SPRAWOZDANIE

Programowanie równoległe

Programowanie asynchroniczne

15.12.2015

Karol Suwalski 125NCI B

https://github.com/SuwalskiKarol/oor.git

Wstęp

Zamrażanie interfejsu użytkownika podczas wykonywania skomplikowanej czynności nie jest dobrym pomysłem. Aplikacja, która "zamiera" nie zachęca do dalszego wykorzystywania przez użytkownika. Programowanie asynchroniczne z drugiej strony wiąże się z większym nakładem pracy oraz bardziej skomplikowanym kodem wynikowym.

Kiedy stosować metody synchroniczne:

- Dla prostych, szybkich operacji nie powodujących blokowania wątków,
- Gdy prostota jest ważniejsza niż efektywność,
- Dla operacji wykorzystujących w większym stopniu procesor (metody asynchroniczne nie obniżają zużycia procesora wręcz przeciwnie zwiększają zużycie procesora).

Kiedy stosować metody asynchroniczne:

- Gdy brakuje wolnych wątków,
- Gdy aplikacja będzie szybciej/lepiej działać przy użyciu metod asynchronicznych,
- Gdy operacja opiera się w głównej mierze na operacjach I/O (zapisu odczytu) na dysku lub na operacjach sieciowych, które mogą zablokować wątek poprzez wolne działanie,
- Aby zaimplementować mechanizm zatrzymywania lub przerywania długich żądań (np. przesyłanie plików).

Starsze wersje C#

We wczesnych wersjach .NET, jedyną możliwością wykonania serii zadań była obsługa tzw. **callback** – metod wywołanych po zakończeniu zadania. Przepływ informacji, był dość skomplikowany, ponieważ wymagało to stworzenia dodatkowych metod (callback), przekazania parametrów a często również stworzenia nowych klas czy kontenerów(w callback można było przekazać wyłącznie jeden parametr typu object). Pisanie kodu wykorzystując callback jest skomplikowane, próbowałem cuś napisać, ale po 5 minutach pracy to olałem i przeszedłem na async☺.

C# 5.0

Słowa kluczowe **async** i **await** to nowość wprowadzona w C# 5.0. Użycie tych słów kluczowych pozwala na bardzo prostą implementacje metod asynchronicznych.

```
privateintBeginCalculate(objectnumbersTuple)
        {
Thread.Sleep(20000);
Tuple<int, int> numbers = (Tuple<int, int>)numbersTuple;
return numbers.Item1 + numbers.Item2;
        }

privateasyncvoidbtnCalculate_Click(object sender, RoutedEventArgs e)
        {
int number1 = int.Parse(txtNumber1.Text);
int number2 = int.Parse(txtNumber1.Text);
int result = 0;
result = await Calculate(number1, number2);

txtAnswer.Text = result.ToString();
awaitUploadResult(result);

}

privateasyncTask<int> Calculate(int number1, int number2)
        {
returnawaitTask.Run(() =>
            {
Thread.Sleep(4000);
return number1 + number2;
            });
        }
```

`btnCalculate_Click`oraz `Calculate`wykonywane sąasynchronicznie Calculate jest również wykonywany w osobnym wątku nie obciążając tym samym UI Thread.Słowo kluczowe awaitczeka na wynik wykonania metody. Obserwuje wątek i wznawia wykonanie kodu, gdy zakończy on działanie. Jednak, w przeciwieństwie do Thread.Join, await nie blokuje całej metody.
Każda metoda, która zawiera w sobie przynajmniej jedno wywołanie await, musi zostać oznaczona async.W powyższym przypadku, `BeginCalculate` wykonywana jest synchronicznie, aż do momentu napotkania wywołania await. Tutaj pojawia się różnica między await a starym Thread.Join. Join po prostu blokował wywołanie, aż do zakończenia wątku. W przypadku async oraz await, .NET framework będzie sprawdzał, czy wątek `Calculate` nie zakończył działania, przy czym metoda async nie obciąża głównego wątku.
Wykorzystując

Eventqueue oraz Eventloop

W C# istnieją metody to tworzenia samych kolejek jak np. (jak nie trudno się domyślić nazwy) queue:

```
Queue<int> queue = newQueue<int>();
SyncEventssyncEvents = newSyncEvents()

Producer producer = newProducer(queue, syncEvents);
Consumer consumer = newConsumer(queue, syncEvents);
Thread producerThread = newThread(producer.ThreadRun);
Thread consumerThread = newThread(consumer.ThreadRun);
```

W przypadku Eventów, jeżeli tworzymy aplikację Windows Forms lub WPF i korzystamy z gotowych kontrolek to wystarczy w kodzie takiegobuttona dodać słowo async

```
privateasyncvoidbtnCalculate_Click(object sender, RoutedEventArgs e)
```

Program od tej pory będzie wiedział, że jest to event wykonywany asynchronicznie i będzie tworzył kolejki automatycznie.

Możemy również tworzyć własne eventloopy.

```
EventLoop.Start(() => {
//Czytaj wszystko w pliku asynchronicznie.
File.ReadAllText(@"C:\oor.txt", (text) => {
Console.WriteLine(text);
```

Async w ASP.NET MVC

W MVC asynchroniczność jest bardzo ważnym elementem. Zazwyczaj jest ona stosowana podczas tworzenia kontrolerów np. przy operacjach CRUD.

```
publicasyncTask<ActionResult> Delete(int ?id)
        {
if (id == null)
            {
returnnewHttpStatusCodeResult(HttpStatusCode.BadRequest);
            }
Departmentdept = awaitdb.Departments.FindAsync(id);
if (dept == null)
            {
returnHttpNotFound();
            }
return View(dept);
        }

        [HttpPost,ActionName("Delete")]
        [ValidateAntiForgeryToken]
publicasyncTask<ActionResult>DeleteConfirmed( int id)
        {
Departmentdept = awaitdb.Departments.FindAsync(id);
db.Departments.Remove(dept);
awaitdb.SaveChangesAsync();
returnRedirectToAction("Index");
        }
```

Jest to podstawowa metoda do asynchronicznej obsługi zdarzenia Delete.