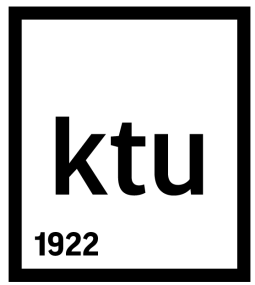
Gijos

Karolis Ryselis

Kauno Technologijos Universitetas



Some people, when confronted with a problem, think, "I know, Ill use threads", and then two they hav erpoblesms.

Paskaitos turinys

ktu 1922

Gijos programavime

- Procesas visada turi bent vieną giją. Jei pasibaigia visos proceso gijos, pasibaigia ir pats procesas.
- Procesas gali valdyti papildomas savo gijas — sukurti, palaukti, užbaigti, komunikuoti ir pan.
- Dauguma programavimo kalbų turi priemones gijų valdymui.

Gijos objektinėse programavimo kalbose

- Vienas iš būdų suteikti galimybę programuotojui valdyti gijas — naudoti gijų klases ir objektus.
- Vienas objektas atitinka vieną giją, gijos valdymui kviečiami to objekto metodai.
- Taip gijos vaizduojamos daugelyje programavimo kalbų, palaikančių objektinį modelį:
  - C++
  - Python
  - Java
  - Ruby
  - C#
  - kt.

Gijų valdymas naudojant objektus

Veiksmai, galimi su gijomis:

- Nurodyti gijai, kokius veiksmus atlikti (nurodoma funkcija ar metodas);
- Paleisti giją;
- Palaukti, kol gija baigs darbą;
- Patikrinti, ar gija dar dirba.

Tiesioginis gijų valdymas dažnai yra sudėtingas ir turėtų būti naudojamas tik tada, kai netinka kitokios priemonės.

Gijas atitinkančios klasės

C++ `std::thread`

C# `System.Threading.Thread`

Gijų metodai

	C++	C#
Sukūrimas	`thread t(fn)`	`new Thread(fn)`
Paleidimas	paleidžia konstruktorius	`t.Start()`
Laukimas	`t.join()`	`t.Join()`
Būsenos patikrinimas	—	`t.IsAlive`

Elementari programa C++

```
void execute(const string& name){
    cout << name << ": one" << endl;
    cout << name << ": two" << endl;
    cout << name << ": three" << endl;
}
```

Elementari programa C++

```
int main(){
    vector<string> names = {"First", "Second"};
    vector<thread> threads;
    vector<thread> threads(names.size());
    transform(names.begin(), names.end(), threads.begin(),
        [](auto &name) {
            return thread(execute, name);
        }
    );
    for_each(threads.begin(), threads.end(),
        mem_fn(&thread::join));
    cout << "Program finished execution" << endl;
    return 0;
}
```

Galimas programos rezultatas

```
First: oneSecond: one
First: two
First: three

Second: two
Second: three
Program finished execution
```

C++ gijos

- C++ gijos valdymas negali priklausyti dviems kintamiesiems vienu metu.
- Jei norėsime priskirti `thread` objektą į kitą kintamąjį, to neleis kompiliatorius.
- Tokiu atveju reikia naudoti funkciją `move`, kuri perkelia objektą iš vieno kintamojo į kitą.
- Po perkėlimo senasis kintamasis lieka nebenaudotinas.
- `thread t1 = move(t);`

C# gijos

- C# `Thread` klasės konstruktorius priima delegatus.
- Jei norime gijoje paleisti metodą be parametrų, galima patį metodą perduoti gijos konstruktoriui.
- Jei norime gijoje paleisti metodą su parametrais, galima patį metodą perduoti gijos konstruktoriui, o parametrus perduoti į `.Start()` kreipinį.
- Galima paduoti lambda funkciją.

Elementari programa

```
public class Executor
{
    private readonly string _name;
    public Executor(string name) => _name = name;

    public void Execute()
    {
        Console.WriteLine($"{_name}: one");
        Console.WriteLine($"{_name}: two");
        Console.WriteLine($"{_name}: three");
    }
}
```

Elementari programa

```
public static void Main()
{
    var names = new List<string> {"First", "Second"};
    var threads = names
        .Select(name =>
        {
            var executor = new Executor(name);
            return new Thread(executor.Execute);
        })
        .ToList();
    foreach (var thread in threads)
    {
        thread.Start();
    }
    foreach (var thread in threads)
    {
        thread.Join();
    }
    Console.WriteLine("Program finished execution");
}
```

Go paprogramės (*goroutines*)

- Go funkcijos gali būti iškviečiamos paprastai (nuosekliai) arba naudojant go sakinį.
- go sakiniu kviečiamos funkcijos vykdomos atskirose gijose, t. y., lygiagrečiai.
- go sakinys nelaukia, kol pasibaigs gijos vykdymas.

Go paprogramės

- go sakinys nieko negrąžina, todėl nėra objekto, kuriam būtų galima kviesti `join()` kaip objektinėse kalbose.
- Tokiam sinchronizavimui Go naudojami `WaitGroup`.
- `WaitGroup` objektui pranešama, kad užduotis įvykdyta iškviečiant `Done()`
- `WaitGroup` objektui iškvietus `Wait()` gija blokuojama, iki bus iškviestas laukiamas kiekis `Done` kreipinių.
- Laukiamų kreipinių kiekį galima nurodyti `Add()` funkcija, kuriai paduodamas parametras, kiek laukiamų `Done` kreipinių pridedama. Pvz., iškvietus `wg.Add(5)` programa lauks, kol tam pačiam `wg` kintamajam bus 5 kartus iškviesta `wg.Done()`.
- `Add()` funkciją galima kviesti kiek norima kartų, pvz., cikle.

Lygiagretumo pavyzdys

```
func main()  {
    var names = [2]string {"First", "Second"}
    var waitGroup = sync.WaitGroup{}
    waitGroup.Add(len(names))
    for _, name := range names {
        go execute(name, &waitGroup)
    }
    waitGroup.Wait()
    fmt.Println("Program finished execution")
}

func execute(name string, group *sync.WaitGroup)  {
    defer group.Done()
    fmt.Println(name + ": one")
    fmt.Println(name + ": two")
    fmt.Println(name + ": three")
}
```

Go privalumai

- Gijų valdymas Go kalboje skiriasi nuo objektinių kalbų, nes nėra jokio kintamojo, kuris yra susietas su gija.
- Jei gijos sinchronizuojamos naudojant `WaitGroup`, toks valdymo būdas yra lankstesnis. Pvz., jei turime gijoje vykdomą funkciją, kuri rašo į failą ir po to atlieka papildomus veiksmus, o pagrindinei gijai reikia palaukti tik įrašymo į failą, kad galėtų tęsti darbą, tą galima paprasčiau padaryti su Go.

Rust gijos

- Rust gijos kuriamos naudojant funkciją `thread::spawn`, kuriai paduodama funkcija, kuri bus vykdoma gijoje.
- Rust programavimo kalba akcentuoja patikimumą, todėl daugelis dalykų gijose neleidžiami, pvz., jei kintamasis būtų naudojamas ir pagrindinėje, ir sukurtoje gijoje, programa nesikompiliuos. Tokios apsaugos priverčia kurtis programas „teisinguoju“ būdu.
- `thread::spawn` funkcija grąžina `JoinHandle` kintamąjį, kuriuo naudojantis galima palaukti gijos darbo pabaigos.
- Jei gijos funkcija grąžina reikšmę, ją galima pasiekti kaip `JoinHandle::join()` rezultatą — tai didelis Rust gijų modelio privalumas.

Rust lygiagretumo pavyzdys

```
fn execute(name: &str) {
    println!("{}: one", name);
    println!("{}: two", name);
    println!("{}: three", name);
}

fn main() {
    let names = vec!["First", "Second"];
    let threads: Vec<JoinHandle<_>> = names
        .iter()
        .map(|name| {
            let owned_name = name.to_owned();
            thread::spawn(move || execute(owned_name))
        }).collect();
    threads
        .into_iter()
        .for_each(|t| t.join().unwrap());
    println!("Program finished execution.");
}
```

Nuosekliojo programavimo bruožai

- Programos sakiniai, užrašyti pirmiau, yra įvykdomi anksčiau, nei sakiniai, užrašyti vėliau.
- Programos rezultatai priklauso nuo išorinės būsenos (failų turinio, duomenų bazės turinio, laiko (pvz., `datetime.now()`)).
- Jei funkcija yra gryna (angl. *pure function*), rezultatas visada bus tas pats su tais pačiais parametrais.

Grynos funkcijos ir pašalinis poveikis (*side effects*)

- Funkcija vadinama gryna tada, kai ji neturi jokio pašalinio poveikio.
- Pašalinis poveikis — bet koks programos išorės modifikavimas ar išorinės būsenos patikrinimas.
- Funkcija, skaitanti iš failo nėra gryna, nes jos rezultatas priklauso nuo failo turinio.
- Grynos funkcijos visada grąžina tą patį rezultatą, jei buvo perduoti tie patys parametrai.

Lygiagrečiojo programavimo bruožai

- Programos lygiagrečiosios dalies sakiniai, užrašyti pirmiau, yra įvykdomi anksčiau, nei sakiniai, užrašyti vėliau tik **vienos gijos ribose**.
- Tarp skirtingų gijų sakiniai gali būti vykdomi kiekvieną kartą vis kita tvarka.
- Funkcija, paleidžianti gijas ar su jomis komunikuojanti, nėra gryna.
- Programos rezultatai gali priklausyti ne tik nuo išorinės būsenos, bet ir nuo programos vykdymo (operacinės sistemos, virtualios mašinos ir kt.).

Programų derinimas

- Lygiagrečiųjų programų derinimas dažnai būna sudėtingesnis, nei nuoseklių.
- Lygiagrečioje programoje rezultatas gali priklausyti nuo dar vieno papildomo faktoriaus – kokia tvarka ir kaip yra vykdomos gijos.
- Gijų vykdymas gali skirtis priklausomai nuo operacinės sistemos, esamų kompiuterio resursų, kitų veikiančių procesų ir kitų faktorių.
- Dėl šių priežasčių aptikti klaidas lygiagrečioje programoje kartais būna daug sunkiau, nei nuoseklioje.

Gijų sąveika

- Gijoms gali reikėti dirbti su tais pačiais resursais.
- Resursas gali būti tiek kintamasis, tiek išorinis objektas (pvz., konsolė, failas).
- Gijų prieigos prie resursų valdymas priklauso nuo pasirinkto atminties modelio — bendros ar atskiros atminties.

Bendra gijų atmintis

- Procesai komunikuoja per bendrus kintamuosius — „mato“ tą pačią atmintį.
- Pakeitus kintamojo reikšmę ar resurso būseną vienoje gijoje, pasikeitimai matomi ir kitose gijose.
- Kintamojo lygiagretus keitimas dažniausiai nepageidaujamas, reikia naudoti *sinchronizavimo* priemones.

Gijų sinchronizavimas

Gijų sinchronizavimas

Gijų veiksmų eilės tvarkos arba vykdymo laiko nustatymas.

- Galima nurodyti, kad kažkurį veiksmą vienu metu gali vykdyti **tik viena** gija.
- Galima nurodyti, kad gija turi laukti signalo prieš tęsdama darbą.

Sąvokos

Atominis veiksmas (angl. *atomic operation*)

Nedalomas veiksmas, į kurio vykdymą negali įsiterpti kitų veiksmų vykdymas.

- Veiksmas yra atominis, jei jis įvykdomas viena CPU instrukcija.
- Neatominį veiksmą galima padaryti atominiu kitų gijų atžvilgiu naudojant gijų sinchronizavimo priemones.

Kritinė sekcija (angl. *critical section*)

Programos kodo dalis, kuri kitų gijų atžvilgiu yra atominis veiksmas.

- Programuotojas kalbos ar bibliotekos priemonėmis nurodo kritinės sekcijos ribas.

Sąvokos

Tarpusavio išskyrimas (angl. *mutual exclusion*)

Priemonė, neleidžianti daugiau nei vienai gijai vienu metu naudotis tam tikru resursu.

- Kritinė sekcija išskiriama naudojantis tarpusavio išskyrimu.

Lenktynių sąlygos (angl. *race condition*)

Sąlygos, kai daugiau nei viena gija vienu metu siekia naudotis tuo pačiu resursu.

Sąvokos

Užimtas laukimas (angl. *busy wait*)

Būsena, kai gija laukia tam tikro resurso atlaisvinimo pastoviai tikrindama jo būseną.

- Jei užimtas laukimas realizuojamas tikrinant resurso būseną cikle, toks ciklas vadinamas besisukančiu užraktu (angl. *spin lock*).

Badavimas (angl. *starvation*)

Būsena, kai gijai pastoviai neleidžiama naudotis bendrais resursais.

- Tokios būsenos siekiama išvengti, nes badaujanti gija negali baigti savo darbo.

Sąvokos

Aklavietė (angl. *deadlock*)

Būsena, kai procesas laukia įvykio, kuris niekada neįvyks.

- Dažna aklavietės priežastis — dviems gijoms reikia dviejų tų pačių resursų. Pirma gija užrakina pirmą resursą, antra — antrą, tuomet pirma gija laukia antro resurso, antra — pirmo.