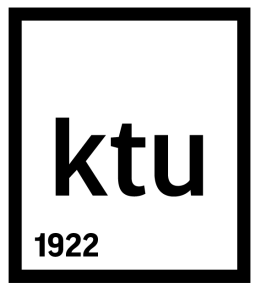
# Dažnai pasitaikančios problemos. Duomenų padalijimas gijoms

Karolis Ryselis

Kauno Technologijos Universitetas




*A man, when confronted with a problem, thinks, "I know, I'll use locks". Now he has two prob*

# Paskaitos turinys

# Lenktynių sąlygos dėl netinkamo interfeiso

- Apsaugoti visus metodus nuo lygiagrečios prieigos gali nepakakti, norint apsaugoti duomenis nuo lygiagrečios prieigos.
- Svarbu, *kaip* metodai yra kviečiami.

# Lenktynių sąlygos: eilė

- Tarkime, turime realizuotą eilės klasę, kuri turi metodus, apsaugotus nuo lygiagrečios prieigos, skirtus duomenims eilėje tvarkyti:
  - Elementui įdėti;
  - Elementui išimti;
  - Elementų kiekiui eilėje patikrinti.

# Lenktynių sąlygos: eilė

```
public class ThreadSafeQueue<T>
{
    private readonly Queue<T> _queue = new Queue<T>();
    public void AddItem(T item)
    {
        lock (this) { _queue.Enqueue(item); }
    }
    public T PopItem()
    {
        lock (this) { return _queue.Dequeue(); }
    }
    public int ItemCount
    {
        get
        {
            lock (this) { return _queue.Count; }
        }
    }
}
```

# Lenktynių sąlygos: eilė

```
while (itemsRemoved < itemsToProcess / popperThreadCount)
{
    if (threadSafeQueue.ItemCount <= 0) continue;
    threadSafeQueue.PopItem();
    itemsRemoved++;
}
```

# Lenktynių sąlygos: eilė

- Kai kviečiamas `PopItem()`,`ItemCount` jau gali būti pasikeitęs.
- Kad programa veiktų teisingai, reikia, kad `ItemCount` tikrinimas ir `PopItem` kreipinys būtų atominė operacija kitų gijų atžvilgiu.
- Sprendimo būdai:
  - Eilės klasėje realizuoti metodą, kuris grąžina `null` ar kitą specialią reikšmę, jei eilė tuščia — tokiu atveju `ItemCount` tikrinimo nereikia.
  - Nekviesti `ItemCount` ir sugauti metamą klaidą, kai eilė tuščia.
  - Jei neturime galimybės keisti eilės klasės, reikia realizuoti kritinę sekciją, kuri apimtų ir `ItemCount` tikrinimą, ir `PopItem` kreipinį. Rizikinga, nes gali susidaryti aklavietė.

# Nuorodų grąžinimas iš funkcijų

- Pavojus kyla, kai iš funkcijos ar metodo grąžinama nuoroda į monitoriumi saugomus duomenis.
- Jei yra metodas, kuris tiesiogiai grąžina tokius duomenis, kitoje programos vietoje, kur grąžinta reikšmė yra naudojama, galima su ta reikšme atlikti veiksmus už kritinės sekcijos ribų.

# Nuorodų grąžinimas iš funkcijų

```
template<typename T>
class SortedThreadSafeList {
private:
    vector<T> storage;
    mutex m;
public:
    void add(T item) {
        unique_lock<mutex> locker(m);
        if (storage.empty()) { storage.push_back(item); return; }
        auto current = storage.begin();
        while (item > *current && current < storage.end()) { current++; }
        storage.insert(current, item);
    }

    vector<T>* get_storage() {
        unique_lock<mutex> locker(m);
        return &storage;
    }
};
```

# Nuorodų grąžinimas iš funkcijų

```
int main() {
    SortedThreadSafeList<int> thread_safe_list;
    auto storage = thread_safe_list.get_storage();
    thread thread(insert, ref(thread_safe_list));
    for (auto i = 10000; i < 15000; i++) {
        storage->insert(storage->begin(), i);
    }
    thread.join();
    for_each(storage->begin(), storage->end(), [](auto item) {
        cout << item << endl;
    });
}

void insert(SortedThreadSafeList<int> &list) {
    vector<int> items(10000);
    iota(items.begin(), items.end(), 0);
    random_device rd;
    default_random_engine engine(rd());
    shuffle(items.begin(), items.end(), engine);
    for_each(items.begin(), items.end(), [&](auto item) { list.add(item); });
}
```

# Nuorodų grąžinimas iš funkcijų

- Metodas `get_storage()` grąžina rodyklę į objektą, kuris saugomas monitoriaus, ir per grąžintą rodyklę objektą kitos gijos gali tvarkyti kaip nori — nėra garantijų, kad grąžintas objektas bus modifikuojamas su kritinės sekcijos apsauga.
- Geriausias variantas — sukurti monitoriaus viduje esančio sąrašo kopiją ir grąžinti ją, tuomet nebeliks bendrų duomenų.
- Galima operacijas, kurias su vidine struktūra norima atlikti už monitoriaus ribų, perkelti į monitoriaus klasę, kad ji būtų vienintelė, kuri tvarko duomenis.

# Naudotojo paduodamos funkcijos

- Metodas, tvarkantis monitoriaus saugančius duomenis, gali priimti parametrą — funkciją, kuri bus kviečiama su vidine monitoriaus struktūra.
- Parametru perduota funkcija gali „pavogti“ nuorodą į vidinę struktūrą.

# Naudotojo paduodamos funkcijos

```
template <class T>
class ThreadSafeStack {
private:
    stack<T> storage;
    mutex m;
public:
    void push(T item) {
        unique_lock<mutex> lock(m);
        storage.push(item);
    }
    template <typename Function>
    void call_user_function(Function func) {
        unique_lock<mutex> lock(m);
        func(storage);
    };
};
```

# Naudotojo paduodamos funkcijos

```
int main() {
    ThreadSafeStack<int> safe_stack;
    vector<int> items;
    items.resize(10000);
    iota(items.begin(), items.end(), 0);
    thread t([&]() {
        for_each(items.begin(), items.end(), [&](auto item) {
            safe_stack.push(item);
        });
    });
    stack<int>* stolen_data;
    safe_stack.call_user_function([&](auto& storage) {
        stolen_data = &storage;
    });
    for_each(items.begin(), items.end(), [&](auto item) {
        stolen_data->push(item);
    });
    t.join();
}
```

# Naudotojo paduodamos funkcijos

- Į `call_user_function` kreipinį galime paduoti funkciją, su kuria pasiimsime rodyklę į monitoriaus saugomą dėklą.
- Toliau su dėklu galima dirbti be monitoriaus apsaugos.
- Nuo tokios klaidos galima apsisaugoti tik nerašant kodo, kuris rodykles ar nuorodas perduoda už monitoriaus ribų.

# Aklavietės dėl keleto užraktų

- Jei yra dvi gijos $T_1$ ir $T_2$ bei du užraktai $L_1$ ir $L_2$, gali susidaryti aklavietė dėl netinkamo užraktų rakinimo.
- Jei $T_1$ užrakina $L_1$ ir $T_2$ užrakina $L_2$, o po to $T_1$ bando užrakinti $L_2$, o $T_2$ — $L_1$, susidarys aklavietė.
- Situaciją sunkiau pastebėti turint užraktų masyvą, kai reikia rakinti ne po vieną užraktą.

# Aklavietės dėl keleto užraktų

```
type BankAccount struct {
    balance decimal.Decimal
    mutex   sync.Mutex
}

func Transfer(accountFrom *BankAccount, accountTo *BankAccount,
    amount decimal.Decimal) bool {
    accountFrom.mutex.Lock()
    accountTo.mutex.Lock()
    defer accountFrom.mutex.Unlock()
    defer accountTo.mutex.Unlock()
    if amount.GreaterThan(accountFrom.balance) {
        return false
    }
    accountFrom.balance = accountFrom.balance.Sub(amount)
    accountTo.balance = accountFrom.balance.Add(amount)
    return true
}
```

# Aklavietės dėl keleto užraktų

```
func main() {
    var bankAccounts = make([]BankAccount, 100)
    for i := 0; i < len(bankAccounts); i++ {
        bankAccounts[i].balance, _ = decimal.NewFromString("1000")
    }
    var waiter = sync.WaitGroup{}
    waiter.Add(1000)
    for i := 0; i < 1000; i++ {
        go func() {
            defer waiter.Done()
            var indexFrom = rand.Intn(100)
            var indexTo = rand.Intn(100)
            amount, _ := decimal.NewFromString("10")
            Transfer(&bankAccounts[indexFrom], &bankAccounts[indexTo],
                amount)
        }()
    }
    waiter.Wait()
}
```

# Aklavietės dėl keleto užraktų

- Aklavietė susidarys, kai norėsime vienu metu pervesti pinigus tarp priešingų sąskaitų, pvz., iš 1 į 2 ir iš 2 į 1.
- Aklavietės išvengti tokiu atveju galima visada rakinant užraktus ta pačia tvarka.
- Tai galima pasiekti perduodant papildomą parametrą, rodantį, kuris užraktas turi būti rakinamas pirmas, arba parametru priimti masyvą ir sąskaitų indeksus ir visada pirma rakinti, pvz., mažesnio indekso užraktą.
- C++ standartinė biblioteka turi funkciją `lock`, kuri priima bet kokį kiekį užraktų ir juos užrakina taip, kad nebūtų gaunama aklavietė.

# Daugiausia skaitomos duomenų struktūros

- Jei duomenų struktūra palaiko įrašymo ir skaitymo operacijas, dažniausiai, kai įrašomi duomenys, kitos operacijos vykti negali.
- Jei duomenų skaitymas struktūros nekeičia, tai gijos gali laisvai skaityti vienu metu, bet negalima skaityti, kai vyksta rašymas.
- Šią problemą sprendžia skaitymo — rašymo užraktai.

# Skaitymo — rašymo užraktai

| | |
|---|---|
| C++ | `shared_mutex` |
| C# | `ReaderWriterLockSlim` |
| Rust | `RwLock` |

# Skaitymo — rašymo užraktai

```
public class RarelyUpdatedDictionary<TK, TV>
{
    private readonly Dictionary<TK, TV> _storage;
    private readonly ReaderWriterLockSlim _lock;
    public TV this[TK key]
    {
        get
        {
            _lock.EnterReadLock();
            try { return _storage[key]; }
            finally { _lock.ExitReadLock(); }
        }
        set
        {
            _lock.EnterWriteLock();
            try { _storage[key] = value; }
            finally { _lock.ExitWriteLock(); }
        }
    }
}
```

# Rekursinis lygiagretumas

- Norint išlygiagretinti rekursinį algoritmą, gijas tenka kurti rekursiškai kviečiamose funkcijose.
- Jei kiekviena gija sukuria po 2 gijas, o rekursijos gylis 10, bus paleistos **1024** gijos.
- Kuriant rekursiškai gijas paleidžiančius algoritmus svarbu išvengti milžiniško gijų kiekio.

# Rekursinis lygiagretumas

```
private static List<T> RunSort(List<T> input, int depth)
{
    if (input.Count == 0)
    {
        return input;
    }

    var pivot = input[0];
    var itemLists = input.Skip(1)
        .GroupBy(t => t.CompareTo(pivot) < 0)
        .ToDictionary(g => g.Key, g => g.ToList());
    var lowerItems = itemLists.TryGetValue(true, out var list) ?
        list : new List<T>();
    var higherItems = itemLists.TryGetValue(false, out var itemList) ?
        itemList : new List<T>();
    List<T> sortedLowerItems = null;
    Thread thread = null;
```

# Rekursinis lygiagretumas

```
    if (depth >= 0)
    {
        thread = new Thread(() =>
        {
            sortedLowerItems = RunSort(lowerItems, depth - 1);
        });
        thread.Start();
    }
    else
    {
        sortedLowerItems = RunSort(lowerItems, -1);
    }
    var sortedHigherItems = RunSort(higherItems, depth - 1);
    thread?.Join();
    sortedLowerItems.Add(pivot);
    sortedLowerItems.AddRange(sortedHigherItems);
    return sortedLowerItems;
}
```

# Duomenų lygiagretumas

- Lygiagretumo modelis, kai visos gijos daro tą patį su skirtingais duomenų poaibiais.
- 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.

# Duomenų 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 skaičiuos 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?

# Duomenų padalinimo gijoms variantas

```
const int DATA_SIZE = ...;
int data[DATA_SIZE] = ...;
auto hw_threads = thread::hardware_concurrency();
auto threads_to_use = hw_threads > 0 ? hw_threads : 1;
vector<thread> threads;
threads.reserve(threads_to_use);
int chunk_size = DATA_SIZE / hw_threads;
for (int i = 0; i < threads_to_use; i++){
    int end_index = (i == threads_to_use - 1
      ? DATA_SIZE - 1 : (i+1) * chunk_size - 1);
    thread t(process, data, i * chunk_size, end_index);
    threads.push_back(move(t));
}
for_each(threads.begin(), threads.end(),
    mem_fn(&thread::join));
```

# Duomenų lygiagretumas

- Duomenų paskirstymą gijoms galima spręsti panaudojant gijų telkinį (*thread pool*).
- Sukuriant gijų telkinį paleidžiamas fiksuotas kiekis gijų.
- Gijų telkiniui galima priskirti darbą, kurį gijos pasidalija pačios.
- Gijų telkinio privalumai — nereikia kiekvieną kartą kurti gijų, gijos pačios pasiskirsto darbus.

# Funkcinis lygiagretumas

- 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.

# Funkcinio lygiagretumo problemos

- Kaip išlygiagretinti skaičiavimus, kai vieno veiksmo rezultatai priklauso nuo kito veiksmo rezultatų? Jei veiksmai priklauso vienas nuo kito, gali kilti sunkumų.
- Kaip išskirti tokius veiksmus, kurie gali būti atliekami lygiagrečiai? Veiksmus lengviau lygiagretinti tada, kai jie nepriklausomi.

# *fork-join* šablonas

- Duomenų lygiagretumo šablonas, kuris paskirsto gijoms užduotims ir po to susirenka visų gijų rezultatus.
- Susideda iš tokių pagrindinių žingsnių:
  1. Padalinti užduotį į keletą sub-užduočių, kurios vykdomos lygiagrečiai;
  2. Palaukti, kol paleistos gijos baigs darbą;
  3. Apjungti gijų suskaičiuotus rezultatus.

# *fork-join* taikymas OpenMP

```
#pragma omp parallel for
for (auto col = 0; col < COLS; col++) {
    for (auto row = 0; row < ROWS; row++) {
        auto x = compute_row(row);
        auto y = compute_column(col);
        complex<double> c(x, y);
        auto colour = is_mandelbrot(c, 100) ? 0 : 255;
        auto offset = (col * COLS * 3) + (3 * row);
        png_buffer[offset] = colour;
        png_buffer[offset + 1] = colour;
        png_buffer[offset + 2] = colour;
    }
}
```

# Lygiagrečiojo programavimo nauda

- programinis modelis geriau atitinka realųjį pasaulį;
- daugiaprocesorinėje sistemoje galima padidinti vykdymo greitį;
- vienprocesorinėje sistemoje viena programa praktiškai neišnaudoja viso procesoriaus laiko;
- iš anksto nustatyta veiksmų tvarka kai kuriems taikymams yra nepriimtina.

# Kada nenaudoti lygiagrečiojo programavimo?

Kai gaunamas programos pagreitėjimas 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;

# Amdalo dėsnis

- Amdalo dėsnis teigia, kad yra fundamentali riba, kiek programą galima pagreitinti didinant gijų ir skaičiavimo branduolių kiekį.
- $S_{latency} = \frac{1}{(1-p)+\frac{p}{s}}$
- $S_{latency}$ — programos pagreitėjimas, priklausantis nuo lygiagrečios dalies pagreitėjimo $s$;
- $p$ — programos dalis, kurią galima lygiagretinti;
- $s$ — lygiagretinamos programos dalies pagreitėjimas; dažnai lygus branduolių skaičiui;

# Amdalo dėsnis

- Iš Amdalo dėsnio seka išvados:
- $S_{latency}(s) \leq \frac{1}{1-p}$
- Jei turėtume neribotą kiekį branduolių, vienos gijos vykdymo laikas artėtų prie nulio ir visą programos vykdymo laiką sudarytų nuoseklios dalies vykdymo laikas.
- Jei galima išlygiagretinti pusę programos, tai jos vykdymo laiko **neįmanoma** pagerinti daugiau, nei dvigubai.
- Jei galima išlygiagretinti 95% programos, tai su **32** gijomis galima pasiekti **12.5** karto pagreitėjimą, su 64 gijomis — **15.4** karto.

# Gustafsono dėsnis

- Gustafsono dėsnis apibrėžia, kiek duomenų galima apdoroti per tą patį laiką didinant branduolių skaičių.
- $S_{latency} = (1 - P) + (N \times P)$
- $P$ — lygiagreti dalis, $N$ — branduolių skaičius.
- Jei 90% programos išlygiagretinta ir turime 24 branduolius, tai per tą patį laiką galime apdoroti $S_{latency} = (1 - 0.9) + (24 \times 0.9) = 21.7$ karto daugiau duomenų, nei su 1 branduoliu.