Politechnika Wrocławska

Platformy programistyczne

Wykorzystanie platformy .NET w kontekście programowania równoległego oraz asynchroniczności

Dr inż. Radosław Idzikowski

Katedra Automatyki, Mechatroniki i Systemów Sterowania
Wydział Informatyki i Teleinformatyki

12 kwietnia 2024

HR EXCELLENCE IN RESEARCH

Motywacja

- **responsywny interfejs użytkowania** – wykonywanie czasochłonnych obliczeń w tle;
- **jednoczesne przetwarzanie żądań** – obsługa zapytań po stronie serwera;
- **obliczenia równoległe** – przyśpieszenie obliczeń w środowiskach wieloprocesorowych;
- **wykonywanie spekulatywne** – przyśpieszenie wydajności poprzez przewidywanie zadań;

Równoległość a współbieżność?

- Programowanie sekwencyjne,
- Programowanie współbieżne,
- Programowanie równoległe,
- Programowanie rozproszone.

Obliczenia wielowątkowe

Ogólny mechanizm pozwalający programowi na równoległe wykonywanie kodu jest określany mianem wielowątkowości. Wspominana wielowątkowość jest obsługiwana zarówno przez środowisko uruchomieniowe CLR, jak i system operacyjny oraz stanowi podstawową koncepcję współbieżności.[1]

Tworzenie wątku

- Nasz program rozpoczyna działanie w wątku (utworzonym przez system operacyjny) nazywanym głównym.
- Tworzenie i uruchamianie nowych wątków można użyć klasy `Thread`.
- Konstruktor klasy `Thread` jako parametr przyjmuje delegat bez parametrów.

```
Thread thread = new Thread(couting);
thread.Start();
couting();
void couting()
{
    for (int i = 0; i < 20; i++)
    {
        Console.Write(i + "␣");
    }
}
```

Własnośći wątku

- Wątek kończy "żywot" w momencie gdy przekazany delegat kończy prace.
- Zakończony wątek nie może zostać drugi raz uruchomiony.
- Podczas działania wątku jego atrybut `isAlive` przyjmuje wartość `true`.
- Każdy wątek ma składową `Name`, która można zostać zdefiniowana tylko raz.

Tworzenie większej liczby wątków

```
int n = 5;
Thread[] threads = new Thread[n];
for(int i = 0; i < n; i++)
{
    threads[i] = new Thread(couting);
    threads[i].Name = $"Thread␣{i}";
}
foreach (Thread th in threads)
    th.Start();

static void couting()
{
    for (int i = 0; i < 500; i++)
    {
        Console.WriteLine(Thread.CurrentThread.Name + ":␣" + i);
    }
}
```

Inne operacje na wątkach

- `Join()` – czekanie na zakończenie pracy innego wątku.
- `Sleep()` – czekanie ze zwolnieniem zasobów procesora.
- `lock()` – nałożenie blokady.
- przekazywanie argumentów do wątku przy użyciu wyrażenia lambda.

klasa `Thread`

- narzędzie niskiego poziomu,
- brak łatwego dostępu do informacji zwrotniej z wątku,
- problem z przechwyceniem i obsługą wyjątków,
- brak możliwości wznowienia zakończonego wątku,
- brak możliwości przydzielenia nowej operacji,
- konieczność synchronizacji poprzez metodę `Join()`.

klasa Task

.NET Framework 4.0

- wyższy poziom abstrakcji,
- reprezentuję współbieżną operacje (która nie musi być wykonywana w nowym wątku),
- zadania mogą być łączone,
- zadanie może używać puli wątków,

Funkcje asynchroniczne w C# wykorzystują typ `Task`.

klasa Task

Uruchamianie zadań

```
using System.Threading.Tasks;
Task task = Task.Run(() => {
    Thread.Sleep(1000);
    Console.WriteLine("Hello, World!");
    });
Console.WriteLine($"Task is compelted?: {task.IsCompleted}");
task.Wait();
Console.WriteLine($"Task is compelted? {task.IsCompleted}");
Console.Read();
```

klasa Task

Wartość zwrotna

```
using System.Threading.Tasks;
Task<int> task = Task.Run(() => {
    Thread.Sleep(1000);
    Console.WriteLine("Hello, World!");
    return 1;
    });
int number = task.Result;
Console.WriteLine($"Task result is: {number}");

Console.Read();
```

klasa Task

Kontynuacja zadania

```
using System.Threading.Tasks;
Task<int> task = Task.Run(() => {
    Thread.Sleep(1000);
    Console.WriteLine("Hello, World!");
    return 1;
    });

var awaiter = task.GetAwaiter();
awaiter.OnCompleted(() => {
    Thread.Sleep(1000);
    Console.WriteLine("Hello, World!");
});
Console.Read();
```

PKX

Parallel Framework

- równoległa biblioteka zadań,
- zastosowanie do programowania równoległego,
- Parallel LINQ.

Parallel.For w C#

- `Parallel.For` jest wygodnym mechanizmem do wykonywania operacji równoległych w C#.
- Pozwala na równoległe przetwarzanie elementów pętli w sposób efektywny.
- Automatycznie zarządza wieloma wątkami, dostosowując liczbę do dostępnych rdzeni procesora.
- Zapewnia prosty interfejs do tworzenia i zarządzania operacjami równoległymi.

Parallel.For a Parallel.ForEach

- `Parallel.For` jest używany do iterowania przez kolekcję w sposób równoległy, gdzie możemy określić początek i koniec zakresu iteracji.
- `Parallel.ForEach` jest używany do iterowania przez kolekcję w sposób równoległy, gdzie określamy pojedynczy element kolekcji, który będzie przetwarzany równolegle.
- `Parallel.For` jest bardziej elastyczny, jeśli chodzi o kontrolę nad zakresem iteracji.
- `Parallel.ForEach` jest bardziej czytelny, gdy chcemy operować na pojedynczych elementach kolekcji.
- Wybór między nimi zależy od kontekstu i preferencji programisty.

Zarządzanie ustawieni biblioteki `Parallel`

```
var options = new ParallelOptions
        {
            MaxDegreeOfParallelism = Environment.ProcessorCount
        };
Parallel.ForEach(myList, options, element =>
        {
            element.modify()
        });
```

Programowania równoległego z użyciem Thread a Parallel.For

- `Thread` jest niskopoziomowym mechanizmem do tworzenia i zarządzania wątkami, co wymaga ręcznej synchronizacji i kontroli.
- `Parallel.For` jest wyższopoziomowym mechanizmem, który automatycznie zarządza wieloma wątkami, dzięki czemu programista nie musi martwić się o synchronizację ani zarządzanie wątkami.
- Programowanie równoległe z użyciem `Thread` może być bardziej skomplikowane i podatne na błędy związane z synchronizacją dostępu do współdzielonych zasobów.
- `Parallel.For` oferuje prostszy i bardziej bezpieczny sposób wykonywania operacji równoległych, co może prowadzić do łatwiejszego utrzymania i debugowania kodu.
- W przypadku `Parallel.For` biblioteka .NET automatycznie dostosowuje liczbę wątków do dostępnych rdzeni procesora, co może prowadzić do lepszej wydajności w przypadku operacji równoległych.

PLINQ

```
using System.Threading.Tasks;
Random random = new Random();
int[] numbers = Enumerable.Range(0,100).ToArray();

numbers = numbers.AsParallel().Select(x => x *x ).ToArray();

foreach (int number in numbers)
    Console.WriteLine(number);
```