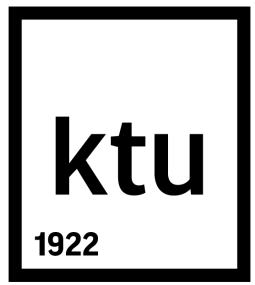
# Asinchroninis programavimas

Karolis Ryselis

Kauno Technologijos Universitetas




*Q: How do you change a lightbulb in concurrent programming?*
*A: You take the lamp to a secure area so nobody else can try to change the lightbulb while you're changing it.*

# Paskaitos turinys



ktu 1922

# Gijų naudojimo trūkumai

Tiesioginis gijų naudojimas turi trūkumų:

- Nėra galimybės grąžinti suskaičiuotą reikšmę iš gijos;
- Jei gijoje įvyksta išimtinė situacija (*exception*), nėra galimybės jos apdoroti.

Šias problemas sprendžia asinchroninės užduotys.

# Asinchroninės užduotys

- Asinchroninės užduotys vykdomos paleidžiant vieną giją lygiagrečiai, o pagrindinė gija toliau tęsia darbą.
- Asinchroninės užduotys naudingos, kai reikia vykdyti ilgai trunkančią operaciją, jos vykdymo metu lygiagrečiai atlikti kitus veiksmus, kuriems nereikia asinchroninės užduoties rezultato, arba su rezultatu atliekamus veiksmus atlikti nepagrindinėje gijoje.
- Kiekviena asinchroninė užduotis nebūtinai turi atliekama atskiros gijos.

# Asinchroninės užduotys

- Asinchroninės užduotys gali būti taikomos programoms su naudotojo sąsaja.
- Naudotojui atlikus tam tikrą veiksmą gali būti paleidžiama asinchroninė užduotis, kuri neblokuoja naudotojo sąsajos, o apskaičiavus rezultatą naudotojo sąsaja atsinaujina.

# Task

- .NET karkasas turi klasę `Task`. Klasė yra naudojama asinchroninėms užduotims kurti ir vykdyti.
- Taip pat palaikomi **`async`** ir **`await`** raktiniai žodžiai, skirti palengvinti darbą su asinchroninėmis užduotimis.

# Pagrindiniai `Task` metodai ir savybės

## Konstruktorius

Klasė turi keletą konstruktorių. Visi jie priima parametrą `action`, kuris nurodo, kokį metodą ar delegatą reikia vykdyti asinchroniškai.
Papildomai galima paduoti tokius parametrus:

- `cancellationToken` — skirtas užduoties atšaukimo valdymui.
- `creationOptions` — skirtas užduoties vykdymo parametrų valdymui, pvz., galima nurodyti, kad užduočių vykdymo planuoklė stengtųsi vykdytų užduotis ta tvarka, kuria jos buvo paleistos.
- `state` — jei `action` priima parametrą, `state` nurodo jo reikšmę.

## `Result`

Blokuoja giją ir, pasibaigus užduoties vykdymui, grąžina `Task` rezultatą.

## `Wait()`

Blokuoja kviečiančią giją kol užduotis baigs darbą.

# Pagrindiniai `Task` metodai

### `ContinueWith(Func<Task, TResult> continuation)`

Metodas priima delegatą, kuris bus iškviečiamas pasibaigus užduoties vykdymui su tos užduoties rezultatu. Delegatas taip pat bus vykdomas asinchroniškai, o pats metodas grąžina naują užduotį. Kadangi metodas kviečiamas vienai užduočiai ir grąžina naują užduotį, galima sudaryti visą `ContinueWith` kreipinių grandinę.

### `Run(Action action)`

Parametru priima metodą ar delegatą ir jį vykdo asinchroniškai, grąžina naują `Task` objektą. Metodas yra statinis. Metodas paleidžia naują giją ir veiksmus vykdo joje, dėl to naudingiau operacijoms, naudojančioms CPU resursus.

# Pagrindiniai `Task` metodai

### `WhenAll(Task[] tasks)`

Priima `Task` objektų masyvą ir grąžina naują `Task` objektą, kuris bus įvykdytas, kai bus įvykdytos visos perduotos užduotys. Sukurtos naujos užduoties rezultatas yra visų užduočių rezultatų masyvas.

### `WhenAny(Task[] tasks)`

Priima `Task` objektų masyvą ir grąžina naują `Task` objektą, kuris bus įvykdytas, kai bus įvykdyta greičiausia užduotis. Sukurtos naujos užduoties rezultatas yra greičiausios užduoties rezultatas.

# Pagrindiniai `Task` metodai

**`Exception`**

Jei vykdant užduotį įvyko išimtinė situacija, ši savybė turės `AggregateException` tipo reikšmę, kuria naudojantis galima pasiekti ir viduje iškilusią išimtinę situaciją.

# Lygiagretus funkcinio programavimo stiliaus quicksort algoritmas

```
private static List<int> GetSorted(List<int> listToSort, int depth)
{
    if (listToSort.Count == 0) {
        return listToSort;
    }
    var pivot = listToSort[0];
    var lowerPart = listToSort.Skip(1).Where(item => item < pivot).ToList();
    var newLower = depth >= 0 ?
        Task.Run(() => GetSorted(lowerPart, depth - 1)) :
        Task.FromResult(GetSorted(lowerPart, depth - 1));
    var higherPart = listToSort.Skip(1).Where(item => item >= pivot).ToList();
    var newHigher = GetSorted(higherPart, depth - 1);
    return newLower.Result
        .Concat(new List<int> { pivot })
        .Concat(newHigher)
        .ToList();
}
```

# Asinchroniniai metodai

- Kuriant asinchronines programas nėra būtina pačiam kurti `Task` objektų ir naudoti tęsinio metodų ar kitų priemonių.
- Užduotis kurti ir rezultatus iš jų paimti galima naudojantis raktiniais žodžiais `async` ir `await`.
- Metodą galima pažymėti raktiniu žodžiu `async`. Tokio metodo grąžinamas tipas turi būti įdėtas į `Task` tipą, kitaip C# kompiliatorius nekompiliuos programos.
- `async` metodo `return` sakinys turi grąžinti ne patį `Task` objektą, o reikšmę, kuri jame saugoma — C# kompiliatorius ją įdės į `Task` objektą ir grąžins jį.

# Asinchroniniai metodai

- Jei metodas pažymėtas `async` raktiniu žodžiu, jo grąžinamą reikšmę „išsiimti“ iš užduoties galima naudojant raktinį žodį `await`.
- `await` sustabdo metodo vykdymą, kol `async` metodo rezultatas bus suskaičiuotas.
- `await` raktinį žodį galima naudoti tik `async` metode.
- Naudojant asinchroninius metodus nebūtinai kuriamos papildomos gijos – išnaudojamas esamas gijų telkinys, o kol vieni metodai sustabdyti su `await`, vykdomi kiti.
- Dėl to asinchroniniai metodai geriau tinka ten, kur nėra išnaudojamas CPU.

# Orų patikrinimas OpenWeatherAPI asinchroniškai

```
public class OpenWeatherApi
{
    private readonly string _apiKey = File.ReadAllText("api_key.txt");

    private async Task<Stream> MakeOpenWeatherRequest(int cityId)
    {
        const string apiRoot = "http://api.openweathermap.org/";
        using var client = new HttpClient();
        return await client.GetStreamAsync(
            $"{apiRoot}data/2.5/weather?id={cityId}&appid={_apiKey}"
        );
    }
```

# Orų patikrinimas OpenWeatherAPI asinchroniškai

```
    private async Task<double> ParseResponse(Stream jsonResponse)
    {
        var jsonObject = await JsonDocument.ParseAsync(jsonResponse);
        var root = jsonObject.RootElement;
        var main = root.GetProperty("main");
        return main.GetProperty("temp").GetDouble();
    }

    private double KelvinToCelsius(double kelvins)
    {
        return kelvins - 273.15;
    }

    public async Task<double> GetTemperatureAsync(int cityId)
    {
        var jsonStream = await MakeOpenWeatherRequest(cityId);
        var temperatureInKelvins = await ParseResponse(jsonStream);
        return KelvinToCelsius(temperatureInKelvins);
    }
}
```

# Orų patikrinimas OpenWeatherAPI asinchroniškai

```
    var cities = new Dictionary<int, string>
    {
        {598098, "Klaipėda"},
        {593116, "Vilnius"},
        {596128, "Panevėžys"},
        {597231, "Marijampolė"},
        {598316, "Kaunas"},
    };
    var tasks = cities
        .Select(async cityData =>
        {
            var (cityId, cityName) = cityData;
            var response = await api.GetTemperatureAsync(cityId);
            return new Tuple<string, double>(cityName, response);
        })
        .ToList();
    var responses = await Task.WhenAll(tasks);
    foreach (var response in responses)
    {
        var (cityName, temperature) = response;
        Console.WriteLine($"{cityName}: {temperature:F2}");
    }
}
```

# Kas vyksta asinchroniniame metode

```
{
    const string apiRoot = "http://api.openweathermap.org/";
    using var client = new HttpClient();
    return await client.GetStreamAsync(
        $"{apiRoot}data/2.5/weather?id={cityId}&appid={_apiKey}"
    );
}
```

1. Vykdomas kodas iki `await` sakinio
2. Paleidžiamas `GetStreamAsync` vykdymas
3. `GetStreamAsync` vykdymo metu reikia laukti, kol grįš atsakymas iš serverio – kontrolė grąžinama `MakeOpenWeatherRequest` metodui.
4. `MakeOpenWeatherRequest` metodas naudoja `await`, dėl to kontrolė grąžinama kviečiančiam metodui
5. Kai kviečiantis metodas nebeturi jokio darbo ir `GetStreamAsync` jau grąžino reikšmę, tęsiamas `MakeOpenWeatherRequest` vykdymas.

# Programavimas naršyklėje

- Naršyklėje atvaizuojamas turinys aprašomas HTML.
- Aprašyto HTML stilius aprašomas CSS.
- Užkrautą HTML turinį galima modifikuoti naudojant JavaScript programavimo kalbą.

# JavaScript

- Kalba sukurta 1995 m.
- Vėliau kalbos standartizavimą perėmė ECMA.
- ECMA formalizavo kalbos specifikaciją ECMAScript, o JavaScript yra ECMAScript realizacija.
- JavaScript buvo sukurta naudoti naršyklėje.
- Atsiradus ES6 standartui JavaScript kalba pradėta naudoti serveriuose.
- Pagrindinė implementacija — Node.
- Kalba daugiausia naudojama saityno serveriuose, bet galima kurti ir kitokias programas.

# JavaScript kalbos bruožai

- JavaScript buvo kuriama kaip kalba papildyti Java. Java turėjo būti naudojama serveryje, JavaScript — naršyklėje. Dėl to panašūs kalbų pavadinimai bei sintaksė.
- JavaScript yra dinamiškai tipizuota kalba, kintamieji paskelbiami naudojant raktinius žodžius `let` ir `const`.
- JavaScript palaiko objektinį programavimą, bet objektus galima kurti ir neturint klasės.
- Javascript palaiko funkcinį programavimą — masyvai turi `map`, `filter`, `reduce` metodus, funkcijas galima perduoti kaip parametrus kitoms funkcijoms.

# Įvykiais paremtas programavimas

- JavaScript programavimas naršyklėje remiasi įvykių modeliu — yra aibė įvykių, kuriuos sukelia tam tikri veiksmai, kuriems įvykus galima vykdyti norimas funkcijas.
- Įvykiams veiksmai priskiriami iškviečiant norimo įvykio `addEventListener` funkciją.
- Tokią sąsają teikia naršyklė, pvz., priskirti funkciją, kuri turi būti vykdoma paspaudus mygtuką.
- Kitokiems veiksmams taikomas atgalinio iškvietimo (*callback*) modelis — funkcija atlieka tam tikrą veiksmą ir po to įvykdo parametru perduotą funkciją (*callback*).
- Toks modelis taikomas, pvz., darant asinchroninę užklausą į serverį — gavus atsakymą reikia įvykdyti norimą funkciją.

# ECMAScript 6 — *Promise*

- ES6 standartas palengvino darbą su asinchroninėmis funkcijomis.
- Buvo pridėtas API, panašus į C# *Task* modelį.
- Asinchroninės funkcijos rezultatą, kuris dar nebūtinai suskaičiuotas, savyje turi `Promise` objektas.
- Darbas su `Promise` rezultatu vykdomas tam objektui kviečiant metodą `then`.
- `then` metodas grąžina naują `Promise` objektą, todėl `then` kreipinius galima jungti į grandinę, panašiai kaip C# `ContinueWith`.

# Promise

## Konstruktorius `Promise(fn)`

`fn` — funkcija, priimanti parametrus `resolve` ir `reject` (neprivalomas). Suskaičiuotas `fn` rezultatas perduodamas į `resolve`, įvykusi klaida — į `reject`.

## `resolve(value)`

Sukuria `Promise` objektą, kurio rezultatas bus konstanta `value`.

## `then(fn)`

Funkcija `fn` priima 1 parametrą — `Promise` suskaičiuotą reikšmę, ir su ja atlieka norimus veiksmus. `fn` grąžintas rezultatas bus įdėtas į naują `Promise` ir pasiekiamas tęsiant `then` grandinę.

# Promise

**`catch(err)`**

Apdoroja *promise* grandinės vykdymo metu įvykusias klaidas.

# Promise

```
const generateNumbers = new Promise(resolve => {
    setTimeout(() => {
        resolve([...Array(100000).keys()]);
    }, 1000);
});

generateNumbers.then(numbers => {
    return numbers
        .map(num => num * num)
        .reduce((acc, num) => acc + num);
}).then(squaredSum => {
    console.log(squaredSum);
}).catch(err => {
    console.log(err);
});
```

# *Promise*

- Naudojant *Promise* nedidėja funkcijų sudėjimo gylis.
- Lengviau pačiam susikurti asinchroninę funkciją.
- Vis tiek reikia naudoti *callback* funkcijas `then` parametrams.

# ES8: async / await

- ECMAScript 8 standartas (2017 m.) pridėjo `async` ir `await` raktinių žodžių palaikymą.
- Raktinis žodis `async` yra rašomas funkcijos apibrėžime. Jei funkcija pažymėta `async`, jos grąžinama reikšmė automatiškai įdedama į `Promise` ir tas `Promise` objektas yra grąžinamas.
- Raktinis žodis `await` yra rašomas prieš `async` funkcijos kreipinį. Jis palaukia, kol `async` funkcijos grąžintas `Promise` taps suskaičiuotas ir paima jo reikšmę.
- `await` galima naudoti **tik** `async` funkcijose.

# ES8: async / await

- Dirbant naršyklėje visos įvesties ir išvesties operacijos (kreipimaisi į serverį ir pan.) turėtų būti asinchroninės.
- Naudojant JavaScript ar kitą kalbą, palaikančią `async` / `await` raktinius žodžius serveryje ar asmeniniame kompiuteryje tokios operacijos gali būti ir darbas su disku ir pan.
- `async` pažymėta funkcija grąžins `Promise` objektą, t.y., jei funkcija yra `async`, tai jos rezultatui galima tiek kviesti `then` metodą, tiek naudoti `await` raktinį žodį.

# async / await

```
async function generateNumbers(count) {
    return [...Array(count).keys()];
}

async function getSquaredAsync() {
    try {
        const numbers = await generateNumbers(10000000);
        const squaredSum = numbers
            .map(num => num * num)
            .reduce((acc, num) => acc + num);
        console.log(squaredSum);
    } catch (e) {
        console.error(e);
    }
}

getSquaredAsync().then(()  => {
    console.log("Finished");
});
console.log("Calculating");
```

# async / await

- Naudojant `async` / `await` nebelieka *callback* funkcijų.
- Klaidų apdorojimas tampa analogiškas nuosekliam kodui: įvykus išimtinei situacijai galima ją suvaldyti su *try-catch* bloku.
- JavaScript, asinchroninės funkcijos vykdymo metu įvykus išimtinei situacijai, grąžina tą pačią klaidą, o ne įdeda į kitą klaidą kaip C#.
- Kodą lengva skaityti, nes jis panašus į nuoseklų kodą.

# JavaScript

- JavaScript varikliai palaiko tik vieną giją, tad asinchroninės funkcijos vykdomos pakaitomis įvykių cikle, o ne lygiagrečiai vienu metu.
- Dėl šios priežasties asinchronines funkcijas JavaScript verta naudoti tada, kai jų vykdymas blokuoja giją, nes reikia laukti, pvz., atsakymo iš serverio.

# Gijos naršyklėje

- Asinchroninis programavimas naršyklėje leidžia išnaudoti vieną CPU giją.
- Naršyklė suteikia programuotojui sąsają kurti gijas.
- Gijos kuriamos pasinaudojant `WebWorker` sąsaja.
- `WebWorker` nėra ECMAScript standarto dalis, jį aprašo W3C ir WHATWG.

# `WebWorker`

- `WebWorker` tipo objektas turi sąsają kurti gijoms ir apsikeisti žinutėmis tarp pagrindinės gijos ir sukurtos su `WebWorker`.
- Gijoje paleistas kodas turi apribojimų, pvz., jis negali dirbti su naršyklėje rodomu turiniu, kodas turi būti atskirame JavaScript faile.

# WebWorker sąsaja

### WebWorker(script)

Sukuria WebWorker, kurio vykdomas kodas nurodytas faile script. Faile turi būti aprašytas kintamasis pavadinimu onmessage, tipas — funkcija, priimanti įvykį su žinute.

### onmessage(event)

Metodas, kuris priima žinutę iš WebWorker.

### postMessage(message)

Siunčia žinutę tarp gijų. Jei kviečiamas kaip WebWorker metodas, siunčia žinutę tam WebWorker, jei kviečiamas iš WebWorker skripto, siunčia žinutę iš jo į pagrindinę giją.

# WebWorker sumos skaičiavimas: pagrindinė gija

```
async function getSumAsync(numbers) {
    return new Promise(resolve => {
        const threadCount = 8;
        const chunkSize = numbers.length / threadCount;
        const partialSums = [];
        for (let i = 0; i < threadCount; i++) {
            const chunkStart = i * chunkSize;
            const chunkEnd = (i + 1) * chunkSize;
            const worker = new Worker("worker.js");
            worker.onmessage = event => {
                partialSums.push(event.data);
                if (partialSums.length === threadCount) {
                    resolve(partialSums);
                }
            };
            worker.postMessage(numbers.slice(chunkStart,
                                             chunkEnd));
        }
    })
}
```

# WebWorker sumos skaičiavimas: WebWorker gija

```
onmessage = function (event) {
    const numbers = event.data;
    const sum = numbers.reduce((acc, num) => acc + num);
    postMessage(sum);
};
```