


1. Procesai ir gijos, jų skirtumai. (pirma tema)

Gijos	Gijos
Gijos ir proceso sąsaja	Gijos ir proceso sąsaja
<ul style="list-style-type: none">Procesas turi resursus programos vykdymui — adresų erdvę, vykdomą kodą, saugumo kontekstą, unikalių identifikatorių, prioritetą ir bent vieną vykdymo giją.Paleidus procesą jis paleidžiamas su viena vykdymo gija, bet gali susikurti daugiau vykdymo gijų.	<ul style="list-style-type: none">Gija yra proceso dalis. Visos vieno proceso gijos dalijasi virtualia adresų erdve ir sistemos resursais.Gijoje programos kodas vykdomas nuosekliai, bet gijos tarpusavyje gali būti vykdomos lygiagrečiai.
2/34	3/34

2. Konteksto prejungimas, gijų planavimas (pirma tema)

Gijos	Gijos
Gijų vykdymas - užduočių perjungimas	Gijų planavimas OS
<p>Jei procesorius turi vieną branduolį, gijos vykdomos po truputį. Operacinė sistema gijoms išskiria po laiko kvantą, jam pasibaigus perjungiamas kontekstas iš vienos gijos į kitą ir atgal, kol gijų darbas baigiasi.</p>	<ul style="list-style-type: none">Vienos programos gijos turi dalinti skaičiavimo branduolius su kitais procesais.Jei aktyvių procesų / gijų sistemoje veikia daugiau, nei yra CPU branduolių, vis tiek bus vykdomas užduočių perjungimas, nors vienas procesas gijų vykdo mažiau, nei CPU branduolių skaičius.
<p>Core 1</p> 	
5/34	6/34

3. Darbas su gijomis C++, Java arba C# (paleidimas, palaukimas). (pirma tema)

C++ gijos	Java gijos
Pagrindiniai <code>std::thread</code> metodai	Pagrindiniai Thread klasės metodai
<p>Konstruktorius <code>thread(fn, args)</code> Sukuria gijos objektą ir pradeda gijos vykdymą <code>fn</code> Gijoje vykdoma funkcija. <code>args</code> Funkcijai perduodamų argumentų sąrašas.</p>	<p>Konstruktorius <code>Thread()</code> Sukuria gijos objektą.</p> <p>Konstruktorius <code>Thread(Runnable target)</code> Sukuria gijos objektą, kuris vykdys <code>target</code> interfeisu aprašytus veiksmus.</p> <p><code>start()</code> Paleidžia naują giją, kurioje vykdoma tai, kas aprašyta <code>run()</code> metode. Metodas neturėtų būti realizuojamas, tik iškviečiamas.</p> <p><code>run()</code> Aprašo veiksmus, kurie turi būti vykdomi gijoje. Metodas turėtų būti realizuojamas, bet ne iškviečiamas.</p>
<p><code>join()</code> Sustabdo gijos, kurioje kviečiamas, darbą, kol pasibaigs gija, kurios <code>join()</code> metodas kviečiamas.</p> <ul style="list-style-type: none">Tokie metodai, kurie laukia tam tikro įvykio sustabdydami gijos vykdymą, vadinami <i>blokuojančiais</i> (angl. <i>blocking</i>).	<p><code>isAlive()</code> Patikrina, ar gijos objektas vis dar vykdomas, ar jau baigė darbą.</p> <p><code>join()</code> Sustabdo gijos, kurioje kviečiamas, darbą, kol pasibaigs gija, kurios <code>join()</code> metodas kviečiamas.</p>
10/34	24/34

Pagrindiniai Thread klasės metodai

Konstruktorius Thread (mt)

Sukuria gijos objektą.

mt - gijoje vykdomas metodas.

C #

Start ()

Pradedą gijos vykdymą.

Join ()

Sustabdo gijos, kurioje kviečiamas, darbą, kol pasibaigs gija, kurios `join()` metodas kviečiamas.

4. Darbas su gijomis OpenMP (parallel blokai, jų parametrai).

OpenMP	OpenMP
<ul style="list-style-type: none">• OpenMP – tai papildomos priemonės, suteikiančios galimybes kurti lygiagrečias programas Fortran, C, C++ kalbomis;• OpenMP skirta kurti lygiagrečias programas, kuriose procesai (gijos) naudojami bendra atmintimi;• OpenMP sudaro: kompiliatoriaus direktyvos, specialių funkcijų biblioteka, aplinkos kintamųjų rinkinys.	<ul style="list-style-type: none">• Nuo programos darbo pradžios iki pabaigos vykdoma pagrindinė gija (angl. <i>master thread</i>).• Naujos gijos kuriamos kompiliatoriaus direktyvomis.• Vienu metu vykdomų gijų rinkinys sudaro lygiagrečią sritį (angl. <i>parallel region</i>).
<ul style="list-style-type: none">• Lygiagrečioje srityje vykdomų gijų skaičių galima nurodyti programos vykdymo metu.• Visos lygiagrečios srities gijos vykdo tą patį struktūrinį bloką.• Visos lygiagrečios srities gijos pradedamos vykdyti tuo pačiu metu ir vykdomos lygiagrečiai.• Lygiagreti sritis baigiama vykdyti, kai baigiamos vykdyti visos tos srities gijos.• Kiekvienai lygiagrečios srities gijai suteikiamas unikalus numeris; <i>master</i> gijos numeris – 0.• Programos vykdymo metu nėra galimybių keisti gijos numerį.	<p>Direktyva <code>#pragma omp parallel</code> Nurodo lygiagrečiai vykdomą sritį. Neprivalomu atributu <code>num_threads(n)</code> galima nurodyti vykdomų gijų skaičių <code>n</code>.</p> <p>Funkcija <code>omp_set_num_threads(n)</code> Nurodo, kiek gijų bus vykdomas <code>parallel</code> blokas.</p> <p>Funkcija <code>omp_get_num_threads()</code> Grąžina, kiek gijų bus vykdoma <code>parallel</code> bloke.</p> <p>Funkcija <code>omp_get_thread_num()</code> Grąžina gijos, kurioje iškviesta, numerį.</p>

5. Sugebėti paleisti gijas (užrašyti kodą) OpenMP ir C++, Java arba C#.

6. Nuosekliojo ir lygiagrečiojo programavimo skirtumai (sakinių, vykdymo tvarka, grynų funkcijų rezultatai).

Lygiagrečiojo programavimo bruožai

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.
- 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.
- Programos rezultatai gali priklausyti ne tik nuo išorinės būsenos, bet ir nuo programos vykdymo (operacinės sistemos, virtualios mašinos ir kt.).
- Grynų funkcijos ne visada gali grąžinti tą patį rezultatą.
- Programos specifikaciją gali atitikti tik kai kurie rezultatai.

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.
- Nuoseklioje programose grynos funkcijos visada grąžina tą patį rezultatą, jei buvo perduoti tie patys parametrai.

7. Duomenų lygiagretumas ir funkcinis lygiagretumas, su jais susijusios problemos.

Duomenų lygiagretumas

Funkcinis lygiagretumas

- Lygiagretumo modelis, kai visos gijos daro tą patį su skirtingais duomenų poalbiais.
- Pagrindinė sprendžiama problema - su dideliu kiekiu elementų reikia atlikti tuos pačius veiksmus.
- Apdorojant elementus lygiagrečiai gaunamas pagreitėjimas, nes vienu metu apdorojamas ne vienas elementas, o daugiau.
- Lygiagretumo modelis, kai visos gijos atlieka skirtingus veiksmus su tais pačiais duomenimis.
- Pagrindinė sprendžiama problema - su dideliu kiekiu duomenų reikia atlikti keletą skirtingų veiksmų.
- Atliekant keletą veiksmų lygiagrečiai gaunamas pagreitėjimas, nes vienu metu vykdomas ne vienas veiksmas, o keletas.

Lygiagretumo variantai	Lygiagretumo variantai
Duomenų lygiagretumo problemos	Funkcinio lygiagretumo problemos

- Kiek gijų sukurti? Kai gijų per mažai, skaičiavimai vyksta lėčiau, kai gijų per daug - vyksta konteksto perjungimas, gijų valdymas sunaudoja didelę dalį skaičiavimų laiko.
- Kaip išdalinti duomenis gijoms? Jei duomenys išdalinti nelygiai - vienos gijos skaičiavimai ilgiau, kitos - trumpiau; skirtingų elementų apdorojimo laikas gali būti skirtingas.
- Kaip valdyti pašalinius gijų vykdomų funkcijų poveikius, jei gijose vykdomos ne grynos funkcijos ir naudoja tuos pačius resursus?
- Kaip išlygiagretinti skaičiavimus, kai vieno veiksmo rezultatai priklauso nuo kito veiksmo rezultatų? Jei veiksmas priklauso vienas nuo kito, gali kilti sunkumų.
- Kaip išskirti tokius veiksmus, kurie gali būti atliekami lygiagrečiai? Veiksmus lengviau lygiagretinti tada, kai jie nepriklausomi.

8. Kada nenaudoti lygiagrečiojo programavimo.

Kada nenaudoti lygiagrečiojo programavimo?

Kai gaunamas programos pagreitis nevertas:

- sudėtingesnio programos palaikymo;
- didesnių kūrimo kaštų;

Kiti faktoriai:

- OS gijų valdymas užima laiko;
- Per trumpas vienos gijos užduoties vykdymo laikas;
- Per daug gijų gali išnaudoti visus sistemos resursus (gijos dėklas užima ~1MB atminties);
- Per dažnas konteksto perjungimas užima laiko;

9. Bendra atmintis ir paskirstyta atmintis — kas tai yra, kokios problemos kyla, kada naudinga, kaip apsieikiama duomenimis, kaip vykdoma sinchronizacija.

Bendra procesų atmintis

- Procesai komunikuoja per bendrus kintamuosius — “mato” tą pačią atmintį.
- Procesai gali būti sinchronizuojami naudojant bendrus kintamuosius.
- Problema: bendrų kintamųjų apsauga.

Atskira procesų atmintis

- Procesai komunikuoja siųsdami ir priimdami pranešimus.
- Procesai sinchronizuojami siunčiant sinchronizavimo signalus.
- Problema: priimti pranešimus iš kelių procesų.

Informacijos apsikeitimas tarp procesų

Bendra procesų atmintis

- Neapibrėžtas rezultatas gaunamas, kai veiksmas, kurie turėtų būti atominiai kitų gijų atžvilgiu, tokie nėra.
- Pavyzdyje viena ciklo iteracija turėtų būti atominis veiksmas kitų gijų atžvilgiu.
- Vienas iš būdų tai padaryti - pritaikyti kritinės sekcijos apsaugą.
- Kritinės sekcijos apsauga garantuos **teisingą rezultatą**, bet **sulėtina programą**, nes kritinę sekciją vienu metu vykdo viena gija, o kitos laukia.

- Per bendrus kintamuosius. Gijos turi prieigą prie tų pačių kintamųjų, keičia jų būseną, o pasikeitimus mato kitos gijos.
- Apsieičiant žinutėmis. Gijos viena kitai siunčia žinutes.

Bendri kintamieji	Procesų sąveika
Tarpusavio išskyrimas	Lygiagrečiųjų procesų sinchronizavimas

- Vienalaikiai skaitymo veiksmas netrukdo vienas kitam.
- Jei dvi gijos vienu metu rašo į kintamąjį, tai rezultatas — kuri nors iš rašomų reikšmių (bet ne jų junginys).
- Jei viena gija rašo, o kita — skaito, tai skaitanti gija mato arba seną, arba naują reikšmę (bet ne jų junginį).

- Procesai laukia įvykio kartodami veiksmus (besisukantis užraktas).
- Procesai laukia nevykdydami jokių veiksmų (blokuojami).
- Procesų blokavimas yra geresnis sunaudojamų sistemos resursų atžvilgiu.

10. Sąvokos (atominis veiksmas, kritinė sekcija, tarpusavio išskyrimas, lenktynių sąlygos, užimtas laukimas, badavimas, aklavietė).

Sąvokos

Atominis veiksmas (angl. *atomic operation*)

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

- Veiksmas yra atominis, jei jis įvykdomas viena CPU instrukcija.
- Neatominį veiksmą galima padaryti atominių 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

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

Tarpusavio išskyrimas (angl. *mutual exclusion*)

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

- Kritinė sekcija išskiriama naudojanti tarpusavio išskyrimu.

Lenktynių sąlygos (angl. *race condition*)

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

Sąvokos

Aklavietė (angl. *deadlock*)

Būsena, kai procesas laukia resurso, kuris niekada neatsilaivins.

- Dažna aklavietės priežastis — dviems gijoms reikia dviejų tų pačių resursų. Pirmą giją užrakina pirmą resursą, antrą — antrą, tuomet pirmą giją laukia antro resurso, antrą — pirmo.

11. Kritinė sekcija — kodėl ją reikia saugoti, kaip reikia saugoti.

Kritinė sekcija

Kritinė sekcija (angl. *critical section*)

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

Tarpusavio išskyrimas (angl. *mutual exclusion*)

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

- Jei procesai turi bendrą kintamąjį, tai veiksmai su šiuo kintamuoju abiejuose procesuose turi būti paskelbiami kritinėmis sekcijomis ir atliekami su tarpusavio išskyrimu.
- Jei procesai P_1 ir P_2 turi bendrą kintamąjį k_1 , o procesai P_2 ir P_3 — bendrą kintamąjį k_2 , tai proceso P_1 veiksmai su k_1 ir proceso P_2 veiksmai su k_2 gali būti atliekami be tarpusavio išskyrimo.

Be kritinės sekcijos apsaugos

- Visi gijų objektai modifikuoja tą patį duomenų klasės objektą.
- Prieiga prie modifikuojamo kintamojo nėra apsaugota.
- Programos rezultatas yra **neapibrėžtas**.

- Sąlyginis kintamasis — gijų, laukiančių įeiti į kritinę sekciją, rinkinys.
- Naudojantis sąlyginiais kintamaisiais galima užtikrinti kritinės sekcijos apsaugą arba leisti gijoms laukti įvykio.

Daugiau info apie kritinės sekcijos apsaugą yra 5-temos skaidrėse...

12. Petersono algoritmas.

Petersono algoritmas

```
class Locker {
    private boolean[] flag = {false, false};
    private int turn;
    void setFlag(int i) {
        flag[i] = true;
    }
    void resetFlag(int i) {
        flag[i] = false;
    }
    int getTurn() {
        return turn;
    }
    void setTurn(int turn) {
        this.turn = turn;
    }
    boolean flagIsSet(int i) {
        return flag[i];
    }
}
```

Petersono algoritmas

```
public class DemoThread extends Thread {
    private Data data;
    private int id;
    private Locker locker;
    DemoThread(Data d, int id, Locker locker){
        this.data = d;
        this.id = id;
        this.locker = locker;
    }
    @Override
    public void run() {
        int turnNumber = 1 - id;
        for (int i = 0; i < 15; i++) {
            locker.setFlag(id);
            locker.setTurn(turnNumber);
            while (locker.flagIsSet(turnNumber) &&
                locker.getTurn() == turnNumber) {}
            int counter = data.getCounter();
            counter++;
            data.setCounter(counter);
            locker.resetFlag(id);
        }
    }
}
```

Petersono algoritmas

- Užtikrina tarpusavio išskyrimą.
- Naudoja besisukantį užraktą — švaistomi CPU resursai.
- Veikia tik su dviem gijom.

13. Bakery algoritmas.

Bakery algoritmas

```
class Locker {
    private volatile boolean[] choosing;
    private volatile int[] ticket;
    Locker(int threadCount) {
        choosing = new boolean[threadCount];
        ticket = new int[threadCount];
        for (int i = 0; i < threadCount; i++) {
            choosing[i] = false;
            ticket[i] = 0;
        }
    }
    void setChoosing(int k) {choosing[k] = true;}
    void unsetChoosing(int k) {choosing[k] = false;}
    void setTicket(int k) {
        ticket[k] = Arrays.stream(ticket).max().getAsInt() + 1;
    }
    void unsetTicket(int k) {ticket[k] = 0;}
    /*...*/
}
```

Bakery algoritmas

```
/*...*/
void wait(int k) {
    for (int i = 0; i < choosing.length; i++) {
        if (i == k) {
            continue;
        }
        while (choosing[i]) ;
        while (ticket[i] != 0 && ( ticket[k] > ticket[i] ||
            (ticket[k] == ticket[i] && k > i))) {}
    }
}
```

Bakery algoritmas

```
public class DemoThread extends Thread {
    private Data data;
    private int id;
    private Locker locker;
    DemoThread(Data d, int id, Locker locker) {
        this.data = d;
        this.id = id;
        this.locker = locker;
    }
    @Override
    public void run() {
        for (int i = 0; i < 15; i++) {
            locker.setChoosing(id);
            locker.setTicket(id);
            locker.unsetChoosing(id);
            locker.wait(id);
            int counter = data.getCounter();
            counter++;
            data.setCounter(counter);
            locker.unsetTicket(id);
        }
    }
}
```

Bakery algoritmas

- Prieš įeinant į kritinę sekciją procesas gauna skaičių. Tas, kuris turi mažiausią skaičių, įleidžiamas į kritinę sekciją.
- Jei du procesai turi tą patį skaičių, į kritinę sekciją įleidžiamas tas, kurio eilės numeris mažesnis.
- Procesas, skaičiaus paprašęs anksčiau, negali gauti didesnio skaičiaus už tą, kuris paprašė vėliau.
- Algoritmas pirmiau aptarnauja tuos procesus, kurie pirmi pareiškė norą įeiti į kritinę sekciją.

14. Monitorius — kas tai yra, kam skirtas, kaip veikia, kokie pavojai kyla su juo dirbant.

- Monitorius bendrąja prasme — įtaisas kažkam stebėti, testuoti, užrašinėti.
- Programinėje įrangoje terminą suformavo C. A. Hoare 1974 m.
- Kritinis regionas, monitorius — programos kodo dalis, visada vykdoma su tarpusavio išskyrimu.
- Kiekvienas Java objektas turi su juo susietą monitorių.
- Kiekviena Java klasė turi su ja susietą monitorių.
- Monitorius nerealizuojamas, jei nenaudojami sinchronizuoti (angl. *synchronized*) metodai.
- Monitorius – tai užraktas (angl. *lock*), nustatantis objekto (klasės) panaudojimo tvarką ir vienu metu leidžiantis jį naudoti tik vienai gijai.



Monitorių sudaro:

- bendri saugomi duomenys;
- atominių veiksmų, skirtų duomenų apdorojimui, rinkinys;
- sąlyginių kintamųjų rinkinys.

- Tarpusavio išskyrimas (*mutual exclusion*) - "užrakinant" objektą su *synchronized* (realizuoja JVM).
- Sąlyginė sinchronizacija (*conditional synchronization*) — naudojant *Object* klasės *wait*, *notify*, *notifyAll* metodus.

Java monitorių veikimas

- JVM naudoja "signal-and-continue" ("wait-and-notify") tipo monitorius.
- Gija, naudojanti monitorių, gali sustabdyti save vykdydama `wait` metodą.
- Gija, vykdydama `wait`, palieka monitorių ir pereina į laukiančiųjų eilę (*wait set*).
- Gija lieka laukiančiųjų eilėje iki tol, kol kita gija neįvykdys `notify` metodo.
- Gija, įvykdžiusi `notify`, lieka monitoriuje.
- Laukianti gija gali pereiti į monitorių po to, kai gija, įvykdžiusi `notify`, palieka monitorių.

15. Sąlyginiai kintamieji — kas tai yra, kam skirti, kaip veikia.

- Sąlyginis kintamasis — gijų, laukiančių įeiti į kritinę sekciją, rinkinys.
- Naudojantis sąlyginiais kintamaisiais galima užtikrinti kritinės sekcijos apsaugą arba leisti gijoms laukti įvykio.

16. Mokėti naudotis C++, Java arba C# kritinės sekcijos apsaugos mechanizmais bei sąlygine sinchronizacija.

17. Kritinės sekcijos apsauga OpenMP — mokėti naudotis.

18. OpenMP bendri ir privatūs kintamieji.

OpenMP bendri kintamieji

- Kompiuterio atmintyje yra tik viena bendrojo (*shared*) kintamojo kopija. Bendrasis kintamasis matomas kiekvienoje gijų rinkinio gijoje.
- Vienoje gijoje pakeista bendrojo kintamojo reikšmė gali būti matoma kitoje gijoje.
- Jei nenurodyta kitaip, visi programos kintamieji yra bendri visoms lygiagrečios srities gijoms.

`private` parinktis

Parinktis nurodo, kad sukuriami po vieną kintamojo kopiją kiekvienai gijai; pradinė reikšmė – numatytoji to kintamojo tipo konstruktoriuje (gali būti ir neapibrėžta).

`firstprivate` parinktis

Parinktis skiriasi nuo `private` tuo, kad į kiekvieną giją kopijuojama kintamojo reikšmė naudojant kopijos konstruktorių.

OpenMP privatūs kintamieji

- Privatus (*private*) kintamasis turi savo kopijas kiekvienoje gijoje. Kiekviena kopija matoma tik vienoje gijoje.
- Vienoje gijoje pakeista privataus kintamojo reikšmė nematoma kitose gijose.
- Kintamieji yra privatūs trimis atvejais:
 - lygiagrečiojo (`for`) ciklo indeksas yra privatus;
 - lygiagrečios srities bloke paskelbti lokalūs kintamieji yra privatūs;
 - visi kintamieji, išvardinti `pragma omp directive` kaip `private`, `firstprivate`, `lastprivate` arba `reduction`, yra privatūs.

`lastprivate` parinktis

Parinktis skiriasi nuo `private` tuo, kad paskutinėje iteracijoje ar sekcijoje gauta reikšmė kopijos priskyrimo operatoriumi perduodama į pagrindinę giją.

`reduction` parinktis

Parinktis skiriasi nuo `private` tuo, kad kartu su kintamuoju perduodamas ir operatorius; `reduction` kintamasis turi būti skaliarinis kintamasis, inicializacijos metu įgyja reikšmę, numatytą tam operatoriui. Bloko gale `reduction` operatorius pritaikomas visoms kopijoms ir pradinei kintamojo reikšmei.

19. Kitos OpenMP direktyvos (parallel for, barrier, master).

OpenMP `pragma omp for`

- Direktyva nurodo kompiliatoriui paskirstyti ciklo operacijas gijoms.

OpenMP `pragma omp barrier`

- Direktyva nurodo sinchronizacijos tašką, ties kuriuo visos gijos lygiagrečiame regione laukia, kol kitos gijos toje sekcijoje pasieks tą patį tašką.
- Tolesni veiksmai po barjero tęsiami lygiagrečiai.

OpenMP `pragma omp master`

- Nurodo, kad kodo sekcija turi būti vykdoma tik pagrindinėje (*master*) gijoje.

20. Asinchroninių užduočių ir gijų palyginimas — kuris variantas kokius pranašumus turi.

Gijų naudojimo trūkumai

Asinchroninės užduotys

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 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, ir po to palaukti asinchroninės užduoties rezultato.

21. Asinchroninių užduočių programavimo bruožai — rezultatų paėmimas, išimčių valdymas.

Išimtinės situacijos naudojant asinchronines užduotis

- Jei asinchroninės užduoties funkcija išmeta išimtinę situaciją, ta pati išimtinė situacija išmetama ir kviečiančioje gijoje. Jos valdymas tampa toks pat, kaip ir nuosekliame kode.
- Išimtinė situacija atkartojama iškvietus `future::get()` metodą.

```
int main() {
    auto task = async(square_root, -1);
    double result;
    try {
        result = task.get();
    } catch (out_of_range& err) {
        cerr << err.what() << endl;
        return 1;
    }
    cout << result << endl;
    return 0;
}

double square_root(double x) {
    if (x < 0) {
        throw out_of_range("x < 0");
    }
    return sqrt(x);
}
```

22. Funkcinis programavimas ir lygiagretusis programavimas — kodėl gerai juos naudoti kartu, kokie trūkumai.

- Funkcinis programavimas — programavimo paradigma, kurioje funkcijos grąžinami rezultatai priklauso vien tik nuo jos parametrų ir nepriklauso nuo išorinės būsenos.
- Tokios funkcijos su tais pačiais parametrais **visada** grąžins tuos pačius rezultatus.
- Grynos funkcijos dar ir *nekeičia* išorinės būsenos, visas funkcijos poveikis yra grąžinama reikšmė.
- Funkcinis programavimas — programavimo ideologija, kai viskas yra funkcija ir funkcijų pašaliniai poveikiai yra aiškiai atskirti nuo skaičiavimų logikos.
- Programa įsivaizduojama kaip funkcija, turinti parametrus ir pagal juos apskaičiuojanti reikšmę.
- Funkcinis programavimas yra **deklaratyvus** — programuotojas aprašo, **ką** nori padaryti, bet ne **kaip** padaryti.
- Vienas iš skiriamųjų funkcinio programavimo bruožų — visos duomenų struktūros yra nekeičiamos (*immutable*).
- Naudojant grynas funkcijas lengviau realizuoti lygiagrečią sistemą, nes jei nėra bendrų resursų keitimo, tai nelieka lenktynių sąlygų, dėl to nereikia naudoti užraktų.
- Toks programavimo stilius eliminuoja visą kategoriją galimų padaryti klaidų, susijusių su prieigos prie bendros atminties valdymu.

23. Java ir JavaScript asinchroninių užduočių modelis(CompletableFuture ir Promise) — kaip naudojama, kaip konstruojamos užduočių grandinės, kaip valdomos išimtinės situacijos.

Pagrindiniai CompletableFuture metodai

`get()`

Blokuoja giją ir laukia `CompletableFuture` rezultato.

`complete(T result)`

Rankiniu būdu įvykdo asinchroninę užduotį — galima naudotis visa asinchroninių skaičiavimų sąsaja. Asinchroninės operacijos rezultatas — parametras `result`.

`runAsync(Runnable runnable)`

Metodas priima `Runnable` interfeisą realizuojantį objektą arba lambda funkciją ir paleidžia asinchroniškai. Metodas yra statinis ir grąžina `CompletableFuture` objektą, kurio pabaigos galima laukti naudojant `get()` metodą. Šis metodas skirtas operacijoms, kurios negrąžina rezultato.

`supplyAsync(Supplier supplier)`

Metodas priima `Supplier` tipo objektą arba lambda funkciją ir paleidžia asinchroniškai. Metodas yra statinis ir grąžina `CompletableFuture` objektą. `get()` metodas grąžins asinchroninės operacijos rezultatą.

`thenApply(Function function)`

Metodas kviečiamas `CompletableFuture` objektui, sukurtam naudojantis `supplyAsync`. Parametru perduodama funkcija, kuri priima `CompletableFuture` grąžinamo tipo parametą. Metodas grąžina naują `Future` objektą, todėl galima sudaryti `thenApply` kreipinių grandinę.

PGR CompatableFuture metodai:

Priima bet kokią kiekį `CompletableFuture` objektų ir grąžina naują `CompletableFuture` objektą, kuris bus įvykdytas, kai bus įvykdytas bent vienas objektas ir pirmojo pabaigusio reikšmė bus priinama per grąžintą `CompletableFuture`.

Leidžia apjungti du nepriklausomus `CompletableFuture` objektus ir suteikia galimybę įvykdyti funkciją, kai jie abu pabaigia darbą.

Priima funkcija, kuri bus iškviesta nepriklausomai nuo to, ar įvyko klaida, ar ne.

26. Go lygiagretumo modelis — kaip Go realizuotas CSP modelis, kaip paleidžiamos gijos, žinučių priėmimas iš kelių kanalų.

- Go naudojamas CSP modelis (*Communicating sequential processes*).
- Viena gija yra vienas nuoseklus procesas, o tarpusavyje jie veikia lygiagrečiai.
- Gijos viena kitai siunčia žinutes pasinaudojant kanalais (*channels*).
- Kanalas yra žinučių siuntimo mechanizmas. Gijos gali į kanalą rašyti arba iš kanalo skaityti duomenis.
- Procesai dalinasi vienu bendru resursu — kanalu.
- Jei procesas rašo duomenis į kanalą, kitas procesas turėtų iš jo skaityti. Programoje turėtų būti tiek kartų skaitoma iš kanalo, kiek kartų į jį yra rašoma.
- Go kanalai palaiko tiek sinchroninį, tiek asinchroninį žinučių siuntimą.
- Sinchroniniam žinučių siuntimui naudojami nebuferizuoti kanalai. Tokiu atveju rašanti gija yra blokuojama, iki žinutė bus gauta.
- Asinchroniniam siuntimui naudojami buferizuoti kanalai.

27. Superkompiuteriai — kas tai yra.

Mazgas *node*. Kompiuteris, turintis vieną ar daugiau skaičiavimo vienetų.

Klasteris *cluster*. Susijusių mazgų rinkinys.

Išteklių tinklas *grid*. Klasterių rinkinys.

Superkompiuteris *supercomputer*. Kompiuteris savo paleidimo metu esantis vienas pirmaujančių pasaulyje pagal skaičiavimo galią.

28. MPI paskirstytos atminties modelis — palyginti kanalus ir komunikatorių, kaip vykdomas point-to-point ir collective komunikavimas (principai), palyginti Go gijų ir MPI procesų modelius (kaip perduodami kintamieji procesams, kaip paleidžiama).

- MPI (*Message Passing Interface*) — pranešimų perdavimo funkcijų, skirtų realizuoti lygiagrečiuosius algoritmus, standartas.
- MPI realizacija — MPI priemonių biblioteka klasteriui, paskirstytis atminties superkompiuteriui ar heterogeniniam tinklui.
- MPI veikia principu SPMD (*Single Program Multiple Data*).
- Procesai yra programos kopijos ir sukuriama prieš programos vykdymą.
- Procesai neturi bendros atminties, bet vykdo tą patį kodą.
- Komunikacija tarp procesų vyksta siuntinėjant žinutes.
- MPI 1.0 išleista 1994 m.
- MPI 3.1.2 — paskutinė versija, išleista 2018 m. rugpjūtį.
- MPI yra standartas, turintis keletą realizacijų, tiek komercinių, tiek nekomercinių.
- Nekomercinės realizacijos:
 - OpenMPI (bus naudojama laboratorinių darbų metu).
 - LAM/MPI — toliau nebevystoma, kūrėjai prisidėjo prie OpenMPI.
 - MPICH
 - MP-MPICH
- **Tiesioginė komunikacija** (angl. *point-to-point communication*) — tokia komunikacija tarp procesų, kurioje dalyvauja **du** procesai: siuntėjas ir gavėjas.
- **Kolektyvinė komunikacija** (angl. *collective communication*) — tokia komunikacija tarp procesų, kurioje dalyvauja **visi** komunikatoriaus procesai.
- Kolektyvinė komunikacija sukuria sinchronizacijos tašką tarp procesų, t.y., visi procesai turi pasiekti tam tikrą tašką kode prieš tęsdami darbą.

29. MPI barjerai, Scatter, Gather — ką daro, kam skirta.

- Barjeras — vienas paprasčiausių kolektyvinės komunikacijos variantų.
- Barjeras skirtas procesų sinchronizacijai — kai procesas pasiekia barjerą, jis laukia, kol visi kiti procesai taip pat pasieks barjerą. Kai visi procesai pasiekia barjerą, galima tęsti darbą.
- MPI barjeras realizuotas siunčiant žetoną ratu — visą ratą galima apsukti tik tada, kai visi procesai jau pasiekė barjerą.

`Comm::Barrier()`

Funkcija, kurios vykdymas baigiamas, kai visi procesai įeina į barjerą.

- Scatter veikia panašiai, kaip `Bcast`, bet siunčia ne visiems procesams tuos pačius duomenis, o visiems skirtingą jų dalį.
- Scatter priima duomenų masyvą ir jo elementus paskirsto visiems procesams jų numerio didėjimo tvarka.
- Procesas, iš kurio siunčiama, taip pat gauna dalį duomenų, nepaisant to, kad jame yra visi duomenys.

Scatter duomenis, saugomus viename procese, persiunčia dalimis visiems komunikatoriaus procesams.

Gather duomenis, saugomus skirtinguose procesuose, surenka viename procese į bendrą duomenų rinkinį.

- Gather yra funkcija, atvirkščia Scatter.

- Gather priima kiekviename procese esantį masyvą ir jų elementus surašo į nurodytą nurodyto proceso masyvą.
- Duomenys bus paimami ir iš surenkančio proceso.

Scatter	Gather
<code>Comm::Scatter(void* send_data, int send_count, Datatype datatype, void* recv_data, int recv_count, Datatype recv_datatype, int root)</code>	<code>Comm::Gather(void* send_data, int send_count, Datatype datatype, void* recv_data, int recv_count, Datatype recv_datatype, int root)</code>
<code>send_data</code> duomenų masyvas, esantis <code>root</code> procese	<code>send_data</code> duomenų masyvas, esantis <code>root</code> procese
<code>send_count</code> kiek elementų siunčiama kiekvienam procesui	<code>send_count</code> kiek elementų siunčiama kiekvienam procesui
<code>send_datatype</code> kokio tipo elementai siunčiami	<code>send_datatype</code> kokio tipo elementai siunčiami
<code>recv_data</code> duomenų masyvas, į kurį bus gaunami duomenys	<code>recv_data</code> duomenų masyvas, į kurį bus gaunami duomenys
<code>recv_count</code> duomenų masyvo, į kurį gaunami duomenys, dydis	<code>recv_count</code> duomenų masyvo, į kurį gaunami duomenys, dydis
<code>recv_datatype</code> priimamų duomenų dydis	<code>recv_datatype</code> priimamų duomenų dydis
<code>root</code> proceso, iš kurio siunčiami duomenys, numeris	<code>root</code> proceso, iš kurio surenka duomenys, numeris

30. Mokėti naudotis Go, MPI arba JavaCSP paskirstytos atminties programavimo modeliais.

31. C kalba — mokėti naudotis rodyklėmis, atminties valdymo funkcijomis.

- Kadangi funkcijos parametrai visada yra kopijuojami, perdavus kintamąjį į funkciją ir pakeitus funkcijoje jo reikšmę, funkcijos išorėje pasikeitimų nesimatys, nes dirbama su kintamojo **kopija**.
- Perdavus rodyklės tipo kintamąjį funkcijai nurodoma ne kintamojo reikšmė, o vieta atmintyje.
- Kai funkcija žino, kurioje atminties vietoje yra kintamasis, ji gali keisti tą atmintį ir pasikeitimai matysis visur, kur naudojama ta atmintis.
- Dėl to, norint perduoti kintamąjį su galimybe funkcijos viduje jo reikšmę keisti, naudojamos rodyklės.
- Ne rodyklės tipo kintamojo adresą galima sužinoti pasinaudojus `&` operatoriumi: `int* foo_ptr = &foo.`
- Rodyklės kintamojo rodomos vietos reikšmę galima sužinoti pasinaudojus `*` operatoriumi: `int foo = *foo_ptr.`

32. Palyginti CPU ir GPU gijas, CPU ir GPU lygiagretumo galimybes.

- **CPU** gijos vykdomos pagrindiniame procesoriuje, GPU gijos — grafiniame procesoriuje.
- **CPU** ir GPU turi atskirą atmintį: **CPU** naudoja RAM atmintinę, GPU — VRAM atmintinę.
- **CPU** efektyviai gali būti vykdoma tik nedidelis kiekis gijų
- Intel Core i9-7980XE turi 18 branduolių ir palaiko **36** lygiagrečias gijas.
- GPU gali būti vykdoma žymiai didesnis kiekis gijų.
- Nvidia GeForce RTX 2080Ti turi **4352** CUDA branduolius.

33. CUDA darbo principai — atminties tipai, duomenų perkėlimas tarp skirtingų atminčių, kernel funkcijos, atominės operacijos.

host — pagrindinis procesorius (CPU)

device — grafinis procesorius (GPU)

kernel — funkcija, vykdoma grafiniame procesoriuje

CUDA skirta tą pačią *kernel* funkciją vykdyti daugelyje gijų.



- CPU ir GPU naudoja skirtingą atmintį, todėl duomenis reikia kopijuoti iš vienos atminties į kitą.
- Darbas su GPU atmintimi CUDA vykdomas naudojant CUDA funkcijas atminties valdymui.

```
cudaError_t cudaMalloc(void **devPtr, size_t size)
```

GPU atmintyje išskiria nurodytą kiekį atminties.

devPtr rodyklė į rodyklę, į kurią bus įrašytas išskirtos atminties adresas.

size kiek baitų atminties išskirti.

```
cudaError_t cudaFree(void **devPtr)
```

Atlaisvina GPU išskirtą atmintį.

devPtr rodyklė į rodyklę, kur buvo išskirta atmintis.

GPU vykdomos funkcijos

- Funkcijos, vykdomos GPU, bet kviečiamos iš CPU, pažymimos raktiniu žodžiu `__global__`.
- `__global__ void run_on_gpu();`
- Funkcijos, vykdomos GPU ir kviečiamos iš GPU, pažymimos raktiniu žodžiu `__device__`.
- `__device__ void run_on_gpu();`

```
cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, enum cudaMemcpyKind kind)
```

Kopijuoja duomenis tarp CPU ir GPU.

dst atminties, į kurią kopijuojami duomenys, pradžios adresas.

src atminties, iš kurios kopijuojami duomenys, pradžios adresas.

count kopijuojamų duomenų dydis.

cudaMemcpyKind kryptis, iš kur ir į kur kopijuojami duomenys.

- GPU funkcija iš CPU kviečiama nurodant gijų bloko, kuriame bus vykdoma programa, dydį.
- Gijų blokas yra dvimatis blokas, kurio dydis nurodomas dviem skaičiais. Jei bloko dydis nurodomas 2, 5, bus paleidžiama 10 gijų.
- Gijos koordinatės bloke pasiekiamas naudojantis `threadIdx.x` ir `threadIdx.y`.
- GPU funkcijos iškviatimo sintaksė:
`run_on_gpu<<<2, 5>>>(parameter1);`

CUDA atominės operacijos

- CUDA remiasi bendros atminties modeliu.
- Rašymas iš kelių CUDA gijų į tą pačią atmintį yra neapibrėžtas (*undefined behaviour*).
- CUDA turi atominių operacijų rinkinį, skirtą operacijų atomiškumui garantuoti.
- Visos CUDA atominės operacijos vykdomos viena tranzakcija ir iš kitų gijų tarpinė operacijos būsena nematoma.
- Visos CUDA atominės operacijos dirba tiek su globalia, tiek su bendra atmintimi.

atomicAdd Sudeda dvi reikšmes

atomicSub Atima vieną reikšmę iš kitos

atomicExch Į nurodytą atmintį įrašo nurodytą reikšmę ir grąžina reikšmę, kuri buvo toje atmintyje

atomicMin Į nurodytą atminties vietą įrašo nurodytą reikšmę, jei ji mažesnė už toje atminties vietoje esančią reikšmę

atomicMax Į nurodytą atminties vietą įrašo nurodytą reikšmę, jei ji didesnė už toje atminties vietoje esančią reikšmę

atomicInc Padidina nurodytą reikšmę 1

atomicDec Sumažina nurodytą reikšmę 1

34. Mokėti išskirti, kopijuoti atmintį CUDA, paleisti kernel funkcijas.

35. Funkcinis programavimas ir lygiagretumas — mokėti naudotis map, filter, reduce ekvivalentais C++ ar kita kalba bei Thrust, suprasti, kaip šios operacijos gali būti lygiagretinamos.

36. Mokėti sukurti funktorių C++.

37. Thrust vektoriai — kas tai, kaip kopijuojami duomenys tarp CPU ir GPU, kokius algoritmus palaiko.

Thrust funkcinio programavimo palaikymas

```
copy_if(InputIterator first, InputIterator last,
        OutputIterator result, Predicate pred)
```

Kopijuoja visus elementus nuo *first* iki *last*, kurie tenkina sąlygą *pred*, į rezultatų rinkinį *result* (*filter* operacija).

```
transform(InputIterator first, InputIterator last,
           OutputIterator result, UnaryFunction op)
```

Kiekvienam elementui nuo *first* iki *last* pritaiko funktorių *op* ir įrašo į rezultatų rinkinį *result* (*map* operacija).

Thrust vektoriai

- Thrust palaiko vektorius CPU ir GPU atmintyje.
- CPU vektorių atitinka klasė `host_vector`.
- GPU vektorių atitinka klasė `device_vector`.
- Priskiriant `device_vector` objektą `host_vector` objektui arba atvirkščiai, kopijos konstruktorius pasirūpina duomenų perkėlimu tarp CPU ir GPU.

Thrust funkcinio programavimo palaikymas

```
reduce(InputIterator first, InputIterator last, T
        init, BinaryFunction binary_op)
```

Kiekvienam elementui nuo *first* iki *last* vykdo operaciją *binary_op* kartu su iki tol sukaupta reikšme, pradedant nuo *init*, ir sukaupytą reikšmę grąžina.

Visos funkcijos yra taikomos tiek `host_vector`, tiek `device_vector`, bet `device_vector` kviečiamos funkcijos bus vykdomos GPU, todėl turėtų būti *kernel* funkcijos.

38. Python programavimo kalba — pagrindiniai bruožai.

Python

- Dinamiškai tipizuota kalba — tam pačiam kintamajam galima priskirti skirtingų tipų reikšmes (*duck typing*).
- Palaiko struktūrinį, objektinį, funkcinį programavimo stilius.
- Bendros paskirties kalba, daugiausia naudojama scenarijams, duomenų apdorojimui, tinklo sistemoms.
- Kalba yra interpretuojama, populiariausias interpretatorius — CPython.
- Vietoj riestinių skliaustų naudojamos įtraukos.
- Yra tik *foreach* stiliaus ir *while* ciklai.

Python sintaksė

- Funkcijos ir metodai apibrėžiami raktiniu žodžiu `def`, grąžinamos reikšmės ir parametrų tipų nurodyti nereikia: `def func(arg1, arg2):`
- Funkcijos kodas rašomas iš naujos eilutės atitraukus per 4 tarpus.
- Atitraukimas per 4 tarpus taikomas ir `if`, `for`, `while` ir kitiems sakiniams.
- Python leidžia iš funkcijos grąžinti keletą kintamųjų ir daryti priskyrimą į kelis kintamuosius.
- Neegzistuojančią reikšmę atitinka `None`, `bool` tipo reikšmės — `True` ir `False`.
- Duomenų struktūros — sąrašai (*list*), žodynai (*dict*), kortežai (*tuple*).
- `str` tipo kintamieji palaiko unikodą, `char` tipas neegzistuoja.

39. Python gijų modelis — panašumai į C++ / Java / C# modelį, skirtumai nuo jų.

Python gijos

- Gijos Python atvaizduojamos `Thread` klase.
- Klase galima naudotis dviem būdais — paveldint klase ir užklojant `run` metodą (analogiškai Java), arba kurti `Thread` objektą jam perduodant norimą vykdyti funkciją `target` parametru.
- Klasė turi klasikinius metodus:
 - `start()`
 - `join()`
 - `is_alive()`

Python užraktai

- Kritinės sekcijos apsauga realizuojama klase `Lock`.
- Pagrindiniai metodai — `acquire()` ir `release()`
- `Lock` objektu galima naudotis kaip konteksto tvarkytoju (*context manager*):

```
lock = Lock()
with lock:
    # this is run with mutual exclusion
```

40. Globalus interpretatoriaus užraktas — kas tai yra, kodėl yra, kokius apribojimus sukelia.

- CPython interpretatorius turi globalų interpretatoriaus užraktą (*global interpreter lock (GIL)*), kuris apsaugo python objektus nuo lygiagrečios prieigos.
- *GIL* reikalingas, nes python atminties valdymas nėra saugus lygiagrečiai prieigai.
- Dėl *GIL* python programos yra vykdomos vienoje gijoje, o programos viduje sukurtos gijos yra vykdomos keičiant kontekstą vienoje CPU gijoje.
- Python realizacijos Jython (python JVM) ir IronPython (python .NET) neturi *GIL* ir gali išnaudoti visus procesoriaus branduolius, bet jos nėra plačiai naudojamos.

41. Python daugiaprocesiško modelis — kokios jo savybės, kaip apsieičia duomenimis tarp procesų, palyginti su Go ir MPI modeliais.

- Python gijos tinka tada, kai lygiagrečiai atliekami veiksmai yra daugiausiai įvestis/išvestis (*IO bound*).
- Python turi modulį *multiprocessing*, kuris leidžia kurti procesus. Kiekvienas procesas yra atskiras Python interpretatoriaus procesas, todėl jais naudojantis galima išnaudoti sistemos lygiagrečias galimybes.

Apsikeitimas duomenimis tarp procesų

Duomenų perdavimas naudojant eiles

- Apsikeitimas duomenimis tarp procesų vykdomas tokiomis priemonėmis:
 - Eilėmis `Queue`;
 - Kanalais `Pipe`.

- Sukuriamas `multiprocessing.Queue` objektas, kurio sąsaja yra beveik identiška standartinės Python eilės sąsajai.
- Į eilę objektai įrašomi metodu `put`.
- Iš eilės objektai išimami metodu `get`.
- Vienas procesas į eilę rašo duomenis, kitas iš eilės duomenis ima.

Duomenų perdavimas naudojant kanalus

- Kviečiama funkcija `multiprocessing.Pipe`, kuri sukuria du kanalo galus (`Connection` tipo) ir juos grąžina:
- `parent_conn, child_conn = Pipe()`
- Kanalo galais galima naudotis kviečiant jų metodus `send` ir `recv`.
- Komunikacija kanalu galima naudotis siunčiant ir gaunant žinutes iš bet kurio kanalo.
- Kanalai yra *One2One* tipo — bandymas į tą patį kanalą rašyti ar iš to paties kanalo skaityti iš kelių procesų vienu metu gali sugadinti siunčiamus duomenis.

42. Gijų / procesų telkiniai — kas tai yra, kada naudingi.

- Procesų telkinys (*process pool*) — fiksuoto dydžio procesų rinkinys, kuriam galima paskirstyti darbus.
- Python procesų telkiniai kuriami pasinaudojant `Pool` klase.

```
Pool(processes, initializer, initargs,
      maxtasksperchild, context)
```

Sukuria procesų telkinį.

processes kiek procesų sukurti, jei nenurodyta, naudojama `os.cpu_count()` reikšmė.

initializer jei nurodyta, kiekvienas procesas sukūrimo metu iškvies `initializer(*initargs)`.

maxtasksperchild kiek užduočių gali įvykdyti procesas, kol bus pakeistas nauju procesu.

context procesų sukūrimo kontekstas.

```
apply(func, args, kwargs)
```

Iškviečia funkciją `func` viename iš procesų telkinio procesų su parametrais `args, kwargs`.

```
map(func, iterable)
```

Iškviečia funkciją `func` kiekvienam `iterable` elementui darbą išdalinant procesams procesų telkinyje.

Funkcijos turi asinchronines versijas `apply_async` ir `map_async`, kurios grąžina `AsyncResult` objektus ir neblokuoja vykdymo.

43. JavaScript — pagrindiniai bruožai.

JavaScript kalbos bruožai

JavaScript serveryje

- Atsiradus ES6 standartui JavaScript kalba pradėta naudoti serveriuose.
- Pagrindinė implementacija — Node.
- Kalba daugiausia naudojama saityno serveriuose, bet galima kurti ir kitokias programas.

- 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.

44. Įvykių ciklai — kas tai yra.

- JavaScript asinchroninės funkcijos vykdomos toje pačioje gijoje.
- Asinchroninių funkcijų vykdymas užregistruojamas įvykių cikle, įvykiai jame apdorojami eilės principu.
- Asinchroninės funkcijos neblokuoja pagrindinės gijos veikimo.

45. JavaScript / C# `async` / `await` asinchroniškumo modelis — kaip naudojamas, kaip siejasi su asinchroninėmis užduotimis.

ES8: `async` / `await`

`async` / `await` palaikymas naršyklėse

- C# 5 (2012 m.) pridėjo raktinius žodžius `async` ir `await`.
- Vėliau juos pridėjo Python 3.5 (2015 m.) ir ECMAScript 8 (2017 m.).
- 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.

- `async` / `await` palaikomas tik naujose naršyklių versijose:
 - Edge 15 (2015 gegužė)
 - Chrome 55 (2016 gruodis)
 - Firefox 52 (2017 kovas)
 - Safari 10.1 (2017 kovas)
- `async` / `await` palaiko 85% vartotojų naršyklių (Lietuvoje — 87%)¹.

`async` / `await`

Gijos naršyklėje

- Naudojant `async` / `await` nebelieka `callback` funkcijų.
- Klaidų apdorojimas tampa analogiškas nuosekliam kodui.
- Kodą lengva skaityti, nes jis panašus į nuoseklų kodą.

- Asinchroninis programavimas 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.