Spis treści

[1. Programowanie równoległe oraz asynchroniczne 2](#_Toc461272626)

[1.1. Klasa Task jako podstawowa jednostka programowania wielowątkowego. 2](#_Toc461272627)

[1.2. Programowanie asynchroniczne w języku C# 3](#_Toc461272628)

# Programowanie równoległe oraz asynchroniczne

Programowanie asynchroniczne oraz równoległe jest jednym z najważniejszych aspektów współczesnego sposobu projektowania aplikacji desktopowych, a co najbardziej istotne aplikacji serwerowych. Obecnie, każdy procesor składa się w rzeczywistości z wielu procesorów zwanych rdzeniami co zapewnia w teorii szybsze działanie aplikacji. Jednakże, aby uzyskać szybsze działanie należy wykorzystać moc obliczeniową procesora, poprzez użycie bibliotek pozwalających na poprawne zarządzanie wątkami procesorów. Jest to szczególnie ważny problem w aplikacjach serwerowych, gdzie skalowalność aplikacji jest rzeczą niezbędną do komercyjnego zastosowania dla wielu użytkowników. W niniejszym rozdziale zostaną przedstawione aspekty programowania równoległego oraz asynchronicznego w języku C#.

## Klasa Task jako podstawowa jednostka programowania wielowątkowego.

Klasa Task, która znajduje się w przestrzeni nazw *System.Threading.Tasks.Task* jest podstawową jednostką wątku w .Net Framework. Pierwotnie do harmonogramowania wątku służyła metoda Task.Factory.StartNew() jednakże od standardu 4.5 została wprowadzona metoda Task.Run() co jest pewnego rodzaju skrótem[[1]](#footnote-2). Obydwie metody przyjmują jako parameter delegate[[2]](#footnote-3), dzięki czemu możemy jako parameter przekazać wyrażenie lambda, jako funkcję anonimową, która zostanie wykonana w nowym wątku.

|  |  |  |
| --- | --- | --- |
|  | Task task = Task.Factory.StartNew(() =>  {  Console.WriteLine("New thread");  }); |  |

W powyższym przypadku nowy wątek zostanie najpierw stworzony, a następnie wprowadzony do zbioru wątków oczekujących na wybranie przez dyspozytora wątków. Inną możliwością jest stworzenie wątku samodzielnie, to znaczy bez mechanizmu fabryki.

|  |  |  |
| --- | --- | --- |
|  | Task task = new Task(() =>  {  Console.WriteLine("New thread");  });  task.Start(); |  |

Jednakże w tym przypadku należy wystartować nowy wątek explicite, w innym razie nie będzie on dostępny dla dyspozytora. Poniższy diagram przedstawia etapy życia wątków.



Diagram 1. Etapy życia wątków. Opracowanie własne na podstawie: SCJP Sun Certified Programmer for Java 6 Study Guide.

Stan nowy jest wówczas kiedy wątek został stworzony jako nowa instancja, bez wywoływania metody *start().* Na tym etapie wątek jest traktowany jako nieżywy (ang. *not alive*). Stan oczekujący jest stanem po wywołaniu metody *start(),* wtedywątek jest dostępny dla dyspozytora jako żywy (ang. *alive*). Wszystkie instancje klasy Task, na których została wywołana metoda *start()* znajdują się w zbiorze wątków oczekujących (ang. *thread pool*). W momencie kiedy dyspozytor wybierze ze zbioru wątek oczekujący, zmienia on status na działający (ang. *running*). Na tym etapie wątek może przejść w dwa stany, oczekujący lub użyty (ang. *dead*). Stan oczekujący/zablokowany jest w momencie kiedy wątek natrafi na blok synchronizujący, który np. zabezpiecza dane przed dostępem kilku wątków równocześnie. W języku C# taki blok oznacza się poprzez użycie słówka kluczowego lock(){ }[[3]](#footnote-4). Tego typu rozwiązania nazywane są blokowymi, ponieważ blokujemy wątki przed dalszą egzekucją w oczekiwana na dostęp do bloku synchronizującego. Wątek zablokowany, wraca do zbioru wątków oczekujących na wybranie przez dyspozytora. Stan użyty jest wówczas kiedy wątek skończy swoje zadanie. Wątek, który raz został użyty nie możesz zostać ponownie wystartowany, ponieważ zostanie rzucony wyjątek *System.InvalidOperationException*.

## Programowanie asynchroniczne w języku C#

Głównym podejściem w programowaniu asynchronicznym jest pisanie tych funkcji, które potencjalnie będą się długo wykonywały, w sposób asynchroniczny. Poprzednią koncepcją było tworzenie osobnych wątków, w których były wykonywane takie funkcje np. pobieranie dużej ilości danych z sieci. Podejście asynchroniczne zwiększa skalowalność aplikacji oraz jej wydajność. Jako przykład weźmy aplikacje desktopową (ang. *rich-client application*). Mając graf metod wywoływanych jedna po drugiej (ang. *call graph*) w tradycyjnym programowaniu synchronicznym, jeżeli jakaś operacja będzie wykonywała się przez długi czas, musimy wywołać cały graf na osobnym wątku, ponieważ w innym przypadku główny wątek aplikacji, który obsługuje cały interfejs użytkownika, przestanie odpowiadać na czas wykonywania metod z grafu. Dodatkowo każda z tych metod musi być bezpieczna w kwestii wielowątkowości. W przypadku asynchronicznego wywołania grafu metod nie musimy tworzyć nowego wątku dopóki nie jest on potrzebny, zazwyczaj tylko dla metody wykonującej operacje odczytu/zapisu. Pozostałe metody mogą być wywołane w wątku głównym. Jest to ważne chociażby z tego względu, że tylko metody które są wykonywane w głównym wątku mają dostęp do obiektów interfejsu graficznego w tym różnego rodzaju kontrolek itd.[[4]](#footnote-5)

Klasa Task idealnie nadaje się do programowania asynchronicznego, ponieważ wspiera kontynuacje co jest esencją asynchroniczności.

1. Źródło internetowe: <https://blogs.msdn.microsoft.com/pfxteam/2011/10/24/task-run-vs-task-factory-startnew/> [↑](#footnote-ref-2)
2. Delegata jest wskaźnikiem na funkcję. Możemy za jej pomocą przekazywać wyrażenia lambda lub zwyczajne metody zadeklarowane w ciele klasy. [↑](#footnote-ref-3)
3. Albahari J., Albahari B.: *C# 6.0 in a Nutshell,* O`Reilly Media, Sebastopol, 2015. [↑](#footnote-ref-4)
4. Albahari J., Albahari B.: *C# 6.0 in a Nutshell,* O`Reilly Media, Sebastopol, 2015, str. 590-591. [↑](#footnote-ref-5)