GRZEGORZ LANG

Asynchroniczność i wielowątkowość w języku C#

Helion

Redaktor prowadzący: Szymon Szwajger
Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Helion SA
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/asynwi
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-7295-5

Printed in Poland.

Spis treści

Wstęp

Programowanie z wykorzystaniem wątków i asynchroniczności nie należy do najprostszych. Jest to dziedzina z natury skomplikowana i błędogenna. Na przestrzeni lat powstało wiele konceptów, które miały na celu uproszczenie zarządzania wątkami. Platforma .NET posiada własny zestaw abstrakcji. Książka ta powstała, aby zgromadzić i zwięźle wyjaśnić wszystkie zagadnienia związane z asynchronicznością i wielowątkowością w świecie C#. Poszczególne zagadnienia uzupełnione zostały przykładami, pomagającymi lepiej zrozumieć dany temat i zastosować go później w praktyce. Każdy koncept został opisany oddzielnie, jednak zakłada przyswojenie wiedzy z poprzednich rozdziałów.

Nie jest to książka dla początkującego programisty, który dopiero wkracza w świat programowania lub nie jest biegły w składni języka C#.

Książka ta została napisana w sposób maksymalnie zwięzły z dwóch powodów. Pierwszy to chęć, aby była używana do szybkiego odświeżenia sobie danego zagadnienia. Drugi to naturalnie minimalistyczny styl autora. Jest to zarazem błogosławieństwo jak i przekleństwo. Dzięki temu książka ta nie ma tysiąca stron, opisy są krótkie i trafiające w sedno, a przykłady kodu są łatwe do analizy. Jednak z drugiej strony czasami czytelnik potrzebuje dłuższego wyjaśnienia i większej ilości przykładów, aby w pełni przyswoić dany koncept. Dlatego cała książka jest obsypana odwołaniami do źródeł, z których czerpał wiedzę. Autor gorąco zachęca do ich odwiedzania. Przy ich pomocy powstała ta książka.

Wszystkie kody zostały napisane z wykorzystaniem C# 9.0 oraz .NET Core 5.0. Są to najnowsze wersje dostępne w trakcie pisania książki. Przeważająca większość kodu będzie działać również na dużo starszych wersjach, po małych modyfikacjach.

Krótka historia powstawania wielowątkowości

Pierwszym językiem, który pozwalał na oddelegowywanie pracy na osobny wątek, był PL/I stworzony przez firmę IBM w 1964 roku. Używał do tego instrukcji `CALL` wraz z opcją `TASK`[1, 2]. Określenie „wątek" powstało dwa lata później, zasugerowane przez Victora A. Vyssotskiego[3, 4].

Pierwszym komputerem domowym wykorzystującym wielozadaniowość była Amiga firmy Commodore, wprowadzona na rynek w 1985 roku[5]. Natomiast pierwszymi procesorami w architekturze x86 posiadającymi rdzenie były Pentium D firmy Intel oraz Athlon 64 X2 firmy AMD, oba przedstawione w 2005 roku[4].

Przez dekady programowanie współbieżne było możliwe, ale skomplikowane. To skutkowało jego unikaniem przez programistów. Obecnie dzięki użyciu współczesnych języków programowania (takich jak C#) możliwe jest wykorzystanie potencjału wątków w wygodny i bezpieczny sposób[6].

Umiejętność programowania wielowątkowego i asynchronicznego jest konieczna, aby wykorzystać wszystkie dostępne zasoby procesora, co pozwala na lepsze skalowanie rozwiązań serwerowych oraz zachowanie responsywności w aplikacjach użytkowych[7].

Wyjaśnienie pojęć związanych z wątkami

Procesor (ang. *central processing unit*) to niezależna fizyczna jednostka centralna odpowiedzialna za wykonywanie instrukcji programów komputerowych[8]. Typowy komputer domowy posiada jeden procesor, na który składa się wiele rdzeni.

Rdzeń (ang. *physical core*) to niezależna część procesora, która powstała, aby umożliwić wykonywanie wielu zadań równolegle w ramach jednej fizycznej struktury[9, 10]. Typowy komputer domowy posiada wiele rdzeni, które mogą, ale nie muszą składać się z wielu wątków sprzętowych.

Wątek sprzętowy (ang. *hardware thread*, *logical core*) pozwala dodatkowo zwielokrotnić możliwości równoległego wykonywania zadań w ramach jednego rdzenia[11, 12]. Typowy komputer domowy posiada taką samą ilość wątków sprzętowych co rdzeni lub jej wielokrotność.

Proces (ang. *process*) reprezentuje uruchomiony program[12]. Każdy proces jest wyizolowany i w pełni niezależny. Ze względów bezpieczeństwa komunikacja jest ograniczona i kontrolowana przez system operacyjny[13, 14]. Typowy program korzysta z pojedynczego procesu, na który składa się jeden lub więcej wątków.

Wątek (ang. *thread*) to najmniejsza jednostka zajmująca się wykonywaniem instrukcji programu[4]. Pozwala dzielić części kodu na niezależne zadania i wykonywać je na osobnych wątkach sprzętowych. Typowy program korzysta z minimum jednego wątku (zwanego głównym)[12].

Współbieżność (ang. *concurrency*) to możliwość wykonywania różnych czynności. Jeśli komputer posiada więcej niż jeden wątek sprzętowy, zadania te mogą być wykonywane w sposób równoległy (o czym więcej w rozdziale 2.)[15].

Ograniczenie procesorem (ang. *CPU-bound*) oznacza, że dane zadanie przez większość czasu wykonuje obliczenia[16]. Pomóc może zastosowanie wielowątkowości lub zrównoleglenie[17].

Ograniczenie systemem wejść/wyjść (ang. *I/O-bound*) oznacza, że dane zadanie przez większość czasu czeka na odpowiedź z zewnętrznego źródła[16]. Pomóc może zastosowanie asynchroniczności[17].

Rozdział 1.

Programowanie wielowątkowe

Programowanie wielowątkowe pozwala wykonywać różne czynności przy wykorzystaniu więcej niż jednego wątku[4]. Rezultatem jest zwiększenie wydajności w operacjach ograniczonych procesorem[17]. Dodatkowo takie programowanie umożliwia wykonywanie zadań w tle bez pogorszenia responsywności aplikacji[7].

Analogią może być sprzątanie domu. Jest wiele różnych czynności, które należy wykonać, robiąc porządki. Mogą one być zrobione przez jedną osobę (czyli jeden wątek sprzętowy) lub więcej osób, co pozwoli zakończyć pracę wcześniej.

Czynnikami spowalniającymi wielowątkowość są:

- narzut (ang. *overhead*), czyli czas potrzebny na przygotowanie wątku do wykonywania czynności,
- przełączanie się między wątkami (ang. *context switching*), jeśli jest ich więcej niż wątków sprzętowych[18].

1.1. Klasa Thread

Klasa `Thread` pozwala na tworzenie wątku oraz zarządzanie nim[19]. Aby sprawdzić numer identyfikacyjny obecnego wątku, można posłużyć się właściwością `Environment.CurrentManagedThreadId`[20].

```
void Main()
{
    // Pracujące wątki: główny.

    var t = new Thread(WykonajPracę);

    // Pracujące wątki: główny.

    t.Start();

    // Pracujące wątki: główny, poboczny.

    t.Join();

    // Pracujące wątki: główny.
}

void WykonajPracę()
{
    Thread.Sleep(5000);
}
```

Każdy program domyślnie wykonuje się na pojedynczym wątku głównym. Aby stworzyć dodatkowe wątki, można skorzystać z klasy `Thread`. Jako parametr przekazuje się delegat (ang. *delegate*), który zostanie uruchomiony na zewnętrznym wątku. Warto podkreślić, że sam fakt utworzenia obiektu `Thread` nie uruchamia wątku. Należy dodatkowo wywołać metodę `Start`[21]. Takie rozdzielenie pozwala oddzielić moment deklaracji pracy od momentu jej uruchomienia.

Wywołanie metody `Join` zawiesza bieżący wątek do momentu zakończenia pracy wątku pobocznego[22]. Takie rozwiązanie jest wygodne w użyciu, ale może ograniczyć skalowalność i responsywność całego systemu, ponieważ wątek główny nie może w tym czasie wykonywać żadnej innej potencjalnej pracy.

Metoda `Sleep` została wykorzystana, aby zasymulować pracę trwającą pięć sekund. Nie zaleca się stosowania jej w rozwiązaniach produkcyjnych, z tego samego powodu, co metody `Join`. Dla zadań, które wykonują się regularnie co określony

czas, zaleca się zastąpienie metody `Sleep` klasą `Timer` (która zostanie szczegółowo opisana w podrozdziale 1.7)[7]. Po okresie pięciu sekund wątek poboczny budzi się i kończy wykonywać metodę `WykonajPracę`. Następnie wątek główny odblokowuje się i kończy pracę programu.

Metoda `WykonajPracę`, jej wariant przyjmujący parametr o typie `object`, metoda `WykonajObliczenia` zwracająca losową liczbę o typie `int` oraz metoda asynchroniczna `WykonajZapytanieAsync` zwracająca `Task<int>` będą używane w dalszej części książki do symulacji prawdziwej pracy.

1.1.1. Wykonywanie pracy w tle

Aby zwiększyć komfort użytkownika, niektóre dodatkowe czynności mogą zostać wykonane na zewnętrznym wątku, w tle. Takie rozwiązanie pozwala zachować pełną responsywność aplikacji. Przykładami takich czynności mogą być: sprawdzanie ciągu tekstowego w poszukiwaniu błędów ortograficznych i gramatycznych, kompilacja kodu źródłowego w celu wyświetlenia błędów i ostrzeżeń kompilatora, defragmentacja dysków.

System operacyjny przydziela czas wątkom w zależności od ich priorytetu. Jeśli czynności wykonywane na danym wątku nie są kluczowe do działania aplikacji, zaleca się obniżenie jego priorytetu przez ustawienie właściwości `Thread.CurrentThread.Priority` na wartość `BelowNormal` albo `Lowest`[23]. Dzięki temu nie zakłóci on pracy innym, ważniejszym wątkom[7].

1.2. Klasa ThreadPool

Samodzielne zarządzanie cyklem życia wątków oraz efektywny rozkład pracy pomiędzy nimi są bardzo skomplikowane i podatne na błędy. Aby rozwiązać ten problem, powstała klasa `ThreadPool`.

Głównym zadaniem `ThreadPool` jest maksymalne, efektywne wykorzystanie rdzeni procesora. Gdy zostaje mu zlecone nowe zadanie do wykonania, a nie posiada żadnego wolnego wątku w puli, któremu mógłby oddelegować to zadanie, tworzy nowy wątek[24]. Zadania zleca się przez użycie metody statycznej `QueueUserWorkItem` i przekazanie delegata jako parametru[25].

```
ThreadPool.QueueUserWorkItem(WykonajPracę);
```

`ThreadPool` dąży do tego, aby maksymalnie aktywnie pracowało tyle wątków, ile procesor posiada rdzeni. To pozwala uniknąć kosztownego przełączania kontekstu pomiędzy rdzeniami.

Warto zwrócić uwagę na to, że `ThreadPool` bierze pod uwagę tylko aktywnie pracujące wątki. Jeśli któryś z wątków będzie zablokowany, stworzy on kolejny wątek. Takie zachowanie może wpłynąć negatywnie na zużycie zasobów pamięci komputera oraz Garbage Collector, jeśli programista będzie umyślnie blokować wątki (czego nigdy nie powinien robić). Natomiast w sytuacji, gdy przed dłuższy okres jest więcej wątków niż pracy do wykonania, `ThreadPool` unicestwia zbędne wątki w wolnej chwili[24].

1.2.1. Rodzaje kolejek

`ThreadPool` posiada globalną kolejkę, dostępną dla wszystkich jego wątków, oraz lokalną, oddzielną dla każdego z nich. Domyślnie metody klasy `ThreadPool` umieszczają pracę w kolejce globalnej. Można to zmienić, używając przeciążeń przyjmujących `boolean`, którego wartość `true` oznacza preferowanie kolejki lokalnej. Zadania zagnieżdżone (utworzone wewnątrz innego zadania) zostają umieszczone w lokalnej kolejce wątku, który wykonuje zadanie-rodzica.

Gdy wątek utworzony przez `ThreadPool` jest gotowy na nowe zadanie, najpierw sprawdza swoją lokalną kolejkę, następnie kolejkę globalną, a na końcu kolejki lokalne innych wątków. Jest to tzw. algorytm kradzieży pracy (ang. *work stealing*)[26].

1.2.2. Metoda UnsafeQueueUserWorkItem

Istnieje wydajniejsza odmiana metody `QueueUserWorkItem`, o nazwie `UnsafeQueue UserWorkItem`. Różnica w wydajności wynika z faktu, że aktualny kontekst `ExecutionContext` nie zostanie przesłany do wątku, który wykona delegowaną pracę. To oznacza także, że nie zostaną przekazane wszelkie restrykcje nałożone na pierwotny wątek. W rezultacie poziom bezpieczeństwa zostaje obniżony, natomiast zyskuje się większą wydajność[27, 28].

Dodatkowo, gdy używa się metody `UnsafeQueueUserWorkItem`, możliwe jest pominięcie alokacji instancji delegata na stercie, jeśli użyje się przeciążenia przyjmującego interfejs `IThreadPoolWorkItem`[29]. Pracę do wykonania deklaruje się jako osobną strukturę implementującą powyższy interfejs, to znaczy posiadającą pojedynczą metodę nazwaną `Execute`, która nie przyjmuje żadnych parametrów oraz niczego nie zwraca[30].

```
struct PracaDoWykonania : IThreadPoolWorkItem
{
    public void Execute()
    {
        WykonajPracę();
    }
}


ThreadPool.UnsafeQueueUserWorkItem(new PracaDoWykonania(), true);
```


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

Przejdź na wyższy poziom C#

Programowanie wielowątkowe pozwala wykonywać różne czynności z wykorzystaniem więcej niż jednego wątku. W efekcie zwiększa się wydajność operacji, które były ograniczone parametrami procesora. W dodatku dzięki zastosowaniu programowania wielowątkowego zadania można wykonywać w tle, bez pogorszenia responsywności aplikacji. Asynchroniczność umożliwia także zachowanie responsywności aplikacji użytkowych oraz zwiększenie skalowalności aplikacji serwerowych.

Brzmi obiecująco. Niestety, programowanie z wykorzystaniem wątków i asynchroniczności nie należy do najprostszych. Jest to dziedzina z natury skomplikowana i błędogenna, także w języku C#. Ten poradnik zbiera i zwięźle tłumaczy wszystkie zagadnienia związane z asynchronicznością i wielowątkowością w C#. Poszczególne tematy uzupełniono przykładami, pomagającymi lepiej zrozumieć problem, którego rozwiązanie można zastosować później w praktyce. Dla wygody w korzystaniu z książki każdy koncept opisano oddzielnie, jednak by go zrozumieć, trzeba przyswoić wiedzę z poprzednich rozdziałów.

- Dowiedz się, na czym polega wielowątkowość w C#
- Opanuj zasady programowania równoległego
- Naucz się programować asynchronicznie
- Poznaj podstawowe zasady synchronizacji

Grzegorz Lang — zawodowy programista .NET, Application Developer w międzynarodowej korporacji. Zajmuje się kluczowym projektem wewnętrznym, jest w nim Lead Backend Developerem. W języku C# programuje od 2013 roku.

Helion

helion.pl

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-7295-5

Cena: 37,00 zł