątki takie można też zakończyć przez przerwanie lub *abort*

Sekcja krytyczna

Wybór obiektu który będzie blokował

- Musi to być typ referencyjny
- Zwykle jest związany z obiektami na których działamy np.:

```
List <string> list = new List <string>();
void Test() {
   lock (list) {
      list.Add ("Item 1");
...
```

- Powinniśmy stosować obiekty, które są `private` by uniknąć niezamierzonej interakcji z zewnątrz
- Z tego samego powodu nie powinniśmy stosować np. `lock (this){}` lub `lock (typeof (Widget)) { ... }`
- Użycie obiektu do zablokowania fragmentu kodu nie powoduje automatycznie blokowania danego obiektu.
  - `lock(list1) { list2.Add ("Item 1"); ...`

Sekcja krytyczna

Blokowanie zagnieżdżone

```
static object x = new object();
  static void Main() {
  lock (x) {
     Console.WriteLine ("I locked");
     Nest();
     Console.WriteLine ("I unlocked");
     }
  //Completly unlocked
  }

static void Nest() {
  lock (x) {
  …//double lock
  }
  //last unlock
  }
```

Wątek blokujący może blokować ile chce, jednak blokowany i tak będzie czekał na tym najbardziej zewnętrznym.

Sekcja krytyczna

`Monitor` – rozwinięcie `lock`

- `lock` faktycznie jest skrótem składniowym czegoś takiego:

```
Monitor.Enter(locker);
  try
        {
        counter++;
        }

finally
        {
        Monitor.Exit(locker);
        }
```

- Wywołanie `Monitor.Exit` bez uprzedniego `Monitor.Enter` spowoduje rzucenie wyjątku

- Monitor posiada też metodę `TryEnter` gdzie możemy podać timeout jeżeli wejdziemy przed końcem czasu to zwróci true lub false jeżeli wystąpi timout.

Sekcja krytyczna

`Monitor` – rozwinięcie `lock`

```
Thread[] tharray3 = new Thread[100];
for (int ii = 0; ii < 100; ii++)
    {
    tharray3[ii] = new Thread(ThreadWithMonitor);
    tharray3[ii].Name = "Thread " + ii.ToString();
    }
sw.Restart();
foreach (var th in tharray3)
    {
    th.Start();
    }
foreach (var th in tharray3)
    {
    th.Join();
    }
Console.WriteLine($"All Monitor threads finished in
{sw.ElapsedMilliseconds} ms counter = {counter}");
```

Sekcja krytyczna

Interlocked – operacje atomowe

```
for (int ii = 0; ii < 1000000; ii++)
    {
    Interlocked.Increment(ref counter);
    }
```

- Dodatkowo mamy do dyspozycji
  - `Add` Dodadawanie dwóch liczb
  - `CompareExchange` porównanie i ewentualna podmiana
  - `Decrement` zmniejszenie
  - `Equals` czy równe
  - `Exchange` Zamiana
  - `Read` odczyt liczby 64b
  - `ReferenceEquals` porównanie dwóch referencji

Sekcja krytyczna

Mutex

```
static Mutex mutex = new Mutex(false, "tu.kielce.pl mutex");
        static void ThreadWithMutex(object o)
        {
            //it is very slow, there is 100 times less iterations
            for (int ii = 0; ii < countFor/100; ii++)
            {
                mutex.WaitOne();
                counter++;
                mutex.ReleaseMutex();
            }
        }
```

Sekcja krytyczna

Semaphore

```
static Semaphore sem = new Semaphore(1, 1);
static void ThreadWithSemaphore()
{
    //it is very slow, there is 100 times less iterations
    for (int ii = 0; ii < countFor/100; ii++)
    {
        sem.WaitOne();
        counter++;
        sem.Release();
    }
}
```

Sekcja krytyczna

SemaphoreSlim

- **SemaphoreSlim** jest lekką alternatywą dla Semaphore i może być używany do synchronizacji tylko w obrębie jednego procesu.
- Nie obsługuje nazwanych semaforów systemowych.

```
static SemaphoreSlim semSlim = new SemaphoreSlim(1, 1);
static void ThreadWithSemaphoreSlim()
{   //Notice. It isn't so slow as Semaphore
    for (int ii = 0; ii < countFor; ii++)
    {
        semSlim.Wait();
        counter++;
        semSlim.Release();
    }
}
```

Sekcja krytyczna

SpinLock

- SpinLock - wykonuje puste instrukcje w oczekiwaniu na zwolnienie blokady.
- Może okazać się wydajniejszy od innych rodzajów blokad, jeżeli uruchamiany jest na wielordzeniowych komputerach, gdy spodziewamy się krótkiego czasu oczekiwania oraz gdy rywalizacja będzie minimalna.
- używaj SpinLock tylko wtedy, gdy stwierdzisz przez profilowanie, że metody z System.Threading.Monitor lub metody Interlocked są znacznie wolniejsze
- Pamiętaj, w celu zapewnienia najlepszej wydajności należy użyć **false** w metodzie wyjścia,
- **true** jest używany w architekturach IA64 przy barierach pamięci, które synchronizują zapisy i odczyty do/z pamięci

```
static SpinLock spinLock = new SpinLock();
static void ThreadWithSpinLock()
{
    bool lockTaken = false;
    for (int ii = 0; ii < countFor; ii++)
    {
        lockTaken = false;
        spinLock.Enter(ref lockTaken);
        {
            counter++;
        }
        spinLock.Exit(false);
    }
}
```

Sekcja krytyczna

- Inny przykład. Synchronizacja przez blokadę, ale dane są podzielone na większe porcje. Lock nie jest wywoływany tak wiele razy.

```
static void ThreadWithLock2()
        {
            for (int ii = 0; ii < 1000; ii++)
            {
                lock (locker)
                {
                    for (int jj = 0; jj < 1000; jj++)
                        counter++;
                }
            }
        }
```

Sekcja krytyczna

- podsumowując

```
--------------- Unsafe thread increment --------------
All unsafe threads finished in 1221 ms counter = 24268196
--------------- lock thread increment --------------
All lock threads finished in 5857 ms counter = 100000000
--------------- Monitor thread increment --------------
All Monitor threads finished in 6363 ms counter = 100000000
--------------- Interlocked thread increment --------------
All Interlocked threads finished in 3504 ms counter = 100000000
--------------- mutex thread increment --------------
All mutex threads finished in 4822 ms counter = 1000000   t x 100
--------------- semaphore thread increment --------------
All semaphore threads finished in 4799 ms counter = 1000000  t x 100
--------------- semaphore slim thread increment --------------
All semaphore slim threads finished in 14417 ms counter = 100000000
--------------- SpinLock thread increment --------------
All SpinLock threads finished in 13190 ms counter = 100000000
--------------- lock thread increment with more granulity --------------
All lock 2 threads finished in 286 ms counter = 100000000
```

Widzimy, że racjonalne dzielenie obliczeń jest najlepsze

Context Bound Object

Tylko w .net framework. Automatyczne blokowanie wywołań metod z jednej instancji klasy.

```
using System.Runtime.Remoting.Contexts;

[Synchronization]
  public class SafeClass : ContextBoundObject
  {
  }
```

CLR (Common Language Runtime) zapewnia, że tylko jeden wątek może wywołać kod tej samej instancji obiektu w tym samym czasie. Sztuczka polega na tym, że podczas tworzenia obiektu klasy **JakasKlasa** tworzony jest obiekt proxy, przez którego przechodzą wywołania metod klasy **JakasKlasa**.

Context Bound Object

```
[Synchronization]
class SafeCounter : ContextBoundObject
{
    public int Counter { get; }
    public SafeCounter()
    {
        Counter = 0;
    }

    public void CounterInc()
    {
        Counter++;
    }
}

static SafeCounter sf = new SafeCounter();
static void ThreadWithoutProtection()
    {
    //it is very slow
    for (int ii = 0; ii < 10000; ii++)
            {
            sf.CounterInc();
            }
    }
```

Context Bound Object

```
static void Main(string[] args)
    {
    Console.WriteLine("--------------- SafeCounter ContextBoundObject
thread increment -------------- ");
    Thread[] tharray = new Thread[100];
    for (int ii = 0; ii < 100; ii++)
        {
        tharray[ii] = new Thread(ThreadWithoutProtection);
        tharray[ii].Name = "Thread " + ii.ToString();
        }
Stopwatch sw = new Stopwatch();
sw.Start();
foreach (var th in tharray)
    {
    th.Start();
    }
foreach (var th in tharray)
    {
    th.Join();
    }
Console.WriteLine($"All ContextBoundObject threads finished in
{sw.ElapsedMilliseconds} ms counter = {sf.Counter}");
            Console.ReadLine();
}
```

Context Bound Object

Automatyczna synchronizacja nie może być stosowana do pól *protect static* ani klas wywodzących się od **`ContextBoundObject`** np. Windows Form

Trzeba też pamiętać, że nadal nie rozwiązuje nam to problemu gdy wywołamy dla kolekcji coś takiego:

```
 BezpiecznaKlasa bezpieka = new
  BezpiecznaKlasa();

...

if (bezpieka.Count > 0) bezpieka.RemoveAt (0);
```

Context Bound Object

Jeżeli z bezpiecznego obiektu tworzony jest kolejny obiekt to automatycznie jest on też bezpieczny w tym samym kontekscie, chyba, że postanowimy inaczej za pomocą atrybutów.

```
[Synchronization (SynchronizationAttribute.REQUIRES_NEW)]

public class JakasKlasaB : ContextBoundObject { ...
```

NOT_SUPPORTED - równoważne z nieużywaniem Synchronized

SUPPORTED - dołącz do istniejącego kontekstu synchronizacji jeżeli jest stworzony z innego obiektu, w innym przypadku będzie niesynchronizowany

REQUIRED - (domyślny) dołącz do istniejącego kontekstu synchronizacji jeżeli jest stworzony z innego obiektu, w innym przypadku stwórz swój nowy kontekst synchronizacji

REQUIRES_NEW - Zawsze twórz nowy kontekst synchronizacji

Przerywanie wątku

- `Thread.Interrupt` – przerywa bieżące czekanie i powoduje rzucenie wyjątku `ThreadInterruptedException`

```
static void InfiniteThread()
      {
      try
        {
        Thread.Sleep(Timeout.Infinite);
        }
      catch (ThreadInterruptedException ex)
        {
        Console.WriteLine("InfiniteThread cought an exception: " + ex.Message);
        }
      Console.WriteLine("InfiniteThread ended normaly");
      }
```

- Należy pamiętać, że przerywanie w ten sposób może być niebezpieczne, chyba, że wiemy dokładnie w którym miejscu jesteśmy i posprzątamy.

Abort

- `Thread.Abort` – Działa podobnie jak Interrupt z tą różnicą, że rzuca wyjątkiem `ThreadAbortException` oraz wyjątek jest ponownie rzucany pod koniec bloku catch, chyba że w bloku catch zastosujemy `Thread.ResetAbort();`
- Działanie jest podobne, jednak w przypadku `Interrupt` wątek przerywany jest tylko w momencie czekania, `Abort` może tego dokonać w dowolnym miejscu wykonywania, nawet w nienaszym kodzie.
- To stara metoda i nie należy jej używać. Jeśli chcem zakończyć jakiś kod w taki dość "brutalny" sposób. Możemy uruchomić kod w osobnym procesie i użyć wywołać **Kill**
- Abort w .net Core nie jest wspierane.

Stany wątków

- **`ThreadState`** – kombinacja bitowa trzech warstw.
- Uruchomienie, blokada, przerwanie wątku (`Unstarted, Running, WaitSleepJoin, Stopped, AbortRequested`)
- Pierwszoplanowość i drugoplanowość wątku (`Background, Foreground`)
- Postęp w zawieszeniu wątku (`SuspendRequested, Suspended`) używane przez przestarzałe metody
- Ostateczny stan wątku określa się przez sumę bitową tych trzech „Warstw”. I tak, może być np wątek
  - `Background, Unstarted`

  lub

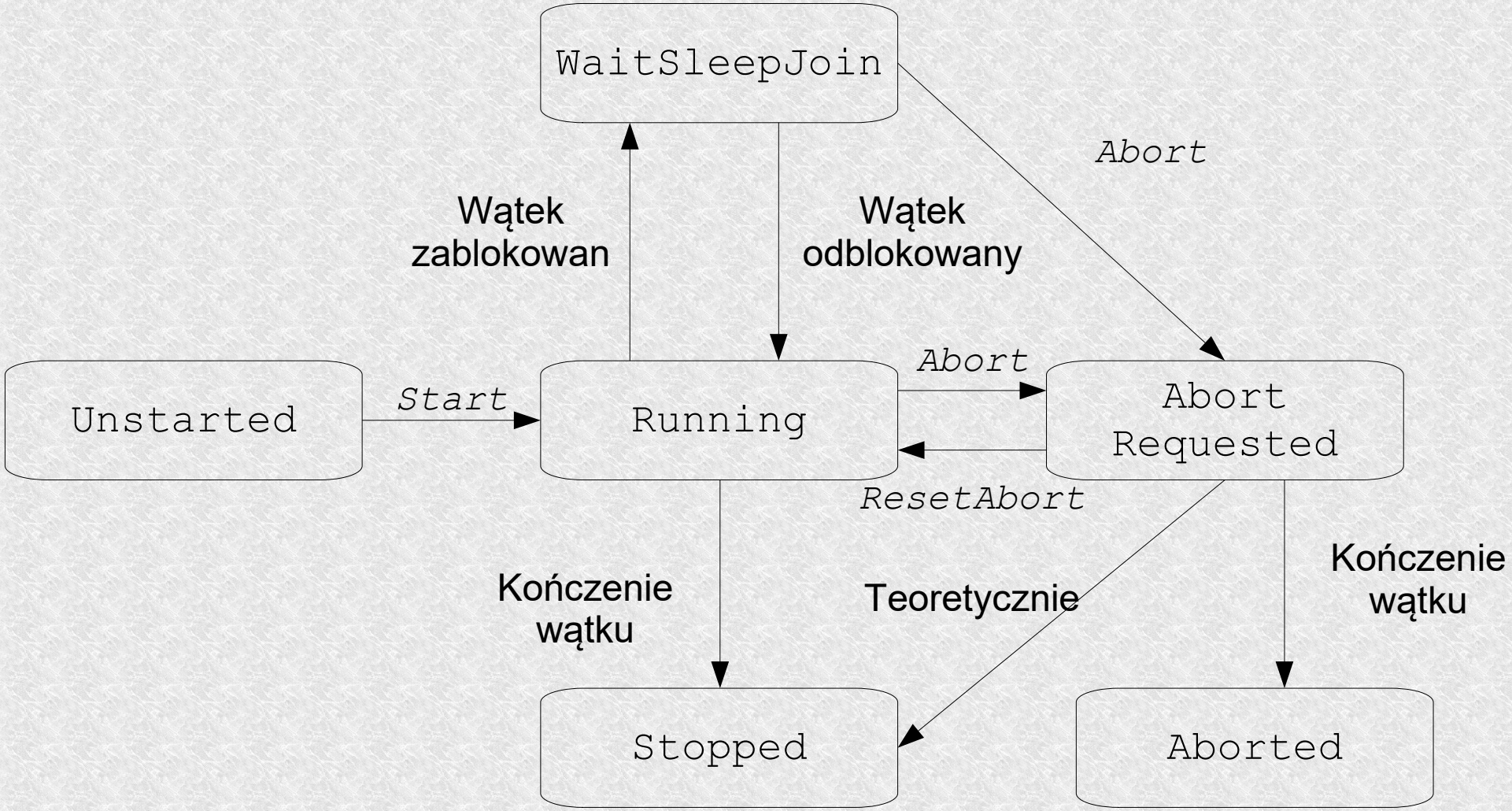
  `SuspendRequested, Background, WaitSleepJoin`

Stany wątków

- W enumeracji ThreadState są też nigdy nie używane dwa stany: **`StopRequested`** i **`Aborted`**
- By jeszcze bardziej skomplikować, *Running* ma wartość 0
  więc porównanie
  `if ((t.ThreadState & ThreadState.Running) > 0)...`
  nic nam nie da
- Można się wspomóc *IsAlive* jednak zwraca *false* tylko przed startem i gdy się zakończy. Gdy jest zablokowany też jest *true*.
- Najlepiej napisać sobie swoją metodę:

```
public static ThreadState SimpleThreadState (ThreadState ts)
    {
    return ts & (ThreadState.Aborted |
                 ThreadState.AbortRequested |
                 ThreadState.Stopped |
                 ThreadState.Unstarted |
                 ThreadState.WaitSleepJoin);
    }
```

Stany wątków



Dziękuję za uwagę