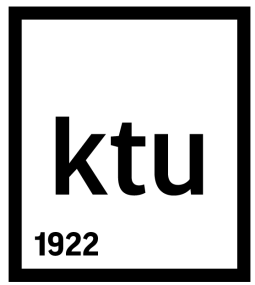
Kritinė sekcija, monitoriai (2). OpenMP gijos

Karolis Ryselis

Kauno Technologijos Universitetas



Knock knock
“Race condition.”
“Who’s there?”

Paskaitos turinys

ktu 1922

Pašto dėžutė 1-n

Pavyzdinės programos (pašto dėžutės) veikimo principas:

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

Pašto dėžutė 1-n

```
public class MailBox
{
    private readonly object _locker;
    private bool _canWrite;
    private bool[] _canRead;
    private int _letter = EmptyMessage;

    public MailBox(int readers)
    {
        _canRead = new bool[readers];
        _canWrite = true;
        _locker = new object();
    }
```

Pašto dėžutė 1-n

```
public void Put(int newLetter)
{
    lock (_locker)
    {
        while (!_canWrite) { Monitor.Wait(_locker); }
        _letter = newLetter;
        _canWrite = false;
        Array.Fill(_canRead, true);
        Monitor.PulseAll(_locker);
    }
}

public int Get(int k)
{
    int newLetter;
    lock (_locker)
    {
        while (!_canRead[k]) { Monitor.Wait(_locker); }
        newLetter = _letter;
        _canRead[k] = false;
        _canWrite = !_canRead.Any(r => r);
        Monitor.PulseAll(_locker);
    }
    return newLetter;
```

Pašto dėžutė 1-n

```
public class Reader
{
    public List<int> Letters { get; }
    private readonly MailBox _mailbox;
    private readonly int _id;
    private readonly int _terminateMessage;

    public Reader(int terminateMessage, MailBox mailbox, int id)
    {
        _mailbox = mailbox;
        _id = id;
        Letters = new List<int>();
        _terminateMessage = terminateMessage;
    }

    public void Read()
    {
        while (true)
        {
            var message = _mailbox.Get(_id);
            if (message == _terminateMessage) { break; }
            Letters.Add(message);
        }
    }
```

Pašto dėžutė 1-n

```
public static void Main(string[] args)
{
    const int readerCount = 5;
    var mailbox = new MailBox(readerCount);
    var readers = Enumerable.Range(0, readerCount).Select(i =>
        new Reader(TerminateMessage, mailbox, i)).ToList();
    var threads = readers.Select(reader => new Thread(reader.Read))
        .ToList();
    foreach (var thread in threads) { thread.Start(); }
    threads.Add(new Thread(() =>
    {
        for (var i = 0; i < 6; i++) { mailbox.Put(i * i); }
    }));
    foreach (var thread in threads) { thread.Join(); }
    var lines = readers.Select(r => string.Join(", ", r.Letters));
    foreach (var line in lines) { Console.WriteLine(line); }
}
```

Gamintojas — vartotojas su ribotu buferiu

Pavyzdinės programos veikimo principas:

- riboto dydžio buferis realizuojama kaip klasė — monitorius;
- buferiu naudojasi du procesai — dedantis į buferį ir šalinantis iš buferio;
- jei buferis yra tuščias, šalinantis procesas laukia;
- jei buferis yra pilnas, dedantis procesas laukia.

Gamintojas — vartotojas su ribotu buferiu

```
type LimitedBuffer struct {
        container   []int
        from        int
        to          int
        currentSize int
        mutex       *sync.Mutex
        cond        *sync.Cond
}
```

Gamintojas — vartotojas su ribotu buferiu

```
func (buffer *LimitedBuffer) Insert(item int) {
        buffer.mutex.Lock()
        for buffer.currentSize == len(buffer.container) {
                buffer.cond.Wait()
        }
        buffer.container[buffer.currentSize] = item
        buffer.to = (buffer.to + 1) % len(buffer.container)
        buffer.currentSize++
        buffer.cond.Broadcast()
        buffer.mutex.Unlock()
}
func (buffer *LimitedBuffer) Remove() int {
        buffer.mutex.Lock()
        for buffer.currentSize == 0 {
                buffer.cond.Wait()
        }
        var item = buffer.container[buffer.from]
        buffer.container[buffer.from] = math.MinInt32
        buffer.from = (buffer.from + 1) % len(buffer.container)
        buffer.currentSize--
        buffer.cond.Broadcast()
        buffer.mutex.Unlock()
        return item
}
```

Gamintojas — vartotojas su ribotu buferiu

```
func produce(buffer *LimitedBuffer, waiter *sync.WaitGroup) {
        defer waiter.Done()
        var itemsToProduce = 13
        var startItem = 10
        for i := 0; i < itemsToProduce; i++ {
                var itemToInsert = startItem + i
                buffer.Insert(itemToInsert)
        }
        buffer.Insert(TerminateMessage)
}

func consume(buffer *LimitedBuffer, consumedItems *[]int, waiter *sync.WaitGroup) {
        defer waiter.Done()
        for {
                var item = buffer.Remove()
                if item == TerminateMessage {
                        break
                }
                *consumedItems = append(*consumedItems, item)
        }
}
```

Gamintojas — vartotojas su ribotu buferiu

```
func main() {
        var buffer = initializeBuffer(5)
        var waiter = sync.WaitGroup{}
        waiter.Add(2)
        go produce(&buffer, &waiter)
        var results = make([]int, 0)
        go consume(&buffer, &results, &waiter)
        waiter.Wait()
        for _, item := range results {
                fmt.Println(item)
        }
}
```

Pavojai dirbant su užraktais

- Jei keletas gijų turi prieigą prie kintamojo, tai kintamąjį modifikuojantys metodai **visada** turi būti vykdomi kritinėje sekcijoje.
- Pavojus kyla tada, kai nuorodą ar rodyklę į modifikuojamą objektą grąžina sinchronizuotas metodas, o vėliau grąžinto kintamojo metodus gali kviesti nesinchronizuotas metodas.
- Svarbu taip suprojektuoti klasės, apsaugančios bendrus duomenis, interfeisą, kad tokie objektai nebūtų prieinami už kritinės sekcijos ribų.

OpenMP

- OpenMP – tai papildomos priemonės, suteikiančios galimybes kurti lygiagrečias programas Fortran, C, C++ kalbomis;
- OpenMP skirta kurti lygiagrečias programas, kuriose procesai (gijos) naudojasi **bendra atmintimi**;
- OpenMP sudaro: kompiliatoriaus direktyvos, specialių funkcijų biblioteka, aplinkos kintamųjų rinkinys.

OpenMP

- Nuo programos darbo pradžios iki pabaigos vykdoma pagrindinė gija (angl. *master thread*).
- Naujos gijos kuriamos kompiliatoriaus direktyvomis.
- Vienu metu vykdomų gijų rinkinys sudaro lygiagrečią sritį (angl. *parallel region*).

OpenMP programos struktūra

```
#include <openmp>

void main() {
    // Nuoseklus kodas
    #pragma omp parallel
    {
        // Lygiagretus kodas
    }
    // Nuoseklus kodas
}
```

OpenMP gijų vykdymas

- 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; *master* gijos numeris – 0.
- Programos vykdymo metu nėra galimybių keisti gijos numerį.

OpenMP direktyvos

Direktyva `#pragma omp parallel`

Nurodo lygiagrečiai vykdomą sritį. Neprivalomu atributu `num_threads(n)` galima nurodyti vykdomų gijų skaičių `n`.

Funkcija `omp_set_num_threads(n)`

Nurodo, kiek gijų bus vykdomas `parallel` blokas.

Funkcija `omp_get_num_threads()`

Grąžina, kiek gijų bus vykdoma `parallel` bloke.

Funkcija `omp_get_thread_num()`

Grąžina gijos, kurioje iškviesta, numerį.

OpenMP kritinės sekcijos apsauga

- OpenMP kritinės sekcijos apsaugai naudojama kompiliatoriaus direktyva `omp critical`.
- Direktyva pažymimas kodo blokas, kuris turi vienu metu būti vykdomas vienos gijos.
- `critical` direktyvai galima nurodyti pavadinimą. Blokai su vienodais pavadinimais laikomi tos pačios kritinės sekcijos dalimi.
- Visos `critical` sritys be vardo taip pat laikomos tos pačios kritinės sekcijos dalimi.

Elementari programa

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

Elementari programa

```
int main() {
    vector<string> thread_names = {"First", "Second"};
    omp_set_num_threads((int) thread_names.size());
#pragma omp parallel shared(thread_names) default(none)
    {
        auto threadNumber = omp_get_thread_num();
        auto name = thread_names[threadNumber];
#pragma omp critical
        {
            execute(name);
        }
    }
    cout << "Program finished execution" << endl;
}
```

OpenMP kritinės sekcijos apsauga naudojant `omp_lock_t`

- `omp_lock_t` tipas turi su savimi susietą užraktą, kuriuo galima manipuliuoti programos vykdymo metu.
- `omp_lock_t` tipas nėra klasė ir savo metodų neturi. Užrakto valdymui naudojamos OpenMP funkcijos, kurioms perduodama rodyklė į `omp_lock_t` kintamąjį.

Su `omp_lock_t` tipu susijusios funkcijos

`void omp_init_lock(omp_lock_t* lock)`

Funkcija inicializuoja užraktą, susietą su parametru `lock`. Po šio funkcijos iškvietimo galima naudoti kitus su užraktu susijusius metodus. Šios funkcijos kvietimas yra vienintelis būdas initicializuoti užraktą. Sukurtas užraktas yra atrakintas.

`void omp_destroy_lock(omp_lock_t* lock)`

Sunaikina su `lock` parametru susietą užraktą. Kviečiant šią funkciją `lock` turi būti inicializuotas ir atrakintas.

Su `omp_lock_t` tipu susijusios funkcijos

`void omp_set_lock(omp_lock_t* lock)`

Funkcija sustabdo gijos darbą iki tol, kol užraktas bus atrakintas, ir tada jį užrakina. `lock` turi būti inicializuotas.

`void omp_unset_lock(omp_lock_t* lock)`

Funkcija atrakina užraktą ir leidžia jį užrakinti kitoms gijoms. `lock` turi būti inicializuotas.

`int omp_test_lock(omp_lock_t* lock)`

Funkcija bando užrakinti užraktą, bet gijos darbo neblokuoja. Jei užraktą užrakinti pavyko, grąžinama nenulinė reikšmė, jei nepavyko — 0.

Paprastas skaitiklis

```
#pragma omp parallel num_threads(total_threads) default(none) shared(increasing_thr
    {
        int thread_id = omp_get_thread_num();
        if (thread_id < increasing_thread_count) {
            for (auto i = 0; i < 50;) {
#pragma omp critical (count_critical)
                {
                    if (count < MAX) {
                        count++;
                        i++;
                    }
                }
            }
        } else {
            for (auto i = 0; i < 50;) {
#pragma omp critical (count_critical)
                {
                    if (count > MIN) {
                        count--;
                        i++;
                    }
                }
            }
        }
```

OpenMP lygiagretusis ciklas

- OpenMP galima automatiškai išlygiagretinti ciklą.
- Tam naudojama direktyva `omp parallel for`.
- Ciklo iteracijos paskirstomos tarp gijų.

OpenMP lygiagretusis ciklas

```
int main() {
    vector<int> v1(VECTOR_SIZE);
    vector<int> v2(VECTOR_SIZE);
    // generate data for vector1 and vector2
    vector<int> v3(VECTOR_SIZE);
#pragma omp parallel for default(none) shared(v1, v2, v3, VECTOR_SIZE)
    for (auto i = 0; i < VECTOR_SIZE; i++) {
        v3[i] = v1[i] + v2[i];
    }
    // do things with v3
}
```

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` direktyvoje kaip `private`, `firstprivate`, `lastprivate` arba `reduction`, yra privatūs.
- Rekomenduojama OpenMP direktyvose naudoti `default none`, kuris priverčia visiems kintamiesiems nurodyti, jie privatūs ar bendri.

OpenMP privatūs kintamieji

`private` parinktis

Parinktis nurodo, kad sukuriama 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

`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.

private veikimas

```
int main() {
    auto c = 99;
    auto *lock = new omp_lock_t;
    omp_init_lock(lock);
#pragma omp parallel num_threads(3) private(c) shared(lock, cout)
    {
        c = omp_get_thread_num();
        omp_set_lock(lock);
        cout << "in thread: " << c << endl;
        omp_unset_lock(lock);
    }
    cout << "after parallel: " << c << endl;
    omp_destroy_lock(lock);
    return 0;
}
```

Programos rezultatas:

```
in thread: 0
in thread: 2
in thread: 1
after parallel: 99
```

lastprivate veikimas

```
int main() {
    auto c = 99;
    auto *lock = new omp_lock_t;
    omp_init_lock(lock);
#pragma omp parallel for lastprivate(c) default(none) shared(lock, cout)
    for (int i = 0; i < 3; i++){
        c = omp_get_thread_num();
        omp_set_lock(lock);
        cout << "in thread: " << c << endl;
        omp_unset_lock(lock);
    }
    cout << "after parallel: " << c << endl;
    omp_destroy_lock(lock);
    return 0;
}
```

Programos rezultatas:

```
in thread: 0
in thread: 2
in thread: 1
after parallel: 2
```

Lygiagreti suma naudojant reduction

```
int main() {
    auto numbers = new int[ARRAY_SIZE];
    fill_array_with_random_numbers(numbers, ARRAY_SIZE);
    auto sum;
#pragma omp parallel reduction(+:sum) default(none) shared(numbers)
    {
        auto total_threads = omp_get_num_threads();
        auto chunk_size = ARRAY_SIZE / total_threads;
        auto thread_number = omp_get_thread_num();
        auto start_index = chunk_size * thread_number;
        auto end_index = thread_number == total_threads - 1 ? ARRAY_SIZE :
            ((thread_number + 1) * chunk_size);
        sum = accumulate(numbers + start_index, numbers + end_index, 0,
                         [](int acc, int curr) { return acc + curr; });

    }
    cout << sum << endl;
}
```