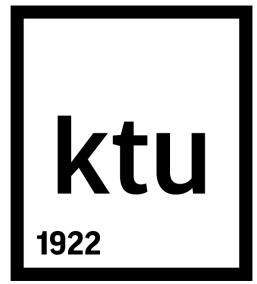
Kritinė sekcija, monitoriai (1)

Karolis Ryselis

Kauno Technologijos Universitetas



Q: *Why did the concurrent chicken cross the road?*
A: *the side other. To to get*

Paskaitos turinys

ktu 1922

Bendra procesų atmintis be kritinės sekcijos

Neteisingo programavimo pavyzdys!

```
void process(int* c){
    for (int i = 0; i < 1000; i++){
        auto k = *c;
        k++;
        *c = k;
    }
}
```

Bendra procesų atmintis be kritinės sekcijos

Neteisingo programavimo pavyzdys!

```
const size_t THREAD_COUNT = 15;

int main() {
    auto a = 0;
    auto c = &a;
    vector<thread> threads;
    threads.reserve(THREAD_COUNT);
    for (auto i = 0; i < THREAD_COUNT; i++){
        threads.emplace_back(process, c);
    }
    for_each(threads.begin(), threads.end(), mem_fn(&thread::join));
    cout << *c << endl;
    return 0;
}
```

Bendra procesų atmintis be kritinės sekcijos

- Programos rezultatas gali būti bet koks nuo 1000 iki 15000.
- Sakykime, c=0. Dvi gijos vienu metu į savo lokalius kintamuosius `k` priskiria esamą c reikšmę 0.
- Abi gijos vienu metu padidina savo lokalius kintamuosius `k`, abiejų reikšmės 1.
- Abi gijos vienu metu pakeičia c reikšmę į savo kintamojo `k` reikšmę.
- c reikšmė tampa 1, nors ciklas iš viso buvo įvykdytas du kartus.
- Tai ne vienintelis galimas variantas - vienu metu gali vykti bet kurie skirtingų gijų veiksmai. Dėl to programos rezultatas yra **neapibrėžtas**.

Bendra procesų atmintis

- Neapibrėžtas rezultatas gaunamas, kai veiksmai, 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.

Kritinė sekcija ir monitorius

- Kritinė sekcija yra **kodo fragmentas**, kuris yra vykdomas ne daugiau kaip vienos gijos vienu metu.
- Monitorius yra **duomenų struktūra**, skirta nuo lygiagrečios prieigos saugoti duomenis.
- Jei reikia kurti monitorių pačiam, tam dažnai naudojamos kritinės sekcijos.
- Realiose programose dažniausiai kritinės sekcijos reikalingos būtent duomenų apsaugai, dėl to programuotojo tikslas ir turimos priemonės kartais prasilenkia.

Monitoriaus struktūra

Monitorių sudaro:

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

Sąlyginiai kintamieji

- Sąlyginis kintamasis — programavimo šablonas, kai reikia gijoms ne tik taikyti kritinės seckijos apsaugą, bet ir laukti, kol kitoje gijoje įvyks tam tikras įvykis.
- Naudojantis sąlyginiais kintamaisiais galima vieną giją užmigdyti, kol kitoje įvyks tam tikras įvykis, tačiau reikia realizuoti tiek gijos užmigdymą, tiek ir jos pažadinimą.

Monitoriai ir kritinės sekcijos programavimo kalbose

- Daugumoje programavimo kalbų sukurtos priemonės kritinei sekcijai kurti. Ja pasinaudojant galima susikurti savo monitorių.
- Rust programavimo kalboje yra tik monitoriaus duomenų struktūra (savo kurti nereikia), kuri kartu teikia ir kritinės sekcijos apsaugą.

C++ klasės `mutex` objektai – teikia kritinės sekcijos sąsają

C# kiekvienas objektas gali veikti kaip monitorius – teikia kritinės sekcijos sąsają

Go tipas `sync.Mutex` – teikia kritinės sekcijos sąsają

Rust tipas `sync::Mutex` – teikia monitoriaus sąsają

Monitorių veikimas

- Monitoriai turi galimybę:
  - Pažymėti kritinę sekciją, kuri bus vykdoma su tarpusavio išskyrimu;
  - Užmigdyti giją;
  - Pažadinti užmigdytas gijas.

Užraktai

- Jei naudojamas kritinės sekcijos variantas, tokia priemonė vadinama *užraktu*.
- Užraktas gali leisti pažymėti kritinės sekcijos ribas — tokiu atveju reikalingas rakinimo kintamasis. Jei skirtingose programos vietose užraktams naudojamas tas pats rakinimo kintamasis, tai vienu metu galima vykdyti tik vieną iš kodo blokų.
- Jei kalbos priemonės leidžia naudoti kritinės sekcijos žymėjimą, rekomenduojama jį ir naudoti, nes, pvz., įvykus išimtinei situacijai (*exception*) kritinės sekcijos viduje užraktas nelieka užrakintas, nors kritinės sekcijos kodas ir nebaigtas vykdyti.

Užraktai — kritinės sekcijos žymėjimas

- C# galima pažymėti kodo blokus raktiniu žodžiu `lock` parametru nurodant objektą, kuris bus naudojamas užrakinimui.
- `lock(monitor) {/*critical section C#*/}`
- Rust turi tipą `Mutex`, kuris, skirtingai nei kitose kalbose, saugo ne tik kodo bloką, bet ir su juo susietus duomenis — sukuriant monitorių jam parametru perduodamas kintamasis, kurį grąžina užraktą atrakinanti funkcija.

Užraktai

- Kritinei sekcijai žymėti gali būti kviečiami atitinkamos monitoriaus klasės metodai.
- Monitorius gali turėti metodus:
  - užrakinimui;
  - atrakinimui;
  - patikrinimui, ar užrakinta.

Užraktai — rankinis valdymas

- C++ turi 3 užraktų klases, reikia susikurti norimos klasės objektą:
  - `mutex` — rakinimas vykdomas kviečiant metodus;
  - `lock_guard` — rakinimas vykdomas sukuriant / sunaikinant objektą;
  - `unique_lock` — pirmų dviejų kombinacija, lanksčiausias.
- C# turi klasę `Monitor`, rakinimas vykdomas kviečiant statinius metodus (objektas nekuriamas, rakinimui naudojamas kitas objektas).
- Go turi tipą `Mutex`, analogiškas C++ `mutex`.

Užraktai — rankinis valdymas

C++ `mutex`, C# `Monitor` ir Go `Mutex` turi metodus kritinės sekcijos užrakinimui ir atrakinimui.

Priemonė	Užrakinimas	Atrakinimas
C++ `mutex`	`m.lock()`	`m.unlock()`
C# `Monitor`	`Monitor.Enter(obj)`	`Monitor.Exit()`
Go `Mutex`	`m.Lock()`	`m.Unlock()`
Rust `Mutex`	`m.lock()`	atrakinamas kodo bloko gale

Užraktus reikėtų naudoti try-finally blokuose, kad užraktas neliktų amžinai užrakintas nulūžus vienai gijai.

C++ papildomi monitoriai

- C++ standartinė biblioteka siūlo klases `lock_guard` ir `unique_lock`, kurių pagalba galima saugiau valdyti užraktus.
- `lock_guard` remiasi ideologija *RAII* (*Resource Acquisition Is Initialization*) — objekto konstruktorius užrakina `mutex` objektą, destruktorius — atrakina.
- `lock_guard` destruktorius kviečiamas tada, kai baigiasi kodo sekcija, pvz., pabaigus vykdyti metodą, todėl užraktas visada atsirakins.
- Tą patį `mutex` objektą gali valdyti keli skirtingi `lock_guard` objektai, papildomų metodų klasė neturi.
- `unique_lock` veikia kaip `mutex` ir `lock_guard` kombinacija: rakinasi automatiškai, bet turi ir metodus rankiniam rakinimui.

Užraktai

- Nepriklausomai nuo to, ar užraktas užrakinamas priėjus sinchronizuotą bloką, ar naudojantis monitoriaus metodais, jei viena gija yra užrakinusi užraktą ir kita gija bando tą patį užrakinti, ji yra **blokuojama**, kol pirmoji gija neatrakins užrakto.

Sąlyginė sinchronizacija

- Monitoriai teikia gijai galimybę užmigti arba pažadinti kitas miegančias gijas.
- Toks gijų valdymas vadinamas *sąlygine sinchronizacija*, nes programuotojas turi kode nurodyti, prie kokių sąlygų giją blokuoti ir prie kokių — pažadinti.
- Sinchronizavime dalyvauja dvi gijų kategorijos: gijos, kurios gali save užmigdyti ir gijos, kurios gali užmigytas gijas pažadinti. **Gija pati savęs pažadinti negali**.
- Toks mechanizmas naudojamas tais atvejais, kai gija turi palaukti, kol įvyks tam tikras veiksmas, kad galėtų tęsti darbą.
- Pvz., jei naudojamas buferis duomenų apsikeitimui tarp gijų, gija negali iš buferio pašalinti elementų, kol kita gija jų neįdėjo.

Sąlyginės sinchronizacijos priemonės

Kalbos turi savo sąlyginės sinchronizacijos priemones:

C++ klasė `std::condition_variable`

C# klasė `Monitor` su statiniais metodais

Go tipas `sync.Cond`

Rust struktūra `sync::Condvar`

- Kiekviena priemonė turi metodą, kuris užmigdo bei metodą, kuris pažadina vieną **bet kurią** miegančią giją ir metodą, kuris pažadina visas miegančias gijas.
- Kai kurios priemonės turi metodus žadinimui, kur galima nurodyti sąlygą, prie kurios bus žadinama gija, t. y., iškvietus žadinimo metodą gija pasitikrina, ar tikrai turi pabusti.
- **Sąlyginę sinchronizaciją galima naudoti tik kritinės sekcijos viduje**.

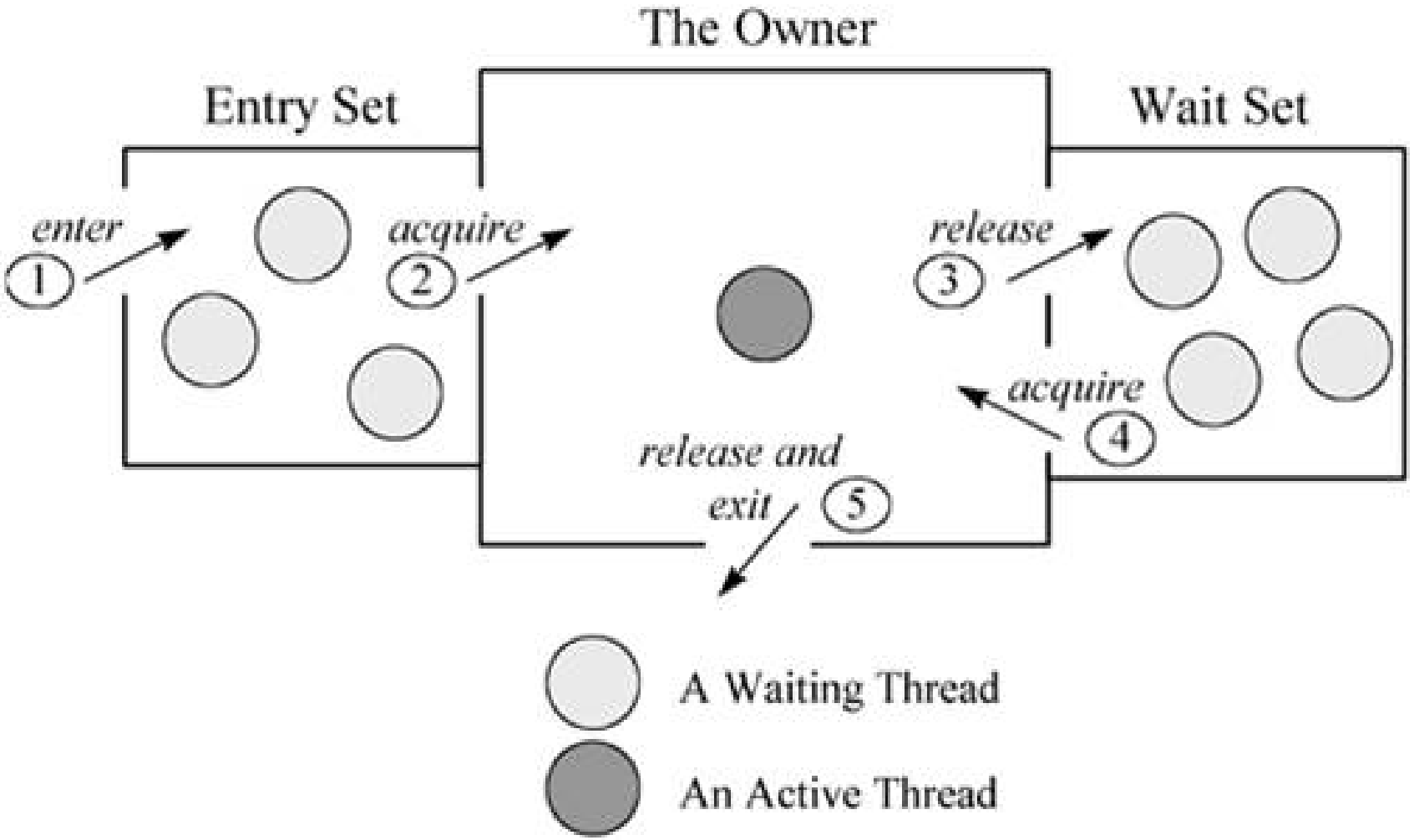
Sąlyginės sinchronizacijos veikimas

- Monitorius viduje turi du sustabdytų gijų rinkinius: *wait set* ir *entry set*.
- *entry set* patenka gijos, kurios sustabdytos dėl to, kad norėjo patekti į kritinę sekciją, pvz., C++ programoje iškvietė `mutex.lock()`.
- *wait set* patenka gijos, kurios buvo užmigdytos ir po to pažadintos naudojantis sąlygine sinchronizacija.
- Jei gija, paliekanti monitorių, nežadino gijų, dėl monitoriaus varžosi tik gijos iš *entry set*.
- Jei gija, paliekanti monitorių, žadino gijas, dėl monitoriaus varžosi gijos iš *entry set* ir *wait set*.
- Kadangi sąlyginei sinchronizacijai reikia valdyti užraktą, tai visoms programavimo priemonėms reikia perduoti, kuris užraktas bus valdomas.

Gijų žadinimas

- Nuo kalbos priemonės realizacijos priklauso:
  - kuri gija iš *wait set* bus „prižadinta“;
  - kokia tvarka „prižadinamos“ gijos iš *wait set*, jei prižadinamos visos iškart;
  - kokia tvarka gijos iš *entry set* įgyja monitorių;
  - kaip pasirinkti tarp *entry set* ir *wait set* gijų pažadinant užmigdytas gijas.

Monitorių veikimas



Sąlyginės sinchronizacijos metodai

Kalba	Gijos migdymas	Vienos gijos žadinimas	Visų gijų žadinimas
C++	`c.wait()`	`c.notify()`	`c.notify_all()`
C#	`Monitor.Wait(m)`	`Monitor.Pulse(m)`	`Monitor.PulseAll(m)`
Go	`c.Wait()`	`c.Signal()`	`c.Broadcast()`
Rust	`c.wait()`	`c.notify_one(m)`	`c.notify_all(m)`

C++ ir Go sąlyginės sinchronizacijos objektų kūrimo metu reikia perduoti `mutex`, kuriuo rakinama kritinė sekcija; C# ir Rust monitorius perduodamas konkretiems metodams.

Paprastas skaitiklis

- Pavyzdyje rodoma programa naudoja monitorių skaitikliui realizuoti.
- Monitorius saugo sveikąjį skaičių.
- Sukuriamos dvi funkcijos, iš kurių viena padidina esamą reikšmę vienetu, kita sumažina vienetu.
- Skaitiklis turi minimalią ir maksimalią reikšmes – jeigu pasiektas minimumas, nebegalima mažinti, jeigu maksimumas – nebegalima didinti reikšmės. Tokiais atvejais gija blokuojama, kol bus galima keisti reikšmę.

Paprastas skaitiklis

```
fn decrease(mutex: Arc<Mutex<i32>>, conditional_variable: Arc<Condvar>) {
    for _ in 0..50 {
        let mut guard = conditional_variable.wait_while(
            mutex.lock().unwrap(),
            |counter| *counter <= 0
        ).unwrap();
        *guard -= 1;
        conditional_variable.notify_all();
    }
}

fn increase(mutex: Arc<Mutex<i32>>, conditional_variable: Arc<Condvar>) {
    for _ in 0..50 {
        let mut guard = conditional_variable.wait_while(
            mutex.lock().unwrap(),
            |counter| *counter >= 50
        ).unwrap();
        *guard += 1;
        conditional_variable.notify_all();
    }
}
```

Paprastas skaitiklis

```
fn main() {
    let mutex = Arc::new(Mutex::new(0));
    let conditional_variable = Arc::new(Condvar::new());
    let mut join_handles: Vec<JoinHandle<_>> = vec![];
    for _ in 0..10 {
        let mutex = Arc::clone(&mutex);
        let conditional_variable = Arc::clone(&conditional_variable);
        let handle = thread::spawn(move || increase(mutex, conditional_variable));
        join_handles.push(handle);
    }
    for _ in 0..9 {
        let mutex = Arc::clone(&mutex);
        let conditional_variable = Arc::clone(&conditional_variable);
        let handle = thread::spawn(move || decrease(mutex, conditional_variable));
        join_handles.push(handle);
    }
    join_handles.into_iter().for_each(|handle| handle.join().unwrap());
    println!("{}", mutex.lock().unwrap());
}
```

Pašto dėžutė 1-1

Pavyzdinės programos (pašto dėžutės) veikimo principas:

- pašto dėžutė realizuojama kaip klasė — monitorius;
- pašto dėžutės talpa yra vienas elementas;
- pašto dėžute naudojasi du procesai: dedantis žinutę ir skaitantis žinutę;
- jei dedantis procesas nori dėti žinutę, bet prieš tai buvusi žinutė dar neperskaityta, procesas laukia;
- skaitantis procesas du kartus tos pačios žinutės neskaito;
- kai visos žinutės apdorotos, abu procesai baigia darbą.

Pašto dėžutė 1-1

```
class MailBox {
private:
    int mail;
    bool exists;
    mutex lock;
    condition_variable cv;
public:
    MailBox();
    void put(int new_mail);
    int get();
};
```

Pašto dėžutė 1-1

```
int main() {
    MailBox mailBox;
    vector<thread> threads;
    threads.emplace_back([&]{
        for (int i = 0; i < 33; i++){
            mailBox.put(i);
        }
        mailBox.put(TERMINATE_MESSAGE);
    });
    threads.emplace_back([&]{
        vector<int> received_messages;
        int received_mail;
        while ((received_mail = mailBox.get()) != TERMINATE_MESSAGE) {
            cout << received_mail << endl;
        }
    });
    for_each(threads.begin(), threads.end(), mem_fn(&thread::join));
    return 0;
}
```

Pašto dėžutė 1-1

```
MailBox::MailBox() {
    mail = EMPTY_VALUE;
    exists = false;
}

void MailBox::put(int new_mail) {
    unique_lock<mutex> guard(lock);
    cv.wait(guard, [&]{ return !exists;});
    mail = new_mail;
    exists = true;
    cv.notify_one();
}

int MailBox::get() {
    unique_lock<mutex> guard(lock);
    cv.wait(guard, [&]{ return exists;});
    auto new_letter = mail;
    exists = false;
    mail = EMPTY_VALUE;
    cv.notify_one();
    return new_letter;
}
```