# 1. Programowanie równoległe i asynchroniczne w C#

## 1.1 Wprowadzenie do programowania współbieżnego

W nowoczesnym tworzeniu oprogramowania aplikacje często muszą wykonywać wiele operacji jednocześnie, aby zmaksymalizować wykorzystanie sprzętu i poprawić responsywność. Ten wykład omawia dwa odrębne, ale uzupełniające się podejścia w C#: programowanie równoległe i programowanie asynchroniczne.

## 1.2 Programowanie równoległe

### 1.2.1 Czym jest programowanie równoległe?

Programowanie równoległe to technika, która umożliwia aplikacji wykonywanie wielu obliczeń **jednocześnie** poprzez wykorzystanie wielu procesorów lub rdzeni procesora. Głównym celem jest poprawa wydajności poprzez podział zadania obliczeniowo intensywnego na mniejsze podzadania, które mogą być przetwarzane równocześnie.

W C# programowanie równoległe koncentruje się na operacjach ograniczonych przez CPU, gdzie czynnikiem ograniczającym jest moc obliczeniowa procesora. Jest idealne w scenariuszach, gdzie potrzebujesz wykonać tę samą operację na różnych elementach danych lub różne operacje na tych samych danych.

### 1.2.2 Kluczowe koncepcje w programowaniu równoległym

- **Równoległość zadań**: Wykonywanie różnych operacji jednocześnie
- **Równoległość danych**: Wykonywanie tej samej operacji na różnych częściach danych jednocześnie
- **Pula wątków**: Zbiór wątków roboczych zarządzanych przez środowisko uruchomieniowe
- **Partycjonowanie**: Dzielenie dużego zadania na mniejsze podzadania
- **Synchronizacja**: Koordynacja dostępu do współdzielonych zasobów wśród wielu wątków

### 1.2.3 Programowanie równoległe w C#: Task Parallel Library (TPL)

C# dostarcza Task Parallel Library (TPL) jako część platformy .NET, aby uprościć programowanie równoległe. TPL abstrahuje złożoności zarządzania wątkami i zapewnia model programowania wyższego poziomu.

**Kluczowe komponenty:**

- Klasa `Parallel`: Zapewnia równoległe wersje powszechnych operacji, takich jak pętle for i foreach
- Klasa `Task`: Reprezentuje operację asynchroniczną
- Klasa `Partitioner`: Pomaga podzielić pracę między wiele wątków
- `Kolekcje współbieżne`: Kolekcje bezpieczne dla wątków zaprojektowane do równoległego dostępu

### 1.2.4 Kwestie do rozważenia w programowaniu równoległym

- **Bezpieczeństwo wątków**: Zapewnienie bezpiecznego dostępu do współdzielonych zasobów w celu uniknięcia warunków wyścigu
- **Narzut**: Tworzenie i zarządzanie wątkami wprowadza dodatkowy narzut
- **Równoważenie obciążenia**: Równomierne rozłożenie pracy między wątki

- **Malejące zyski**: Dodawanie większej liczby wątków nie zawsze poprawia wydajność
- **Skalowalność**: Jak dobrze wydajność poprawia się wraz z dodatkowymi zasobami

# 1.3 Programowanie asynchroniczne

## 1.3.1 Czym jest programowanie asynchroniczne?

Programowanie asynchroniczne to paradygmat programowania, który pozwala operacjom na wykonywanie niezależnie od głównego przepływu aplikacji, bez blokowania wykonania kolejnych operacji. Głównym celem jest poprawa responsywności i wykorzystania zasobów poprzez unikanie bezczynnego oczekiwania.

W C# programowanie asynchroniczne jest szczególnie przydatne dla operacji ograniczonych przez operacje wejścia/wyjścia (I/O), takich jak dostęp do plików, wywołania sieciowe lub zapytania do baz danych, gdzie czynnikiem ograniczającym jest oczekiwanie na zewnętrzny zasób, a nie moc obliczeniowa procesora.

## 1.3.2 Kluczowe koncepcje w programowaniu asynchronicznym

- **Operacje nieblokujące**: Operacje, które nie zatrzymują wykonania programu podczas oczekiwania na ukończenie
- **Wywołania zwrotne (Callbacks)**: Funkcje, które wykonują się po zakończeniu operacji asynchronicznej
- **Obietnice/Zadania (Promises/Tasks)**: Obiekty reprezentujące ewentualne ukończenie (lub niepowodzenie) operacji asynchronicznej
- **Kontynuacje**: Działania zaplanowane do uruchomienia po zakończeniu operacji asynchronicznej
- **Pętla zdarzeń (Event Loop)**: Konstrukcja programistyczna, która czeka i dysponuje zdarzeniami lub wiadomościami

## 1.3.3 Kwestie do rozważenia w programowaniu asynchronicznym

- **Przełączanie kontekstu**: Zrozumienie, jak kontekst wykonania przepływa z async/await
- **Obsługa wyjątków**: Poprawna obsługa wyjątków w kodzie asynchronicznym
- **Zakleszczenia**: Unikanie powszechnych scenariuszy zakleszczenia z async/await
- **Anulowanie**: Obsługa anulowania operacji z CancellationToken
- **Raportowanie postępu**: Dostarczanie informacji zwrotnej podczas długotrwałych operacji

# 1.4 Porównanie programowania równoległego i asynchronicznego

| Aspekt | Programowanie równoległe | Programowanie asynchroniczne |
|---|---|---|
| Główny cel | Poprawa wydajności przez wykorzystanie wielu rdzeni | Poprawa responsywności przez nieblokowanie wątków |
| Najlepsze dla | Operacji ograniczonych przez CPU | Operacji ograniczonych przez I/O |
| Wykonanie | Jednoczesne | Niezależne |
| Użycie zasobów | Wiele wątków aktywnie pracujących | Wątek zwolniony podczas okresów oczekiwania |
| Implementacja w C# | Task Parallel Library, PLINQ | Wzorzec async/await, API oparte na zadaniach |
| Przykładowe zastosowanie | Przetwarzanie obrazów, złożone obliczenia | Żądania sieciowe, operacje na plikach |

## 1.5 Kiedy używać każdego podejścia

- **Używaj programowania równoległego gdy:**
  - Masz operacje intensywne obliczeniowo
  - Chcesz wykorzystać wiele rdzeni dla lepszej wydajności
  - Praca może być podzielona na niezależne fragmenty
  - Narzut paralelizacji jest uzasadniony przez zysk wydajnościowy
- **Używaj programowania asynchronicznego gdy:**
  - Masz operacje ograniczone przez I/O
  - Chcesz zachować responsywność interfejsu użytkownika
  - Potrzebujesz wykonać wiele niezależnych wywołań usług
  - Chcesz uniknąć blokowania wątków podczas okresów oczekiwania
- **Łącz oba podejścia gdy:**
  - Masz mieszankę operacji ograniczonych przez CPU i I/O
  - Chcesz zrównoleglić operacje po zakończeniu operacji asynchronicznej
  - Potrzebujesz wykonać operacje równoległe bez blokowania głównego wątku

## 1.6 Najlepsze praktyki

### 1.6.1 Dla programowania równoległego

- Mierz wydajność przed i po paralelizacji
- Bądź świadomy współdzielonego stanu i synchronizacji
- Rozważ używanie abstrakcji wyższego poziomu, jak PLINQ
- Dostosuj stopień równoległości w oparciu o możliwości systemu
- Używaj kolekcji bezpiecznych dla wątków do współdzielonych danych

### 1.6.2 Dla programowania asynchronicznego

- Zawsze używaj async/await konsekwentnie w całym łańcuchu wywołań
- Unikaj wywołań blokujących w metodach asynchronicznych
- Zwracaj Task/Task<\T> zamiast void dla lepszej obsługi błędów
- Używaj ConfigureAwait(false) gdy to odpowiednie, aby uniknąć przełączania kontekstu
- Implementuj właściwą obsługę anulowania i limitu czasu

# 2. Klasa Thread: Fundament współbieżności

## Wprowadzenie i cel

Klasa `Thread` była jednym z oryginalnych elementów budujących programowanie współbieżne w .NET, wprowadzonym z pierwszą wersją platformy. Zapewnia ona bezpośrednią otoczkę wokół wątków systemu operacyjnego, dając programistom jawną kontrolę nad tworzeniem wątków i zarządzaniem ich cyklem życia.

Klasa `Thread` została zaprojektowana, aby sprostać kilku podstawowym potrzebom w rozwoju aplikacji:

1. **Responsywność**: Utrzymywanie responsywności aplikacji z interfejsem użytkownika poprzez przenoszenie ciężkich zadań poza główny wątek
2. **Przepustowość**: Poprawianie wydajności poprzez wykonywanie zadań równolegle
3. **Wykorzystanie zasobów**: Lepsze wykorzystanie procesorów wielordzeniowych
4. **Przetwarzanie w tle**: Wykonywanie pracy, która nie powinna blokować głównego przepływu aplikacji

## Jak działa Thread

Kiedy tworzysz instancję `Thread`, zasadniczo prosisz system operacyjny o przydzielenie nowego wątku wykonania w ramach twojego procesu. Ten wątek otrzymuje własny stos wywołań i może wykonywać kod niezależnie od innych wątków. Klasa `Thread` zapewnia metody do uruchamiania, wstrzymywania, wznawiania i kończenia tych wątków, wraz z właściwościami pozwalającymi na konfigurowanie ich zachowania.

## Przykład podstawowego użycia

Oto prosty przykład tworzenia i używania wątku:

```
using System;
using System.Threading;

class ThreadExample
{
    static void Main()
    {
        Console.WriteLine($"Main thread ID: {Thread.CurrentThread.ManagedThreadId}");

        // Utworzenie nowego wątku, który wykonuje metodę WorkerMethod
        Thread workerThread = new Thread(WorkerMethod);

        // Uruchomienie wątku
        workerThread.Start("Hello from the main thread!");

        // Główny wątek kontynuuje wykonanie
        for (int i = 0; i < 3; i++)
        {
            Console.WriteLine($"Main thread working... {i}");
            Thread.Sleep(100);  // Symulacja pracy
        }

        // Oczekiwanie na zakończenie wątku roboczego
        workerThread.Join();
```

```
        Console.WriteLine("Main thread exiting.");
    }

    static void WorkerMethod(object parameter)
    {
        string message = parameter as string;
        Console.WriteLine($"Worker thread ID: {Thread.CurrentThread.ManagedThreadId}");
        Console.WriteLine($"Parameter received: {message}");

        for (int i = 0; i < 5; i++)
        {
            Console.WriteLine($"Worker thread working... {i}");
            Thread.Sleep(200);  // Symulacja pracy
        }

        Console.WriteLine("Worker thread exiting.");
    }
}
```

Ten przykład pokazuje:

- Tworzenie wątku z wyznaczoną metodą do wykonania
- Przekazywanie parametrów do metody wątku
- Równoległe wykonanie głównego wątku i wątku roboczego
- Oczekiwanie na zakończenie wątku za pomocą `Join()`

## Zaawansowany przykład Thread: Synchronizacja wątków

Gdy wiele wątków uzyskuje dostęp do współdzielonych zasobów, synchronizacja staje się niezbędna. Oto przykład używający blokady do synchronizacji:

```
using System;
using System.Threading;

class ThreadSynchronizationExample
{
    // Współdzielony zasób
    static int sharedCounter = 0;
    static object lockObject = new object();

    static void Main()
    {
        // Utworzenie wielu wątków uzyskujących dostęp do tego samego zasobu
        Thread[] threads = new Thread[5];

        for (int i = 0; i < threads.Length; i++)
        {
            threads[i] = new Thread(IncrementCounter);
            threads[i].Name = $"Thread {i}";
            threads[i].Start();
        }

        // Oczekiwanie na zakończenie wszystkich wątków
        foreach (Thread thread in threads)
        {
```

```
            thread.Join();
        }

        Console.WriteLine($"Final counter value: {sharedCounter}");
    }

    static void IncrementCounter()
    {
        for (int i = 0; i < 100000; i++)
        {
            // Synchronizacja dostępu do współdzielonego licznika
            lock (lockObject)
            {
                sharedCounter++;
            }
        }

        Console.WriteLine($"{Thread.CurrentThread.Name} completed.");
    }
}
```

Ten przykład pokazuje:

- Wiele wątków uzyskujących dostęp do współdzielonego zasobu (licznika)
- Używanie instrukcji `lock` w celu zapewnienia, że tylko jeden wątek modyfikuje zasób jednocześnie
- Nadawanie nazw wątkom dla łatwiejszego debugowania

## Ograniczenia klasy Thread

Mimo że jest potężna, bezpośrednie zarządzanie wątkami ma kilka wad:

1. **Intensywne wykorzystanie zasobów**: Każdy wątek zużywa pamięć na swój stos (zazwyczaj 1MB w systemie Windows)
2. **Narzut tworzenia**: Tworzenie i niszczenie wątków są stosunkowo kosztownymi operacjami
3. **Trudność w skalowaniu**: Tworzenie zbyt wielu wątków może prowadzić do zagłodzenia wątków i zmniejszenia wydajności
4. **Ręczne zarządzanie**: Programiści muszą obsługiwać cykl życia wątków, wyjątki i koordynację
5. **Brak wbudowanego mechanizmu wyników**: Uzyskiwanie wyników z wątków wymaga niestandardowej synchronizacji

Te ograniczenia doprowadziły do rozwoju abstrakcji wyższego poziomu, zaczynając od ThreadPool.

# 3. ThreadPool: Efektywne zarządzanie wątkami

## Dlaczego wprowadzono ThreadPool

Klasa `ThreadPool` została wprowadzona, aby rozwiązać kilka kluczowych ograniczeń bezpośredniego zarządzania wątkami:

1. **Narzut tworzenia wątków**: Tworzenie i niszczenie wątków jest kosztowne zarówno pod względem czasu, jak i zasobów
2. **Zarządzanie zasobami**: Aplikacje tworzące zbyt wiele wątków mogą wyczerpać zasoby systemowe
3. **Optymalna liczba wątków**: Określenie właściwej liczby wątków jest trudne i zależne od systemu
4. **Krótkotrwałe zadania**: Dla krótkich operacji narzut tworzenia wątku może przekroczyć faktyczny czas pracy

ThreadPool zarządza pulą wątków roboczych, które są tworzone raz i ponownie wykorzystywane do wielu zadań, znacznie zmniejszając narzut związany z tworzeniem i niszczeniem wątków.

## Jak ThreadPool usprawnia działanie wątków

ThreadPool przynosi kilka korzyści:

1. **Ponowne wykorzystanie wątków**: Zamiast tworzyć i niszczyć wątki, są one zwracane do puli po zakończeniu zadania
2. **Automatyczne skalowanie**: Pula rośnie i kurczy się w zależności od obciążenia systemu i zapotrzebowania
3. **Zarządzanie kolejką**: Zadania są umieszczane w kolejce i wykonywane, gdy wątki stają się dostępne
4. **Oszczędzanie zasobów**: Ogranicza maksymalną liczbę wątków, aby zapobiec wyczerpaniu zasobów
5. **Zmniejszone przełączanie kontekstu**: Mniejsza liczba aktywnych wątków oznacza mniejszy narzut z przełączania kontekstu

## Podstawowy przykład ThreadPool

Oto jak używać ThreadPool dla prostych zadań:

```
using System;
using System.Threading;

class ThreadPoolExample
{
    static void Main()
    {
        Console.WriteLine($"Main thread ID: {Thread.CurrentThread.ManagedThreadId}");

        // Pobierz informacje o ThreadPool przed kolejkowaniem zadań
        ThreadPool.GetMaxThreads(out int maxWorkerThreads, out int
maxCompletionPortThreads);
        Console.WriteLine($"Maximum worker threads: {maxWorkerThreads}");

        // Utwórz uchwyt oczekiwania, aby zasygnalizować zakończenie całej pracy
        ManualResetEvent allDone = new ManualResetEvent(false);
        int pendingWork = 5;
```

```
        // Umieść w kolejce wiele elementów pracy do ThreadPool
        for (int i = 0; i < 5; i++)
        {
            int workItemId = i;  // Przechwyć zmienną pętli

            ThreadPool.QueueUserWorkItem(state =>
            {
                ProcessWorkItem(workItemId);

                // Jeśli to było ostatnie zadanie, zasygnalizuj główny wątek
                if (Interlocked.Decrement(ref pendingWork) == 0)
                {
                    allDone.Set();
                }
            });
        }

        Console.WriteLine("All work items have been queued");

        // Poczekaj na zakończenie wszystkich zadań
        allDone.WaitOne();

        Console.WriteLine("All work completed. Main thread exiting.");
    }

    static void ProcessWorkItem(int workItemId)
    {
        Console.WriteLine($"Processing work item {workItemId} on thread ID:
{Thread.CurrentThread.ManagedThreadId}");

        // Symuluj różne obciążenia
        Thread.Sleep(workItemId * 100);

        Console.WriteLine($"Work item {workItemId} completed");
    }
}
```

Ten przykład pokazuje:

- Umieszczanie wielu elementów pracy w kolejce do ThreadPool
- Pobieranie informacji o pojemności ThreadPool
- Koordynowanie zakończenia za pomocą `ManualResetEvent`
- Ponowne wykorzystanie wątków (zauważysz, że te same identyfikatory wątków są używane dla różnych elementów pracy)

## ThreadPool z wynikami zwrotnymi

Jednym z ograniczeń ThreadPool jest brak wbudowanego mechanizmu zwracania wyników. Oto wzorzec do obsługi tego:

```
using System;
using System.Threading;

class ThreadPoolCallbackExample
{
```

```
    static void Main()
    {
        // Utwórz obiekt wyniku wywołania zwrotnego
        WorkCallback callbackObject = new WorkCallback();

        // Umieść pracę w kolejce z obiektem wywołania zwrotnego
        ThreadPool.QueueUserWorkItem(state =>
        {
            // Wykonaj jakieś obliczenia
            int result = PerformCalculation(42);

            // Ustaw wynik i zasygnalizuj zakończenie
            WorkCallback callback = (WorkCallback)state;
            callback.Result = result;
            callback.CompletedEvent.Set();
        }, callbackObject);

        Console.WriteLine("Work queued, waiting for result...");

        // Poczekaj na zakończenie pracy
        callbackObject.CompletedEvent.WaitOne();

        // Użyj wyniku
        Console.WriteLine($"Work completed with result: {callbackObject.Result}");
    }

    static int PerformCalculation(int input)
    {
        Console.WriteLine($"Performing calculation on thread ID:
{Thread.CurrentThread.ManagedThreadId}");
        Thread.Sleep(1000);  // Symulacja pracy
        return input * 2;
    }

    // Klasa pomocnicza dla wyników wywołania zwrotnego
    class WorkCallback
    {
        public ManualResetEvent CompletedEvent { get; } = new ManualResetEvent(false);
        public int Result { get; set; }
    }
}
```

Ten wzorzec pokazuje:

- Mechanizm zwrotny do odbierania wyników z elementów pracy ThreadPool
- Używanie niestandardowej klasy do przechowywania zarówno sygnału zakończenia, jak i wyniku

## Ograniczenia ThreadPool

Chociaż ThreadPool usprawnia bezpośrednie zarządzanie wątkami, wciąż ma ograniczenia:

1. Ograniczona kontrola: Nie można ustawiać priorytetu wątku ani nazywać wątków ThreadPool
2. Brak śledzenia zadań: Brak wbudowanego sposobu sprawdzania, czy element pracy został zakończony lub nie powiódł się
3. Brak kompozycji: Nie można łatwo łączyć ani komponować operacji

4. **Brak obsługi wyjątków**: Wyjątki w wątkach ThreadPool mogą spowodować awarię aplikacji
5. **Brak obsługi anulowania**: Brak standardowego sposobu anulowania elementów pracy

Te ograniczenia doprowadziły do opracowania Task Parallel Library, która bazuje na ThreadPool, ale zapewnia bogatszą funkcjonalność.

# 4. Task Parallel Library (TPL)

## Dlaczego wprowadzono TPL

Task Parallel Library (TPL) została wprowadzona w .NET Framework 4.0, aby rozwiązać ograniczenia ThreadPool i zapewnić podejście wyższego poziomu, bardziej ustrukturyzowane do programowania równoległego. Kluczowe motywacje obejmowały:

1. **Abstrakcja**: Zapewnienie abstrakcji wyższego poziomu ponad wątkami i zarządzaniem wątkami
2. **Komponowalność**: Umożliwienie składania operacji asynchronicznych z kontynuacjami
3. **Wyniki i wyjątki**: Standaryzacja sposobu obsługi wyników i wyjątków
4. **Anulowanie**: Wprowadzenie ujednoliconego modelu anulowania
5. **Ustrukturyzowana równoległość**: Dostarczenie wzorców dla typowych scenariuszy równoległych

## Jak TPL bazuje na ThreadPool

TPL jest zbudowana na podstawie ThreadPool, ale dodaje kilka ważnych funkcji:

1. **Abstrakcja zadań**: Zadania reprezentują operacje asynchroniczne, które mogą zwracać wynik
2. **Kontynuacje**: Zadania mogą być łączone razem z zależnymi operacjami
3. **Obsługa wyników**: Zadania mogą zwracać wartości przez klasę `Task<TResult>`
4. **Propagacja wyjątków**: Wyjątki w zadaniach są przechwytywane i mogą być obserwowane
5. **Anulowanie**: Standardowy wzorzec anulowania zadań za pomocą CancellationToken
6. **Raportowanie postępu**: Standaryzowany mechanizm raportowania postępów
7. **Koordynacja zadań**: Narzędzia do koordynowania wielu zadań (WaitAll, WhenAll, itp.)

## Podstawowy przykład zadania

Oto prosty przykład, który pokazuje tworzenie i oczekiwanie na zadania:

```
using System;
using System.Threading;
using System.Threading.Tasks;

class TaskBasicExample
{
    static void Main()
    {
        Console.WriteLine($"Main thread ID: {Thread.CurrentThread.ManagedThreadId}");

        // Utwórz i uruchom proste zadanie
        Task task = Task.Run(() =>
        {
            Console.WriteLine($"Task running on thread ID:
{Thread.CurrentThread.ManagedThreadId}");

            // Symulacja pracy
            Thread.Sleep(1000);

            Console.WriteLine("Task completed");
        });
```

```
        Console.WriteLine("Task started. Waiting for completion...");

        // Poczekaj na zakończenie zadania
        task.Wait();

        Console.WriteLine("Task has completed. Main thread continuing.");

        // Utwórz zadanie, które zwraca wynik
        Task<int> resultTask = Task.Run(() =>
        {
            Console.WriteLine($"Result task running on thread ID:
{Thread.CurrentThread.ManagedThreadId}");

            // Symulacja obliczeń
            Thread.Sleep(1000);

            return 42;
        });

        Console.WriteLine("Result task started. Waiting for result...");

        // Pobierz wynik (czeka na zakończenie)
        int result = resultTask.Result;

        Console.WriteLine($"Result task completed with value: {result}");
    }
}
```

Ten przykład pokazuje:

- Tworzenie i uruchamianie zadań za pomocą `Task.Run()`
- Oczekiwanie na zakończenie zadania za pomocą `Wait()`
- Tworzenie zadań, które zwracają wartości za pomocą `Task<TResult>`
- Pobieranie wyników za pomocą właściwości `Result`

## Przykład kontynuacji zadań

Jedną z najpotężniejszych funkcji TPL jest możliwość łączenia operacji za pomocą kontynuacji:

```
using System;
using System.Threading;
using System.Threading.Tasks;

class TaskContinuationExample
{
    static void Main()
    {
        Console.WriteLine("Starting task chain...");

        Task.Run(() =>
        {
            Console.WriteLine("First task starting");
            Thread.Sleep(1000);
            return "First task result";
        })
```

```
        .ContinueWith(antecedent =>
        {
            string previousResult = antecedent.Result;
            Console.WriteLine($"Second task received: {previousResult}");
            Thread.Sleep(1000);
            return $"{previousResult} + Second task result";
        })
        .ContinueWith(antecedent =>
        {
            string previousResult = antecedent.Result;
            Console.WriteLine($"Third task received: {previousResult}");
            Thread.Sleep(1000);
            Console.WriteLine("Task chain completed");
        })
        .Wait(); // Poczekaj na zakończenie całego łańcucha

        Console.WriteLine("All tasks completed");
    }
}
```

Ten przykład pokazuje:

- Łączenie zadań za pomocą `ContinueWith()`
- Przekazywanie wyników między zadaniami w łańcuchu
- Oczekiwanie na zakończenie całego łańcucha

## Obsługa wyjątków w zadaniach

Kolejną znaczącą zaletą TPL jest ustrukturyzowana obsługa wyjątków:

```
using System;
using System.Threading.Tasks;

class TaskExceptionExample
{
    static void Main()
    {
        // Utwórz zadanie, które zgłasza wyjątek
        Task task = Task.Run(() =>
        {
            Console.WriteLine("Task starting");
            throw new InvalidOperationException("Deliberate exception in task");
        });

        try
        {
            // Wait ponownie zgłosi wszelkie wyjątki z zadania
            task.Wait();
        }
        catch (AggregateException ae)
        {
            Console.WriteLine("Task failed with exceptions:");

            foreach (Exception innerException in ae.InnerExceptions)
            {
                Console.WriteLine($"- {innerException.GetType().Name}:
```

```
{innerException.Message}");
            }
        }

        // Alternatywnie, możemy obserwować wyjątek bez ponownego zgłaszania
        if (task.IsFaulted)
        {
            Console.WriteLine("\nTask faulted. Examining exceptions:");

            foreach (Exception innerException in task.Exception.InnerExceptions)
            {
                Console.WriteLine($"- {innerException.GetType().Name}:
{innerException.Message}");
            }
        }
    }
}
```

Ten przykład pokazuje:

- Jak wyjątki w zadaniach są przechwytywane
- Używanie `Wait()` do propagacji wyjątków
- `AggregateException`, który opakowuje wyjątki zadań
- Sprawdzanie statusu zadania za pomocą `IsFaulted`
- Badanie wyjątków bez ponownego zgłaszania

## Przykład koordynacji zadań

TPL dostarcza narzędzia do koordynowania wielu zadań:

```
using System;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

class TaskCoordinationExample
{
    static void Main()
    {
        // Utwórz kilka zadań
        Task[] tasks = new Task[5];

        for (int i = 0; i < tasks.Length; i++)
        {
            int taskId = i;
            tasks[i] = Task.Run(() =>
            {
                Console.WriteLine($"Task {taskId} starting");
                Thread.Sleep(taskId * 200); // Różne czasy trwania
                Console.WriteLine($"Task {taskId} completed");
                return taskId * 10;
            });
        }

        Console.WriteLine("Waiting for all tasks to complete...");
```

```
        // Poczekaj na zakończenie wszystkich zadań
        Task.WaitAll(tasks);

        Console.WriteLine("All tasks completed");

        // Utwórz zadania, które zwracają wyniki
        Task<int>[] resultTasks = Enumerable.Range(0, 5)
            .Select(i => Task.Run(() =>
            {
                Thread.Sleep(i * 200);
                return i * 100;
            }))
            .ToArray();

        Console.WriteLine("Waiting for first task to complete...");

        // Poczekaj na zakończenie dowolnego zadania
        int firstCompleted = Task.WaitAny(resultTasks);

        Console.WriteLine($"Task {firstCompleted} completed first with result:
{resultTasks[firstCompleted].Result}");

        // Pobierz wszystkie wyniki
        Console.WriteLine("Getting all results...");
        int[] results = Task.WhenAll(resultTasks).Result;

        Console.WriteLine("All results: " + string.Join(", ", results));
    }
}
```

Ten przykład pokazuje:

- Tworzenie wielu zadań
- Oczekiwanie na wszystkie zadania za pomocą `Task.WaitAll()`
- Oczekiwanie na dowolne zadanie za pomocą `Task.WaitAny()`
- Zbieranie wyników z wielu zadań za pomocą `Task.WhenAll()`

## Programowanie równoległe z TPL

Oprócz klasy `Task`, TPL zawiera narzędzia do równoległości danych:

```
using System;
using System.Diagnostics;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

class ParallelExample
{
    static void Main()
    {
        int[] numbers = Enumerable.Range(1, 1000000).ToArray();

        Console.WriteLine("Processing array with different approaches...");
```

```
        // Przetwarzanie sekwencyjne
        Stopwatch stopwatch = Stopwatch.StartNew();
        long sequentialSum = ProcessArraySequential(numbers);
        stopwatch.Stop();

        Console.WriteLine($"Sequential sum: {sequentialSum}, Time:
{stopwatch.ElapsedMilliseconds}ms");

        // Parallel.For
        stopwatch.Restart();
        long parallelForSum = ProcessArrayParallelFor(numbers);
        stopwatch.Stop();

        Console.WriteLine($"Parallel.For sum: {parallelForSum}, Time:
{stopwatch.ElapsedMilliseconds}ms");

        // Parallel.ForEach
        stopwatch.Restart();
        long parallelForEachSum = ProcessArrayParallelForEach(numbers);
        stopwatch.Stop();

        Console.WriteLine($"Parallel.ForEach sum: {parallelForEachSum}, Time:
{stopwatch.ElapsedMilliseconds}ms");
    }

    static long ProcessArraySequential(int[] array)
    {
        long sum = 0;

        for (int i = 0; i < array.Length; i++)
        {
            sum += ComputeValue(array[i]);
        }

        return sum;
    }

    static long ProcessArrayParallelFor(int[] array)
    {
        long sum = 0;

        Parallel.For(0, array.Length, i =>
        {
            long localSum = ComputeValue(array[i]);
            Interlocked.Add(ref sum, localSum);
        });

        return sum;
    }

    static long ProcessArrayParallelForEach(int[] array)
    {
        long sum = 0;

        Parallel.ForEach(array, item =>
        {
```

```
            long localSum = ComputeValue(item);
            Interlocked.Add(ref sum, localSum);
        });

        return sum;
    }

    static long ComputeValue(int value)
    {
        // Symulacja obliczeń ograniczonych przez CPU
        long result = value;

        for (int i = 0; i < 100; i++)
        {
            result += (long)Math.Sqrt(result);
        }

        return result;
    }
}
```

Ten przykład pokazuje:

- Używanie `Parallel.For` do równoległej iteracji
- Używanie `Parallel.ForEach` do równoległego przetwarzania kolekcji
- Porównanie wydajności sekwencyjnej i równoległej
- Bezpieczną dla wątków akumulację za pomocą `Interlocked.Add`

## Ograniczenia TPL

Mimo że TPL stanowiło duży postęp, wciąż miało pewne wyzwania:

1. **Złożoność**: Komponowanie złożonych przepływów asynchronicznych mogło prowadzić do "piekła wywołań zwrotnych"
2. **Obsługa błędów**: Bloki try-catch nie działają naturalnie w granicach zadań
3. **Przepływ kontekstu**: Zarządzanie kontekstem synchronizacji w kontynuacjach jest skomplikowane
4. **Czytelność**: Złożone łańcuchy zadań mogą być trudne do zrozumienia i utrzymania

Te wyzwania doprowadziły do wprowadzenia wzorca async/await w C# 5.0.

# 5. async/await: Uproszczone programowanie asynchroniczne

## Dlaczego wprowadzono async/await

Wzorzec async/await został wprowadzony w C# 5.0, aby rozwiązać złożoność programowania asynchronicznego z wykorzystaniem zadań. Jego główne cele to:

1. Czytelność: Sprawienie, by kod asynchroniczny wyglądał i zachowywał się jak kod synchroniczny
2. Obsługa błędów: Umożliwienie stosowania znanych wzorców try-catch w operacjach asynchronicznych
3. Przepływ sterowania: Zachowanie naturalnego przepływu sterowania w kodzie asynchronicznym
4. Kompozycja: Uproszczenie składania operacji asynchronicznych
5. Zarządzanie kontekstem: Automatyczne zarządzanie kontekstem synchronizacji

## Jak async/await bazuje na TPL

Wzorzec async/await jest zbudowany na Task Parallel Library, ale przekształca sposób, w jaki programiści piszą kod asynchroniczny:

1. Lukier składniowy: Kompilator przekształca metody asynchroniczne w maszyny stanów
2. Integracja z zadaniami: Działa płynnie z istniejącymi API opartymi na zadaniach
3. Naturalna składnia: Sprawia, że kod asynchroniczny czyta się jak kod synchroniczny
4. Propagacja wyjątków: Umożliwia naturalną obsługę wyjątków za pomocą try-catch
5. Zarządzanie kontynuacjami: Automatycznie zarządza kontynuacjami

## Podstawowy przykład async/await

Oto prosty przykład demonstrujący wzorzec async/await:

```
using System;
using System.Threading;
using System.Threading.Tasks;

class AsyncAwaitBasicExample
{
    static async Task Main()
    {
        Console.WriteLine($"Main thread ID: {Thread.CurrentThread.ManagedThreadId}");

        Console.WriteLine("Before calling DoWorkAsync");

        // Wywołaj metodę asynchroniczną i zaczekaj na jej zakończenie
        int result = await DoWorkAsync(42);

        Console.WriteLine($"After awaiting DoWorkAsync, result: {result}");

        // Wywołaj wiele metod asynchronicznych sekwencyjnie
        await Step1Async();
        await Step2Async();
        await Step3Async();

        Console.WriteLine("All steps completed");
```

```
    }

    static async Task<int> DoWorkAsync(int input)
    {
        Console.WriteLine($"DoWorkAsync started on thread ID:
{Thread.CurrentThread.ManagedThreadId}");

        // Symulacja pracy asynchronicznej
        await Task.Delay(1000);

        Console.WriteLine($"DoWorkAsync resuming on thread ID:
{Thread.CurrentThread.ManagedThreadId}");

        return input * 2;
    }

    static async Task Step1Async()
    {
        Console.WriteLine("Step 1 starting");
        await Task.Delay(500);
        Console.WriteLine("Step 1 completed");
    }

    static async Task Step2Async()
    {
        Console.WriteLine("Step 2 starting");
        await Task.Delay(500);
        Console.WriteLine("Step 2 completed");
    }

    static async Task Step3Async()
    {
        Console.WriteLine("Step 3 starting");
        await Task.Delay(500);
        Console.WriteLine("Step 3 completed");
    }
}
```

Ten przykład demonstruje:

- Deklarowanie metod asynchronicznych za pomocą słowa kluczowego `async`
- Oczekiwanie na zakończenie zadania za pomocą słowa kluczowego `await`
- Sekwencyjne wykonywanie operacji asynchronicznych
- Zmiany kontekstu wątku podczas operacji asynchronicznych

## Obsługa wyjątków z async/await

Jedną z głównych zalet async/await jest naturalna obsługa wyjątków:

```
using System;
using System.Threading.Tasks;

class AsyncExceptionExample
{
    static async Task Main()
```

```
    {
        try
        {
            Console.WriteLine("Calling method that will throw...");
            await ThrowExceptionAsync();
        }
        catch (InvalidOperationException ex)
        {
            Console.WriteLine($"Caught exception: {ex.Message}");
        }

        try
        {
            Console.WriteLine("\nCalling method with nested exceptions...");
            await NestedExceptionAsync();
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Caught top-level exception: {ex.GetType().Name},
{ex.Message}");
        }
    }

    static async Task ThrowExceptionAsync()
    {
        await Task.Delay(500); // Symulacja pracy asynchronicznej

        // Wyjątek zostanie przechwycony przez Task i ponownie zgłoszony podczas
oczekiwania
        throw new InvalidOperationException("Async operation failed");
    }

    static async Task NestedExceptionAsync()
    {
        try
        {
            await Task.Delay(500); // Symulacja pracy asynchronicznej
            throw new InvalidOperationException("Inner operation failed");
        }
        catch (InvalidOperationException ex)
        {
            // Możemy przechwycić i przekształcić wyjątki
            throw new ApplicationException("Wrapped exception", ex);
        }
    }
}
```

Ten przykład pokazuje:

- Używanie try-catch z oczekiwanymi zadaniami
- Jak wyjątki propagują się przez wywołania asynchroniczne
- Przechwytywanie i ponowne zgłaszanie przekształconych wyjątków

## Asynchroniczne operacje na plikach i sieci

async/await naprawdę błyszczy przy operacjach ograniczonych przez I/O:

```
using System;
using System.IO;
using System.Net.Http;
using System.Threading.Tasks;

class AsyncIOExample
{
    static async Task Main()
    {
        await FileOperationsAsync();
        await NetworkOperationsAsync();
    }

    static async Task FileOperationsAsync()
    {
        string filePath = "example.txt";
        string content = "This is a sample text file created asynchronously.";

        Console.WriteLine("Writing to file asynchronously...");
        await File.WriteAllTextAsync(filePath, content);

        Console.WriteLine("Reading from file asynchronously...");
        string readContent = await File.ReadAllTextAsync(filePath);

        Console.WriteLine($"Read {readContent.Length} characters from file");

        // Asynchroniczne kopiowanie pliku
        string backupPath = "example.backup.txt";

        using (FileStream sourceStream = new FileStream(filePath, FileMode.Open,
FileAccess.Read, FileShare.Read, 4096, true))
        using (FileStream destinationStream = new FileStream(backupPath,
FileMode.Create, FileAccess.Write, FileShare.None, 4096, true))
        {
            Console.WriteLine("Copying file asynchronously...");
            await sourceStream.CopyToAsync(destinationStream);
        }

        Console.WriteLine("File operations completed");
    }

    static async Task NetworkOperationsAsync()
    {
        using (HttpClient client = new HttpClient())
        {
            Console.WriteLine("Sending HTTP request asynchronously...");

            try
            {
                HttpResponseMessage response = await
client.GetAsync("https://jsonplaceholder.typicode.com/todos/1");
                response.EnsureSuccessStatusCode();

                string responseBody = await response.Content.ReadAsStringAsync();

                Console.WriteLine($"Received response: {responseBody.Substring(0,
Math.Min(100, responseBody.Length))}...");
```

```
            }
            catch (HttpRequestException ex)
            {
                Console.WriteLine($"Request error: {ex.Message}");
            }
        }
    }
}
```

Ten przykład demonstruje:

- Asynchroniczne czytanie i zapisywanie plików
- Asynchroniczne kopiowanie plików za pomocą strumieni
- Asynchroniczne żądania HTTP
- Poprawne zwalnianie zasobów za pomocą instrukcji using

## Równoległe wykonywanie z async/await

Możemy łączyć async/await z TPL dla równoległych operacji asynchronicznych:

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

class AsyncParallelExample
{
    static async Task Main()
    {
        // Wykonaj wiele operacji asynchronicznych równolegle
        Console.WriteLine("Starting parallel async operations...");

        Task<int> task1 = ComputeValueAsync(10);
        Task<int> task2 = ComputeValueAsync(20);
        Task<int> task3 = ComputeValueAsync(30);

        // Poczekaj na zakończenie wszystkich zadań i zbierz wyniki
        int[] results = await Task.WhenAll(task1, task2, task3);

        Console.WriteLine($"All tasks completed. Results: {string.Join(", ",
results)}");

        // Przetwarzaj kolekcję równolegle
        List<int> inputs = Enumerable.Range(1, 10).ToList();

        Console.WriteLine("\nProcessing collection in parallel...");

        // Utwórz i uruchom wiele zadań równolegle
        IEnumerable<Task<int>> tasks = inputs.Select(ComputeValueAsync);

        // Poczekaj na zakończenie wszystkich
        int[] parallelResults = await Task.WhenAll(tasks);

        Console.WriteLine($"Parallel processing completed. Sum:
{parallelResults.Sum()}");
```

```
    }

    static async Task<int> ComputeValueAsync(int input)
    {
        Console.WriteLine($"Starting computation for input: {input}");

        // Symulacja pracy asynchronicznej
        await Task.Delay(1000);

        int result = input * 10;
        Console.WriteLine($"Computation completed for input: {input}, result:
{result}");

        return result;
    }
}
```

Ten przykład pokazuje:

- Uruchamianie wielu operacji asynchronicznych równolegle
- Używanie `Task.WhenAll` do oczekiwania na wszystkie operacje
- Przetwarzanie kolekcji równolegle z LINQ i metodami asynchronicznymi

## Anulowanie i raportowanie postępu z async/await

Możemy również zintegrować anulowanie i raportowanie postępu:

```
using System;
using System.Threading;
using System.Threading.Tasks;

class AsyncCancellationExample
{
    static async Task Main()
    {
        // Utwórz źródło tokenu anulowania z limitem czasu
        using (CancellationTokenSource cts = new CancellationTokenSource(5000)) // 5-
sekundowy limit czasu
        {
            try
            {
                // Utwórz obiekt raportowania postępu
                Progress<int> progress = new Progress<int>(percent =>
                {
                    Console.WriteLine($"Operation progress: {percent}%");
                });

                Console.WriteLine("Starting long-running operation with cancellation and
progress...");

                // Wykonaj operację z możliwością anulowania i raportowaniem postępu
                int result = await LongRunningOperationAsync(100, progress, cts.Token);

                Console.WriteLine($"Operation completed with result: {result}");
            }
            catch (OperationCanceledException)
```

```
            {
                Console.WriteLine("Operation was cancelled");
            }
        }

        // Przykład ręcznego anulowania
        using (CancellationTokenSource cts = new CancellationTokenSource())
        {
            // Uruchom zadanie, które zostanie anulowane
            Task task = LongRunningOperationAsync(50, new Progress<int>(percent =>
            {
                Console.WriteLine($"Second operation progress: {percent}%");
            }), cts.Token);

            // Zaplanuj anulowanie po 2 sekundach
            Console.WriteLine("Operation started, will cancel in 2 seconds...");
            await Task.Delay(2000);

            Console.WriteLine("Cancelling operation...");
            cts.Cancel();

            try
            {
                await task;
                Console.WriteLine("Operation completed (shouldn't reach here)");
            }
            catch (OperationCanceledException)
            {
                Console.WriteLine("Second operation was successfully cancelled");
            }
        }
    }

    static async Task<int> LongRunningOperationAsync(int iterations, IProgress<int>
progress, CancellationToken cancellationToken)
    {
        int result = 0;

        for (int i = 0; i < iterations; i++)
        {
            // Sprawdź, czy operacja została anulowana
            cancellationToken.ThrowIfCancellationRequested();

            // Wykonaj pracę
            await Task.Delay(100, cancellationToken);
            result += i;

            // Raportuj postęp
            int percentComplete = (i + 1) * 100 / iterations;
            progress?.Report(percentComplete);
        }

        return result;
    }
}
```

Ten przykład demonstruje:

- Używanie `CancellationTokenSource` i `CancellationToken` do anulowania
- Implementowanie raportowania postępu za pomocą `IProgress<T>`
- Obsługę anulowania za pomocą `ThrowIfCancellationRequested()`
- Przekazywanie tokenów anulowania do operacji asynchronicznych
- Przechwytywanie i obsługę `OperationCanceledException`

## Korzyści z async/await

Wzorzec async/await zapewnia znaczące korzyści:

1. **Czytelność**: Kod asynchroniczny wygląda podobnie do kodu synchronicznego
2. **Utrzymywalność**: Przepływ sterowania jest łatwiejszy do śledzenia i zrozumienia
3. **Obsługa błędów**: Naturalna obsługa wyjątków za pomocą bloków try-catch
4. **Zarządzanie zasobami**: Dobrze współpracuje z instrukcjami using dla czyszczenia zasobów
5. **Skalowalność**: Efektywne wykorzystanie zasobów dla operacji ograniczonych przez I/O
6. **Kompozycja**: Upraszcza łączenie i zagnieżdżanie operacji asynchronicznych

## Ograniczenia async/await

Chociaż async/await rozwiązuje wiele wyzwań, wciąż ma pewne kwestie do rozważenia:

1. **Async w całej ścieżce**: Dla maksymalnej korzyści cały stos wywołań powinien być asynchroniczny
2. **Potencjał zakleszczenia**: Mieszanie kodu synchronicznego i asynchronicznego może prowadzić do zakleszczeń
3. **Przechwytywanie kontekstu**: Domyślnie kontynuacje wracają do oryginalnego kontekstu, co może być nieefektywne
4. **Narzut wydajnościowy**: Maszyna stanów wprowadza pewien dodatkowy narzut
5. **Złożoność debugowania**: Przekształcony kod może być trudniejszy do debugowania

# 6. PLINQ: Deklaratywna równoległość danych

## Dlaczego wprowadzono PLINQ

Parallel LINQ (PLINQ) został wprowadzony jako część Task Parallel Library, aby zapewnić deklaratywne podejście do równoległego przetwarzania danych. Jego kluczowe cele to:

1. **Prostota**: Uczynienie równoległości dostępną przy minimalnych zmianach w kodzie
2. **Model deklaratywny**: Umożliwienie programistom wyrażania, co ma być wykonywane równolegle, a nie jak
3. **Integracja**: Bazowanie na znanej składni zapytań LINQ
4. **Wydajność**: Poprawa wydajności dla operacji ograniczonych przez CPU na kolekcjach
5. **Adaptowalność**: Automatyczne dostosowywanie się do dostępnych zasobów sprzętowych

## Jak PLINQ bazuje na TPL i LINQ

PLINQ łączy Task Parallel Library z operatorami zapytań LINQ:

1. **Równoległe zapytania**: Automatycznie dystrybuuje operacje zapytań pomiędzy wiele wątków
2. **Partycjonowanie**: Dzieli dane na fragmenty przetwarzane równolegle
3. **Kradzież pracy**: Wykorzystuje algorytm kradzieży pracy TPL do równoważenia obciążenia
4. **Scalanie**: Łączy wyniki z operacji równoległych
5. **Obsługa wyjątków**: Konsoliduje wyjątki z wielu wątków

## Podstawowy przykład PLINQ

Oto prosty przykład użycia PLINQ:

```
using System;
using System.Diagnostics;
using System.Linq;

class BasicPlinqExample
{
    static void Main()
    {
        // Utwórz dużą tablicę danych
        int[] numbers = Enumerable.Range(1, 10000000).ToArray();

        Console.WriteLine("Performing LINQ and PLINQ queries...");

        // Standardowe zapytanie LINQ
        Stopwatch stopwatch = Stopwatch.StartNew();
        var normalResult = numbers
            .Where(n => n % 2 == 0)
            .Select(n => Compute(n))
            .Take(1000)
            .ToList();
        stopwatch.Stop();

        Console.WriteLine($"Standard LINQ: {normalResult.Count} items, Time:
{stopwatch.ElapsedMilliseconds}ms");
```

```
        // Zapytanie PLINQ - wystarczy dodać AsParallel()
        stopwatch.Restart();
        var parallelResult = numbers
            .AsParallel()
            .Where(n => n % 2 == 0)
            .Select(n => Compute(n))
            .Take(1000)
            .ToList();
        stopwatch.Stop();

        Console.WriteLine($"PLINQ: {parallelResult.Count} items, Time:
{stopwatch.ElapsedMilliseconds}ms");
    }

    static int Compute(int n)
    {
        // Symulacja obliczeń intensywnych procesorowo
        double result = 0;
        for (int i = 0; i < 1000; i++)
        {
            result += Math.Sqrt(n * i);
        }
        return (int)result;
    }
}
```

Ten przykład demonstruje:

- Konwersję zapytania LINQ na PLINQ za pomocą `AsParallel()`
- Poprawę wydajności dla operacji ograniczonych przez CPU
- Prostotę dodawania równoległości do istniejących zapytań LINQ

## Zaawansowane opcje PLINQ

PLINQ zapewnia kilka opcji do dostrajania równoległości:

```
using System;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

class AdvancedPlinqExample
{
    static void Main()
    {
        int[] numbers = Enumerable.Range(1, 1000000).ToArray();

        Console.WriteLine("PLINQ with various options:");

        // Kontrola stopnia równoległości
        var customDopResult = numbers.AsParallel()
            .WithDegreeOfParallelism(Environment.ProcessorCount / 2)
            .Where(n => n % 2 == 0)
            .Select(n =>
            {
```

```
                Console.WriteLine($"Processing {n} on thread
{Thread.CurrentThread.ManagedThreadId}");
                return n * 2;
            })
            .Take(10)
            .ToArray();

        Console.WriteLine("\nPreserving element order:");

        // Zachowanie kolejności z AsOrdered
        var orderedResult = numbers.AsParallel()
            .AsOrdered()
            .Where(n => n % 2 == 0)
            .Select(n => n * 2)
            .Take(10)
            .ToArray();

        Console.WriteLine("Ordered result: " + string.Join(", ", orderedResult));

        Console.WriteLine("\nWithout preserving order:");

        // Bez zachowania kolejności (potencjalnie szybsze)
        var unorderedResult = numbers.AsParallel()
            .Where(n => n % 2 == 0)
            .Select(n => n * 2)
            .Take(10)
            .ToArray();

        Console.WriteLine("Unordered result: " + string.Join(", ", unorderedResult));

        // Anulowanie z PLINQ
        Console.WriteLine("\nPLINQ with cancellation:");

        CancellationTokenSource cts = new CancellationTokenSource();

        try
        {
            Task.Run(() =>
            {
                Thread.Sleep(100); // Anuluj po krótkim opóźnieniu
                cts.Cancel();
                Console.WriteLine("Cancellation requested");
            });

            var cancelResult = numbers.AsParallel()
                .WithCancellation(cts.Token)
                .Where(n =>
                {
                    Thread.Sleep(10); // Spowolnij operację, aby zademonstrować
anulowanie
                    return n % 2 == 0;
                })
                .Select(n => n * 2)
                .ToArray();
        }
        catch (OperationCanceledException)
        {
```

```
            Console.WriteLine("PLINQ operation was cancelled");
        }

        // ForAll dla niestandardowego przetwarzania
        Console.WriteLine("\nUsing ForAll for side effects:");

        int count = 0;
        object lockObj = new object();

        numbers.AsParallel()
            .Where(n => n % 2 == 0)
            .Take(1000)
            .ForAll(n =>
            {
                // Inkrementacja bezpieczna dla wątków
                Interlocked.Increment(ref count);
            });

        Console.WriteLine($"Processed {count} items with ForAll");
    }
}
```

Ten przykład demonstruje:

- Kontrolowanie stopnia równoległości za pomocą `WithDegreeOfParallelism()`
- Zachowanie kolejności elementów za pomocą `AsOrdered()`
- Obsługę anulowania za pomocą `WithCancellation()`
- Używanie `ForAll()` do niestandardowego przetwarzania wyników

## Obsługa wyjątków w PLINQ

PLINQ obsługuje wyjątki z wielu wątków, zbierając je w AggregateException:

```
using System;
using System.Linq;

class PlinqExceptionExample
{
    static void Main()
    {
        int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        try
        {
            // Zapytanie PLINQ, które zgłosi wyjątki
            var result = numbers.AsParallel()
                .Select(n =>
                {
                    if (n % 3 == 0)
                    {
                        throw new InvalidOperationException($"Error processing {n}");
                    }
                    return n * 2;
                })
                .ToArray();
```

```
        }
        catch (AggregateException ae)
        {
            Console.WriteLine("PLINQ operation failed with exceptions:");

            foreach (var ex in ae.InnerExceptions)
            {
                Console.WriteLine($"- {ex.Message}");
            }
        }
    }
}
```

Ten przykład pokazuje:

- Jak wyjątki z operacji równoległych są zbierane
- AggregateException, który zawiera wszystkie zgłoszone wyjątki

## Korzyści z PLINQ

PLINQ oferuje kilka zalet:

1. **Prostota**: Dodawanie równoległości za pomocą pojedynczego wywołania metody
2. **Znajomość**: Bazuje na dobrze znanej składni LINQ
3. **Wydajność**: Poprawia wydajność dla operacji ograniczonych przez CPU
4. **Podejście deklaratywne**: Koncentruje się na tym, co robić, a nie jak równoległe
5. **Automatyczne partycjonowanie**: Obsługuje podział danych i scalanie

## Kwestie do rozważenia przy PLINQ

PLINQ ma kilka ważnych kwestii do rozważenia:

1. **Narzut**: Równoległość wprowadza narzut koordynacji, który może nie być opłacalny dla małych zbiorów danych
2. **Efekty uboczne**: Operacje powinny być bezstanowe i wolne od efektów ubocznych
3. **Bezpieczeństwo wątków**: Każdy współdzielony stan musi być odpowiednio synchronizowany
4. **Porządkowanie**: Zachowanie kolejności za pomocą `AsOrdered()` dodaje narzut
5. **Nieparalelizowalne operacje**: Niektóre operacje wymuszają sekwencyjne wykonanie

# 7. Podsumowanie: Ewolucja współbieżności w CS

Ewolucja mechanizmów współbieżności w C# i .NET odzwierciedla postęp w kierunku abstrakcji wyższego poziomu, które czynią programowanie współbieżne bardziej przystępnym, łatwiejszym w utrzymaniu i solidniejszym:

1. **Thread**: Kontrola niskiego poziomu z wysoką złożonością
2. **ThreadPool**: Bardziej efektywne zarządzanie wątkami z mniejszą kontrolą
3. **Task Parallel Library**: Ustrukturyzowana równoległość oparta na zadaniach z kompozycją
4. **async/await**: Uproszczone programowanie asynchroniczne z naturalnym przepływem sterowania
5. **PLINQ**: Deklaratywna równoległość danych z minimalnymi zmianami kodu

Każda warstwa bazuje na poprzedniej, rozwiązując ograniczenia i zmniejszając złożoność. Dzisiejsi programiści zazwyczaj pracują na wyższych poziomach (async/await i PLINQ), ale zrozumienie całego stosu pomaga w diagnozowaniu problemów i optymalizacji wydajności w razie potrzeby.

## Rekomendacje frameworka

Przy wyborze podejścia do współbieżności w C#:

1. **Dla operacji ograniczonych przez I/O** (plik, sieć, baza danych):
   - Używaj `async/await` z API opartymi na zadaniach
   - Koncentruj się na przepustowości i responsywności
2. **Dla operacji ograniczonych przez CPU** (obliczenia, przetwarzanie):
   - Używaj `Task.Run` z TPL dla prostej równoległości
   - Używaj `Parallel.For/ForEach` dla ustrukturyzowanej równoległości danych
   - Używaj PLINQ dla równoległości danych opartej na zapytaniach
3. **Kiedy używać podejść niższego poziomu**:
   - Używaj ThreadPool bezpośrednio dla specjalistycznych scenariuszy kolejkowania
   - Używaj Thread bezpośrednio tylko wtedy, gdy potrzebujesz pełnej kontroli nad cyklem życia wątku

Nowoczesne podejście generalnie faworyzuje async/await dla większości scenariuszy, łącząc je z TPL lub PLINQ, gdy potrzebna jest równoległość, i powracając do mechanizmów niższego poziomu tylko wtedy, gdy jest to konieczne.

Dzięki zrozumieniu pełnej ewolucji współbieżności w C#, programiści mogą dokonywać świadomych wyborów dotyczących tego, które podejście najlepiej odpowiada ich konkretnym wymaganiom, równoważąc prostotę, wydajność i łatwość utrzymania.