SPRAWOZDANIE

Programowanie równoległe

Wielowątkowość

03.12.2015

Karol Suwalski 125NCI B

https://github.com/SuwalskiKarol/oor.git

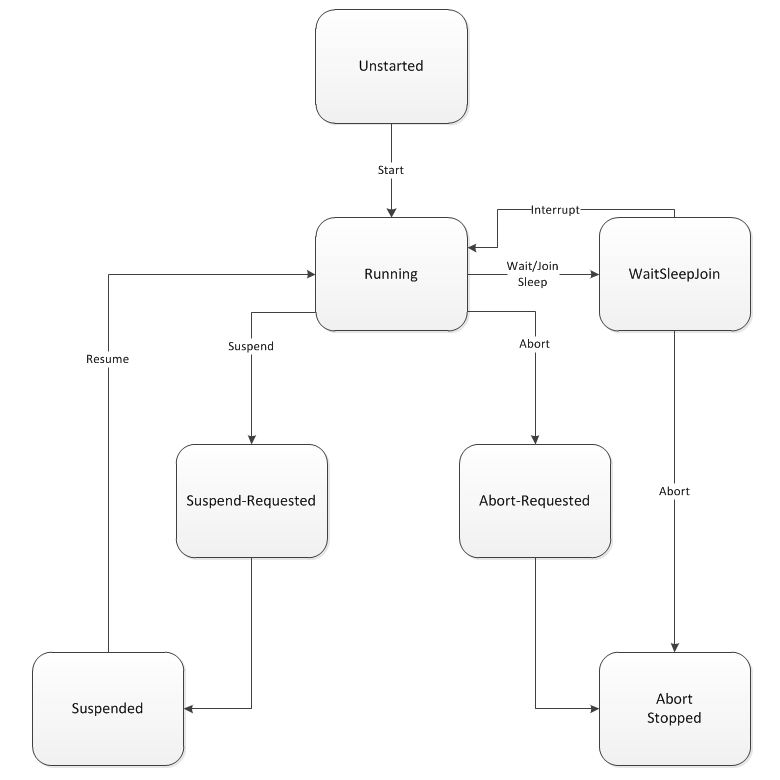
**Wstęp**

Równoległość działania wątków osiągana jest przez mechanizm przydzielania czasu procesora poszczególnym wykonującym się wątkom. Każdy wątek uzyskuje dostęp do procesora na krótki czas (kwant czasu), po czym „oddaje procesor” innemu wątkowi.

Jest to zagadnienie obsługiwane przez większość nowoczesnych języków programowania - w tym również C#. Wątek **- thread** umożliwia nam wykonanie dodatkowych operacji na raz. Obsługiwane jest to właśnie przez **klasę Thread** w przestrzeni nazw **System.Threading.** Ilość wątków teoretycznie jest nieograniczona - praktycznie ogranicza nas ilość zasobów sprzętowych. Każda aplikacja domyślnie posiada jeden wątek - wątek główny - czy to aplikacja konsolowa, czy okienkowa.

**Stany wątków**

Schemat przedstawiający stany wątków oraz metody z nimi powiązane w frameworku .NET:



**Tworzenie i uruchamianie wątków**

ThreadStart startDelegate = new ThreadStart(ds.Start);

Thread t = new Thread(startDelegate);

t.IsBackground = true;

t.Start();

t.Priority = ThreadPriority.AboveNormal;

Uruchamianiem wątków i zarządzaniem nimi zajmuje się klasa **Thread.**

**ThreadStart**  jest delegatem. Czym są delegaty? Najprościej mówiąc, służą one do przekazywania metod, jako argumentów do innych metod. Czyli metoda ThreadStart wskazuję na metodę, która zostanie wykonana, jako Thread w innej metodzie.

Kiedy właściwość **IsBackground** jest prawdą to wątek ten jest niezdolny do przerwania. Kończy działanie dopiero, kiedy proces się skończy. Właściwość raczej rzadko ustawiana właśnie ze względu na to, że takiego wątku nie możemy w żaden sposób zatrzymać, oczyścić czy uruchomić od nowa.

**ThreadPriority** określa priorytet wątku. Własność ta służy do odróżnienia wątków ważnych dla naszej aplikacji od tych pomocniczych. W technologii .Net wyróżniamy 5 priorytetów. Trzeba jednak uważać przy zmianie priorytetu ze względu chociażby na to, że wątki z najwyższym priorytetem mogą konkurować z wątkami systemowymi i możemy sobie ładnie namieszać.

**Thread-Local Storage**

Załóżmy, że piszemy wiele wątków i chcemy, aby każdy wątek posiadał własną kopię danych. Trochę lipa, ponieważ każdy wątek posiada swój stos i po skończeniu wątku dane są niszczone. Możemy oczywiście tworzyć mnóstwo publicznych parametrów, ale zawsze będzie to tylko jedna kopia danych, z których mogą korzystać wszystkie wątki. Thread-Local Storage jest właśnie metodą, dzięki której każdy wątek może mieć własną kopie danych. W .Net mamy 3 metody tworzenia TLS.

**Atrybut ThreadStatic**

Atrybut ten stawiamy przed statycznymi polami. Dzięki niemu każdy wątek będzie posiadał własną kopię podanych danych.

Tak więc zamiast bawić się przy każdym wątku w słowo **Thread.Current:**

Thread.Current["MojaKlasa.liczba"] = 1;

Thread.Current["MojaKlasa.liczba"] += 1;

Możemy na początku użyć właśnie atrubutu ThreadStatic:

class MojaKlasa {

[ThreadStatic]

static int liczba;

}

// .. then

MojaKlasa.liczba = 1;

MojaKlasa.liczba += 1;

Problem w tym, że ta metoda działa tylko z static fieldsami i nie sprawdza się przy instancjach klasy.

**Klasa ThreadLocal <>**

Klasa ta poprawia błędy związane z ThreadStatic i możemy jej użyć przy obiektach klasy.

static ThreadLocal<string> siema;

ThreadLocal<int> liczbaThreads = new ThreadLocal<int>(() => 1); // instance field

**Thread.SetData, Thread.GetData**

Są to metody stosowane do przechowywania oraz do pobierania lokalnych danych z konkretnych ‘slotów’. Jeśli damy slotowi nazwę to dane przechowywane w tym slocie będą dostępne w całej aplikacji właśnie pod tą nazwą.

//setting

LocalDataStoreSlot lds = System.Threading.Thread.AllocateNamedDataSlot("jakasFajnaNazwaSlotu");

System.Threading.Thread.SetData(lds, " jakieś super ważne dane");

//getting

LocalDataStoreSlot lds = System.Threading.Thread.GetNamedDataSlot("jakasFajnaNazwaSlotu");

string somevalue = System.Threading.Thread.GetData(lds).ToString();

**Synchronizacja wątków**

**Lock**

Lock to najprostszy z mechanizmów, który pozwala na zapewnienie odpowiedniego dostępu do zasobów. Stosując lock na wybranym fragmencie kodu, mamy pewność, że w danym momencie tylko jeden wątek uzyska do niego dostęp.

private static readonly object \_locker = new object();

class Synchronizacja

{

static readonly object \_locker = new object();

static int val1, val2;

static void Go()

{

lock (\_locker)

{

if (val2 != 0) Console.WriteLine (val1 / val2);

val2 = 0;

}

}

Zastosowanie obiektu **\_locker** umożliwia odnoszenie blokady do konkretnego, znanego zasobu.

Można by do tego dodać instancję ze słowem kluczowym **volatile**, które zapobiega różnym zawiłościom związanym z cache'owaniem. Mogłaby się zdarzyć sytuacja, w której dwa wątki próbowałyby jednocześnie dostać się do jeszcze nieutworzonej instancji. Jeden z nich by ją utworzył, a drugi w wyniku cache'owania obiektów niepotrzebnie założyłby blokadę.

private static volatile Synchronizacja instancja = null;

**Monitor**

Monitor ma praktycznie takie samo zastosowanie jak lock, jednak jest odrobinę bardziej umolny w implementacji.

private static readonly object \_locker = new object();

class Synchronizacja

{

static readonly object \_locker = new object();

static int val1, val2;

static void Go()

{

Monitor.Enter (\_locker)

try

{

if (val2 != 0) Console.WriteLine (val1 / val2);

val2 = 0;

}

finally

{

Monitor.Exit(\_locker)

}

}

W tym przypadku konieczne będzie zastosowanie klauzuli try-finally, które zapewni zdjęcie blokady nawet w przypadku błędu operacji wykonanej w sekcji krytycznej. Monitor posiada również kilka innych funkcji. Metoda **TryEnter** sprawdza czy wejście do sekcji krytycznej jest możliwe i w razie takiej możliwości, automatycznie tam wchodzi. Metody **Wait** oraz **Pulse**. Pierwsza z metod powoduje wstrzymanie określonego wątku do czasu aż druga z metod wykonywana w innym wątku na to nie pozwoli.

**MUTEX**

Mutex jest w pewnym sensie podobny do poprzednich metod. Zasadniczą jego wadą jest to, że jego użycie kosztuje więcej pamięci oraz zasobów. Mutex posiada jednak za to pewną zaletę niedostępną dla dwóch powyższych rozwiązań, czyli możliwość blokowania dostępu do określonego zasobu pomiędzy różnymi procesami działającymi w systemie. Mutex może mieć, zatem swego rodzaju zasięg globalny w obszarze całego systemu - nie tylko w pojedynczej instancji aplikacji.

private readonly string MUTEX\_GUID = "969f7729-3867-4583-8bd6-e990d262c94e";

private Mutex mutexxx = null;

public Costam ()

{

mutexxx = new Mutex(false, MUTEX\_GUID);

}

public void Synchronizacja ()

{

mutexxx.WaitOne();

try

{

//sekcja krytyczna

}

finally

{

mutexxx.ReleaseMutex();

}

}

Zacznijmy od tego, że aby nasz Mutex zadziałał poprawnie w aspekcie systemowym, to musi mieć odpowiednią nazwę. Nazwa może być dowolna (pod warunkiem, że wszystkie aplikacje/wątki będą się do niej stosować), ale zalecam stosowanie w miarę unikalnej. Dobrym rozwiązaniem może być dlatego GUID. Najpierw oczekujemy na możliwość wejścia do sekcji krytycznej za pomocą metody **WaitOne**. Nasza aplikacja będzie musiała poczekać na swoją kolej. Kiedy tylko dostęp będzie możliwy, nasza instancja zablokuje Mutex dla siebie, a następnie wykona kod umieszczony w sekcji krytycznej. Później powinniśmy koniecznie zwolnić Mutex.

**SEMAFOR**

Semafor to klasa zbliżona i bardzo podobna do Mutexa. Ich podobieństwo związane jest ze wspólnym ojcem, ponieważ obie klasy dziedziczą po **WaitHandle**. W obu przypadkach możemy, zatem korzystać z metod **WaitOne**. Zasadniczą różnicą jest w tym przypadku to, że Semafor w przeciwieństwie do Mutexa umożliwia korzystanie ze strefy krytycznej dowolnej liczbie wątków, którą to my sami definiujemy w konstruktorze.

Najprostszy z konstruktorów, który wystarczy w większości przypadków przyjmuje dwa argumenty. Pierwszy z nich określa nie, jako liczbę zajętych miejsc w semaforze.

Możemy w tym przypadku podać wartość 0 lub większą (0 oznacza, że wszystkie miejsca są zajęte przez wątek główny), drugi natomiast określa maksymalną liczbę dostępnych miejsc.

class Semafor

{

private Semaphore m\_oSemaphore = null;

public Semafor()

{

m\_oSemaphore = new Semaphore(1, 1);

}

public void Run()

{

try

{

m\_oSemaphore.WaitOne();

/// kod sekcji krytycznej

}

finally

{

m\_oSemaphore.Release();

}

}

}

Klasa Semafora posiada również odpowiednie metody. Nasze omówienie zaczniemy od WaitOne. Metoda ta zasadniczo wprowadza nas do sekcji krytycznej. Jeśli liczba dostępnych miejsc w danym momencie jest większa niż 0, nasz wątek po prostu przejdzie przez te metodę bez żadnego oczekiwania i rozpocznie realizację zadań zawartych w sekcji krytycznej. W przeciwnym przypadku, będzie musiał zaczekać aż zwolni się jakieś miejsce.

**MANUALRESETEVENT I AUTORESETEVENT**

Idea działania tych klas, polega na blokowaniu dostępu do określonych sekcji, dopóki nie nastąpi sygnalizacja obiektu.

Dostępu do sekcji krytycznej broni metoda WaitOne, na której przetwarzanie zostanie wstrzymane, dopóki nie otrzymamy sygnału (metoda **Set**) na obiekcie typu **ResetEvent**. Dochodzimy, więc powoli teraz do zasadniczej różnicy pomiędzy obiema klasami.

W przypadku klasy AutoResetEvent, sygnalizacja jest automatycznie wyłączana jak tylko wejdziemy do sekcji krytycznej (to jest przekroczymy metodę WaitOne).

W przypadku ManualResetEvent, dzieje się tak tylko w momencie, gdy na obiekcie zastosujemy metodę **Reset**. W przeciwnym razie obiekt cały czas będzie zasygnalizowany i dowolna liczba wątków będzie w stanie wejść do sekcji krytycznej. Zastosowanie ręcznego mechanizmu daje nam większą kontrolę nad całym procesem.

ManualResetEvent oManualResetEvent = new ManualResetEvent(false);

// Sygnalizacja

oManualResetEvent.Set();

try

{

// Oczekiwanie, ponieważ obiekt jest zasygnalizowany od razu wejdziemy

// do sekcji krytycznej

oManualResetEvent.WaitOne();

// Sekcja krytyczna

}

finally

{

// Blokowanie

oManualResetEvent.Reset();

}

// False, obiekt nie jest zasygnalizowany, musimy wykonać ręczną sygnalizację

AutoResetEvent oAutoResetEvent = new AutoResetEvent(false);

// Sygnalizacja

oAutoResetEvent.Set();

// Oczekiwanie i zarazem blokowanie. Wejście do sekcji krytycznej

// nastąpi natychmiastowo

oManualResetEvent.WaitOne();

**INTERLOCKED**

Dzięki Interlocked możemy bezpiecznie operować na zmiennych oraz ich wartościach. Jest kontenerem dla szeregu statycznych metod, które umożliwiają przeprowadzanie atomowych operacji na zmiennych, z których mogą korzystać różne wątki. Przykłady metod:

Add - umożliwia zwiększenie/zmniejszenie wartości liczby całkowitej. Zmienną przekazujemy przez referencję:

int nTest = 99;

Interlocked.Add(ref nTest, 1);

Equals - porównanie dowolnych dwóch obiektów:

string sA = "a";

string sB = "b";

Interlocked.Equals(sA, sB);

Jest wiele więcej metod synchronizacji wątków w .NET chociażby: blocking, CountdownEvent, WaitAny, WaitAll, SignalAndWait, Reentrancy ale ich nie znam(jeszcze).

**Thread vs ThreadPool vs Task vs Parallel**

**Thread**

Jest metodą najstarszą (co za tym idzie coraz mniej wspieraną). Reprezentuje rzeczywisty wątek z poziomu systemu operacyjnego, wraz z jego zasobami jądra, stosu i takie tam. Thread umożliwia najwyższy stopień kontroli(Abort, Suspend, Resume). Umożliwia zmianę wielu właściwości jak wielkość stosu, pamięci czy tzw ApartmentState(STA i MTA).

Problemem Thread jest fakt, że wątki systemowe są kosztowne, co bardzo obciąża procesor.

**ThreadPool**

Jest wrapperem wprowadzonym przez CLR(środowisko uruchomieniowe dla .NET, dzięki któremu można pracować również na Unixach czy macintoshach). ThreadPool nie daje żadnej kontoli, można tylko kontrolować wielkość puli wątków. Nie można nawet stwierdzić, kiedy zacznie pracować i kiedy zakończy prace(brak czegoś na wzór Thread.Join()).

Plusem jest, że dzięki ThreadPool unikniemy tworzenia zbyt dużo niepotrzebnych wątków, wkońcu to my kontrolujemy ich ilość. ThreadPool nadaje się do zarządzania wieloma, krótkimi operacjami. Nic więcej.

**Task**

O Taskach będę gaworzył przy sprawozdaniu nt. równoległości opartej na procesach także tu się ograniczę do minimum.

Task jest aktualnie prawie zawsze najlepszym rozwiązaniem do zarządzania wątkami. Nie tworzy własnego wątku systemowego. Zamiast tego zadania wykonywane są przez TaskScheduler. Zapewnia o wiele bardziej zaawansowane API i eliminuje marnowanie wątków OS.

**Parallel**

Klasa Parallel służy w sumie tylko jednej rzeczy. Dzięki niej możemy zamienić każdą pętle for lub foreach w wielowątkowy kod. Każda iteracja pętli będzie inteligentnie przypisywana do oddzielnego wątku i będzie wykonywana w sposób możliwie najbardziej efektywny.

**Podsumowanie**

Patrząc na ilość stron(prawie 450) w książce „Programowanie równoległe i asynchroniczne w C# 5.0” (Wydawnictwo Helion) a to co opisałem w tym sprawozdaniu kończy się na 30 stronie mogę stwierdzić, że liczba metod, funkcji, możliwości związanych z multithreadingiem w frameworku .NET jest równa:

