Rozdział 7.
Elementy programowania współbieżnego

Jacek Matulewski
Materiały dla Podyplomowego Studium Programowania i Zastosowania Komputerów,
sekcja Projektowanie i tworzenie aplikacji dla platformy .NET (pod patronatem Microsoft)

Do platformy .NET w wersji 4.0 dodana została biblioteka TPL (ang. *Task Parallel Library*) oraz kolekcje umożliwiające pracę w różnych scenariuszach pojawiających się przy programowaniu współbieżnym. Całość popularnie nazywana *Parallel Extensions*. TPL nadbudowuje klasyczne wątki i pulę wątków, korzystając z nowej klasy `Task` (z ang. zadanie).

W szczegółach biblioteka TPL, mechanizmy synchronizacji „kolekcje równoległe" oraz narzędzia pozwalające na debugowanie i analizowanie programów równoległych opisane zostały w naszej książce *Programowanie równoległe i asynchroniczne w C# 5.0* wydanej w wydawnictwie Helion w grudniu 2013 roku. Tu chciałbym skupić się jedynie na najczęściej używanym jej elemencie — współbieżnej pętli `for` oraz na nowości C# 5.0 ogniskujących się na nowych słowach kluczowych `async` i `await`.

Równoległa pętla for

Załóżmy, że mamy zbiór stu liczb rzeczywistych, dla których musimy wykonać jakieś stosunkowo czasochłonne obliczenia. W naszym przykładzie będzie to obliczenie wartości funkcji $y = f(x) = arcsin(sin(x))$. Funkcja ta powinna z dokładnością numeryczną zwrócić wartość argumentu. I zrobi to, jednak nieźle się przy tym namęczy — funkcje trygonometryczne są bowiem wymagające numerycznie. Powtórzmy te obliczenia kilkukrotnie, aby dodatkowo przedłużyć czas obliczeń. Listing 7.1 prezentuje kod pliku *Program.cs* z aplikacji konsolowej, w której zaimplementowany został powyższy pomysł.

Listing 7.1. Metoda zajmująca procesor

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ProgramowanieRownolegle
{
    class Program
    {
        static private double obliczenia(double argument)
        {
```

```
            for (int i = 0; i < 10; ++i)
            {
                argument = Math.Asin(Math.Sin(argument));
            }
            return argument;
        }

        static void Main(string[] args)
        {
            //przygotowania
            int rozmiar = 10000;
            Random r = new Random();
            double[] tablica = new double[rozmiar];
            for (int i = 0; i < tablica.Length; ++i) tablica[i] = r.NextDouble();

            //obliczenia sekwencyjne
            int liczbaPowtorzen = 100;
            double[] wyniki = new double[tablica.Length];
            int start = System.Environment.TickCount;
            for (int powtorzenia = 0; powtorzenia < liczbaPowtorzen; ++powtorzenia)
                for (int i = 0; i < tablica.Length; ++i)
                    wyniki[i] = obliczenia(tablica[i]);
            int stop = System.Environment.TickCount;
            Console.WriteLine("Obliczenia sekwencyjne trwały " + (stop -
start).ToString() + " ms.");

            /*
            //prezentacja wyników
            Console.WriteLine("Wyniki:");
            for (long i = 0; i < tablica.Length; ++i) Console.WriteLine(i + ". " +
tablica[i] + " ?= " + wyniki[i]);
            */
        }
    }
}
```

Na tym samym listingu, w metodzie `Main` widoczna jest pętla wykonująca obliczenia wraz z przygotowaniem tablicy. Wyniki nie są drukowane — tablica jest zbyt duża, żeby to miało sens (poza tym drukowanie w konsoli jest bardzo wolne). Poniższy kod zawiera dwie zagnieżdżone pętle `for`. Interesuje nas tylko ta wewnętrzna. Zewnętrzna jedynie powtarza obliczenia, co po prostu zwiększa wiarygodność pomiaru czasu, który realizujemy na bazie zliczania taktów procesora. Pętlę wewnętrzną można bez większego wysiłku zrównoleglić dzięki klasie `Parallel` z przestrzeni nazw `System.Threading.Tasks`. W Visual Studio 2013 przestrzeń ta jest uwzględniona w domyślnie zadeklarowanych przestrzeniach w sekcji `using` na początku pliku. Zrównoleglona wersja tej pętli widoczna jest na listingu 7.2. Dodajmy ją do metody `Main`.

Listing 7.2. Przykład zrównoleglonej pętli for

```
//obliczenia równoległe
start = System.Environment.TickCount;
for (int powtorzenia = 0; powtorzenia < liczbaPowtorzen; ++powtorzenia)
{
```

```
        Parallel.For(0, tablica.Length, (int i) => wyniki[i] = obliczenia(tablica[i]));
    }
    stop = System.Environment.TickCount;
    Console.WriteLine("Obliczenia równoległe trwały " + (stop - start).ToString() + " ms.");
```

Jak wspomniałem, zrównoleglamy tylko wewnętrzną pętlę. Użyta do tego metoda `Parallel.For` jest dość intuicyjna w użyciu. Jej dwa pierwsze argumenty określają zakres zmiany indeksu pętli. W naszym przypadku jest on równy [0,10000). Wobec tego do metody podanej w trzecim argumencie przekazywane są liczby od 0 do 9999. Trzeci argument jest natomiast delegacją, do której można przypisać metodę (lub jak w naszym przypadku wyrażenie lambda) realizującą polecenia w każdej iteracji pętli. Powinna się tam zatem znaleźć zawartość oryginalnej pętli. Argumentem wyrażenia lambda jest bieżący indeks pętli.

Proces zrównoleglania kodu może być tak prosty, jak pokazałem powyżej, ale są to rzadkie przypadki. Często w ogóle nie jest on możliwy (tak jest w przypadku pętli `for` w tabeli 3.5 obliczającej silnię podanej liczby, w której kolejne iteracje zależą od poprzednich — muszą być wobec tego wykonywane sekwencyjnie), albo wymaga synchronizacji. Metoda `Parallel.For` automatycznie synchronizuje zadania po zakończeniu wszystkich iteracji, dlatego nie ma zagrożenia zamazania danych w ramach kolejnych jej powtórzeń. Jeżeli jednak zajdzie taka potrzeba, synchronizację można zrealizować za pomocą operatora `lock`, tego samego, którego używa się w przypadku wątków.

Warto przestrzec, że nie należy się spodziewać, że dzięki użyciu równoległej pętli `for` nasze obliczenia przyspieszą tyle razy, ile rdzeni procesora mamy do dyspozycji. Tworzenie i usuwanie zadań również zajmuje nieco czasu. Eksperymentując z rozmiarem tablicy i liczbą obliczanych sinusów, można sprawdzić, że zrównoleglanie opłaca się tym bardziej, im dłuższe są obliczenia wykonywane w ramach jednego zadania. Dla krótkich zadań użycie równoległej pętli może wręcz wydłużyć całkowity czas obliczeń. Na komputerze z jednym procesorem dwurdzeniowym uzyskane przez mnie uśrednione przyspieszenie dla powyższych parametrów było równe 66,4%. Z kolei w przypadku ośmiu rdzeni czas obliczeń równoległych spadł do zaledwie 34,8%.

Przerywanie pętli

Podobnie jak w klasycznej pętli `for`, również w przypadku wersji równoległej możemy w każdej chwili przerwać jej działanie. Służy do tego klasa `ParallelLoopState`. Aby to umożliwić należy dodać drugi argument do metody lub wyrażenia lambda wykonywanego w każdej iteracji – właśnie obiekt typu `ParallelLoopState`. Udostępnia on dwie metody: `Break` i `Stop`. Różnią się one tym, że `Break` pozwala na wcześniejsze zakończenie bieżącej iteracji bez uruchamiania następnych, a `Stop` nie tylko natychmiast kończy bieżący wątek, ale również podnosi flagę `IsStopped`, która powinna być sprawdzona we wszystkich uruchomionych wcześniej iteracjach, i która jest sygnałem do ich zakończenia (to leży jednak w gestii programisty przygotowujący kod wykonywany w każdej iteracji). Listing 7.3 pokazuje przykład, w którym pętla przerywana jest, jeżeli wylosowana zostanie liczba 0.

Listing 7.3. Przerywanie pętli równoległej

```
static void Main(string[] args)
{
    Random r = new Random();
    long suma = 0;
    long licznik = 0;
    string s = "";

    //iteracje zostaną wykonane tylko dla liczb parzystych
    //pętla zostanie przerwana wcześniej, jeżeli wylosowana liczba jest równa 0
    Parallel.For(0, 10000, (int i, ParallelLoopState stanPetli) =>
    {
        int liczba = r.Next(7); //losowanie liczby oczek na kostce
        if (liczba == 0)
```

```
        {
            s += "Stop:";
            stanPetli.Stop();
        }
        if (stanPetli.IsStopped) return;
        if (liczba % 2 == 0)
        {
            s += liczba.ToString() + "; ";
            obliczenia(liczba);
            suma += liczba;
            licznik++;
        }
        else s += "[" + liczba.ToString() + "]; ";
    });

    Console.WriteLine(
        "Wylosowane liczby: " + s +
        "\nLiczba pasujących liczb: " + licznik +
        "\nSuma: " + suma +
        "\nŚrednia: " + (suma / (double)licznik).ToString());
}
```

Programowanie asynchroniczne. Modyfikator async i operator await (nowość języka C# 5.0)

Programowanie asynchroniczne to niezwykle istotny element aplikacji *Windows Store* uruchamianych na nowym ekranie start Windows 8. Jednak również w „zwykłych” aplikacjach mechanizm ten może być z powodzeniem używany. Mniej jest w tym przypadku gotowych metod korzystających z tej techniki, ale można go bez problemu użyć we własnych rozwiązaniach. Dlatego w dalszej części rozdziału postaram się opisać mechanizm asynchroniczności właśnie w taki sposób, w jaki jego poznanie jest potrzebne, aby używać go we własnych projektach. Muszę jednak uprzedzić, że w kolejnych rozdziałach nie znalazł się żaden pretekst żeby tego mechanizmu użyć.

Jak wspomniałem, język C# 5.0 wyposażony został w nowy operator `await`, ułatwiający synchronizację dodatkowych uruchomionych przez użytkownika zadań. Poniżej zaprezentuję prosty przykład jego użycia, który chyba najlepiej wyjaśni jego działanie. Nie omówiłem wprawdzie zadań (mam na myśli bibliotekę TPL i jej sztandarową klasę `Task`), jednak podobnie jak w przypadku opisanej wyżej pętli równoległej `Parallel.For`, tak i w przypadku operatora `await` dogłębna znajomość biblioteki TPL nie jest do tego konieczna. Spójrzmy na przykład widoczny na listingu 7.4. Przedstawia on metodę `Main`, która definiuje przykładową czynność, zapisuje referencję do niej w zmiennej `akcja` i wykonuje ją synchronicznie. Czynność ta wprowadza półsekundowe opóźnienie metodą `Thread.Sleep`, które oczywiście opóźnia zwrot informacji odsyłanej przez serwer do klienta (przeglądarki).

Listing 7.4. Synchroniczne wykonywanie kodu zawartego w akcji

```
static void Main(string[] args)
{
    //czynność
    Func<object, long> akcja =
```

```
            (object argument) =>
            {
                Console.WriteLine("Początek działania akcji - " + argument.ToString());
                System.Threading.Thread.Sleep(500); //opóźnienie 0.5s
                Console.WriteLine("Koniec działania akcji - " + argument.ToString());
                return DateTime.Now.Ticks;
            };

        long wynik = akcja("synchronicznie");
        Console.WriteLine("Synchronicznie: " + wynik.ToString());
    }
```

W listingu 7.5 ta sama akcja wykonywana jest asynchronicznie w osobnym wątku utworzonym na potrzeby zdefiniowanego przez nas zadania. Synchronizacja następuje w momencie odczytania wartości zwracanej przez czynność, tj. w momencie odczytania właściwości `Result`. Jej sekcja `get` czeka ze zwróceniem wartości aż do zakończenia zadania i tym samym wstrzymuje wątek, w którym wykonywana jest metoda `Main`. Jest to zatem punkt synchronizacji. Zwróćmy uwagę, że po instrukcji `zadanie.Start`, a przed odczytaniem właściwości `Result` mogą być wykonywane dowolne czynności, o ile są niezależne od wartości zwróconej przez zadanie.

Listing 7.5. Użycie zadania do asynchronicznego wykonania kodu

```
    static void Main(string[] args)
    {
        //czynność
        Func<object, long> akcja =
            (object argument) =>
            {
                Console.WriteLine("Początek działania akcji - " + argument.ToString());
                System.Threading.Thread.Sleep(500); //opóźnienie 0.5s
                Console.WriteLine("Koniec działania akcji - " + argument.ToString());
                return DateTime.Now.Ticks;
            };

        long wynik = akcja("synchronicznie");
        Console.WriteLine("Synchronicznie: " + wynik.ToString());

        //w osobnym zadaniu
        Task<long> zadanie = new Task<long>(akcja, "zadanie");
        zadanie.Start();
        Console.WriteLine("Akcja została uruchomiona");
        //właściwość Result czeka ze zwróceniem wartości, aż zadanie zostanie zakończone
        //(synchronizacja)
        long wynik = zadanie.Result;
        Console.WriteLine("Zadanie: " + wynik.ToString());
    }
```

Ponadto nie jest konieczne, aby instrukcja odczytania właściwości `Result` znajdowała się w tej samej metodzie co uruchomienie zadania – należy tylko do miejsca jej odczytania przekazać referencję do zadania (w naszym przypadku zmienną typu `Task<long>`). Zwykle referencję tę przekazuje się jako wartość zwracaną przez metodę uruchamiającą zadanie. Zgodne z konwencją metody tworzące i uruchamiające zadania powinny zawierać w nazwie przyrostek `..Async` (por. listing 7.6).

Listing 7.6. Wzór metody wykonującej jakąś czynność asynchronicznie

```
Task<long> ZróbCośAsync(object argument)
{
    //czynność, która będzie wykonywana asynchronicznie
    Func<object, long> akcja =
        (object _argument) =>
        {
            Console.WriteLine("Początek działania akcji - " + _argument.ToString());
            System.Threading.Thread.Sleep(500); //opóźnienie 0.5s
            Console.WriteLine("Koniec działania akcji - " + _argument.ToString());
            return DateTime.Now.Ticks;
        };

    Task<long> zadanie = new Task<long>(akcja, argument);
    zadanie.Start();
    return zadanie;
}

static void Main(string[] args)
{
    Task<long> zadanie2 = ZróbCośAsync("zadanie-metoda");
    Console.WriteLine("Akcja została uruchomiona (metoda)");
    long wynik = zadanie2.Result;
    Console.WriteLine("Zadanie-metoda: " + wynik.ToString());
}
```

Wraz z wersjami 4.0 i 4.5 w platformie .NET (oraz w platformie Windows Runtime) pojawiło się wiele metod, które wykonują długotrwałe czynności asynchronicznie. Znajdziemy je w klasie `HttpClient`, w klasach odpowiedzialnych za obsługę plików (`StorageFile`, `StreamWriter`, `StreamReader`, `XmlReader`), w klasach odpowiedzialnych za kodowanie i dekodowanie obrazów, czy też w klasach WCF. Asynchroniczność jest wręcz standardem w aplikacjach Windows 8 z interfejsem Modern UI (aplikacje *Windows Store*). I właśnie aby ich użycie było (prawie) tak proste jak metod synchronicznych, wprowadzony został w C# 5.0 (co odpowiada platformie .NET 4.5) operator `await`. Ułatwia on synchronizację dodatkowego zadania tworzonego przez te metody. Należy jednak pamiętać, że metodę, w której chcemy użyć operatora `await`, musimy oznaczyć modyfikatorem `async`. A ponieważ modyfikatora takiego nie można dodać do metody wejściowej `Main`, stworzyłem dodatkową, wywoływaną z niej metodę `ProgramowanieAsynchroniczne`. Prezentuje to listing 7.7.

Listing 7.7. Przykład użycia modyfikatora async i modyfikatora await

```
static async void ProgramowanieAsynchroniczne()
{
    Task<long> zadanie2 = ZróbCośAsync("zadanie-metoda");
    Console.WriteLine("Akcja została uruchomiona (metoda)");
    long wynik = await zadanie2;
    Console.WriteLine("Zadanie-metoda: " + wynik.ToString());
}

static void Main(string[] args)
{
    ProgramowanieAsynchroniczne();
    Console.WriteLine();
```

```
        Console.Write("Naciśnij Enter..."); Console.ReadLine();
    }
```

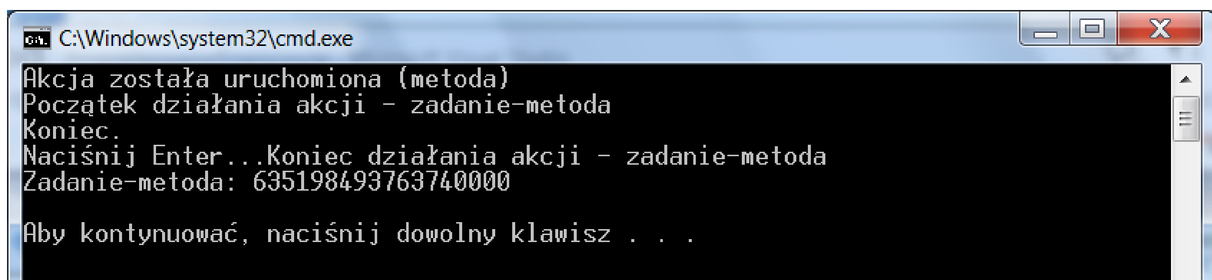
Operator await zwraca parametr typu Task<> (czyli long w przypadku Task<long>) lub void, jeżeli użyta została wersje nieparametryczna klasy Task.

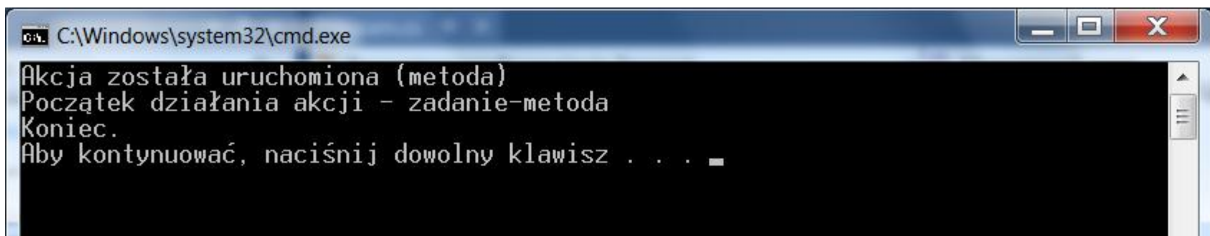
Metody oznaczone modyfikatorem async nazywane są w angielskiej dokumentacji MSDN *async methods*. Może to jednak wprowadzać pewne zamieszanie. Metody z modyfikatorem async (w naszym przypadku metoda ProgramowanieAsynchroniczne) mylone są bowiem z metodami wykonującymi asynchronicznie jakieś czynności (w naszym przypadku ZróbCośAsync). Osobom poznającym dopiero temat często wydaje się, że aby metoda wykonywana była asynchronicznie, wystarczy dodać do jej sygnatury modyfikator async. To nie jest prawda.

Możemy oczywiście wywołać metodę ZróbCośAsync w taki sposób, że umieścimy ją bezpośrednio za operatorem await, np. long wynik = await ZróbCośAsync("async/await");. Czy to ma sens? Wykonywanie metody ProgramowanieAsynchroniczne, w której znajduje się to wywołanie, zostanie wstrzymane aż do momentu zakończenia metody ZróbCośAsync, więc efekt, jaki zobaczymy na ekranie, będzie identyczny jak w przypadku synchronicznym (listing 7.5). Różnica jest jednak zasadnicza, bowiem instrukcja zawierająca operator await nie blokuje wątku, w którym wywołana została metoda ProgramowanieAsynchroniczne. Kompilator zawiesza jej wywołanie, przechodząc do kolejnych czynności aż do momentu zakończenia uruchomionego zadania. W momencie gdy to nastąpi, wątek wraca do metody ProgramowanieAsynchroniczne i kontynuuje jej działanie.

„Zawieszenie działania metody ProgramowaniaAsynchroniczne” to obrazowe, ale mało precyzyjne sformułowanie. Tak naprawdę kompilator tnie tę metodę w miejscu wystąpienia operatora await i tę część, która ma być wykonana po zakończeniu zadania i odebraniu wyniku umieszcza w metodzie zwrotnej (ang. *callback*). Co więcej ta metoda zwrotna wcale nie będzie wykonywana w tym samym wątku, co pierwsza część metody ProgramowanieAsynchroniczne i metoda Main; wykorzystany zostanie wątek, w którym pracowało zadanie tworzone przez metodę ZróbCośAsync.

Z tego wynika, że efekt działania operatora await zobaczymy dopiero, gdy metodę ProgramowanieAsynchroniczne wywołamy z innej metody, w której będą dodatkowe instrukcje wykonywane w czasie wstrzymania metody ProgramowanieAsynchroniczne. W naszym przykładzie wywołujemy ją z metody Main, która po wywołaniu metody ProgramowanieAsynchroniczne wstrzymuje działanie aż do naciśnięcia klawisza *Enter*. W serii instrukcji wywołanie metody oznaczonej modyfikatorem async nie musi się zakończyć przed wykonaniem następnej instrukcji — w tym sensie jest ona asynchroniczna. Aby tak się stało, musi w niej jednak zadziałać operator await, w naszym przykładzie czekający na wykonanie metody ZróbCośAsync. W efekcie, jeżeli w metodzie Main usuniemy ostatnie polecenia wymuszające oczekiwanie na naciśnięcie klawisza *Enter*, metoda ta zakończy się przed zakończeniem ProgramowanieAsynchroniczne, kończąc tym samym działanie całego programu i nie pozwalając metodzie ZróbCośAsync wykonać całego zadania. Jeżeli polecenia te są obecne, instrukcje z metody ZróbCośAsync zostaną wykonane już po wyświetleniu komunikatu o konieczności naciśnięcia klawisza *Enter* wyświetlonego przez metodę Main. Dowodzi tego rysunek 7.1 (por. diagram przy listingu 7.8).

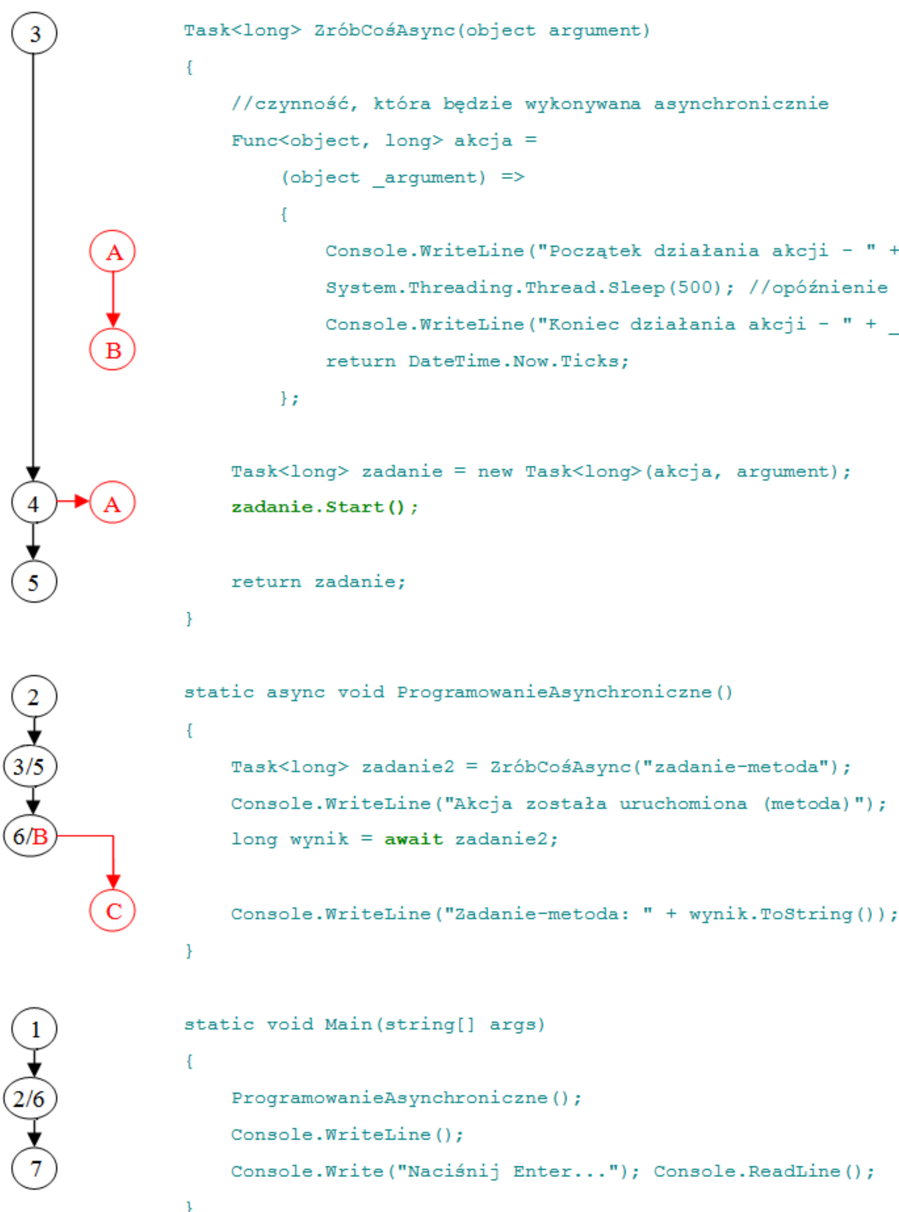




Rysunek 7.1. Zadania utworzone w ten sposób nie blokują wątku głównego – działają jak wątki tła

Diagram na listingu 7.8 obrazuje przebieg programu. Liczby w okręgach oznaczają kolejne ważne punkty programu m.in. te, w których następuje przepływ wątku z metody do metody (dwa okręgi z tą samą liczbą) lub uruchamiane jest zadanie. Lewa kolumna odpowiada wątkowi głównemu (w nim wykonywana jest metoda `Main`), a prawa – wątkowi tworzonemu na potrzeby zadania, a potem wykorzystywanemu do dokończenia metody zawierającej operator `await`. Warto nad tym diagram spędzić chwilę czasu z ołówkiem w ręku starając się zrozumieć jak działa operator `await` i które fragmenty kodu czekają na zakończenie zadania, a które nie.

Listing 8.8. Diagram przebiegu programu



* * *

Zgodnie z zapowiedzią to tylko najbardziej podstawowe wiadomości o podstawowych elementach TPL i programowaniu asynchronicznym w C# 5.0. Więcej informacji na ten temat, w szczególności o bardzo ważnym zagadnieniu synchronizacji wątków, znajdzie Czytelnik we wspomnianej już książce *Programowanie równoległe i asynchroniczne w C# 5.0* (Helion 2013).