packt

Aplikacje i usługi na platformie .NET 7

Tworzenie praktycznych projektów opartych na programach Blazor, .NET MAUI, gRPC, GraphQL i innych zaawansowanych technologiach

Helion

Mark J. Price

Tytuł oryginału: Apps and Services with .NET 7: Build practical projects with Blazor, .NET MAUI, gRPC, GraphQL, and other enterprise technologies

Tłumaczenie: Andrzej Watrak

ISBN: 978-83-8322-716-0

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/aplius
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

Rozdział 4

Ocena wydajności kodu, wielozadaniowość i współbieżność

Ten rozdział jest poświęcony jednoczesnemu wykonywaniu wielu operacji przez aplikację, aby praca jej użytkownika była bardziej wydajna, skalowalna i efektywna.

W tym rozdziale są opisane następujące zagadnienia:

- procesy, wątki, zadania,
- monitorowanie wydajności aplikacji i wykorzystania zasobów,
- asynchroniczne wykonywanie zadań,
- synchronizacja dostępu do współdzielonych zasobów,
- instrukcje `async` i `await`.

Procesy, wątki, zadania

Proces to na przykład aplikacja konsolowa, taka jak jedna z utworzonych wcześniej, której są przydzielane zasoby, m.in. pamięć i wątki.

Wątek to kod wykonywany instrukcja po instrukcji. Domyślnie proces ma tylko jeden wątek, co może powodować problemy, gdy kilka zadań musi być wykonanych w tym samym czasie. Wątki są również wykorzystywane do kontrolowania różnych rzeczy, na przykład uwierzytelnionego w danym czasie użytkownika lub zasad internacjonalizacji właściwych dla bieżącego języka i regionu.

Większość nowoczesnych systemów operacyjnych, w tym Windows, wykorzystuje **wielozadaniowość z wywłaszczaniem** (ang. *preemptive multitasking*), która symuluje równoległe wykonywanie zadań. Polega ona na przydzielaniu czasu procesora każdemu wątkowi, jednemu po drugim. Procesor zawiesza bieżący wątek po upływie przydzielonego mu czasu, po czym wykonuje następny wątek przez zadany okres.

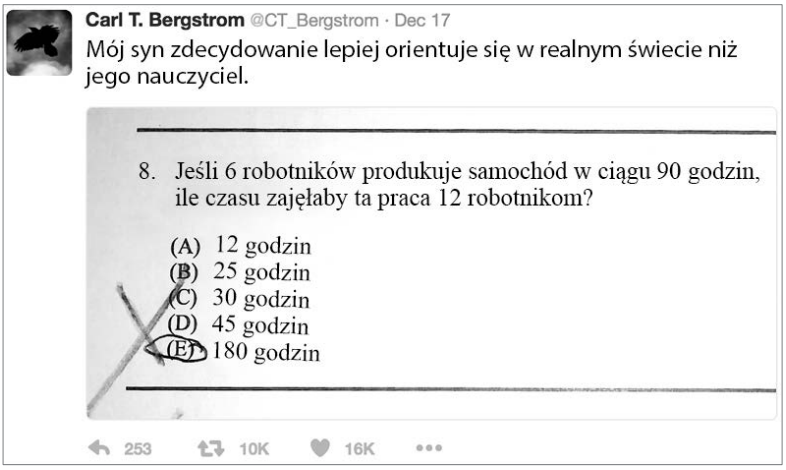
System Windows podczas przełączania umieszczonych w kolejce wątków zapisuje ich konteksty, a następnie ponownie je ładuje. Wymaga to zarówno czasu, jak i zasobów.

Jeżeli programista musi zaimplementować niewielką liczbę złożonych operacji i chce mieć nad nimi pełną kontrolę, może utworzyć kilka instancji klasy `Thread` i zarządzać nimi. Jeżeli jest jeden główny wątek i wiele małych operacji, które mogą być wykonywane w tle, można użyć klasy `ThreadPool`, utworzyć instancje delegatów wskazujących te operacje zaimplementowane jako metody i umieścić je w kolejce. Instancje te są automatycznie przydzielane do wątków znajdujących się w puli.

W tym rozdziale użyjesz typu `Task` do zarządzania wątkami na wyższym poziomie abstrakcji.

Wątki mogą konkurować między sobą o dostęp do współdzielonych zasobów, takich jak zmienne, pliki czy obiekty bazy danych. Można nimi zarządzać, o czym się przekonasz w praktyce w dalszej części rozdziału.

Nie zawsze podwojenie liczby wątków skraca czas wykonania danej operacji o połowę. To zależy od rodzaju operacji. Może się zdarzyć, że ten czas nawet się wydłuży, co ilustruje rysunek 4.1.



Rysunek 4.1. Tweet na temat wykonywania pracy w prawdziwym świecie

Wskazówka

Dobra praktyka: nie zakładaj, że zwiększenie liczby wątków poprawi wydajność aplikacji. Sprawdź wydajność bazowej implementacji z jednym wątkiem, a następnie porównaj ją z implementacją wielowątkową. Oprócz tego mierz wydajność w środowisku testowym możliwie zbliżonym do produkcyjnego.

Monitorowanie wydajności aplikacji i wykorzystania zasobów

Abyś mógł zwiększać wydajność kodu, musisz umieć go monitorować i tworzyć linię odniesienia, która pozwoli Ci mierzyć postępy.

Ocena efektywności typów danych

Jaki typ danych jest najlepszy w konkretnym przypadku? Aby odpowiedzieć na to pytanie, trzeba dokładnie rozważyć, co oznacza pojęcie *najlepszy*. W związku z tym należy wziąć pod uwagę następujące czynniki:

- **Funkcjonalność**: czy dany typ zapewnia potrzebne funkcje?
- **Wielkość**: ile bajtów pamięci zajmuje dany typ?
- **Wydajność**: jak szybki jest dany typ?
- **Przyszłe potrzeby**: czynnik zależny od zmian w wymaganiach i od możliwości utrzymania.

W niektórych przypadkach, na przykład przetwarzania liczb, te same funkcjonalności są dostępne w kilku typach. Wówczas podczas dokonywania wyboru trzeba wziąć pod uwagę ich wielkość i wydajność.

Jeśli liczb jest kilka milionów i trzeba je zapisać, najlepszym typem będzie ten, który zajmuje najmniej pamięci. Jeżeli natomiast liczb jest zaledwie kilka, ale trzeba na nich wykonywać wiele obliczeń, najlepszy będzie taki typ, który jest najszybszy dla danego procesora.

Funkcja `sizeof` zwraca liczbę bajtów pamięci zajmowanych przez pojedynczą instancję danego typu. Jeżeli wiele wartości ma być przechowywanych w bardziej złożonych strukturach, na przykład tablicach lub listach, potrzebny jest lepszy sposób określania zajętości pamięci.

W książkach i w internecie można znaleźć wiele porad na ten temat, jednak najlepszym rozwiązaniem jest samodzielne porównanie typów i wybór najlepszego dla danego kodu.

W następnym punkcie dowiesz się, jak pisać kod, aby monitorować rzeczywistą zajętość pamięci i wydajność różnych typów.

Na przykład dzisiaj najlepszy może się wydawać typ `short`. Jednak typ `int`, mimo że zajmuje dwa razy więcej miejsca w pamięci, będzie jeszcze lepszym wyborem, ponieważ w przyszłości może być konieczne przechowywanie szerszego zakresu wartości.

Jak wspomniałem, jest jeszcze jeden czynnik, o którym programiści często zapominają: możliwości utrzymania kodu. Jest to miara wysiłku, jaki musi włożyć inny programista, aby zrozumieć i zmodyfikować Twój kod. Jeśli dokonasz nieoczywistego wyboru i nie umieścisz komentarza z uzasadnieniem, programista, który będzie musiał później na przykład poprawić błąd lub dodać nową funkcje, będzie zdezorientowany.

Monitorowanie wydajności kodu i zajętości pamięci za pomocą przestrzeni Diagnostics

Przestrzeń nazw `System.Diagnostics` oferuje wiele przydatnych typów do monitorowania kodu. Pierwszym, któremu się przyjrzymy, jest `Stopwatch`.

1. W wybranym edytorze kodu utwórz projekt biblioteki klas z następującymi ustawieniami:
   - szablon projektu: *Biblioteka klas* (*Class Library*)/`classlib`,
   - plik/katalog rozwiązania/ obszaru roboczego: *Chapter04*,
   - plik/katalog projektu: *MonitoringLib*.
2. Utwórz projekt aplikacji konsolowej z następującymi ustawieniami:
   - szablon projektu: *Aplikacja konsoli* (*Console App*)/`console`,
   - plik/katalog rozwiązania/ obszaru roboczego: *Chapter04*,
   - plik/katalog projektu: *MonitoringApp*.
3. Wskaż aktywny projekt:
   - W środowisku Visual Studio 2022 oznacz bieżący projekt jako startowy.
   - W środowisku Visual Studio Code ustaw *MonitoringApp* jako aktywny projekt OmniSharp.
4. W projekcie *MonitoringLib* zmień nazwę pliku *Class1.cs* na *Recorder.cs*.
5. W tym samym projekcie zaimportuj statycznie i globalnie przestrzeń nazw `System.Console`.
6. W projekcie *MonitoringApp* zaimportuj statycznie i globalnie przestrzeń nazw `System.Console` i dodaj odwołanie do biblioteki klas *MonitoringLib*, jak niżej:

```
<ItemGroup>
  <Using Include="System.Console" Static="true" />
</ItemGroup>

<ItemGroup>
  <ProjectReference
    Include="..\MonitoringLib\MonitoringLib.csproj" />
</ItemGroup>
```

7. Skompiluj projekt *MonitoringApp*.

Przydatne elementy typów Stopwatch i Process

Typ `Stopwatch` ma kilka przydatnych elementów opisanych w poniższej tabeli.

Element	Opis
Metoda `Restart`	Metoda zerująca pomiar czasu i uruchamiająca stoper
Metoda `Stop`	Metoda zatrzymująca stoper
Właściwość `Elapsed`	Czas, który upłynął, zapisany w formacie `TimeSpan` (np. godziny:minuty:sekundy)
Właściwość `ElapsedMilliseconds`	Czas w milisekundach, który upłynął, zapisany jako wartość `Int64`

Typ `Process` ma kilka przydatnych elementów opisanych w poniższej tabeli.

Element	Opis
Właściwość `VirtualMemorySize64`	Liczba bajtów wirtualnej pamięci, która została przydzielona procesowi
Właściwość `WorkingSet64`	Liczba bajtów fizycznej pamięci, która została przydzielona procesowi

Implementacja typu Recorder

Utwórz typ `Recorder`, który ułatwi Ci monitorowanie czasu działania kodu i zajętości pamięci. Wykorzystaj w tym celu typy `Stopwatch` i `Process`.

1. W pliku *Recorder.cs* wpisz poniższy kod, który za pomocą instancji typu `Stopwatch` i `Process` mierzy, odpowiednio, czas działania kodu i zajętość pamięci:

```
using System.Diagnostics; // Stopwatch

using static System.Diagnostics.Process; // GetCurrentProcess

namespace Packt.Shared;

public static class Recorder
{
  private static Stopwatch timer = new();

  private static long bytesPhysicalBefore = 0;
  private static long bytesVirtualBefore = 0;

  public static void Start()
  {
    // Jawne uporządkowanie pamięci w celu usunięcia obiektów, które nie są już używane.
```

```
        GC.Collect();
        GC.WaitForPendingFinalizers();
        GC.Collect();
        GC.WaitForPendingFinalizers();
        GC.Collect();

        // Zapisanie bieżącego wykorzystania pamięci wirtualnej i fizycznej.
        bytesPhysicalBefore = GetCurrentProcess().WorkingSet64;
        bytesVirtualBefore = GetCurrentProcess().VirtualMemorySize64;

        timer.Restart();
      }

      public static void Stop()
      {
        timer.Stop();

        long bytesPhysicalAfter =
          GetCurrentProcess().WorkingSet64;

        long bytesVirtualAfter =
          GetCurrentProcess().VirtualMemorySize64;

        WriteLine("Liczba zajętych bajtów pamięci fizycznej: {0:N0}.",
          bytesPhysicalAfter - bytesPhysicalBefore);

        WriteLine("Liczba zajętych bajtów pamięci wirtualnej: {0:N0}.",
          bytesVirtualAfter - bytesVirtualBefore);

        WriteLine("Czas działania: {0}.", timer.Elapsed);

        WriteLine("Czas działania w milisekundach: {0:N0}.",
          timer.ElapsedMilliseconds);
      }
    }
```

Uwaga

Metoda `Start` klasy `Recorder`, zanim odczyta ilość zajętej pamięci, usuwa z niej wszystkie przydzielone, ale nie wykorzystane obszary z użyciem klasy `GC` (ang. *garbage collector*, kolektor śmieci). Jest to zaawansowana operacja, której nie należy implementować w kodzie, ponieważ mechanizm GC więcej „wie" o wykorzystaniu pamięci niż programista i jemu należy powierzyć podejmowanie decyzji o porządkowaniu niewykorzystywanych obszarów. Konieczność przejęcia kontroli przez kod jest w tym przykładzie wyjątkiem.

2. Usuń z pliku *Program.cs* istniejące instrukcje, a następnie wpisz poniższe, które uruchamiają i zatrzymują rejestrator czasu oraz tworzą tablicę 10 000 liczb całkowitych:

```
using Packt.Shared; // Recorder

WriteLine("Przetwarzanie danych. Czekaj...");
Recorder.Start();

// Symulacja procesu zajmującego pewną ilość pamięci...
int[] largeArrayOfInts = Enumerable.Range(
  start: 1, count: 10_000).ToArray();

// ...i trwającego jakiś czas.
Thread.Sleep(new Random().Next(5, 10) * 1000);
Recorder.Stop();
```

3. Uruchom kod. Uzyskasz wynik podobny do poniższego:

```
Przetwarzanie danych. Czekaj...
Liczba zajętych bajtów pamięci fizycznej: 889 824.
Liczba zajętych bajtów pamięci wirtualnej: 131 072.
Czas działania: 00:00:06.0074221.
Czas działania w milisekundach: 6 007.
```

Uwaga

Zwróć uwagę, że w programie jest wprowadzana zwłoka o losowej długości od 5 do 10 sekund. Ponadto wielokrotnie uruchamiając ten kod na tym samym komputerze, za każdym razem uzyskasz różne wyniki. Na przykład na moim laptopie Mac Mini M1 program zajął mniej pamięci fizycznej, ale więcej wirtualnej:

```
Przetwarzanie danych. Czekaj...
Liczba zajętych bajtów pamięci fizycznej: 294 912.
Liczba zajętych bajtów pamięci wirtualnej: 10 485 760.
Czas działania: 00:00:06.0016951.
Czas działania w milisekundach: 6 001.
```

Monitorowanie wydajności przetwarzania ciągów znaków

Teraz, gdy wiesz już, jak używać typów `Stoper` i `Process` do monitorowania wydajności kodu, wykorzystaj je do wybrania najlepszego sposobu przetwarzania ciągów znaków.

1. W projekcie *MonitoringApp* utwórz plik klasy *Program.Helpers.cs*.
2. W nowym pliku zdefiniuj częściową klasę `Program` z metodą wyświetlającą tytuł sekcji w ciemnożółtym kolorze, jak niżej:

```
partial class Program
{
  static void SectionTitle(string title)
  {
    ConsoleColor previousColor = ForegroundColor;
    ForegroundColor = ConsoleColor.DarkYellow;
    WriteLine("*");
    WriteLine($"* {title}");
    WriteLine("*");
    ForegroundColor = previousColor;
  }
}
```

3. W pliku *Program.cs* oznacz jako komentarz wpisane wcześniej instrukcje przez umieszczenie ich wewnątrz znaków /* */.

4. Wpisz poniższe instrukcje, które tworzą tablicę 50 000 liczb całkowitych, a następnie rozdzielają je przecinkami i dołączają do ciągu znaków:

```
int[] numbers = Enumerable.Range(
  start: 1, count: 50_000).ToArray();

SectionTitle("Klasa StringBuilder");
Recorder.Start();

System.Text.StringBuilder builder = new();
for (int i = 0; i < numbers.Length; i++)
{
  builder.Append(numbers[i]);
  builder.Append(", ");
}

Recorder.Stop();
WriteLine();

SectionTitle("Typ string i operator +");
Recorder.Start();

string s = string.Empty; // Ciąg "".

for (int i = 0; i < numbers.Length; i++)
{
  s += numbers[i] + ", ";
}

Recorder.Stop();
```

5. Uruchom kod. Uzyskasz wynik podobny do poniższego:

```
*
* Klasa StringBuilder
*
```

```
Liczba zajętych bajtów pamięci fizycznej: 1 150 976.
Liczba zajętych bajtów pamięci wirtualnej: 0.
Czas działania: 00:00:00.0010796.
Czas działania w milisekundach: 1.

*
* Typ string i operator +
*
Liczba zajętych bajtów pamięci fizycznej: 11 849 728
Liczba zajętych bajtów pamięci wirtualnej: 1 638 400.
Czas działania: 00:00:01.7754252.
Czas działania w milisekundach: 1 775.
```

Powyższe wyniki można podsumować następująco:

- Kod wykorzystujący klasę `StringBuilder` zajął ok. 1 MB pamięci fizycznej, pamięci wirtualnej nie zajął w ogóle, a opisaną operację wykonywał przez mniej więcej 1 milisekundę.
- Kod wykorzystujący typ `strings` i operator + zajął ok. 11 MB pamięci fizycznej, 1,5 MB pamięci wirtualnej, a opisaną operację wykonywał przez mniej więcej 1,7 milisekundy.

W tym przykładzie klasa `StringBuilder` jest ponad 1000 razy szybsza i zajmuje ok. 10 razy mniej pamięci niż typ `string`. Wynika to stąd, że typ ten jest niemutowalny i podczas każdorazowego dołączania ciągu znaków jest tworzony nowy ciąg. Natomiast klasa `StringBuilder` tworzy w pamięci jeden bufor, w którym umieszcza dołączane znaki.

Wskazówka

Dobra praktyka: nie stosuj w pętlach metody `String.Concat` ani operatora +. Zamiast nich używaj klasy `StringBuilder`.

Teraz, gdy wiesz już, jak mierzyć wydajność kodu i efektywność wykorzystania zasobów z użyciem wbudowanych typów platformy .NET, poznaj pakiet NuGet, który oferuje bardziej zaawansowane techniki pomiarów.

Monitorowanie wydajności kodu i zajętości pamięci za pomocą pakietu Benchmark.NET

Benchmark.NET to popularny pakiet NuGet do wykonywania testów porównawczych aplikacji opartych na platformie .NET. Microsoft propaguje go na swoim blogu we wpisach poświęconych poprawianiu wydajności kodu. Dlatego warto, abyś wiedział,

jak działa ten pakiet, i mógł go używać we własnych testach wydajności. Zobacz teraz, jak możesz go użyć do porównania wydajności operacji łączenia ciągów znaków za pomocą typu `string` i klasy `StringBuilder`.

1. W wybranym edytorze kodu otwórz rozwiązanie/ obszar roboczy o nazwie *Chapter04* i dodaj do niego aplikację konsolową o nazwie *Benchmarking*.
   - W środowisku Visual Studio 2022 oznacz bieżący projekt jako startowy.
   - W środowisku Visual Studio Code ustaw *Benchmarking* jako aktywny projekt OmniSharp.
2. W pliku konfiguracyjnym projektu dodaj odwołanie do pakietu `Benchmark.NET`, jak niżej. Pamiętaj, aby użyć jego najnowszej wersji:

```
<ItemGroup>
  <PackageReference Include="BenchmarkDotNet" Version="0.13.1" />
</ItemGroup>
```

3. Skompiluj projekt, aby został pobrany wymagany pakiet.
4. Utwórz plik klasy *StringBenchmarks.cs*.
5. Wpisz poniższy kod definiujący klasę z metodami, których użyjesz w pomiarach porównawczych. Metody te wykorzystują, odpowiednio, typ `string` i klasę `StringBuilder` do utworzenia ciągu 20 liczb oddzielonych przecinkami.

```
using BenchmarkDotNet.Attributes; // [Benchmark]

public class StringBenchmarks
{
  int[] numbers;

  public StringBenchmarks()
  {
    numbers = Enumerable.Range(
      start: 1, count: 20).ToArray();
  }

  [Benchmark(Baseline = true)]
  public string StringConcatenationTest()
  {
    string s = string.Empty; // Ciąg "".

    for (int i = 0; i < numbers.Length; i++)
    {
      s += numbers[i] + ", ";
    }

    return s;
  }
```

```
  [Benchmark]
  public string StringBuilderTest()
  {
    System.Text.StringBuilder builder = new();

    for (int i = 0; i < numbers.Length; i++)
    {
      builder.Append(numbers[i]);
      builder.Append(", ");
    }

    return builder.ToString();
  }
}
```

6. Usuń z pliku *Program.cs* istniejące instrukcje, a następnie wpisz poniższe, które importują przestrzeń nazw `BenchmarkDotNet.Running` i wywołują metodę klasy pomiarowej:

```
using BenchmarkDotNet.Running;
BenchmarkRunner.Run<StringBenchmarks>();
```

7. Uruchom aplikację w trybie *Release*:
   - W środowisku Visual Studio 2022 wybierz z rozwijanej listy *Konfiguracje rozwiązania* (*Solution Configurations*) pozycję *Release*, a następnie kliknij polecenie menu *Debug/Start Without Debugging* (*Debuguj/Uruchom bez debugowania*).
   - W środowisku Visual Studio Code otwórz terminal i wpisz w nim polecenie **`run --configuration Release`**.

8. Uzyskasz wyniki zawierające kilka nieistotnych informacji, m.in. nazwy plików z raportami. Ważna jest natomiast tabela w sekcji *Summary* (podsumowanie), według której łączenie ciągów z użyciem typu `string` i klasy `StringBuilder` średnio zajęło, odpowiednio, 413 ns i 275,1 ns. Ilustruje to poniższy częściowy wynik.

```
// ***** BenchmarkRunner: Finish  *****

// * Export *
  BenchmarkDotNet.Artifacts\results\StringBenchmarks-report.csv
  BenchmarkDotNet.Artifacts\results\StringBenchmarks-report-github.md
  BenchmarkDotNet.Artifacts\results\StringBenchmarks-report.html

// * Detailed results *
StringBenchmarks.StringConcatenationTest: DefaultJob
Runtime = .NET 7.0.0 (7.0.22.22904), X64 RyuJIT; GC = Concurrent
Workstation
Mean = 412.990 ns, StdErr = 2.353 ns (0.57%), N = 46, StdDev = 15.957 ns
Min = 373.636 ns, Q1 = 413.341 ns, Median = 417.665 ns, Q3 = 420.775 ns,
```

```
Max = 434.504 ns
IQR = 7.433 ns, LowerFence = 402.191 ns, UpperFence = 431.925 ns
ConfidenceInterval = [404.708 ns; 421.273 ns] (CI 99.9%), Margin = 8.282 ns
(2.01% of Mean)
Skewness = -1.51, Kurtosis = 4.09, MValue = 2
-------------------- Histogram -------------------
[370.520 ns ; 382.211 ns) | @@@@@@
[382.211 ns ; 394.583 ns) | @
[394.583 ns ; 411.300 ns) | @@
[411.300 ns ; 422.990 ns) | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
[422.990 ns ; 436.095 ns) | @@@@@
---------------------------------------------------

StringBenchmarks.StringBuilderTest: DefaultJob
Runtime = .NET 7.0.0 (7.0.22.22904), X64 RyuJIT; GC = Concurrent
Workstation
Mean = 275.082 ns, StdErr = 0.558 ns (0.20%), N = 15, StdDev = 2.163 ns
Min = 271.059 ns, Q1 = 274.495 ns, Median = 275.403 ns, Q3 = 276.553 ns,
Max = 278.030 ns
IQR = 2.058 ns, LowerFence = 271.409 ns, UpperFence = 279.639 ns
ConfidenceInterval = [272.770 ns; 277.394 ns] (CI 99.9%), Margin = 2.312
ns (0.84% of Mean)
Skewness = -0.69, Kurtosis = 2.2, MValue = 2
-------------------- Histogram -------------------
[269.908 ns ; 278.682 ns) | @@@@@@@@@@@@@@@
---------------------------------------------------

// * Summary *

BenchmarkDotNet=v0.13.1, OS=Windows 10.0.22000
11th Gen Intel Core i7-1165G7 2.80GHz, 1 CPU, 8 logical and 4 physical cores
.NET SDK=7.0.100
  [Host]     : .NET 7.0.0 (7.0.22.22904), X64 RyuJIT
  DefaultJob : .NET 7.0.0 (7.0.22.22904), X64 RyuJIT

|                  Method |     Mean |   Error |   StdDev | Ratio |
RatioSD |
|------------------------ |---------:|--------:|---------:|------:|--------:|
| StringConcatenationTest | 413.0 ns | 8.28 ns | 15.96 ns |  1.00 |    0.00 |
|       StringBuilderTest | 275.1 ns | 2.31 ns |  2.16 ns |  0.69 |    0.04 |

// * Hints *
Outliers
  StringBenchmarks.StringConcatenationTest: Default -> 7 outliers
  ↪were removed, 14 outliers were detected (376.78 ns..391.88 ns, 440.79
  ↪ns..506.41 ns)
  StringBenchmarks.StringBuilderTest: Default       -> 2 outliers
  ↪were detected (274.68 ns, 274.69 ns)
```

```
// * Legends *
  Mean    : Arithmetic mean of all measurements
  Error   : Half of 99.9% confidence interval
  StdDev  : Standard deviation of all measurements
  Ratio   : Mean of the ratio distribution ([Current]/[Baseline])
  RatioSD : Standard deviation of the ratio distribution
([Current]/[Baseline])
  1 ns    : 1 Nanosecond (0.000000001 sec)

// ***** BenchmarkRunner: End *****
// ** Remained 0 benchmark(s) to run **

Run time: 00:01:13 (73.35 sec), executed benchmarks: 2
Global total time: 00:01:29 (89.71 sec), executed benchmarks: 2
// * Artifacts cleanup *
```

Szczególnie ciekawa jest sekcja *Outliers* (wartości odstające), która pokazuje, że typ `string` jest nie tylko wolniejszy od klasy `StringBuilder`, ale również uzyskiwanie przy jego użyciu wyniki są mniej spójne. Twoje wyniki mogą być oczywiście inne. Wiedz również, że gdyby wyniki nie zawierały wartości odstających, nie byłoby sekcji *Hints* (wskazówki) i *Outliers*,

Poznałeś dwa sposoby mierzenia wydajności kodu. Teraz dowiedz się, jak możesz ją poprawić poprzez asynchroniczne wykonywanie zadań.

Asynchroniczne wykonywanie zadań

Aby zrozumieć, w jaki sposób kilka zadań może być wykonywanych *w tym samym czasie*, napiszesz kod, który wywoła trzy metody. Pierwsza będzie wykonywana przez 3 sekundy, druga przez 2 sekundy, a trzecia przez 1 sekundę. Użyjesz w tym celu klasy `Thread`, umożliwiającej m.in. wprowadzenie bieżącego wątku w stan uśpienia na zadaną liczbę milisekund.

Synchroniczne wykonywanie operacji

Zanim sprawisz, że zadania zostaną wykonane jednocześnie, uruchom je synchronicznie, czyli jedno po drugim.

1. W wybranym edytorze kodu otwórz rozwiązanie/ obszar roboczy o nazwie *Chapter04* i dodaj do niego aplikację konsolową o nazwie *WorkingWithTasks*.
   - W środowisku Visual Studio 2022 oznacz bieżący projekt jako startowy.
   - W środowisku Visual Studio Code ustaw *WorkingWithTasks* jako aktywny projekt OmniSharp.

2. W tym samym projekcie zaimportuj statycznie i globalnie przestrzeń nazw `System.Console`.
3. Utwórz plik klasy *Program.Helpers.cs*.
4. W nowym pliku zdefiniuj częściową klasę `Program` z metodami wyświetlającymi, dla poprawy czytelności w różnych kolorach, odpowiednio tytuł sekcji, nazwę zadania i informacje o bieżącym wątku, jak niżej:

```
partial class Program
{
  static void SectionTitle(string title)
  {
    ConsoleColor previousColor = ForegroundColor;
    ForegroundColor = ConsoleColor.DarkYellow;
    WriteLine("*");
    WriteLine($"* {title}");
    WriteLine("*");
    ForegroundColor = previousColor;
  }

  static void TaskTitle(string title)
  {
    ConsoleColor previousColor = ForegroundColor;
    ForegroundColor = ConsoleColor.Green;
    WriteLine($"{title}");
    ForegroundColor = previousColor;
  }

  static void OutputThreadInfo()
  {
    Thread t = Thread.CurrentThread;
    ConsoleColor previousColor = ForegroundColor;
    ForegroundColor = ConsoleColor.DarkCyan;

    WriteLine(
      "ID wątku: {0}, priorytet: {1}, w tle: {2}, nazwa: {3}",
      t.ManagedThreadId, t.Priority, t.IsBackground, t.Name ?? "null");

    ForegroundColor = previousColor;
  }
}
```

5. Utwórz plik klasy *Program.Methods.cs*.
6. W nowym pliku zdefiniuj trzy metody symulujące pracę, jak niżej:

```
partial class Program
{
  static void MethodA()
  {
    TaskTitle("Rozpoczęcie metody A...");
```

```
    OutputThreadInfo();
    Thread.Sleep(3000); // Symulacja 3 sekund pracy.
    TaskTitle("Zakończenie metody A.");
  }

  static void MethodB()
  {
    TaskTitle("Rozpoczęcie metody B...");
    OutputThreadInfo();
    Thread.Sleep(2000); // Symulacja 2 sekund pracy.
    TaskTitle("Zakończenie metody B.");
  }

  static void MethodC()
  {
    TaskTitle("Rozpoczęcie metody C...");
    OutputThreadInfo();
    Thread.Sleep(1000); // Symulacja 1 sekundy pracy.
    TaskTitle("Zakończenie metody C.");
  }
}
```

7. Usuń z pliku *Program.cs* istniejące instrukcje, a następnie wpisz poniższe, które wywołują metodę pomocniczą (wyświetlającą informacje o wątku), definiują i uruchamiają czasomierz oraz wywołują trzy metody symulujące pracę i metodę wyświetlającą czas działania kodu:

```
using System.Diagnostics; // Stopwatch

OutputThreadInfo();
Stopwatch timer = Stopwatch.StartNew();

SectionTitle("Synchroniczne wywołanie metod w jednym wątku.");
MethodA();
MethodB();
MethodC();

WriteLine($"Czas działania: {timer.ElapsedMilliseconds:# ##0} ms.");
```

8. Uruchom kod i zwróć uwagę, że na pierwszym planie zostanie uruchomiony tylko jeden wątek bez nazwy, który będzie działał przez 6 sekund:

```
ID wątku: 1, priorytet: Normal, w tle: False, nazwa: null
*
* Synchroniczne wywołanie metod w jednym wątku.
*
Rozpoczęcie metody A...
ID wątku: 1, priorytet: Normal, w tle: False, nazwa: null
Zakończenie metody A.
Rozpoczęcie metody B...
ID wątku: 1, priorytet: Normal, w tle: False, nazwa: null
Zakończenie metody B.
Rozpoczęcie metody C...
```

```
ID wątku: 1, priorytet: Normal, w tle: False, nazwa: null
Zakończenie metody C.
Czas działania: 6 012 ms.
```

Asynchroniczne wykonywanie operacji za pomocą zadań

Klasa `Thread` (wątek) pojawiła się w pierwszej wersji platformy .NET w 2002 r. Służy ona do tworzenia wątków i zarządzania nimi, ale bezpośrednie korzystanie z niej jest dość trudne. W 2010 r. w wersji platformy .NET Framework 4.0 wprowadzono klasę `Task` (zadania), która reprezentuje operację asynchroniczną. **Zadanie** to wyższego rzędu abstrakcja wątku, który wykonuje operację. Tworzenie zadań i zarządzanie nimi jest łatwiejsze niż w przypadku wątków. Dzięki wątkom opakowanym w zadania kod może wykonywać kilka operacji jednocześnie, czyli asynchronicznie.

Klasa `Task` ma właściwości `Status` i `CreationOptions` (opcje utworzenia), jak również metodę `ContinueWith`, którą można dostosowywać przez użycie wartości wyliczeniowej `TaskContinuationOptions` i którą można zarządzać za pomocą klasy `TaskFactory`.

Uruchomienie zadania

Poznaj trzy sposoby wywoływania metody za pomocą instancji klasy `Task`. W repozytorium GitHub przypisanym tej książce znajdziesz odnośniki do artykułów opisujących zalety i wady tych sposobów.

Każdy z tych sposobów wymaga zdefiniowania i uruchomienia zadania z użyciem innej składni.

1. W pliku *Program.cs* dopisz poniższe instrukcje, które definiują i wywołują trzy metody wykonujące poszczególne zadania:

```
SectionTitle("Asynchroniczne wywołanie metod w kilku wątkach.");
timer.Restart();

Task taskA = new(MethodA);
taskA.Start();
Task taskB = Task.Factory.StartNew(MethodB);
Task taskC = Task.Run(MethodC);

WriteLine($"Czas działania: {timer.ElapsedMilliseconds:# ##0} ms.");
```

2. Uruchom kod i zwróć uwagę, że zakończy działanie niemal natychmiast, ponieważ każda metoda zostanie wywoływana w tle w osobnym wątku przydzielonym z puli.

```
ID wątku: 1, priorytet: Normal, w tle: False, nazwa: null
*
```

```
* Asynchroniczne wywołanie metod w kilku wątkach.
*
Rozpoczęcie metody A...
ID wątku: 6, priorytet: Normal, w tle: True, nazwa: .NET ThreadPool
Worker
Rozpoczęcie metody C...
ID wątku: 8, priorytet: Normal, w tle: True, nazwa: .NET ThreadPool
Worker
Rozpoczęcie metody B...
ID wątku: 9, priorytet: Normal, w tle: True, nazwa: .NET ThreadPool
Worker
Czas działania:  6 ms.
```

Uwaga

Może się zdarzyć, że aplikacja zakończy działanie, zanim jedno, czy nawet wszystkie zadania zostaną wykonane i coś wyświetlą w terminalu!

Oczekiwanie na wykonanie zadań

Czasami trzeba wstrzymać działanie kodu do czasu wykonania zadania. W tym celu można użyć metody `Wait` instancji klasy `Task` lub statycznej metody `WaitAll` albo `WaitAny` z tablicą zadań w argumencie, jak opisano w poniższej tabeli:

Metoda	Opis
`t.Wait()`	Oczekiwanie na zakończenie zadania reprezentowanego przez instancję `t` klasy `Task`
`Task.WaitAny(Task[])`	Oczekiwanie na zakończenie dowolnego zadania umieszczonego w tablicy
`Task.WaitAll(Task[])`	Oczekiwanie na zakończenie wszystkich zadań umieszczonych w tablicy

Korzystanie z metod oczekujących na wykonanie zadań

Zobacz teraz, jak możesz użyć powyższych metod do rozwiązania problemu z aplikacją.

1. W pliku *Program.cs*, poniżej instrukcji uruchamiającej trzecie zadanie, ale przed instrukcją wyświetlającą czas działania, wpisz poniższy kod, który tworzy tablicę zadań i umieszcza ją w argumencie metody `WaitAll`:

```
Task[] tasks = { taskA, taskB, taskC };
Task.WaitAll(tasks);
```

2. Uruchom kod i zwróć uwagę, że po wywołaniu metody `WaitAll` wątek zostanie wstrzymany do czasu wykonania wszystkich zadań. Dopiero potem pojawi się informacja o czasie działania kodu, nieco przekraczającym 3 sekundy:

```
Rozpoczęcie metody A...
Rozpoczęcie metody B...
ID wątku: 8, priorytet: Normal, w tle: True, nazwa: .NET ThreadPool
Worker
ID wątku: 6, priorytet: Normal, w tle: True, nazwa: .NET ThreadPool
Worker
Rozpoczęcie metody C...
ID wątku: 4, priorytet: Normal, w tle: True, nazwa: .NET ThreadPool
Worker
Zakończenie metody C.
Zakończenie metody B.
Zakończenie metody A.
Czas działania: 3 005 ms.
```

Trzy nowe wątki uruchamiają się niemal jednocześnie, potencjalnie w dowolnej kolejności. Jako pierwsza kończy działanie metoda C, ponieważ trwa tylko 1 sekundę. Następna jest metoda B, zajmująca 2 sekundy, a na końcu metoda A, działająca przez 3 sekundy.

Duży wpływ na uzyskiwane wyniki ma procesor, który przydziela interwały czasu każdemu wątkowi. Programista nie ma możliwości kontrolowania chwili wywołania poszczególnych metod.

Sekwencyjne wykonanie innego zadania

Po jednoczesnym uruchomieniu kilku zadań wystarczy poczekać, aż wszystkie zostaną wykonane. Jednak często kolejne zadanie jest uzależnione od wyników poprzedniego. W takich sytuacjach definiuje się **zadanie kontynuacyjne**.

W kolejnym przykładzie utworzysz dwie metody symulujące usługę internetową. Pierwsza będzie zwracać pewną kwotę, a druga liczbę produktów, które kosztują więcej, niż wynosi ta kwota. Wynik zwrócony przez pierwszą metodę musi być przekazany drugiej metodzie.

Tym razem żadna z metod nie będzie wstrzymywała działania wątków na stały przedział czasu. Użyjesz klasy `Random`, dzięki której każda z nich będzie symulowała pracę przez losowy okres od 2 do 4 sekund.

1. W pliku *Program.Methods.cs* wpisz poniższy kod definiujący dwie metody. Pierwsza symuluje wywołanie usługi internetowej, a druga wywołanie procedury składowanej w bazie danych.

```
static decimal CallWebService()
{
  TaskTitle("Wywołanie usługi internetowej");
  OutputThreadInfo();
  Thread.Sleep((new Random()).Next(2000, 4000));
  TaskTitle("Koniec wywołania usługi internetowej.");
  return 89.99M;
}

static string CallStoredProcedure(decimal amount)
{
  TaskTitle("Wywołanie procedury składowanej...");
  OutputThreadInfo();
  Thread.Sleep((new Random()).Next(2000, 4000));
  TaskTitle("Koniec wywołania procedury składowanej.");
  return $"12 produktów kosztuje więcej niż {amount:C}.";
}
```

2. W pliku *Program.cs* wpisz poniższy kod, który uruchamia zadanie odwołujące się do usługi internetowej, a zwrócony wynik przekazuje zadaniu wywołującemu procedurę składowaną:

```
SectionTitle("Przekazanie wyniku jednego zadania na wejście innego.");
timer.Restart();

Task<string> taskServiceThenSProc = Task.Factory
  .StartNew(CallWebService) // Zwraca wynik typu Task<decimal>.
  .ContinueWith(previousTask => // Zwraca wynik typu Task<string>.
    CallStoredProcedure(previousTask.Result));

WriteLine($"Wynik: {taskServiceThenSProc.Result}");

WriteLine($"Czas działania: {timer.ElapsedMilliseconds:# ##0} ms.");
```

3. Uruchom kod. Uzyskasz wynik podobny do poniższego:

```
Wywołanie usługi internetowej...
ID wątku: 4, priorytet: Normal, w tle: True, nazwa: .NET ThreadPool Worker
Koniec wywołania usługi internetowej.
Wywołanie procedury składowanej...
ID wątku: 6, priorytet: Normal, w tle: True, nazwa: .NET ThreadPool Worker
Koniec wywołania procedury składowanej.
Wynik: 12 produktów kosztuje więcej niż 89,99 zł.
Czas działania: 5 005 ms.
```

Usługa internetowa i procedura składowana mogą zostać wywołane w dwóch różnych wątkach, w powyższym przykładzie oznaczonych numerami 4 i 6, albo w jednym, jeżeli będzie wolny w chwili rozpoczęcia drugiego zadania.

Zadania zagnieżdżone i podrzędne

Oprócz zależności między zadaniami można definiować **zadania zagnieżdżone** i **podrzędne**. Zadanie zagnieżdżone jest wykonywane wewnątrz innego zadania. Zadanie podrzędne jest podobne, ale musi zostać wykonane przed zakończeniem zadania nadrzędnego.

Przyjrzyj się obu rodzajom zadań.

1. W pliku *Program.Methods.cs* wpisz poniższy kod definiujący dwie metody. Pierwsza uruchamia zadanie, które inicjuje inne zadanie.

```
static void OuterMethod()
{
  TaskTitle("Wywołanie metody zewnętrznej...");
  Task innerTask = Task.Factory.StartNew(InnerMethod);
  TaskTitle("Koniec metody zewnętrznej.");
}

static void InnerMethod()
{
  TaskTitle("Wywołanie metody wewnętrznej...");
  Thread.Sleep(2000);
  TaskTitle("Koniec metody wewnętrznej.");
}
```

2. W pliku *Program.cs* wpisz poniższy kod, który wywołuje metodę zewnętrzną, czeka na jej wykonanie, po czym kończy działanie:

```
SectionTitle("Zadania zagnieżdżone i podrzędne");

Task outerTask = Task.Factory.StartNew(OuterMethod);
outerTask.Wait();
WriteLine("Zakończenie działania aplikacji.");
```

3. Uruchom kod. Uzyskasz wynik podobny do poniższego:

```
Wywołanie metody zewnętrznej...
Wywołanie metody wewnętrznej...
Koniec metody zewnętrznej.
Zakończenie działania aplikacji.
```

Uwaga

Chociaż kod czeka na zakończenie zewnętrznego zadania, jego wewnętrzne zadanie nie musi być wykonane do końca. Co więcej, zewnętrzne zadane może zostać wykonane, a aplikacja może zakończyć działanie, jeszcze zanim wewnętrzne zadanie zostanie rozpoczęte!

4. Aby połączyć zadania relacją nadrzędne-podrzędne, trzeba użyć specjalnej opcji. Dodaj wyróżniony w poniższym kodzie argument `TaskCreationOptions.AttachedToParent`:

```
Task innerTask = Task.Factory.StartNew(InnerMethod,
  TaskCreationOptions.AttachedToParent);
```

5. Ponownie uruchom kod. Zwróć uwagę, że tym razem zewnętrzna metoda zaczeka, aż zostanie wykonana metoda wewnętrzna:

```
Wywołanie metody zewnętrznej...
Wywołanie metody wewnętrznej...
Koniec metody zewnętrznej.
Koniec metody wewnętrznej.
Zakończenie działania aplikacji.
```

Uwaga

Metoda zewnętrzna może zakończyć działanie przed metodą wewnętrzną, jak pokazuje powyższy wynik, ale jej zadanie musi czekać na wykonanie zadania wewnętrznego.

Opakowanie zadania w obiekcie

Czasami potrzebna jest metoda asynchroniczna, której wynik nie jest zadaniem. W takim przypadku wynik można umieścić w pomyślnie wykonanym zadaniu, a następnie zwrócić wyjątek lub wskazać z użyciem jednej ze statycznych metod wymienionych w poniższej tabeli, że zadanie zostało anulowane.

Metoda	Opis
`FromResult<TResult>(TResult)`	Utworzenie obiektu `Task<TResult>`, którego właściwość `Result` zawiera wynik niebędący zadaniem, a właściwość `Status` zawiera wartość `RanToCompletion`
`FromException<TResult>(Exception)`	Utworzenie obiektu `Task<TResult>` reprezentującego zadanie, po którego wykonaniu jest zgłaszany zadany wyjątek
`FromCanceled<TResult>(CancellationToken)`	Utworzenie obiektu `Task<TResult>` reprezentującego zadanie, którego wykonanie można przerwać za pomocą zadanego tokenu

Powyższe metody mają następujące zastosowania:

- Synchroniczna implementacja interfejsu zawierającego metody asynchroniczne. Jest to typowe podejście stosowane w witrynach internetowych i usługach.
- Symulowanie asynchronicznych implementacji w testach jednostkowych.

Załóżmy, że trzeba utworzyć metodę sprawdzającą poprawność danych wejściowych XML. Metoda musi być zgodna z interfejsem i zwracać wynik typu `Task<T>`, jak niżej:

```
public interface IValidation
{
  Task<bool> IsValidXmlTagAsync(this string input);
}
```

Aby zwrócić wyniki opakowane w zadanie, można użyć pomocnych metod `FromX`, jak w poniższym kodzie:

```
using System.Text.RegularExpressions;

namespace Packt.Shared;

public static class StringExtensions : IValidation
{
  public static Task<bool> IsValidXmlTagAsync(this string input)
  {
    if (input == null)
    {
      return Task.FromException<bool>(
        new ArgumentNullException($"Brak parametru {nameof(input)}"));
    }

    if (input.Length == 0)
    {
      return Task.FromException<bool>(
        new ArgumentException($"Parameter {nameof(input)} jest pusty."));
    }

    return Task.FromResult(Regex.IsMatch(input,
      @"^<([a-z]+)([^<]+)*(?:>(.*)<\/\1>|\s+\/>)$"));
  }
}
```

Jeżeli zaimplementowana metoda zwraca wynik typu `Task` (odpowiednik `void` w metodzie synchronicznej), może to być predefiniowany obiekt reprezentujący ukończone zadanie, jak w poniższym kodzie:

```
public Task DeleteCustomerAsync()
{
  // ...
  return Task.CompletedTask;
}
```

Synchronizacja dostępu do współdzielonych zasobów

Jeżeli kilka wątków jest wykonywanych w tym samym czasie, może się zdarzyć, że jednocześnie odwołają się do tej samej zmiennej lub zasobu i spowodują problem. Dlatego trzeba zwracać baczną uwagę, aby kod był **wątkowo bezpieczny**. Najprostszym rozwiązaniem jest użycie zmiennej obiektowej w charakterze flagi oznaczającej blokadę współdzielonego zasobu i uzyskanie do niego dostępu na wyłączność.

W książce *Władca much* Williama Goldinga Prosiaczek i Ralph znajdują muszlę i używają jej do przeprowadzenia zebrania. Określają przy tym „zasadę muszli", że mówić może tylko ten, kto trzyma muszlę.

Obiekt, którego używam do implementacji kodu wątkowo bezpiecznego, mam zwyczaj nazywać muszlą. Gdy jakiś wątek „ma muszlę", żaden inny wątek nie powinien uzyskać dostępu do reprezentowanych przez nią współdzielonych zasobów. Zwróć uwagę, że użyłem tu zwrotu „nie powinien". Zsynchronizowany dostęp jest możliwy tylko wtedy, gdy w kodzie jest respektowana „zasada muszli". Muszla sama w sobie nie jest blokadą.

Przyjrzymy się dwóm typom obiektów używanych do synchronizowania dostępu do współdzielonych zasobów:

- `Monitor`: obiekt wykorzystywany przez wiele wątków do sprawdzania, czy mogą uzyskać dostęp do zasobu w ramach tego samego procesu.
- `Interlocked`: obiekt wykorzystywany do wykonywania operacji na prostych typach liczbowych na poziomie procesora.

Synchronizacja dostępu do zasobu

Utwórz aplikację konsolową, która pozwoli Ci dowiedzieć się, jak zasoby są współdzielone między wątkami.

1. W wybranym edytorze kodu otwórz rozwiązanie/ obszar roboczy o nazwie *Chapter04* i dodaj do niego aplikację konsolową o nazwie *SynchronizingResourceAccess*.
   - W środowisku Visual Studio 2022 oznacz bieżący projekt jako startowy.
   - W środowisku Visual Studio Code ustaw *SynchronizingResourceAccess* jako aktywny projekt OmniSharp.
2. W tym samym projekcie zaimportuj statycznie i globalnie przestrzeń nazw `System.Console`.
3. Utwórz plik klasy *SharedObjects.cs*.

4. W nowym pliku zdefiniuj statyczną klasę zawierającą pole `Message`, które będzie współdzielonym zasobem, jak niżej:

```
static class SharedObjects
{
  public static string? Message; // Współdzielony zasób.
}
```

5. Utwórz plik klasy *Program.Methods.cs*.

6. W nowym pliku wpisz poniższy kod definiujący dwie metody. Każda z nich zawiera pętlę, która pięciokrotnie dołącza do pola `Message` literę `A` lub `B`:

```
partial class Program
{
  static void MethodA()
  {
    for (int i = 0; i < 5; i++)
    {
      Thread.Sleep(Random.Shared.Next(2000));
      SharedObjects.Message += "A";
      Write(".");
    }
  }

  static void MethodB()
  {
    for (int i = 0; i < 5; i++)
    {
      Thread.Sleep(Random.Shared.Next(2000));
      SharedObjects.Message += "B";
      Write(".");
    }
  }
}
```

7. Usuń z pliku *Program.cs* istniejące instrukcje, a następnie wpisz poniższy kod, który importuje przestrzeń nazw `System.Diagnostics`, wywołuje obie powyższe metody wykonujące zadania w osobnych wątkach, oczekuje na zakończenie ich działania i wyświetla czas, jaki upłynął:

```
using System.Diagnostics; // Stopwatch

WriteLine("Oczekiwanie na zakończenie zadań.");
Stopwatch watch = Stopwatch.StartNew();
Task a = Task.Factory.StartNew(MethodA);
Task b = Task.Factory.StartNew(MethodB);

Task.WaitAll(new Task[] { a, b });
WriteLine();
WriteLine($"Wynik: {SharedObjects.Message}.");
WriteLine($"Czas działania: {timer.ElapsedMilliseconds:N0} ms.");
```

8. Uruchom kod. Uzyskasz wynik podobny do poniższego:

```
Oczekiwanie na zakończenie zadań.
..........
Wynik: BABABAABBA.
Czas działania: 6 740 ms.
```

Wynik dowodzi, że dwa wątki jednocześnie modyfikowały właściwość `Message`. W aplikacji z prawdziwego zdarzenia oznaczałoby to problem. Jednoczesnemu odwoływaniu się do zasobu można zapobiec przez założenie na obiekt muszli blokady dostępu na wyłączność oraz dodanie do obu metod kodu sprawdzającego tę blokadę przed próbą modyfikacji zasobu. Tę operację zaimplementujesz w następnym podrozdziale.

Zakładanie na muszlę blokady dostępu na wyłączność

Teraz użyj obiektu muszli, aby tylko jeden wątek w danej chwili miał dostęp do zasobu.

1. W pliku *SharedObjects.cs* wpisz poniższy kod tworzący obiekt, który będzie pełnił funkcję muszli:

```
public static object Conch = new();
```

2. W pliku *Program.Methods.cs* wpisz w metodach `MethodA` i `MethodB` instrukcję **`lock`** przed instrukcją `for`:

```
lock (SharedObjects.Conch)
{
  for (int i = 0; i < 5; i++)
  {
    Thread.Sleep(Random.Shared.Next(2000));
    SharedObjects.Message += "A";
    Write(".");
  }
}
```

Wskazówka

Dobra praktyka: zwróć uwagę, że sprawdzanie blokady muszli jest kwestią „dobrej woli". Jeśli instrukcja `lock` zostanie użyta tylko w jednej z dwóch metod, obie będą miały jednoczesny dostęp do współdzielonego zasobu.

3. Uruchom kod. Uzyskasz wynik podobny do poniższego:

```
Oczekiwanie na zakończenie zadań.
..........
Wynik: BBBBBAAAAA.
Czas działania: 10 345 ms.
```

Tym razem kod działał dłużej, ale w każdej chwili tylko jedna z metod miała dostęp do współdzielonego zasobu. Każda z nich może być wywołana w pierwszej kolejności. Gdy jedna kończy przetwarzanie współdzielonego zasobu, zwalnia blokadę muszli, aby druga metoda mogła zacząć wykonywać swoje operacje.

Instrukcja lock

Zapewne się zastanawiasz, co robi poniższa instrukcja `lock`, gdy blokuje zmienną obiektową (podpowiedź: nie blokuje zasobu):

```
lock (SharedObjects.Conch)
{
  // Operacje wykonywane na współdzielonym zasobie.
}
```

Kompilator języka C# zmienia instrukcję `lock` na instrukcje `try-finally`, które używają klasy `Monitor` do wejścia do obiektu muszli i wyjścia z niego (mam zwyczaj nazywać to założeniem i zwolnieniem blokady muszli), jak niżej:

```
try
{
  Monitor.Enter(SharedObjects.Conch);
  // Operacje wykonywane na współdzielonym zasobie.
}
finally
{
  Monitor.Exit(SharedObjects.Conch);
}
```

Metoda `Monitor.Enter` sprawdza, czy wątek inny niż jej własny założył blokadę na obiekcie muszli umieszczonym w jej argumencie. Jeżeli tak, wstrzymuje działanie. W przeciwnym razie zakłada blokadę i wątek może wykonywać operacje na współdzielonym zasobie. Gdy kończy, wywołuje metodę `Monitor.Exit`, która zwalnia blokadę muszli. Jeżeli w tym czasie czekał inny wątek, może teraz założyć swoją blokadę i wykonać operacje. Wymagane jest przy tym, aby wszystkie wątki przestrzegały „zasady muszli", a więc wywoływały metody, odpowiednio, `Monitor.Enter` i `Monitor.Exit`.

Wskazówka

Dobra praktyka: nie można w charakterze muszli stosować typów wartości (`struct`). Metoda `Monitor.Enter` wymaga typu referencyjnego, ponieważ blokuje adres obszaru pamięci.

Zapobieganie zakleszczeniu blokad

Sposób, w jaki kompilator zamienia instrukcję `lock` na wywołania metod klasy `Monitor`, warto znać również dlatego, aby wiedzieć, jak zapobiec zakleszczeniu blokad.

Zakleszczenie ma miejsce wtedy, gdy współdzielonych zasobów jest kilka, z każdym jest powiązana muszla wykorzystywana do kontrolowania, który wątek może wykonywać na nim operacje, i zachodzi następująca sekwencja zdarzeń:

- Wątek X blokuje muszlę A i zaczyna wykonywać operacje na współdzielonym zasobie A.
- Wątek Y blokuje muszlę B i zaczyna wykonywać operacje na współdzielonym zasobie B.
- Wątek X musi w trakcie przetwarzania zasobu A wykonać operację na zasobie B i w tym celu usiłuje założyć blokadę na muszli B. Muszla ta jednak jest zablokowana przez wątek Y.
- Wątek Y musi w trakcie przetwarzania zasobu B wykonać operację na zasobie A i w tym celu usiłuje założyć blokadę na muszli A. Muszla ta jednak jest zablokowana przez wątek X.

Jednym ze sposobów uniknięcia zakleszczenia blokad jest określenie maksymalnego czasu oczekiwania na zwolnienie blokady. W tym celu zamiast instrukcji `lock` należy jawnie użyć klasy `Monitor`.

1. W pliku *Program.Methods.cs* zastąp instrukcję `lock` wyróżnionym niżej kodem, który podejmuje próbę wejścia do obiektu muszli na określony czas. Jeżeli to się nie uda, kod wyświetla komunikat o błędzie. W obiekt może wtedy wejść inny wątek.

```
try
{
  if (Monitor.TryEnter(SharedObjects.Conch, TimeSpan.FromSeconds(15)))
  {
    for (int i = 0; i < 5; i++)
    {
      Thread.Sleep(Random.Shared.Next(2000));
      SharedObjects.Message += "A";
      Write(".");
    }
  }
  else
  {
    WriteLine("Przekroczony czas blokowania muszli przez metodę A.");
  }
}
finally
{
  Monitor.Exit(SharedObjects.Conch);
}
```

2. Uruchom kod. Powinieneś uzyskać taki sam wynik jak poprzednio (aczkolwiek muszlę jako pierwsza może przejąć każda z metod). Ten kod jest jednak lepszy od poprzedniego, ponieważ nie powoduje zakleszczenia blokad.

Wskazówka

Dobra praktyka: używaj instrukcji `lock` tylko wtedy, gdy możesz napisać kod, nie narażając się na zakleszczenia. Jeżeli nie jest to możliwe, stosuj kombinację metody `Monitor.TryEnter` i instrukcji `try-finally`, dzięki którym możesz określić maksymalny czas oczekiwania wątku na zwolnienie blokady i uniknąć potencjalnego zakleszczenia. Więcej dobrych praktyk pisania wielowątkowego kodu znajdziesz na stronie *https://learn.microsoft.com/pl-pl/dotnet/standard/threading/managed-threading-best-practices*.

Synchronizacja zdarzeń

Zdarzenia generowane przez platformę .NET nie są wątkowo bezpieczne, dlatego nie należy z nich korzystać w wielowątkowym kodzie.

Czasami programiści próbują zakładać blokady na wyłączność podczas dodawania i usuwania procedur obsługi zdarzeń, jak również podczas zgłaszania zdarzeń, na przykład:

```
// Pole delegata zdarzenia.
public event EventHandler? Shout;

// Muszla.
private object eventConch = new();

// Metoda.
public void Poke()
{
  lock (eventConch) // Zły pomysł.
  {
    // Jeżeli coś nasłuchuje...
    if (Shout != null)
    {
      // ...wywołaj delegata i zgłoś zdarzenie.
      Shout(this, EventArgs.Empty);
    }
  }
}
```

Wskazówka

Dobra praktyka: Czy to dobrze, czy źle, że programiści tak robią? Zależy to od wielu skomplikowanych czynników, więc nie sposób wydać jednoznacznej oceny. Więcej o zdarzeniach i bezpieczeństwie wątków dowiesz się na stronie *https://docs.microsoft.com/en-us/archive/blogs/cburrows/field-like-events-considered-harmful*.

O tym, jak złożone jest to zagadnienie, pisze Stephen Cleary na swoim blogu pod adresem *https://blog.stephencleary.com/2009/06/threadsafe-events.html*.

Kodowanie atomicznych operacji procesora

Termin *atomiczny* pochodzi od greckiego słowa *atomos* — niepodzielny. Znajomość pojęcia atomiczności w wielowątkowym kodzie jest ważna, ponieważ operacja nieatomiczna może zostać w trakcie jej wykonywania przerwana przez inny wątek. Na przykład czy operacja inkrementacji w poniższym kodzie jest atomiczna?

```
int x = 3;
x++; // Czy jest to atomiczna operacja procesora?
```

Nie jest! Inkrementacja liczby całkowitej wymaga wykonania przez procesor następujących kroków:

1. Załadowanie zmiennej do rejestru.
2. Inkrementacja wartości.
3. Zapisanie wartości zmiennej.

Wątek po wykonaniu dwóch pierwszych kroków może zostać przerwany przez inny wątek, który wykona wszystkie kroki. Gdy pierwszy wznowi działanie, nadpisze wartość zmiennej i w ten sposób zniweczy inkrementację dokonaną przez drugi wątek.

Istnieje typ `Interlocked`, umożliwiający wykonywanie atomicznych operacji `Add`, `Increment`, `Decrement`, `Exchange`, `CompareExchange`, `And`, `Or` i `Read` na liczbach całkowitych następujących typów:

- `System.Int32` (`int`), `System.UInt32` (`uint`),
- `System.Int64` (`long`), `System.UInt64` (`ulong`).

Nie można go jednak używać z liczbami typu `byte`, `sbyte`, `short`, `ushort` i `decimal`.

Operacje `Exchange` i `CompareExchange`, polegające na zamianie wartości zapisanych w pamięci, można wykonywać na danych następujących typów:

- `System.Single` (`float`), `System.Double` (`double`),
- `nint`, `nuint`,
- `T`, `System.Object` (`object`).

Zobacz teraz typ `Interlocked` w akcji.

1. W klasie `SharedObjects` zadeklaruj pole, w którym będzie zapisywana liczba wykonanych operacji:

   ```
   public static int Counter; // Inny współdzielony zasób.
   ```

2. W metodach `MethodA` i `MethodB` wpisz wewnątrz pętli `for`, pod instrukcją modyfikującą ciąg znaków, poniższy kod, który w bezpieczny sposób inkrementuje licznik:

   ```
   Interlocked.Increment(ref SharedObjects.Counter);
   ```

3. W pliku *Program.cs*, pod instrukcją wyświetlającą czas działania programu, wpisz poniższy kod wyświetlający wartość licznika:

```
WriteLine($"Liczba modyfikacji ciągu znaków:{SharedObjects.Counter}.");
```

4. Uruchom kod. Uzyskasz wynik podobny do poniższego:

```
Oczekiwanie na zakończenie zadań.
..........
Wynik: BBBBBAAAAA.
Czas działania: 12 129 ms.
Liczba modyfikacji ciągu znaków: 10.
```

Gdy się uważnie przyjrzysz, zauważysz, że chronione są wszystkie zasoby, do których kod odwołuje się w bloku zabezpieczonym z użyciem obiektu muszli. Zatem w tym konkretnym przypadku użycie typu `Interlocked` nie jest konieczne. Inaczej byłoby, gdyby nie był chroniony inny współdzielony zasób, na przykład pole `Message`.

Inne techniki synchronizacji dostępu

`Monitor` i `Interlocked` to wzajemnie wykluczające się blokady dostępu na wyłączność. Są proste i skuteczne, ale czasami potrzebne są bardziej zaawansowane techniki synchronizowania dostępu do współdzielonego zasobu. Są one opisane w poniższej tabeli.

Typ	Opis
`ReaderWriterLock`, `ReaderWriterLockSlim`	Kilka wątków może działać w trybie odczytu. Jeden wątek może działać w trybie zapisu z blokadą zasobu na wyłączność. Wątek działający w trybie odczytu może przejść w tryb zapisu bez rezygnacji z dostępu do odczytu
`Mutex`	Typ ten, podobnie jak `Monitor`, daje wyłączny dostęp do udostępnionego zasobu, chyba że jest on używany do synchronizowania dostępu między procesami
`Semaphore`, `SemaphoreSlim`	Ograniczenie za pomocą slotów liczby wątków, które mogą jednocześnie uzyskiwać dostęp do zasobu lub puli zasobów
`AutoResetEvent`, `ManualResetEvent`	Procedury obsługi zdarzeń wykorzystywane przez wątki do synchronizowania swoich działań przez wysyłanie do siebie nawzajem sygnałów i oczekiwanie na nie

Instrukcje async i await

W wersji języka C# 5 wprowadzono dwie instrukcje wykorzystujące typ `Task`, szczególnie użyteczne w następujących sytuacjach:

- implementacja wielozadaniowości w graficznym interfejsie użytkownika,
- skalowanie aplikacji i usług internetowych.

W rozdziale 18., „Tworzenie aplikacji mobilnych i stacjonarnych z użyciem .NET MAUI", dowiesz się, jak z użyciem instrukcji `async` i `await` implementować wielozadaniowość w graficznym interfejsie użytkownika.

Teraz poznaj podstawy teoretyczne uzasadniające wprowadzenie tych instrukcji, a później zastosuj je w praktyce.

Poprawa responsywności aplikacji konsolowej

Instrukcję `await` można stosować wyłącznie w metodach asynchronicznych. W wersjach języka C# 7 i starszych nie można było oznaczać instrukcją `async` metody `Main` w aplikacji konsolowej. Na szczęście jedną z nowości wprowadzonych w wersji C# 7.1 jest możliwość deklarowania tej metody jako asynchronicznej.

1. W wybranym edytorze kodu otwórz rozwiązanie/ obszar roboczy o nazwie *Chapter04* i dodaj do niego aplikację konsolową o nazwie *AsyncConsole*.
   - W środowisku Visual Studio 2022 oznacz bieżący projekt jako startowy.
   - W środowisku Visual Studio Code ustaw *AsyncConsole* jako aktywny projekt OmniSharp.
2. Usuń z pliku *Program.cs* istniejące instrukcje, zaimportuj statycznie przestrzeń nazw `System.Console` i wpisz poniższy kod, który tworzy instancję klasy `HttpClient`, wysyła zapytanie na adres strony Apple i wyświetla liczbę odebranych bajtów:

```
using static System.Console;

HttpClient client = new();

HttpResponseMessage response =
  await client.GetAsync("http://www.apple.com/");

WriteLine("Wielkość strony głównej Apple w bajtach: {0:N0}.",
  response.Content.Headers.ContentLength);
```

3. Uruchom kompilację projektu i zwróć uwagę, że zostanie wykonana pomyślnie. W wersjach platformy .NET 5 i starszych szablon projektu zawierał jawną klasę `Program` z synchroniczną metodą `Main`, więc próba kompilacji kończyła się wyświetleniem następującego komunikatu:

```
Program.cs(14,9): error CS4033: The 'await' operator can only be used
within an async method. Consider marking this method with the 'async'
modifier and changing its return type to 'Task'. [/Users/markjprice/
appsservices-net7/Chapter04/AsyncConsole/AsyncConsole.csproj]
```

(Operatora 'await' można używać tylko w metodzie asynchronicznej. Oznacz tę metodę modyfikatorem 'async' i zmień typ zwracanego przez nią wyniku na 'Task').

4. Począwszy od wersji.NET 6 szablon aplikacji konsolowej wykorzystuje funkcje najwyższego poziomu i automatycznie definiuje klasę `Program` z asynchroniczną metodą `<Main>$`.
5. Uruchom kod. Możesz uzyskać inny wynik od poniższego, ponieważ Apple często zmienia swoją stronę główną.

```
Wielkość strony głównej Apple w bajtach: 162 480.
```

Operacje na strumieniach asynchronicznych

Wraz z wersją platformy .NET Core 3.0 pojawiła się możliwość wykonywania asynchronicznych operacji na strumieniach.

Uwaga

Na stronie *https://learn.microsoft.com/pl-pl/dotnet/csharp/asynchronous-programming/generate-consume-asynchronous-stream* znajduje się przewodnik po strumieniach asynchronicznych.

W starszych wersjach języka i platformy niż, odpowiednio, C# 8.0 i .NET Core 3.0 instrukcję `await` można było stosować tylko w metodach zwracających wartości skalarne. W platformie .NET Standard 2.1 metoda asynchroniczna może zwracać wartości z użyciem strumienia.

Przeanalizujmy przykładową metodę, która zwraca trzy losowe liczby całkowite jako strumień asynchroniczny.

1. W wybranym edytorze kodu otwórz rozwiązanie/ obszar roboczy o nazwie *Chapter04* i dodaj do niego aplikację konsolową o nazwie *AsyncEnumerable*.
   - W środowisku Visual Studio 2022 oznacz bieżący projekt jako startowy.
   - W środowisku Visual Studio Code ustaw *AsyncEnumerable* jako aktywny projekt OmniSharp.
2. Zaimportuj globalnie i statycznie przestrzeń nazw `System.Console`.
3. Usuń z pliku *Program.cs* istniejące instrukcje i wpisz poniższy kod, który definiuje metodę zwracającą asynchronicznie za pomocą instrukcji `yield` trzy losowe liczby całkowite:

```
async static IAsyncEnumerable<int> GetNumbersAsync()
{
  Random r = Random.Shared;
```

```
    // Symulacja działania.
    await Task.Delay(r.Next(1500, 3000));
    yield return r.Next(0, 1001);

    await Task.Delay(r.Next(1500, 3000));
    yield return r.Next(0, 1001);
    await Task.Delay(r.Next(1500, 3000));
    yield return r.Next(0, 1001);
  }
```

4. Powyżej metody `GetNumbersAsync` wpisz kod wyświetlający sekwencję liczb:

```
await foreach (int number in GetNumbersAsync())
{
  WriteLine($"Liczba: {number}");
}
```

5. Uruchom kod. Uzyskasz wynik podobny do poniższego:

```
Liczba: 820
Liczba: 70
Liczba: 629
```

Poprawa responsywności aplikacji graficznej

Do tej pory podczas pracy z tą książką tworzyłeś wyłącznie aplikacje konsolowe. Życie programisty jest trudniejsze, gdy musi kodować aplikacje i usługi internetowe, programy z graficznym interfejsem użytkownika, na przykład dla komputerów z systemem Windows i urządzeń przenośnych. Jest tak m.in. dlatego, że aplikacja graficzna wykorzystuje specjalny wątek UI (ang. *user interface*, interfejs użytkownika). Obowiązują przy tym dwie zasady:

- Wątek ten nie może wykonywać długotrwałych zadań.
- Do obiektów graficznych może się odwoływać tylko wątek UI.

Wcześniej, aby spełnić te wymagania, programiści musieli pisać złożony kod, w którym długotrwałe zadania były wykonywane przez dodatkowy wątek, a ten po zakończeniu działania przekazywał wyniki wątkowi UI w celu przedstawienia ich użytkownikowi. W takim kodzie bałagan pojawiał się bardzo szybko.

Na szczęście począwszy od wersji języka C# 5 można korzystać z instrukcji `async` i `await`. Dzięki nim można pisać kod, który wygląda jak synchroniczny, jest czysty i zrozumiały. Jednak pod spodem kompilator C# tworzy złożoną maszynę stanów i śledzi uruchomione wątki. To niemal czary! Kombinacja tych dwóch instrukcji sprawia, że metoda asynchroniczna jest wywoływana w wątku roboczym, a po zakończeniu działania zwraca wyniki do wątku UI.

Przeanalizujmy przykład aplikacji przeznaczonej dla komputerów stacjonarnych, opartej na platformie WPF. Aplikacja ta będzie odczytywała z bazy SQL Server *Northwind* dane pracowników z wykorzystaniem niskopoziomowych typów `SqlConnection`, `SqlCommand` i `SqlDataReader`.

Uwaga

Northwind jest średnio złożoną bazą danych, zawierającą przyzwoitą liczbę przykładowych rekordów. Poznałeś ją, skonfigurowałeś i intensywnie wykorzystywałeś w rozdziale 2., „Zarządzanie relacyjną bazą danych z użyciem oprogramowania SQL Server".

Wskazówka

Opisany przykład będziesz mógł wykonać, jeżeli korzystasz z systemu Microsoft Windows i masz bazę Microsoft SQL Server *Northwind*. To jedyny rozdział książki, w którym aplikacja nie jest wieloplatformowa i nowoczesna (platforma WPF ma już 17 lat!). Możesz użyć środowiska Visual Studio 2022 lub Visual Studio Code.

Teraz skupimy się na responsywności aplikacji graficznej. Język XAML i techniki tworzenia uniwersalnych aplikacji graficznych poznasz w rozdziale 18., „Tworzenie aplikacji mobilnych i stacjonarnych z użyciem .NET MAUI". Ponieważ nigdzie indziej w tej książce nie będziemy się zajmować platformą WPF, uznałem, że teraz będzie dobra okazja, aby poznać — choćby ogólnie — opartą na niej przykładową aplikację.

Do dzieła!

1. Jeżeli używasz środowiska Visual Studio 2022, utwórz w rozwiązaniu/obszarze roboczym *Chapter04* projekt aplikacji WPF (C#) o nazwie *WpfResponsive*. Jeżeli zaś korzystasz ze środowiska Visual Studio Code, utwórz nowy projekt za pomocą polecenia `dotnet new wpf` i ustaw go jako aktywny projekt OmniSharp.
2. Dodaj w projekcie odwołanie do pakietu `Microsoft.Data.SqlClient`.
3. Otwórz plik konfiguracyjny projektu i zwróć uwagę, że docelowa aplikacja jest typu Windows EXE oraz że wykorzystuje ona platformy .NET 7 dla Windows (nie uruchomi się w systemie macOS ani Linux) i WPF.

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net7.0-windows</TargetFramework>
    <Nullable>enable</Nullable>
    <UseWPF>true</UseWPF>
  </PropertyGroup>
```

```
  <ItemGroup>
    <PackageReference Include="Microsoft.Data.SqlClient" Version="5.0.0" />
  </ItemGroup>

</Project>
```

4. Skompiluj projekt, aby został pobrany wymagany pakiet.

5. Otwórz plik *MainWindow.xaml* i w sekcji `<Grid>` wpisz poniższy kod definiujący dwa przyciski, pole tekstowe i listę. Elementy te będą ułożone pionowo, jeden nad drugim:

```
<StackPanel>
  <Button Name="GetEmployeesSyncButton"
          Click="GetEmployeesSyncButton_Click">

    Synchroniczny odczyt danych pracowników</Button>
  <Button Name="GetEmployeesAsyncButton"
          Click="GetEmployeesAsyncButton_Click">
    Asynchroniczny odczyt danych pracowników </Button>
  <TextBox HorizontalAlignment="Stretch" Text="Wpisz tutaj" />
  <ListBox Name="EmployeesListBox" Height="350" />
</StackPanel>
```

Uwaga

Środowisko Visual Studio 2022 dla systemu Windows, w odróżnieniu od Visual Studio Code, dobrze współpracuje z platformą WPF i oferuje funkcję IntelliSense ułatwiającą edytowanie kodu i znaczników XAML.

6. W pliku *MainWindow.xaml.cs* wpisz instrukcje importujące przestrzenie nazw **`System.Diagnostics`** i **`Microsoft.Data.SqlClient`**.

7. W klasie `MainWindow` wpisz poniższy kod definiujący dwie stałe tekstowe zawierające, odpowiednio, ciąg połączenia z bazą danych oraz polecenie SQL:

```
private const string connectionString =
  "Data Source=.;" +
  "Initial Catalog=Northwind;" +
  "Integrated Security=true;" +
  "Encrypt=false;" +
  "MultipleActiveResultSets=true;";

private const string sql =
  "WAITFOR DELAY '00:00:05';" +
  "SELECT EmployeeId, FirstName, LastName FROM Employees";
```

8. Wpisz poniższy kod, który definiuje metody obsługujące zdarzenia kliknięcia przycisków. Metody te wykorzystują powyższe stałe tekstowe do połączenia się z bazą *Northwind*, odczytania identyfikatorów i nazwisk pracowników, a następnie wyświetlenia ich w liście.

```
private void GetEmployeesSyncButton_Click(object sender, RoutedEventArgs e)
{
  Stopwatch timer = Stopwatch.StartNew();
  using (SqlConnection connection = new(connectionString))
  {
    try
    {
      connection.Open();

      SqlCommand command = new(sql, connection);
      SqlDataReader reader = command.ExecuteReader();

      while (reader.Read())
      {
        string employee = string.Format("{0}: {1} {2}",
          reader.GetInt32(0), reader.GetString(1), reader.GetString(2));

        EmployeesListBox.Items.Add(employee);
      }

      reader.Close();
      connection.Close();
    }
    catch (Exception ex)
    {
      MessageBox.Show(ex.Message);
    }
  }
  EmployeesListBox.Items.Add($"Czas odczytu synchronicznego:
{timer.ElapsedMilliseconds:N0} ms");
}

private async void GetEmployeesAsyncButton_Click(
  object sender, RoutedEventArgs e)
{
  Stopwatch timer = Stopwatch.StartNew();

  using (SqlConnection connection = new(connectionString))
  {
    try
    {
      await connection.OpenAsync();

      SqlCommand command = new(sql, connection);
      SqlDataReader reader = await command.ExecuteReaderAsync();

      while (await reader.ReadAsync())
      {
        string employee = string.Format("{0}: {1} {2}",
          await reader.GetFieldValueAsync<int>(0),
          await reader.GetFieldValueAsync<string>(1),
```

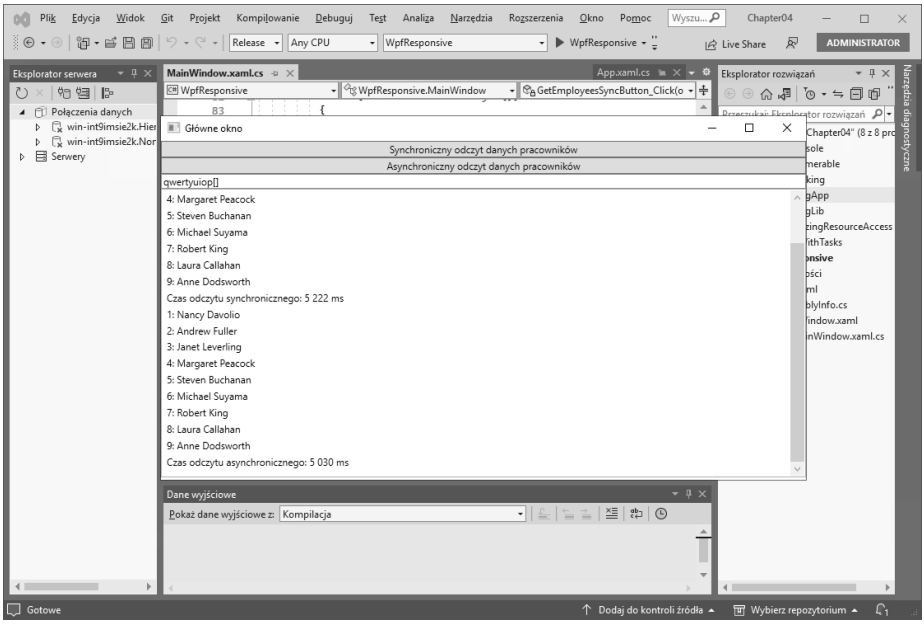
```
            await reader.GetFieldValueAsync<string>(2));

          EmployeesListBox.Items.Add(employee);
        }
        await reader.CloseAsync();
        await connection.CloseAsync();
      }
      catch (Exception ex)
      {
        MessageBox.Show(ex.Message);
      }
    }
    EmployeesListBox.Items.Add($"Czas odczytu asynchronicznego:
{timer.ElapsedMilliseconds:NO} ms");
}
```

Zwróć uwagę na następujące kwestie:

- Definiowanie metody `async void` jest złą praktyką „wywołaj i zapomnij". Nie uzyska się powiadomienia o jej zakończeniu i nie ma możliwości przerwania jej działania, ponieważ nie zwraca ona wyniku typu `Task` ani `Task<T>`, który dawałby nad nią kontrolę.
- Polecenie SQL symuluje pracę przez 5 sekund z wykorzystaniem instrukcji `WAITFOR DELAY`, a następnie odczytuje rekordy z trzech kolumn tabeli `Employees`.
- Metoda `GetEmployeesSyncButton_Click`, obsługująca zdarzenie kliknięcia przycisku, wykorzystuje synchroniczne metody do nawiązania połączenia z bazą i odczytania danych pracowników.
- Metoda `GetEmployeesAsyncButton_Click`, obsługująca zdarzenie kliknięcia przycisku, wykorzystuje asynchroniczne metody oznaczone instrukcją `async` do nawiązania połączenia z bazą i odczytania danych pracowników.
- Obie powyższe metody wykorzystują klasę `Stopwatch` do zarejestrowania czasu działania, a następnie wyświetlenia go w liście.

9. Uruchom aplikację w trybie *Release* bez debugowania.
10. Kliknij pole tekstowe i wpisz w nim dowolny ciąg. Zwróć uwagę, że interfejs graficzny jest responsywny.
11. Kliknij przycisk *Synchroniczny odczyt danych pracowników*.
12. Kliknij pole tekstowe i zwróć uwagę, że interfejs graficzny nie jest responsywny.
13. Zaczekaj 5 sekund, aż w liście pojawią się dane pracowników.
14. Ponownie kliknij pole tekstowe i wpisz w nim dowolną treść. Interfejs z powrotem jest responsywny.
15. Kliknij przycisk *Asynchroniczny odczyt danych pracowników*.

16. Kliknij pole tekstowe i wpisz w nim dowolną treść. Zwróć uwagę, że w trakcie wykonywania operacji interfejs graficzny jest responsywny. Wpisuj znaki, aż pojawią się dane pracowników (patrz rysunek 4.2).



Rysunek 4.2. Synchroniczny i asynchroniczny odczyt danych w aplikacji WPF

17. Zwróć uwagę na różne czasy wykonania obu operacji. Na czas synchronicznego odczytywania danych interfejs graficzny jest blokowany, natomiast w trakcie odczytu asynchronicznego jest responsywny.

18. Zamknij aplikację.

Poprawa skalowalności aplikacji i usług internetowych

Instrukcje `async` i `await` można również stosować w kodzie generującym strony internetowe, w aplikacjach i usługach uruchamianych na serwerach. Z perspektywy aplikacji klienckiej nic się nie zmienia (może z wyjątkiem nieznacznego wydłużenia czasu przetwarzania zapytania). Natomiast efektem implementacji wielozadaniowości na serwerze za pomocą instrukcji `async` i `await` mogą być gorsze wrażenia użytkownika.

Po stronie serwera tworzone są dodatkowe, tańsze wątki robocze, które czekają na zakończenie wykonania długotrwałych zadań. Dzięki temu drogie wątki wykonujące operacje wejścia/wyjścia nie są blokowane i mogą obsługiwać inne żądania klientów.

Poprawia to ogólną skalowalność aplikacji lub usługi internetowej, która może obsługiwać większą liczbę klientów jednocześnie.

Popularne typy obsługujące wielozadaniowość

Istnieje wiele popularnych typów zawierających metody asynchroniczne, które można stosować z instrukcją `await`. Przedstawia je poniższa tabela.

Typ	Metody
`DbContext<T>`	`AddAsync`, `AddRangeAsync`, `FindAsync`, `SaveChangesAsync`
`DbSet<T>`	`AddAsync`, `AddRangeAsync`, `ForEachAsync`, `SumAsync`, `ToListAsync`, `ToDictionaryAsync`, `AverageAsync`, `CountAsync`
`HttpClient`	`GetAsync`, `PostAsync`, `PutAsync`, `DeleteAsync`, `SendAsync`
`StreamReader`	`ReadAsync`, `ReadLineAsync`, `ReadToEndAsync`
`StreamWriter`	`WriteAsync`, `WriteLineAsync`, `FlushAsync`

Wskazówka

Dobra praktyka: widząc metodę o nazwie zakończonej przyrostkiem `Async`, zawsze sprawdzaj, czy zwraca ona wynik typu `Task` lub `Task<T>`. Jeżeli tak, możesz jej używać zamiast synchronicznej metody z innym sufiksem w nazwie. Pamiętaj, aby wywoływać ją z użyciem instrukcji `await`, a swoją metodę oznaczać instrukcją `async`.

Instrukcja await w bloku catch

W wersji języka C# 5 instrukcję `await` można było stosować w bloku `try`, ale nie w `catch`. Począwszy od wersji C# 6 można to robić w obu blokach.

Nauka i praktyka

Sprawdź swoją wiedzę i umiejętności: odpowiedz na kilka pytań, wykonaj praktyczne ćwiczenia i pogłęb wiedzę na tematy omówione w tym rozdziale.

Ćwiczenie 4.1. Sprawdź swoją wiedzę

Odpowiedz na poniższe pytania:

1. Jakie informacje można pozyskać o procesie?
2. Jak dokładna jest klasa `Stopwatch`?

3. Jaki sufiks powinna mieć zgodnie z przyjętą konwencją nazwa metody zwracającej wynik typu `Task` lub `Task<T>`?
4. Jakiej instrukcji należy użyć w deklaracji metody, aby w jej wnętrzu można było stosować instrukcję `await`?
5. Jak się tworzy zadanie podrzędne?
6. Dlaczego należy unikać używania instrukcji `lock`?
7. Kiedy należy stosować klasę `Interlocked`?
8. Kiedy należy stosować klasę `Mutex` zamiast `Monitor`?
9. Jakie są zalety stosowania instrukcji `async` i `await` w kodzie aplikacji lub usługi internetowej?
10. Czy można przerwać zadanie? Jeżeli tak, to w jaki sposób?

Ćwiczenie 4.2. Zbadaj temat

Odwiedź stronę *https://github.com/markjprice/apps-services-net7/blob/main/book-links. md#chapter-4---improving-performance-and-scalability-using-multitasking*, aby dowiedzieć się więcej na tematy poruszone w tym rozdziale.

Ćwiczenie 4.3. Dowiedz się więcej o programowaniu równoległym

Tematy poruszone w tym rozdziale są dokładniej omówione w książce *Parallel Programming and Concurrency with C# 10 and .NET 6: A modern approach to building faster, more responsive, and asynchronous .NET applications using C#* (Programowanie równoległe i współbieżność w C# 10 i .NET 6: nowoczesne techniki tworzenia szybszych, responsywnych i asynchronicznych aplikacji .NET z użyciem C#), Alvin Ashcraft, wyd. Packt (*https://www.packtpub.com/product/parallel-programming-and-concurrency-with-c-10-andnet-6/9781803243672*).

Podsumowanie

W tym rozdziale nauczyłeś się:

- definiować i uruchamiać zadania,
- kodować oczekiwanie na zakończenie jednego lub kilku zadań,
- kontrolować kolejność wykonywania zadań,

- synchronizować dostęp do współdzielonych zasobów,
- stosować magiczne instrukcje `async` i `await`.

W następnym rozdziale dowiesz się, jak korzystać z kilku popularnych zewnętrznych bibliotek.

Skorowidz

A

B

C

D

E

F

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

X

Z

Aplikacje w .NET, wydajne, skalowalne, bezpieczne — poznaj najnowsze rozwiązania!

C# i wieloplatformowy framework .NET sprawiają, że praca programisty jest efektywna i satysfakcjonująca. Podobnie jak w wypadku innych zaawansowanych technologii, nauka obsługi .NET może przysporzyć sporo trudności. Wielu deweloperów odkrywa, że oficjalna dokumentacja nie wystarczy do nabrania wprawy w tworzeniu złożonych projektów.

To książka przeznaczona dla programistów zaznajomionych z podstawami języka C# i platformy .NET, chcących zdobyć umiejętność tworzenia rzeczywistych aplikacji i usług. Opisuje wyspecjalizowane biblioteki, które umożliwiają monitorowanie i zwiększanie wydajności aplikacji, zabezpieczanie ich wraz z danymi, a także internacjonalizowanie ich kodu. Zawiera również omówienie najnowszych rozwiązań, bibliotek i technologii w połączeniu z ich praktycznym zastosowaniem — między innymi Web API, OData, gRPC, GraphQL, SignalR i Azure Functions. Nie zabrakło prezentacji technik pracy z .NET MAUI, programem, który służy do tworzenia aplikacji mobilnych dla systemów iOS i Android, a także stacjonarnych dla systemów Windows i macOS.

Najciekawsze zagadnienia:

- wydajność, bezpieczeństwo i skalowalność aplikacji i usług
- specjalistyczne biblioteki .NET i biblioteki zewnętrzne, takie jak Serilog i FluentValidation
- tworzenie wielosystemowych aplikacji i ich integracja z natywnymi funkcjami mobilnych systemów operacyjnych
- stosowanie różnych technologii, między innymi bibliotek komponentów Blazor
- praca z danymi w bazach SQL Server i Cosmos DB

Mark J. Price specjalizuje się w programowaniu w języku C#. Pracuje w Microsofcie, tworzy rozwiązania dla MS Azure. Zdał ponad 80 egzaminów Microsoftu. Zajmuje się też dydaktyką — specjalizuje się w przygotowywaniu kandydatów do egzaminów certyfikacyjnych.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-8322-716-0
9 788383 227160
Cena: 169,00 zł