Mateusz Warczak, Jacek Matulewski
Rafał Pawłaszek, Piotr Sybilski,
Dawid Borycki, Tomasz Dziubak

Programowanie równoległe i asynchroniczne w C# 5.0

Programowanie współbieżne — wykorzystaj w pełni moc procesorów!
Opanuj wątki, zadania i TPL
Poznaj sprytne rozwiązania z użyciem bibliotek DSS i CCR
Wejdź na wyższy poziom z Reactive Extensions i CUDAfy.NET

Helion

Redaktor prowadzący: Ewelina Burska
Projekt okładki: Studio Gravite/Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/proch5
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-246-6698-0

Printed in Poland.

Spis treści

Wstęp **9**

Przedmowa **11**

Rozdział 1. Dla niecierpliwych: asynchroniczność i pętla równoległa **13**
Programowanie asynchroniczne. Operator await i modyfikator async (nowość języka C# 5.0 i platformy .NET 4.5) 13
Klasa Parallel z biblioteki TPL (nowość platformy .NET 4.0) 19
Równoległa pętla For 20
Przerywanie pętli 22

Rozdział 2. Wątki **25**
Monte Carlo 25
Obliczenia bez użycia dodatkowych wątków 26
Przeniesienie obliczeń do osobnego wątku 28
Wątki, procesy i domeny aplikacji 30
Usypianie bieżącego wątku 31
Przerywanie działania wątku (Abort) 32
Wstrzymywanie i wznawiane działania wątku 34
Wątki działające w tle 35
Zmiana priorytetu wątku 36
Użycie wielu wątków i problemy z generatorem liczb pseudolosowych 36
Pamięć lokalna wątku i bezpieczeństwo wątku 39
Czekanie na ukończenie pracy wątku (Join) 40
Sekcje krytyczne (lock) 43
Przesyłanie danych do wątku 45
Pula wątków 47
Jeszcze raz o sygnalizacji zakończenia pracy wątków 50
Operacje atomowe 51
Tworzenie wątków za pomocą System.Threading.Timer i imitacja timera w wątku z wysokim priorytetem 54
Zadania 57

Rozdział 3. Zmienne w aplikacjach wielowątkowych **59**
Atrybut ThreadStatic 59
Opóźniona inicjacja i zmienne lokalne wątku 60
Volatile 64
Zadania 65

Rozdział 4. Więcej o synchronizacji wątków. Blokady i sygnały 67
Problem ucztujących filozofów 68
Problem czytelników i pisarzy 73
Komunikacja między wątkami. Problem producenta i konsumenta 78
Sygnalizacja za pomocą metod Monitor.Pulse i Monitor.Wait 81
EventWaitHandle i AutoResetEvent 85
Bariera 86
Synchronizacja wątków z różnych procesów. Muteksy i semafory nazwane 88
Kontrola ilości instancji aplikacji 89
Mutex 89
Semafor 91
Zadania 93

Rozdział 5. Wątki a interfejs użytkownika 95
Wątki robocze w aplikacjach desktopowych 95
Przygotowanie projektu aplikacji oraz danych wejściowych 96
Wykorzystanie wątków w długotrwałych metodach zdarzeniowych 99
Synchronizacja wątków z interfejsem użytkownika w aplikacjach Windows Forms ... 104
BackgroundWorker 110
Synchronizacja wątków z komponentami Windows Presentation Foundation 114
Projekt graficznego interfejsu użytkownika 115
Implementacja metod zdarzeniowych 117
Bezpieczny dostęp do kontrolek WPF 125
Kontekst synchronizacji 128
Groźba zagłodzenia wątku interfejsu i asynchroniczna zmiana stanu współdzielonych zasobów 135
Zadania 136

Rozdział 6. Zadania 137
Tworzenie zadania 137
Praca z zadaniami 138
Dane przekazywane do zadań 140
Dane zwracane przez zadania 141
Przykład: test liczby pierwszej 141
Synchronizacja zadań 143
Przykład: sztafeta zadań 144
Przerywanie zadań 145
Stan zadania 149
Fabryka zadań 152
Planista i zarządzanie kolejkowaniem zadań 155
Ustawienia zadań 159
Zadania 160

Rozdział 7. Klasa Parallel. Zrównoleglanie pętli 161
Równoległa pętla for 162
Równoległa pętla foreach 163
Metoda Invoke 164
Ustawienia pętli równoległych. Klasa ParallelOptions 166
Przerywanie pętli za pomocą CancelationToken 166
Kontrola wykonywania pętli 168
Synchronizacja pętli równoległych. Obliczanie π metodą Monte Carlo 169
Partycjonowanie danych 175
Zadania 177

Rozdział 8. Synchronizacja zadań ... 179
Blokady (lock) ... 179
Sygnały (Monitor.Pulse i Monitor.Wait) ... 182
Bariera ... 184

Rozdział 9. Dane w programach równoległych ... 187
Praca ze zbiorami danych w programowaniu równoległym ... 187
Współbieżne struktury danych ... 187
Kolekcja ConcurrentBag ... 189
Współbieżne kolejka i stos ... 189
Praca z BlockingCollection ... 190
Własna kolekcja współbieżna ... 193
Agregacja ... 197
Agregacje dla kolekcji równoległych ... 199
PLINQ — zrównoleglone zapytania LINQ ... 203
Przykład zapytania PLINQ ... 204
Jak działa PLINQ? ... 205
Kiedy PLINQ jest wydajne? ... 207
Metody przekształcające dane wynikowe ... 208
Przerywanie zapytań ... 209
Metoda ForAll ... 212
Zadania ... 213

Rozdział 10. Synchronizacja kontrolek interfejsu z zadaniami ... 215
Zadania w aplikacjach Windows Forms ... 215
Zadania w aplikacjach WPF ... 219
Aktualizacja interfejsu z wykorzystaniem operatora await ... 221
Zadania ... 223

Rozdział 11. Analiza aplikacji wielowątkowych. Debugowanie i profilowanie ... 225
Okno wątków (Threads) ... 226
Okno zadań równoległych (Parallel Tasks) ... 228
Okno stosów równoległych (Parallel Stacks) ... 229
Okno równoległego śledzenia zmiennych (Parallel Watch) ... 230
Concurrency Visualizer ... 232
Widok Wykorzystanie CPU ... 232
Widok Wątki ... 233
Widok Rdzenie ... 236
Profilowanie aplikacji zewnętrznych ... 237
Znaczniki ... 238
Zadania ... 241

Rozdział 12. Wstęp do CCR i DSS ... 243
Instalacja środowiska Microsoft Robotics ... 245
Możliwe problemy z uruchomieniem środowiska Robotics ... 247
Kompilacja i uruchamianie projektów dołączonych do książki ... 248
CCR i DSS w pigułce ... 249
Czujniki i urządzenia — tworzenie pierwszej usługi ... 249
Serwisy partnerskie ... 265

Rozdział 13. Skalowalne rozwiązanie dla systemów rozproszonych na bazie technologii CCR i DSS ... 277
Opóźnione uruchamianie ... 291
Uruchamianie obliczeń na klastrze ... 293
Podsumowanie ... 298
Zadania ... 299

Rozdział 14. Wprowadzenie do Reactive Extensions. Zarządzanie sekwencjami zdarzeń ... 301
Programowanie reaktywne ... 302
IObservable<T> ... 303
IObserver<T> ... 303
Dualizm interaktywno-reaktywny ... 304
Obserwator — wzorzec projektowy ... 305
Platforma Rx ... 306
Biblioteki Rx ... 307
Gramatyka Rx ... 309
Jak korzystać z interfejsów w Rx? ... 309
Subskrypcje ... 312
LINQ do zdarzeń ... 315
Zimne i gorące obserwable ... 329

Rozdział 15. Współbieżność w Rx ... 333
Zarządzanie równoległością ... 333
Interfejs IScheduler ... 334
Planiści ... 335
Metody SubscribeOn i ObserveOn ... 339
Słowo o unifikacji ... 343

Rozdział 16. Przykłady użycia technologii Rx w aplikacjach WPF ... 345
Rysowanie z użyciem Rx ... 346
Wyszukiwarka ... 353

Rozdział 17. CUDA w .NET ... 365
Konfiguracja środowiska dla CUDAfy.NET ... 366
Pierwsze kroki ... 368
Hello World, czyli pierwszy program CUDAfy.NET ... 370
Emulator GPU ... 375
Własności GPU ... 376
Przekazywanie parametrów do kerneli ... 378
Operacje na pamięci globalnej karty graficznej ... 380
Pomiar czasu wykonania ... 383
Dostęp zwarty do pamięci globalnej i pamięć współdzielona ... 386
Generator liczb pseudolosowych ... 390
FFT na GPU ... 392
BLAS ... 394
Zadania ... 395

Dodatek A Biblioteka TPL w WinRT ... 397
Zadania ... 398
Struktura SpinWait ... 400
Usypianie wątków ... 400
Pula wątków ... 401
ThreadPoolTimer ... 402
Podobieństwa ... 403
Przenośna biblioteka ... 404
Zadania ... 406

Dodatek B Dobre praktyki programowania aplikacji wielowątkowych 407
Wprowadzenie 407
Sekcje krytyczne i zakleszczenia 407
Wyścig 411
Słowo kluczowe volatile i kontrola pętli wykonywanej w ramach funkcji wątku 417
Bezpieczeństwo wątków a konstruktory i pola statyczne 419

Dodatek C Menadżer pakietów NuGet 423
Instalacja NuGet 423
Korzystanie z NuGet 425

Skorowidz 427

Rozdział 1.

Dla niecierpliwych: asynchroniczność i pętla równoległa

Jacek Matulewski

Zgodnie z zasadą Pareto, w większości przypadków czytelnicy będą potrzebowali tylko znikomej części wiedzy przedstawionej w tej książce. Postanowiłem wobec tego w rozdziale 1. opisać dwie nowości platformy .NET 4.0 i 4.5, które wydają mi się najważniejsze i które prawdopodobnie będą najczęściej używane w programach czytelników.

Programowanie asynchroniczne. Operator await i modyfikator async (nowość języka C# 5.0 i platformy .NET 4.5)

Język C# 5.0 wyposażony został w nowy operator `await`, ułatwiający synchronizację dodatkowych zadań uruchomionych przez użytkownika. Poniżej zaprezentuję prosty przykład jego użycia, który powinien wyjaśnić jego działanie. Działanie tego operatora związane jest ściśle z biblioteką TPL (ang. *Task Parallel Library*) i jej sztandarową klasą `Task`, które zostaną omówione w kolejnych rozdziałach. Jednak podobnie jak w przypadku opisanej niżej pętli równoległej `Parallel.For`, tak i w przypadku operatora `await` dogłębna znajomość biblioteki TPL nie jest konieczna.

Spójrzmy na przykład widoczny na listingu 1.1, w którym przedstawiam metodę zdarzeniową przycisku. Zdefiniowana jest w niej przykładowa akcja pobierająca obiekt typu `object`, a zwracająca liczbę całkowitą `long`. Referencję do niej zapisuję w zmiennej `akcja` i uruchamiam ją (synchronicznie). Czynność owa wprowadza jednosekundowe opóźnienie za pomocą metody `Thread.Sleep` (należy zadeklarować użycie przestrzeni nazw `System.Threading`[1]), które — oczywiście — opóźnia wykonywanie całej metody zdarzeniowej po kliknięciu przycisku. W efekcie na jedną sekundę aplikacja zamiera.

Listing 1.1. *Synchroniczne wykonywanie kodu zawartego w akcji*

```
private void button1_Click(object sender, EventArgs e)
{
    Func<object, long> akcja =
        (object argument) =>
        {
            msgBox("Akcja: Początek, argument: " + argument.ToString());
            Thread.Sleep(1000); //opóźnienie
            msgBox("Akcja: Koniec");
            return DateTime.Now.Ticks;
        };

    msgBox("button1_Click: Początek");
    msgBox("Wynik: "+akcja("synchronicznie"));
    msgBox("button1_Click: Koniec");
}

void msgBox(string komunikat)
{
    string taskID = Task.CurrentId.HasValue ? Task.CurrentId.ToString() : "UI";
    MessageBox.Show("! " + komunikat + " (" + taskID + ")");
}
```

W metodzie przedstawionej na listingu 1.2 ta sama akcja wykonywana jest asynchronicznie w osobnym wątku utworzonym przez platformę .NET na potrzeby zdefiniowanego tu zadania (instancja klasy `Task` z TPL). Synchronizacja następuje w momencie odczytania wartości `zadanie.Result`, czyli wartości zwracanej przez czynność `akcja`. Jej sekcja `get` czeka ze zwróceniem wartości aż do zakończenia akcji wykonywanej przez zadanie, wstrzymując do tego czasu wątek, w którym wykonywana jest metoda `button1_Click`. Jest to zatem typowy punkt synchronizacji, choć trochę ukryty. Warto zwrócić uwagę, że po instrukcji `zadanie.Start()`, a przed odczytaniem własności `zadanie.Result` mogą być wykonywane dowolne czynności, o ile są niezależne od wartości zwróconej przez zadanie.

Listing 1.2. *Użycie zadania do asynchronicznego wykonania kodu*

```
private void button1_Click(object sender, EventArgs e)
{
    Func<object, long> akcja =
        (object argument) =>
```

[1] Alternatywnie moglibyśmy użyć instrukcji `await Task.Delay(1000);`, ale wówczas musielibyśmy oznaczyć wyrażenie lambda jako `async`, a wtedy należałoby referencję do niego zapisać w zmiennej typu `Func<object, Task<long>>`.

```
        {
            msgBox("Akcja: Początek, argument: " + argument.ToString());
            Thread.Sleep(1000); //opóźnienie
            msgBox("Akcja: Koniec");
            return DateTime.Now.Ticks;
        };

    Task<long> zadanie = new Task<long>(akcja, "zadanie");
    zadanie.Start();
    msgBox("Akcja została uruchomiona");
    if (zadanie.Status != TaskStatus.Running &&
        zadanie.Status!=TaskStatus.RanToCompletion)
            msgBox("Zadanie nie zostało uruchomione");
    else msgBox("Wynik: "+zadanie.Result);
    msgBox("button1_Click: Koniec");
}
```

Nie jest konieczne, aby instrukcja odczytania własności `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. Przykład takiej metody widoczny jest na listingu 1.3. Jeżeli używamy angielskich nazw metod, jest zwyczajem, aby metoda tworząca i uruchamiająca zadanie miały przyrostek `..Async`.

Listing 1.3. *Wzór metody wykonującej jakąś czynność asynchronicznie*

```
Task<long> DoSomethingAsync(object argument)
{
    Func<object, long> akcja =
        (object _argument) =>
        {
            msgBox("Akcja: Początek, argument: " + _argument.ToString());
            Thread.Sleep(1000); //opóźnienie
            msgBox("Akcja: Koniec");
            return DateTime.Now.Ticks;
        };

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

protected void button1_Click(object sender, EventArgs e)
{
    msgBox("button1_Click: Początek");
    Task<long> zadanie = DoSomethingAsync("zadanie-metoda");
    msgBox("Akcja została uruchomiona");
    if (zadanie.Status != TaskStatus.Running &&
        zadanie.Status!=TaskStatus.RanToCompletion)
            msgBox("Zadanie nie zostało uruchomione");
    else msgBox("Wynik: " + zadanie.Result);
    msgBox("button1_Click: Koniec");
}
```

Po tym wprowadzeniu możemy przejść do omówienia zasadniczego tematu. Wraz z wersjami 4.0 i 4.5 w platformie .NET (oraz w platformie Windows Runtime) pojawiło się wiele metod podobnych do przedstawionej powyżej metody `DoSomethingAsync` (ale — oczywiście — w odróżnieniu od niej robiących coś pożytecznego). Metody te wykonują asynchronicznie różnego typu długotrwałe czynności. Znajdziemy je w klasie `HttpClient`, w klasach odpowiedzialnych za obsługę plików (`StorageFile`, `StremWriter`, `Stream` ↪`Reader`, `XmlReader`), w klasach odpowiedzialnych za kodowanie i dekodowanie obrazów czy w klasach WCF. Asynchroniczność jest wręcz standardem w aplikacjach Windows 8 z interfejsem Modern UI. I właśnie po to, 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`. Prezentuję to na listingu 1.4.

Listing 1.4. *Przykład użycia modyfikatora async i modyfikatora await*

```
protected async void button1_Click(object sender, EventArgs e)
{
    msgBox("button1_Click: Początek");
    Task<long> zadanie = DoSomethingAsync("async/await");
    msgBox("Akcja została uruchomiona");
    long wynik = await zadanie;
    msgBox("Wynik: " + wynik);
    msgBox("button1_Click: Koniec");
}
```

Operator `await` zwraca parametr użyty w klasie parametrycznej `Task<>`. Zatem w przypadku zadania typu `Task<long>` będzie to zmienna typu `long`. Jeżeli użyta została wersja nieparametryczna klasy `Task`, operator zwraca `void` i służy jedynie do synchronizacji (nie przekazuje wyniku; nieparametryczna klasa `Task` nie ma także własności `Result`).

Metody oznaczone modyfikatorem `async` nazywane są w angielskiej dokumentacji MSDN *async method*. Może to jednak wprowadzać pewne zamieszanie. Z powodu tej nazwy metody z modyfikatorem `async` (w naszym przypadku metoda `Button1_Click`) utożsamiane są z metodami wykonującymi asynchronicznie jakieś czynności (a taką w naszym przypadku jest `DoSomethingAsync`). 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 wywołać metodę `DoSomethingAsync` w taki sposób, że umieścimy ją bezpośrednio za operatorem `await`, np. `long wynik = await DoSomethingAsync("async/` ↪`await");`. Jaki to ma sens? Wykonywanie metody `button1_Click`, w której znajduje się to wywołanie, zostanie wstrzymane aż do momentu zakończenia metody `DoSomething` ↪`Async`, więc efekt, jaki zobaczymy na ekranie, będzie identyczny z wynikiem w przypadku synchronicznym (listing 1.1). Różnica jest jednak wyraźna i to jest zasadnicza nowość, bo instrukcja zawierająca operator `await` nie blokuje wątku, w którym wywołana została metoda `button1_Click`. Kompilator zawiesza wywołanie metody `button1_Click`, przechodząc do kolejnych czynności w miejscu jej wywołania aż do momentu zakończenia uruchomionego zadania. W momencie, gdy to nastąpi, wątek wraca do metody

button1_Click i kontynuuje jej działanie[2]. Jednak w programie, na którym w tej chwili testujemy operator await, efektów tego nie zobaczymy. Efekt będzie widoczny dopiero wtedy, gdy metodę button1_Click wywołamy z innej metody — niech będzie to metoda zdarzeniowa button2_Click związana z drugim przyciskiem. Należy zauważyć, że w serii instrukcji wywołanie metody oznaczonej modyfikatorem async nie musi się zakończyć przed wykonaniem następnej instrukcji — i w tym sensie jest ona asynchroniczna. Aby tak się stało, musi w niej jednak zadziałać operator await czekający na wykonanie jakiegoś zadania (w naszym przykładzie metody DoSomethingAsync). W efekcie, w scenariuszu przedstawionym na listingu 1.5 metoda button2_Click zakończy się przed zakończeniem button1_Click.

Listing 1.5. *Działanie modyfikatora async*

```
private async void button1_Click(object sender, EventArgs e)
{
    msgBox("button1_Click: Początek");
    long wynik = await DoSomethingAsync("async/await");
    msgBox("Wynik: " + wynik.ToString());
    msgBox("button1_Click: Koniec");
}

private void button2_Click(object sender, EventArgs e)
{
    msgBox("button2_Click: Początek");
    button1_Click(null, null);
    msgBox("button2_Click: Koniec");
}
```

Ważna rzecz: samo użycie operatora await i modyfikatora async nie powoduje utworzenia nowych zadań lub wątków! Powoduje jedynie przekazanie na pewien czas sterowania z metody, w której znajduje się operator await i oznaczonej modyfikatorem async, do metody, która ją wywołała, i powrót w momencie ukończenia zadania, na jakie czeka await. Koszt jest zatem niewielki i rozwiązanie to może być z powodzeniem stosowane bez obawy o utratę wydajności. Ponadto, właśnie z uwagi na wydajność, operator await sprawdza, czy w momencie, w którym dociera do niego sterowanie, metoda asynchroniczna nie jest już zakończona. Jeżeli tak, praca kontynuowana jest synchronicznie bez zbędnych skoków.

Metoda z modyfikatorem async może zwracać wartość void — tak jak w przedstawionej wyżej metodzie zdarzeniowej button1_Click. Jednak w takim przypadku jej działanie nie może być żaden sposób synchronizowane. Po uruchomieniu nie mamy nad nią żadnej kontroli. Szczególnie nie można użyć operatora await ani metody Wait klasy Task, aby poczekać na jej zakończenie. Żeby to było możliwe, metoda z modyfikatorem async musi zwracać referencję Task lub Task<>. Wówczas możliwe jest użycie operatora await, za którym można zresztą ustawić dowolne wyrażenie o wartości Task

[2] Aby taki efekt uzyskać bez operatora await, należałoby użyć konstrukcji opartej na funkcjach zwrotnych (ang. *callback*). W efekcie kod stałby się raczej skomplikowany i przez to podatny na błędy. Warto też zauważyć, że await nie jest prostym odpowiednikiem metody Task.Wait, która po prostu zatrzymałaby bieżący wątek do momentu zakończenia zadania. W przypadku operatora await nastąpi przekazanie sterowania do metody wywołującej i powrót w momencie zakończenia zadania.

lub Task<> (zmienne i własności tego typu oraz metody lub wyrażenia lambda zwracające wartość tego typu[3]). Przekazane zadanie umożliwia synchronizację. Ponadto użycie wersji parametrycznej pozwala na zwrócenie wartości przekazywanej potem przez operator await.

Sprawdźmy to, tworząc odpowiednik metody button1_Click ze zmienioną sygnaturą (nie możemy tego zrobić z oryginałem, bo jest związany ze zdarzeniem button1.Click). Nowa metoda o nazwie DoSomethingMoreAsync widoczna jest na listingu 1.6[4]. Usunąłem argumenty, których i tak nie używaliśmy, i zmieniłem zwracaną wartość z void na Task. Dzięki temu metoda ta nie jest już typu „wystrzel i zapomnij", a może być kontrolowana z miejsca uruchomienia (zob. widoczna również na listingu 1.6 metoda button2_Click). Zdziwienie może budzić jednak fakt, że za słowem kluczowym return w metodzie DoSomethingMoreAsync wcale nie ma instrukcji tworzącej zwracane przez tą metodę zadanie (instrukcji return mogłoby wcale nie być). W metodach z modyfikatorem async i zwracających wartość Task zadanie jest przypisywane przez kompilator. W ten sposób ułatwiona jest wielostopniowa obsługa metod asynchronicznych. Należy jednak pamiętać, że te metody nie tworzą nowych zadań, a jedynie je przekazują.

Listing 1.6. *Metoda async zwracająca zadanie*

```
private async Task DoSomethingMoreAsync()
{
    msgBox("DoSomethingMoreAsync: Początek");
    long wynik = await DoSomethingAsync("async/await");
    msgBox("DoSomethingMoreAsync: Wynik: " + wynik.ToString());
    msgBox("DoSomethingMoreAsync: Koniec");
    return;
}

private async void button2_Click(object sender, EventArgs e)
{
    msgBox("button2_Click: Początek");
    await DoSomethingMoreAsync();
    msgBox("button2_Click: Koniec");
}
```

A co w przypadku metod async, które miałyby zwracać wartość? Załóżmy, że metoda DoSomethingMore miałaby zwracać wartość typu long (np. wartość zmiennej wynik). Wtedy należy zmienić typ tej metody na Task<long>, a za słowem kluczowym return wstawić wartość typu long. Pokazuję to na listingu 1.7. Warto zapamiętać, choć to uproszczone stwierdzenie, że w metodach async operator await wyłuskuje z typu Task<> parametr, a słowo kluczowe return w metodach async zwracające wartość typu Task<> działa odwrotnie — otacza dowolne obiekty typem Task<>.

[3] Prawdę mówiąc, należałoby to stwierdzenie uściślić, bo nie tylko zadania mogą być argumentem operatora await, a każdy typ, który zwraca metodę GetAwaiter. Więcej informacji dostępnych jest na stronie FAQ zespołu odpowiedzialnego za implementację mechanizmu async/await w platformie .NET (*http://blogs.msdn.com/b/pfxteam/archive/2012/04/12/10293335.aspx*).

[4] Warto zwrócić uwagę na przyrostek „Async". W końcu jest to teraz metoda, która działa asynchronicznie, choć żadnego zadania nie tworzy.

Listing 1.7. *Metoda async zwracająca wartość long*

```
private async Task<long> DoSomethingMoreAsync()
{
    msgBox("DoSomethingMoreAsync: Początek");
    long wynik = await DoSomethingAsync("async/await");
    msgBox("DoSomethingMoreAsync: Wynik: " + wynik.ToString());
    msgBox("DoSomethingMoreAsync: Koniec");
    return wynik;
}

private async void button2_Click(object sender, EventArgs e)
{
    msgBox("button2_Click: Początek");
    msgBox("button2_Click: Wynik: " + await DoSomethingMoreAsync());
    msgBox("button2_Click: Koniec");
}
```

I kolejna sprawa. Co w metodach `async` dzieje się w przypadku błędów? Nieobsłużone wyjątki zgłoszone w metodzie z modyfikatorem `async` i zwracające zadania (`Task` lub `Task<>`) są za pośrednictwem tych zadań przekazywane do metody wywołującej. Można zatem użyć normalnej konstrukcji `try..catch`, jak na listingu 1.8. Gorzej jest w przypadku metod `async` zwracających `void` (typu „wystrzel i zapomnij”, jak `button1_Click` z naszego przykładu). Wówczas wyjątek przekazywany jest do puli wątków kryjącej się za mechanizmem zadań i przechwytywanie wyjątków nic nie da.

Listing 1.8. *Obsługa wyjątków zgłaszanych przez metody async*

```
private async void button2_Click(object sender, EventArgs e)
{
    msgBox("button2_Click: Początek");
    try
    {
        msgBox("button2_Click: Wynik: " + await DoSomethingMoreAsync());
    }
    catch(Exception exc)
    {
        msgBox("button2_Click: Błąd!\n" + exc.Message);
    }
    msgBox("button2_Click: Koniec");
}
```

Klasa Parallel z biblioteki TPL (nowość platformy .NET 4.0)

Do platformy .NET w wersji 4.0 dodana została biblioteka TPL (ang. *Task Parallel Library*), która wraz ze zrównoleglonym PLINQ i kolekcjami przystosowanymi do konkurencyjnej obsługi składa się na tzw. *Parallel Extensions*. Biblioteka TPL nadbudowuje klasyczne wątki, korzystając z poznanej już przed chwilą klasy `Task` (z ang.

zadanie). Biblioteka ta zostanie dokładnie opisana w następnych rozdziałach. Tu chciałbym skupić się tylko na najczęściej używanym jej elemencie — implementacji współbieżnej pętli `For`.

Równoległa pętla For

Załóżmy, że mamy zbiór stu liczb rzeczywistych, dla których musimy wykonać jakieś stosunkowo czasochłonne czynności. W naszym przykładzie będzie to obliczanie wartości funkcji $f(x) = arcsin(sin(x))$. Funkcja ta powinna z dokładnością numeryczną zwrócić wartość argumentu x. Zrobi to, ale nieźle się przy tym namęczy — funkcje trygonometryczne są dość wymagające numerycznie. Dodatkowo powtórzymy te obliczenia kilkakrotnie, aby jeszcze bardziej wydłużyć czas obliczeń. Kod odpowiedniej metody z projektu aplikacji konsolowej widoczny jest na listingu 1.9.

Listing 1.9. *Metoda zajmująca procesor*

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

Z kolei na listingu 1.10 widoczna jest pętla wykonująca owe obliczenia wraz z przygotowaniem tablicy z wynikami. Wyniki te nie są jednak drukowane — tablica jest zbyt duża, żeby to miało sens. Poniższy kod zawiera dwie zagnieżdżone pętle `For`. Interesuje nas tylko wewnętrzna. Zadaniem zewnętrznej jest wielokrotne powtórzenie obliczeń, co pozwoli nam bardziej wiarygodnie zmierzyć czas obliczeń. Pomiary te realizujemy na bazie zliczania taktów procesora (`System.Environment.TickCount`).

Listing 1.10. *Obliczenia sekwencyjne*

```
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 iloscPowtorzen = 100;
    double[] wyniki = new double[tablica.Length];
    int start = System.Environment.TickCount;
    for(int powtorzenia = 0; powtorzenia<iloscPowtorzen;++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
```

```
        string s = "Wyniki:\n";
        for(int i=0;i<tablica.Length;++i)
            s += i + ". " + tablica[i] + " ?= " + wyniki[i] + "\n";
                Console.WriteLine(s);
        */
    }
```

Przy użyciu klasy `Parallel` z przestrzeni nazw `System.Threading.Tasks` można bez większego wysiłku zrównoleglić pętlę `for` z metody `Main` (tę z indeksem `i`). Pokazuje to kod z listingu 1.11. Należy go dodać do metody z listingu 1.10.

Listing 1.11. *Przykład zrównoleglonej pętli for*

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

Metoda `Parallel.For` jest dość intuicyjna w użyciu. Jej dwa pierwsze argumenty określają zakres zmiany indeksu pętli. W naszym przypadku jest on równy [0,1000). Wobec tego do metody podanej w trzecim argumencie przekazywane są liczby od 0 do 999. Trzeci argument jest delegatem, do którego można przypisać metodę lub, jak w naszym przypadku, wyrażenie lambda wywoływane w każdej iteracji pętli. Powinna się tam zatem znaleźć zawartość oryginalnej pętli.

Metoda `Parallel.For` automatycznie synchronizuje używane przez nią zadania przed zakończeniem, dlatego nie ma zagrożenia zamazania danych w ramach kolejnych powtórzeń (zewnętrzna pętla `for`).

To, że tworzenie równoległej pętli `Parallel.For` jest, jak to mówią Anglicy, *out of the box*, nie oznacza, że automatycznie unikamy wszystkich problemów, jakie w równoległych pętlach mogą powstać. Szczególnie należy zwrócić uwagę na sprawę podstawową: między iteracjami pętli nie może być rekurencyjnej zależności, a więc kolejna iteracja nie może zależeć od wartości jakieś zmiennej policzonej w poprzedniej iteracji. Iteracje w równoległej pętli nie są przecież wykonywane w kolejności indeksów. Należy także uważać na ukryte zależności rekurencyjne. Przykładem, w którym kryją się takie zależności, jest choćby klasa `Random`.

Nie należy się spodziewać, że dzięki użyciu równoległej pętli 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. W moich testach na komputerze z jednym procesorem dwurdzeniowym czas obliczeń zmniejszył się do mniej więcej ⅔ czasu obliczeń sekwencyjnych. Z kolei przy aż ośmiu rdzeniach czas obliczeń równoległych spadł tylko do nieco ponad ⅓.

Przedstawione w tym rozdziale informacje o klasie `Parallel` i jej metodzie `For` należy traktować jedynie jako zapowiedź rozdziału 7., w którym klasa ta zostanie omówiona bardziej wyczerpująco.

Przerywanie pętli

Podobnie jak w klasycznej pętli `for`, również w jej równoległej wersji możemy w każdej chwili przerwać działanie pętli. Służy do tego klasa `ParallelLoopState`, która może być przekazana w dodatkowym argumencie metody wykonywanej w każdej iteracji. Klasa ta udostępnia dwie ważne metody: `Break` i `Stop`. Różnią się one tym, że pierwsza pozwala na wcześniejsze zakończenie bieżącej iteracji, a następne nie będą już uruchamiane, podczas gdy metoda `Stop` nie tylko natychmiast kończy bieżące zadanie, ale również podnosi flagę `IsStopped`, która może być sprawdzona we wszystkich uruchomionych wcześniej iteracjach, co powinno być dla nich sygnałem do zakończenia działania (jeżeli programista uwzględni to w ich kodzie). Na listingu 1.12 pokazuję przykład, w którym pętla jest przerywana, jeżeli wylosowana zostanie liczba 0.

Listing 1.12. *Przerywanie pętli równoległej*

```
private static void przerywaniePetli()
{
    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 większa od 90
    Parallel.For(
        0,
        10000,
        (int i, ParallelLoopState stanPetli) =>
            {
                int liczba = r.Next(7); //losowanie liczby oczek na kostce
                if(liczba == 0)
                {
                    s += "0 (Stop);";
                    stanPetli.Stop();
                }
                if(stanPetli.IsStopped) return;
                if(liczba % 2 == 0)
                {
                    s += liczba.ToString() + "; ";
                    obliczenia(liczba);
                    suma += liczba;
                    licznik += 1;
                }
                else
                {
                    s += liczba.ToString() + "; ";
                }
            });
```

```
        Console.WriteLine(
            "Wylosowane liczby: " + s + "\n" +
            "Liczba pasujących liczb: " + licznik + "\n" +
            "Suma: " + suma + "\n" +
            "Średnia: " + (suma / (double)licznik).ToString());
    }
```

Skorowidz

A

ActiveX, 124
adres
  http, 256
  URL, 256
agregacja kolekcji równoległych, 199
Albahari Joe, 64
algorytm
  braci Borwein, 47
  spigot, 47
Apartment Threaded Model, *Patrz:* ATM
aplikacja
  desktopowa, 95, 124, 215, 251
  domena, *Patrz:* domena aplikacji
  GitHub, 302
  instancja, 89
  kliencka, 302
  konsolowa, 26, 28, 80, 215, 237, 307, 339, 397
  przebieg pracy, 419
  równoległa, 225
    profiler, 225, 232
  rysująca, 350
  sieciowa, 215
  webowa, 237
  wielowątkowa, 40, 187, 189, 407, 411
  Windows Forms, 96, 104, 105
  Windows Store, 397
  WinRT, 397
  WPF, 116, 219
  z interfejsem graficznym, 397
async method, 16
ATI Stream, 365
ATM, 124

B

BackgroundWorker, 110, 114
bariera, 86, 104, 184
Bart de Smet, 346
Base Class Library, *Patrz:* BCL
Basic Linear Algebra Subprograms, *Patrz:* BLAS
bazą danych SQL, 212
BCL, 303
bezpieczeństwo, 40, 77, 104, 124, 419
biblioteka
  Bing Search API, 355
  BLAS, *Patrz:* BLAS
  CCR, *Patrz:* CCR
  cuBLAS, 394
  CUDAfy.Net, 376
  CUDAfy.NET, 366
  Cudafy.NET.dll, 371
  cuFFT, 392
  cuRAND, 390, 392
  DLL, 404
  DSS, *Patrz:* DSS
  Kinect for Windows, 246
  klas podstawowa, *Patrz:* BCL
  kontrolek WPF, 345
  licencja, 423
  Microsoft Silverlight, 246
  Portable Class Library, 307
  ReactiveCocoa, *Patrz:* ReactiveCocoa
  Rx, *Patrz:* Rx
  System.Data.Services.Client.dll, 357
  TPL, *Patrz:* TPL
  Windows Forms, 96
Bing, 345, 353, 357

Bing Search API, 353
BLAS, 394
blokada, 77, 179, 181, 188, 191
  wirująca, 45
broadcast, *Patrz:* rozgłaszanie

C

C for CUDA, 365
callback function, *Patrz:* funkcja odpowiedzi
CCR, 243, 244, 249, 276, 277, 291, 298
CLR, 31
cold observable, *Patrz:* obserwabla zimna
COM, 124
Common Language Runtime, *Patrz:* CLR
Component Object Model, *Patrz:* COM
compute capability, *Patrz:* karta graficzna potencjał obliczeniowy
Compute Unified Device Architecture, *Patrz:* CUDA
Concurrency and Coordination Runtime, *Patrz:* CCR
Concurrency Visualizer, 225, 232, 237
Console Application, *Patrz:* aplikacja konsolowa
CUDA, 365
CUDAfy.NET, 366, 368, 376
czas
  obliczeń, 20, 383
  wirtualny, 334, 335

D

dane
  metody przekształcające, 208
  partycjonowanie, 175
  podział, 205, 213
  przekazywane do zadania, 140
  przesyłanie do wątku, 45
  spychane, 303
  SQL, 212
  struktura współbieżna, 187, 188
  w programowaniu równoległym, 187, 188
  współdzielone przez wątki, 40, 187
  wyciąganie, 302
  zwracane przez zadanie, 141
DCOM, 124
deadlock, *Patrz:* zakleszczenie
debugowanie, 93, 148, 225, 227
Decentralized Software Services, *Patrz:* DSS
dekompilator ILSpy, *Patrz:* ILSpy
delegat, 108, 110, 138
diagram koralikowy, 315, 316, 320
dokumentacja MSDN, 16, 33, 149, 155, 205, 399
domena aplikacji, 31
DSS, 243, 244, 249, 276, 277, 291, 298
  konsola Command Prompt, 293

E

edytor XAML, *Patrz:* XAML
Euler Leonhard, 47
extension method, *Patrz:* metoda rozszerzająca

F

factory method, *Patrz:* metoda tworząca
FFT, 392
FIFO, *Patrz:* kolejka FIFO
flaga, 34, 239, 302
  IsStopped, 22
Fouriera transformata szybka, *Patrz:* FFT
funkcja
  odpowiedzi, 302
  WinAPI InterlockedAdd, 52

G

General-Purpose computing on Graphics Processor Units, *Patrz:* GPGPU
generator liczb
  losowych, 36, 37
  pseudolosowych, 59, 390, 392
GPGPU, 365
GPU, 365, 372
  emulator, 375
Graphical User Interface, *Patrz:* interfejs użytkownika
GUI, *Patrz:* interfejs:użytkownika

H

Hadamarda iloczyn, 378
hot observable, *Patrz:* obserwabla gorąca
Hybrid DSP, 366

I

identyfikator
  kontraktu, 245
  sekcji krytycznej, 44
  usługi, 245

iloczyn
Hadamarda, 378
po współrzędnych, 378
Schura, 378
ILSpy, 368, 369
inicjacja
leniwa, *Patrz:* inicjacja z opóźnieniem
z opóźnieniem, 60, 61, 62, 63
instancja
aplikacji, *Patrz:* aplikacja instancja
programu, 31, *Patrz też:* wątek
interfejs
graficzny, 397
ICollection, 304
IEnumerable, 156, 188, 203, 304
IEnumerator, 304
implementacja, 309
IObservable, 303, 304, 305, 309, 315, 316, 334, 346
IObserver, 303, 305, 309, 312, 315, 334
IProducerConsumerCollection, 78, 155, 187, 188, 191, 193, 195
IScheduler, 334
stron internetowych, 251
użytkownika, 95, 96
aktualizacja, 110
wątek, *Patrz:* wątek interfejsu użytkownika

J

jądro, *Patrz:* kernel
język XAML, *Patrz:* XAML

K

karta graficzna, 365, 371, 376, 390
czas obliczeń, 383
pamięć, 380, 381, 387
potencjał obliczeniowy, 368, 370, 375
uchwyt, 372
kernel, 366, 372, 373, 386
wywołanie, 373, 374
Kinect, 246
klasa
AutoResetEvent, 85, 184
BackgroundWorker, 111
Barrier, 86, 88
BlockingCollection, 78, 190, 191, 192
CancellationToken, 145, 154, 166, 209
CancellationTokenSource, 209
ConcurrentBag, 188, 189
ConcurrentDictionary, 188
ConcurrentQueue, 188, 190
ConcurrentStack, 188, 190
CountdownEvent, 51
CudafyHost, 372, 376
CudafyModule, 372
CudafyTranslator, 369, 373
Dictionary, 337
Dispatcher, 345
DispatcherTimer, 402
EnlightenmentProvider, 343
Enumerable, 204
EventWaitHandle, 85, 184
FFTPlan1D, 392
GPGPU, 372, 384
GPGPUProperties, 376
GThread, 378, 388
HttpClient, 16
instancja, 45
Interlocked, 64, 188, 412
Lazy, 60
leniwa, 61
List, 399
ManualResetEvent, 85, 184
ManualResetEventSlim, 184
Monitor, 44, 50, 413
Mutex, 88, 89
Observable, 316, 339
odpowiedzialna za obsługę plików, 16
Parallel, 22, 138, 161, 403
ParallelEnumerable, 199, 203
ParallelLoopResult, 168
ParallelLoopState, 22, 168
ParallelOptions, 166
ParallelQuery, 199
Partitioner, 175, 187
Queue, 189
Random, 21
ReaderWriterLock, 73
ReaderWriterLockSlim, 73, 77
SemaphorSlim, 93
SpinLock, 45
Stack, 189
statyczna, 49, 316
StorageFile, 16
StreamReader, 16
StremWriter, 16
SynchronizationContext, 128, 334, 345

klasa
System.Threading.Interlocked, 52, 53
System.Threading.LazyInitializer, 63
System.Threading.Timer, 54
Task, 13, 19, 138, 144, 398, 399, 400
TaskContinuationOptions, 154
TaskCreationOptions, 154
TaskFactory, 138, 144, 152, 153
TaskScheduler, 138, 154, 155
Thread, 29, 334, 399, 400
ThreadPool, 48, 343, 401
ThreadPoolTimer, 402
Timer, 402
WCF, 16
WindowsFormsSynchronizationContext, 130
XmlReader, 16
klaster obliczeniowy, 277
kod XAML, 116
kolejka
FIFO, 155, 156, 189, 191
wiadomości, 244
współbieżna, 189
kolekcja, 189, 191
równoległa, 199
współbieżna, 189, 193
własna, 193, 195
kompilator, 18
C#, 64
JIT, 64
komponent wizualny, 124
komunikat, 239, 251
konsola
DSS Command Prompt, 293
Xbox 360, 404
kontrolka, 103, 108, 110, 215, 339
BackgroundWorker, 25
Timer, 25
WPF, 345
kursor myszy, 346, 348

L

Language INtegrated Query, *Patrz:* LINQ
Lego Mindstorms, 243, 249
liczba
losowa, 37
pierwsza, 141, 215
π, 25, 47, 115, 170, 277
LIFO, *Patrz:* stos
linia obrazu, 95, 99
LINQ, 203, 205, 209, 212, 301, 315
do zdarzeń, 302, 306, 315, 316
lock, *Patrz:* blokada
log, 419

M

macierz, 378, 379, 380, 388, 394
manifest, 251
Manifest Load Results, *Patrz:* manifest
marble diagram, *Patrz:* diagram koralikowy
marmurki, 315
maszyna wirtualna, 31
MATLAB, 365
metoda
Add, 191
Aggregate, 199
AllocateShared, 388
AsOrdered, 208, 209
AsParallel, 138, 199, 209
AsSequential, 209
AsUnordered, 208, 209
async, 18
błędy, 19
zwracająca wartość, 18
asynchroniczna, 131
BackgroundWorker.CancelAsync, 110
BackgroundWorker.DoWork, 110, 114
BackgroundWorker.ProgressChanged, 110
BackgroundWorker.RunWorkerAsync, 110
BackgroundWorker.RunWorkerCompleted, 110
blokująca, 131, 191, 357
Break, 22
Buffer, 324, 326, 327
Cancel, 210
CancellationToken.ThrowIfCancellationRequested, 146, 147
CancellationTokenSource.Cancel, 145, 168
CombineLatest, 323
Console.WriteLine, 382
ContinueWhenAny, 144
Control.BeginInvoke, 107, 108, 131
Control.Dispatcher.BeginInvoke, 128, 131
Control.Dispatcher.Invoke, 128
Control.EndInvoke, 131
Control.Invoke, 104, 107, 108, 124, 130, 131
CountDownEvent, 188
Create, 392
Cudafy, 369

Delay, 400
DropHandler, 244
EnsureInitialized, 63
Eulera, 47
ForEach, 138, 399
FromCurrentSynchronizationContext, 221
GetConsumingEnumerable, 192
GetDevice, 372
GetDeviceProperties, 376
GetEnumerator, 305
Interlocked.Add, 52, 53
Interlocked.Increment, 412
Leave, 240
LoadModule, 372
Log, 274
LogError, 274
LogInfo, 274
LogVerbose, 274
LogWarning, 274
MessageBox.Show, 26
Monitor.Enter, 44, 45
Monitor.Exit, 44, 45, 410
Monitor.Pulse, 50, 81, 84, 86, 182
Monitor.Wait, 81, 84, 86, 182
Monitor.WaitOne, 50
Monte Carlo, 25, 47, 115, 170
MoveNext, 305
nieblokująca, 263
Observable.Create, 310
Observable.FromAsyncPattern, 357
Observable.FromEventPattern, 348
Observable.Generate, 310
Observable.Interval, 317
Observable.Range, 309, 323
Observable.Timer, 319
Observable.Timestamp, 318
ObservableRange, 339
ObserveOn, 339
obsługi zdarzeń, 245
OnCompleted, 316
OnError, 304, 316
OnNext, 304, 316
Parallel.For, 21, 22, 161, 162, 166, 176, 403
Parallel.ForEach, 161, 163, 166, 176, 212
Parallel.Invoke, 161, 164
ParallelQuery.ForAll, 212
Post, 128, 132
przekształcająca dane wynikowe, 208
przełączenie widoku, 229
Publish, 330
rozszerzająca, 199, 203, 204, 206, 316, 348
Salamina i Brenta, 47
Schedule, 334
SemaphoreSlim, 188
Send, 128, 132
Skip, 320
Sleep, 399, 400
SpinLock, 188
SpinLock.Enter, 45
SpinLock.Exit, 45
SpinOnce, 400
SpinWait, 188, 399
StartTimer, 384
statyczna, 31, 64, 181, 373, 413
Stop, 22
StopTimer, 384
SubscribeOn, 339
Switch, 359
SynchronizationContext.Post, 131
SynchronizationContext.Send, 131
System.Threading.Thread.VolatileRead, 64
System.Threading.Thread.VolatileWrite, 64
Take, 191
TakeWhile, 206
Task.ContinueWith, 143, 145, 147
Task.Delay, 399
Task.Factory.ContinueWhenAll, 152, 153
Task.Factory.ContinueWhenAny, 152, 153
Task.Factory.StartNew, 152, 153, 154, 179, 181
Task.Wait, 138, 143, 147
Task.WaitAll, 143, 147
Task.WaitAny, 143, 147
TaskFactory.ContinueWhenAny, 153
TaskScheduler.FromCurrentSynchronization
↪Context, 219
Thread.Abort, 30, 32, 33, 44, 103
Thread.Interrupt, 44
Thread.Join, 40, 135, 136
Thread.MemoryBarrier, 64
Thread.ResetAbort, 34
Thread.Resume, 30, 34, 80
Thread.Sleep, 14, 31, 181
Thread.SpinWait, 140, 399, 400
Thread.Suspend, 30, 34, 80
ThreadPool.QueueUserWorkItem, 49, 333
ThreadPool.SetMaxThreads, 49
Throttle, 358
ThrowIfCancellationRequested, 210

metoda
  tworząca, 309, 319
  Wait klasy Task, 17
  Window, 326
  WithCancellation, 209
  WithDegreeOfParallelism, 205, 213
  WithExecutionMode, 213
  WithMergeOptions, 213
  Wolfa, 47
  zdarzeniowa, 110, 114, 117
    przycisku, 14
  Zip, 321, 323
Microsoft OLE, 124
Microsoft Robotics, 243, 248, 249, 251, 276, 298
  instalacja, 246
  uruchamianie, 247
  zabezpieczenia, 293
model STA, *Patrz:* STA
modyfikator async, 16, 17
MTA, 124
Multi-Threaded Apartment, *Patrz:* MTA
multithreading, *Patrz:* wielowątkowość
murmelki, 315
muteks, 88, 89, 91, 93, 104
  lokalny, 89
  tworzenie, 90
MySpace, 243
mysz, 346, 348

N

NA, 124
Neutral Apartment, *Patrz:* NA
NuGet, 307, 345, 423, 425
  instalacja, 423
NVidia, 365, 375

O

obiekt
  CancellationTokenSource, 145
  COM, 124, *Patrz:* COM
  interfejsu, 345
  jądra, 88, 89
  synchronizacji, 45
  Task, 152
  timer, *Patrz:* timer
  typu referencyjnego, 45
  zarządzany, 124
obserwabla, 305, 310, 312
  czasu, 316
  gorąca, 329, 330
  Observable.Interval, 317
  Observable.Timer, 319
  zimna, 329, 330
obserwator, 305
odległość w przestrzeni euklidesowej, 199
okno
  stosów równoległych, 229
  śledzenia zmiennych, 230
  wątków, 226, 227
  zadań równoległych, 228
opakowanie, 190, 365, 366, 368, 369
operacja
  algebraiczna, 394
  asynchroniczna, 302
  atomowa, 51, 55, 64
operator
  async, 403
  await, 13, 16, 17, 18, 221, 403
  lock, 53, 188
  using, 240
optymalizacji wyłączanie, 64

P

pamięci bariera, 64
Parallel Extensions, 19, 137, 188
Parallel Stacks Window, *Patrz:* okno stosów równoległych
Parallel Tasks, *Patrz:* okno zadań równoległych
Parallel Watch Window, *Patrz:* okno równoległego śledzenia zmiennych
pełnomocnictwo, 108
pętla, 161
  For, 20
  liczba kroków, 175, 176
  Parallel.For, 13, 189, 403
  przerywanie, 166, 168
  równoległa, 13, 20
  współbieżna, *Patrz:* pętla równoległa
planista, 334
  CurrentThreadScheduler, 336, 337
  DispatcherScheduler, 346
  HistoricalScheduler, 335
  ImmediateScheduler, 336, 337, 339
  Reactive Extensions, 335, 336, 339
platforma CLR, *Patrz:* CLR
PLINQ, 19, 161, 199, 203, 204, 205, 207, 209, 212

pole statyczne, 40, 45
port TimeoutPort, 291
powiadomienia, 267
problem
  czytelników i pisarzy, 73
  konsumenta i producenta, 78, 155, 188, 191
  pięciu ucztujących filozofów, 68
proces, 31
program
  DssHost.exe, 245, 247
    administrator, 251
  oparty na wyciąganiu danych, 302
  w którym dane spływają, 303
programowanie
  interaktywne, 302, 304, 357
  reaktywne, 303, 304
protokół
  DSS Protocol, 245, 251
  komunikacji między procesami, 243
  TCP/IP, 245, 251
przedstawicielstwo, 108
przeglądarka internetowa, 249, 251
przekrój linii obrazu, 95
przestrzeń nazw
  System.Collections.Concurrent, 187, 188, 189, 190
  System.Reactive.Concurrency, 335
  System.Reactive.Linq, 316
  System.Reactive.Windows.Threading, 346
  System.Threading, 14, 29, 181, 209, 401, 402
  System.Threading.Tasks, 138, 161
  System.Windows.Shapes, 119
pull-based, *Patrz:* program oparty na wyciąganiu danych
punkt synchronizacji, 14
Python, 365

R

race condition, *Patrz:* wątek wyścig
Reactive Extensions, *Patrz:* Rx
  planista, 335, 336, 339
ReactiveCocoa, 302
Representational State Transfer, *Patrz:* REST
Resource Diagnostics, *Patrz:* usługa diagnostyki zasobów
rozgłaszanie, 267
rozszerzenie, *Patrz:* metoda rozszerzająca
Rx, 301, 307, 345, 361
  gramatyka, 309
  platforma, 306
  rysowanie, 346
  unifikacja, 343
  warstwa, *Patrz:* warstwa
  zarządzanie współbieżnością, 333
Rx-Cor, 343
Rx-Interfaces, 343
Rx-Linq, 343
Rx-PlatformServices, 343
Rx-Silverlight, 345
Rx-WPF, 345
Rx-Xaml, 345

S

Schura iloczyn, 378
sekcja krytyczna, 44, 53, 55, 89, 91, 104, 170, 188, 234, 407, 410
semafor, 91, 92, 93, 104
  lokalny, 93
serwis WCF, 215
silnik wyszukiwania, 345, 353, 357
Silverlight, 246, 404
Single-Threaded Apartment, *Patrz:* STA
słownik, 188, 337
słowo kluczowe
  delegate, 108
  lock, 44, 45, 51, 64, 81, 84, 181, 412
  params, 143
  return, 18
  volatile, 64, 417
spinning, 188
STA, 124
starvation, *Patrz:* wątek zagłodzony
stos, 189, 191, 195
  okno, *Patrz:* okno stosów równoległych
  współbieżny, 189
struktura CancellationToken, 34
subskrypcja, 312, 339
sygnał, 182
system
  operacyjny planista, 31
  rozproszony, 277

T

tablica
  deklaracja, 64
  sortowanie, 212
Task, *Patrz:* zadanie

Task Parallel Library, *Patrz:* TPL
technologia
  niezarządzana, 124
  REST, *Patrz:* REST
thread, *Patrz:* wątek
Threads, *Patrz:* okno wątków
timer, 55
token przerwania, 148, 209
TPL, 13, 19, 137, 161, 175, 204, 205, 215, 221, 225, 243, 361, 398, 404
transformata Fouriera szybka, *Patrz:* FFT

U

układ
  kartezjański lewoskrętny, 122
  współrzędnych, 115, 122
usługa, 245
  diagnostyki zasobów, 252
  identyfikacja, 256
  identyfikator, *Patrz:* identyfikator usługi
  partnerska, 265, 266, 291
  port TimeoutPort, 291
  port główny, 245
  rozpraszanie, 277
  stan, 245
  synchronizacja, 291
  tworzenie, 250, 284

V

Visual Studio, 225, 245, 307, 368, 397, 423

W

warstwa
  LINQ do zdarzeń, 306, 315
  sekwencji zdarzeń, 306, 315, 320
  zarządzania współbieżnością, 306, 334
Watch Window, *Patrz:* okno śledzenia zmiennych
wątek, 25, 28, 30, 334, 399
  aktywny, 227, 229
  bezpieczeństwo, *Patrz:* bezpieczeństwo
  blokada wirująca, 45
  budzenie, *Patrz:* wątek wznawianie
  CUDA, 378
  czas wykonania, 234
  dane współdzielone, 40
  interfejsu
    uprzywilejowany, 339
    użytkownika, 95, 130
  kontekst
    działania, *Patrz:* ATM
    synchronizacji, 128, 130, 132, 215, 218, 221
  obsługa zakończenia, 110
  oflagowanie, 226
  okno, *Patrz:* okno wątków
  pamięć lokalna, 39
  pobieranie danych, 45
  pomocniczy, 234
  priorytet, 35, 36, 56, 57
  przerywanie działania metody, 110
  pula, 25, 47, 48, 50, 54, 55, 155, 179, 185, 205, 401
  raportowanie postępu pracy, 110
  sekcja krytyczna, *Patrz:* sekcja krytyczna
  synchronizacja, 25, 34, 43, 45, 67, 84, 88, 169, 179, 234, 262, 291, 407, 413, 414, 416
    z interfejsem użytkownika, 104
    za pomocą blokad, 68
  tła, 35, 56
  usypianie, 31, 78, 81, 188, 400
  wstrzymanie, 34, 400
  wyścig, 104, 124, 208, 411, 412, 413, 416
  wznawianie, 78, 81
  zagłodzony, 71, 135
  zakleszczenie, 68, 135, 104, 124, 407
  zamrażanie, 188
  zmienna lokalna, *Patrz:* zmienna lokalna
  zrównoleglenie, 36
wektor, 394
wiadomość, 245, 261, 262
  Timeout, 291
widok
  Wątki, 233, 239
  Wykorzystanie CPU, 232
Widok Rdzenie, 236
wielowątkowość, 25
Windows Azure Marketplace, 353
Windows Communication Foundation, 243
Windows Forms, 124, 131, 215, 218, 345, 407
Windows Phone, 404
Windows Presentation Foundation, 96, 114, 116, 122, 131, 132, 218, 219, 345
WinRT, 397, 400, 402, 403, 404
własność
  BackgroundWorker.CancellationPending, 114
  Control.InvokeRequired, 104
  Control.InvokeRequired, 107, 128
  Environment.ProcessorCount, 206
  Task.Status, 149

WPF, *Patrz:* Windows Presentation Foundation
wrapper, *Patrz:* opakowanie
wyjątek, 103, 304, 410
  IndexOutOfRangeException, 189
  InvalidOperationException, 103, 107, 124
  OperationCanceledException, 146, 147, 209
  przechwytywanie, 148
wyjątki, 33
wyszukiwarka internetowa, 345, 353
wzorzec projektowy, 125
  obserwator, 305

X

XAML, 116, 345, 355, 397

Z

zadanie, 137, 138, 334, 398, 399
  dane, 140, 141
  fabryka, 152, 154
  oflagowanie, 226
  okno, *Patrz:* okno zadań równoległych
  planista, 153, 154, 155, 159, 219, 221
  planowanie, 334
  priorytet, 159
  przerywanie, 145
  stan, 149
  synchronizacja, 179
  sztafeta, 144
zakleszczenie, *Patrz:* wątek zakleszczenie
zależność rekurencyjna, 21
zapytanie, 302, 357
  czas wykonania, 203
  LINQ, *Patrz:* LINQ
  PLINQ, *Patrz:* PLINQ
  przerywanie, 209
  wydajność, 207
  zrównoleglone, 203, 205, 207
    zintegrowane z językiem programowania, 161
zasada Pareto, 13
zdarzenie, 245, 302, 304, 345, *Patrz też:* wiadomość
  kolekcja, 304
  MouseMove, 348
  sekwencja, 306, 315, 320, 334, 336
  strumień, 304
ziarno, 390
zmienna
  globalna, 40
  lokalna, 39, 59
  statyczna, 59
  typu referencyjnego, 44
znacznik, 238, 239

Programowanie współbieżne jest w dzisiejszych czasach absolutnym standardem, jedyną drogą pozwalającą w pełni wykorzystać moc wielordzeniowych procesorów, umieszczanych we współczesnych komputerach. Jednak nadal niewielu programistów potrafi poprawnie i efektywnie korzystać z tej technologii. Czas to zmienić!

Jeśli chcesz być naprawdę świetnym programistą C#, tworzącym aplikacje na platformę .NET, a do tej pory nie przestudiowałeś jeszcze porządnie kwestii współbieżności, pora to nadrobić. W tej książce znajdziesz informacje o podstawach tej technologii, takich jak wątki, a także o klasycznych pułapkach związanych z programowaniem współbieżnym. Kolejne rozdziały odkryją przed Tobą tajemnice zadań oraz bibliotek TPL. Poznasz również technologie DSS i CCR oraz dowiesz się co nieco na temat asynchroniczności. Potem przyjdzie kolej na omówienie technologii Reactive Extensions oraz biblioteki CUDAfy.NET, pozwalającej efektywnie wykorzystać karty graficzne do obliczeń niezwiązanych z grafiką. Odkryj zalety programowania równoległego!

- Dla niecierpliwych: asynchroniczność i pętla równoległa
- Wątki i zadania
- Zmienne w aplikacjach wielowątkowych
- Synchronizacja wątków, zadań i kontrolek interfejsu z zadaniami
- Wątki i zadania a interfejs użytkownika
- Dane w programach równoległych
- Analiza aplikacji wielowątkowych. Debugowanie i profilowanie
- Wstęp do CCR i DSS
- Skalowalne rozwiązanie dla systemów rozproszonych na bazie technologii CCR i DSS
- Wprowadzenie do Reactive Extensions. Zarządzanie sekwencjami zdarzeń
- Współbieżność w Rx
- Przykłady użycia technologii Rx w aplikacjach WPF
- CUDA w .NET
- Biblioteka TPL w WinRT
- Dobre praktyki programowania aplikacji wielowątkowych

Zostań mistrzem programowania współbieżnego!

Książka aktualna dla edycji VS 2010, 2012, 2013

ISBN 978-83-246-6698-0

9 788324 666980

Cena: 69,00 zł