# Objektno orijentirano programiranje C++

## Predavanje 12 - višedretveno programiranje, 2. dio

Marko Živković

Prirodoslovno-matematički fakultet
Matematički odsjek

12. lipnja 2026.

# std::future

**Problem:** treba nam rezultat izračuna za koji je potrebno dulje vrijeme da se izračuna. Recimo da nam točan rezultat ne treba odmah, ali želimo imati varijablu koja drži budući rezultat.

**Rješenje:** Možemo koristiti mutexe, lokote i uvjetne varijable. No C++ ima ugrađen tip baš za ovo: `std::future<T>`, definiran u zaglavlju `future`.

# std::future primjer

```
1  #include <iostream>
2  #include <future>
3
4  int izracunaj() {
5      return 42;
6  }
7
8  int main() {
9      std::future<int> rezultat
10         = std::async(izracunaj);
11
12     //glavni program radi nesto drugo
13
14     int vrijednost = rezultat.get();
15     std::cout << "Rezultat je: " << vrijednost
16         << std::endl;
17 }
```

# std::async

- Osnovna funkcija za stvaranje futura je `std::async`.
- Ona kao parametar može primati funkciju, lambda izraz i funkcijski objekt, slično kao konstruktor of `std::thread`: `std::future<int> fut=std::async([]return 12;);`
- Ako funkcija prima parametre, oni se dodaju iza funkcije u pozivu `std::async`:

```
1  int izracunaj(int x) {
2      // dugotrajno izracunavanje
3      return x * 42;
4  }
5
6  int main() {
7      std::future<int> rezultat
8          = std::async(izracunaj, 10);
9      ....
10 }
```

# Čekanje rezultata

Funkcije definirane na futuru:

- `get()`: Čeka da se funkcija izvrši i vraća rezultat funkcije. Može se pozvati samo jednom.
- `wait()`: Čeka da se funkcija izvrši, ali ne vraća rezultat. Može se pozvati više puta.
- `wait_for(vrijeme)`: Čeka da se funkcija izvrši, ali najviše određeno vrijeme. Vraća status `std::future_status::ready`, `std::future_status::timeout` ili `std::future_status::deferred`.
- Nema funkcije koja postavlja rezutat!

# Kreiranje nove dretve

- Standardno, prevodioc sam odlučuje hoće li `std::async` kreirati novu dretvu ili će se zadatak izvršavati sinkrono kada čekamo na `future`.
- Programer može definirati ponašanje postavljanjem parametra tipa `std::launch`. On može imati ili vrijednost `std::launch::deferred`, koja označava da se izvršavanje funkcije odgađa do naredbi `wait()` ili `get()`, ili `std::launch::async` što označava da se funkcija izvodi u zasebnoj dretvi.
- Ako je izvođenje funkcije odgođeno i ne pozovemo rezultat, može se dogoditi da se funkcija nikada ne izvrši.

# Prosljeđivanje iznimke

Ako funkcijski poziv izvršavan koristeći `std::async` prijavi iznimku, ta se iznimka sprema u `future` umjesto povratne vrijednosti, `future` postaje spreman za korištenje i poziv funkcije `get()` ponovo prijavljuje spremljenu iznimku.

```
1  double korjen(double x) {
2      if(x<0) {
3          throw std::out_of_range("x<0");
4      }
5      return sqrt(x);
6  }
7
8  ...
9  std::future<double> future = std::async(korjen, -2);
10 double rezultat = future.get();
```

Standard ne garantira da će biti vraćen isti objekt iznimke, moguće je dobivanje i kopije.

# std::promise

Umjesto korištenja std::async i funkcije, možemo koristiti objekt tipa std::promise za ručno upisivanje rezultata:

```
1  void sumiraj(std::vector<int>::iterator first,
2               std::vector<int>::iterator last,
3               std::promise<int> promiseSuma) {
4    int sum = std::accumulate(first, last, 0);
5    promiseSuma.set_value(sum);
6  }
7
8  int main() {
9    std::vector<int> brojevi = {2, 4, 6, 8, 10, 12};
10   std::promise<int> promiseSum;
11   std::future<int> futureSum = promiseSum.get_future();
12   std::thread sumator(sumiraj, brojevi.begin(),
13     brojevi.end(), std::move(promiseSum));
14   std::cout << "rez=" << futureSum.get() << '\n';
15   sumator.join();
16 }
```

# Funkcije na `std::promise`

`std::promise` služi za postavljanje tražene vrijednosti. Postavljena se vrijednost ne može direktno pročitati, ali se može dobiti future koji je čita. Funcije na `std::promise`:

- `get_future()`: Vraća future pomoću kojeg ćemo pročitati rezultat. Može se pozvati samo jednom. Mora biti pozvan prije postavljanja rezultata.
- `set_value(vrijednost)`: Postavlja obećanu vrijednost. Može se pozvati samo jednom.
- `set_exception(izuzetak)`: Postavlja izuzetak. Taj će izuzetak biti bačen kada se na future pozove `get()`.

# Dojavljivanje

std::future<void> može se koristiti za dojavljivanje:

```
1  void zadatak(std::promise<void> dojavi) {
2    std::this_thread::sleep_for(
3      std::chrono::seconds(1));
4    dojavi.set_value();
5  }
6
7  int main() {
8    std::promise<void> dojavi;
9    std::future<void> dojavi_future =
10     dojavi.get_future();
11   std::thread radnik(zadatak, std::move(dojavi));
12   dojavi_future.wait();
13   radnik.join();
14 }
```

# std::shared_future

- Funkcija get može se pozvati samo jednom na futuru.
- Ako želimo da rezultat bude dostupa na više mjesta može se koristiti std::shared_future:

```
1 std::shared_future<int> shared_future
2     = future.share();
```

- Funkcija get može se pozvati proizvoljan broj puta na shared futuru.

# Atomic tipovi

**Problem:** Imamo `int brojac` i želimo osigurati da se on mijenja u više dretvi.

**Rješenje:** Možemo koristiti pridruženi mutex kod svake promjene:

```
1 mutex.lock();
2 brojac++;
3 mutex.unlock();
```

**Bolje rješenje:** Definiramo atomski tip `std::atomic_int atomic_brojac(brojac);` i onda jednostavno koristimo

```
1 atomic_brojac++;
```

# Atomic tipovi

- Postoje i drugi atomic tipovi, općenito `std::atomic<T>` za podržani tip `T`.
- Funkcije na atomic tipovima:
  - `void store(T x)` - upisuje vrijednost u atomic,
  - `T load()` - vraća vrijednost upisano u atomic,
  - `T exchange(T x)` - vraća staru vrijednost i postavlja novu,
  - operatori (++, +=, ...), ...
- Atomic tipovi osiguravaju da samo jedna dretva njima pristupa pri jednoj operacije.
- Kod više operacija i dalje je potreban mutex. Npr. atomski tip nije dovoljna zaštita za

```
1 if (brojac > 0)
2     brojac--;
```

# Memorijski redovi

Kod novijih prevodilaca redosljed u kodu, npr.

```
1 a = 5;
2 b = 7;
```

ne garantira redosljed operacija u izvršavanju. Ako su operacije neovisno prevodilac može iz optimizacijskih razloga operacije raditi istovremeno, ili redosljed potpuno zamijeniti. Zato sljedeći kod

```
1 Dretva A:
2 x = 5;
3 flag = true;
4
5 Dretva B:
6 while (!flag);
7 std::cout << x;
```

ne garantira ispis 5!

# Atomic i memorijski redovi

Atomic rješava problem:

```
1  Dretva A:
2  x = 5;
3  flag.store(true);
4
5  Dretva B:
6  while (!flag.load());
7  std::cout << x;
```

garantira ispis 5.
To je zato jer funkcije na atomicu garantiraju i redosljed u memoriji.

# std::memory_order

U biti operacije na atomicu imaju dodatan parametar tipa `std::memory_order` koji može biti:

- `memory_order_relaxed` - ne garantira nikakav redoslijed,
- `memory_order_release` - na operaciji koja upisuje (npr. store) svi memorijski pristupi prije operacije ne smiju se izvršiti nakon nje,
- `memory_order_acquire` - na operaciji koja čita (npr. load) svi memorijski pristupi nakon operacije ne smiju se izvršiti prije nje,
- `memory_order_acq_rel` - kao acquire i release,
- `memory_order_seq_cst` - kao acq_release + sve seq_cst operacije moraju se izvršiti tim redoslijedom - *default*.

# happens-before veza

Minimalni uvjeti koji rade:

```
1  Dretva A:
2  x = 5;
3  flag.store(true, memory_order_release);
4
5  Dretva B:
6  while (!flag.load(memory_order_acquire));
7  std::cout << x;
```

# Memorijske barijere

Moguće je garantirati redoslijede izvršavanja i bez atomskih operacije, koristići *memorijske barijere* `atomic_thread_fence`:

- `atomic_thread_fence(memory_order_release)` - memorijski pristupi prije barijere ne smiju se izvršiti poslije pisanja u memoriju poslije barijere,
- `atomic_thread_fence(memory_order_acquire)` - memorijski pristupi nakon barijere ne smiju se izvršiti prije operacije čitanja iz memorije prije barijere,
- `atomic_thread_fence(memory_order_acq_rel)` - kao acquire i release...

# Implementacija paralelnog QuickSort algoritma

```
1  #include <iostream>
2  #include <list>
3  #include <algorithm>
4  #include <chrono>
5  #include <cstdlib>
6  #include <future>
7  #include <mutex>
8
9  std::mutex lokot;
10
11 template<typename T>
12 std::list<T> pQuickSort(std::list<T> ulaz,
13                         std::atomic_int *brNiti){
14
15    if(ulaz.empty()){ return ulaz; }
```

# Implementacija paralelnog QuickSort algoritma

```
16    std::list<T> rezultat;
17    rezultat.splice(rezultat.begin(),ulaz,
18                                    ulaz.begin());
19    T const& pivot=*rezultat.begin();
20    auto dijeljenje=std::partition(ulaz.begin(),
21      ulaz.end(),[&](T const& t){return t<pivot;});
22      std::list<T> donji_dio;
23      donji_dio.splice(donji_dio.end(),ulaz,
24      ulaz.begin(), dijeljenje);
25
26      std::list<T> gornji_dio;
27      gornji_dio.splice(gornji_dio.end(),ulaz,
28                  dijeljenje,ulaz.end());
29      lokot.lock();
30      if(*brNiti>0){
31          --(*brNiti);
32          lokot.unlock();
```

# Implementacija paralelnog QuickSort algoritma

```
33          std::future<std::list<T> > novi_donji(
34          std::async(std::launch::async,
35          &pQuickSort<T>,std::move(donji_dio),
36                                          brNiti));
37          lokot.lock();
38          if(*brNiti>0){
39              --(*brNiti);
40              lokot.unlock();
41              std::future<std::list<T>> novi_gornji(
42              std::async(std::launch::async,
43              &pQuickSort<T>,
44              std::move(gornji_dio),brNiti));
45              rezultat.splice(rezultat.end(),
46                                  novi_gornji.get());
47              rezultat.splice(rezultat.begin(),
48                                  novi_donji.get());}
```

# Implementacija paralelnog QuickSort algoritma

```
49          else{
50              lokot.unlock();
51              auto novi_gornji(
52              pQuickSort(std::move(gornji_dio),
53                                          brNiti));
54              rezultat.splice(rezultat.end(),
55                                      novi_gornji);
56              rezultat.splice(rezultat.begin(),
57                          novi_donji.get()); } }
58      else{
59          lokot.unlock();
60          auto novi_donji(
61          pQuickSort(std::move(donji_dio),brNiti));
62          rezultat.splice(rezultat.begin(),
63                                      novi_donji);
```

# Implementacija paralelnog QuickSort algoritma

```
64          auto novi_gornji(
65          pQuickSort(std::move(gornji_dio),
66                                          brNiti));
67          rezultat.splice(rezultat.end(),
68                                        novi_gornji);
69      }
70      return rezultat;
71  }
```

# Sekvencijalni QuickSort algoritam

```
1  template<typename T>
2  std::list<T> sQuickSort(std::list<T> ulaz){
3      if(ulaz.empty()){ return ulaz; }
4      std::list<T> rezultat;
5      rezultat.splice(rezultat.begin(),ulaz,
6                                  ulaz.begin());
7      T const& pivot=*rezultat.begin();
8
9      auto dijeljenje=std::partition(ulaz.begin(),
10         ulaz.end(), [&](T const& t){
11                             return t<pivot;});
12     std::list<T> donji_dio;
13     donji_dio.splice(donji_dio.end(),ulaz,
14                 ulaz.begin(), dijeljenje);
15     auto novi_donji(
16         sQuickSort(std::move(donji_dio)));
```

# Sekvencijalni QuickSort algoritam

```
17      auto novi_gornji(
18          sQuickSort(std::move(ulaz)));
19      rezultat.splice(rezultat.end(),novi_gornji);
20      rezultat.splice(rezultat.begin(),novi_donji);
21      return rezultat; }
```

Ispitivanje vremena izvršavanja:

```
1  std::atomic_int bNiti(400);
2
3  int main(void){
4    srand((unsigned) time(NULL));
5    std::list<int> lista, rezultat, rezultat1,
6                                          lista1;
7    int random=0;
```

# QuickSort glavni program

```
8   for(int i=0;i<10000000;i++){
9       random = rand();
10      lista.push_back(random); }
11
12  std::list<int>::iterator it;
13
14  for(it = lista.begin();it!=lista.end();it++)
15      lista1.push_back(*it);
16
17  auto pocetak = std::chrono::steady_clock::now();
18  rezultat = sQuickSort(lista);
19  auto kraj = std::chrono::steady_clock::now();
20  std::cout << "Vrijeme (s): "
21            << std::chrono::duration_cast
22            <std::chrono::seconds>
23           (kraj - pocetak).count()
24            << " s"<<std::endl;
```

# QuickSort glavni program

```
25 std::cout<<provjera(rezultat)<<std::endl;
26 std::cout<<"Velicina: "<<rezultat.size()<<
27                                     std::endl;
28
29 pocetak = std::chrono::steady_clock::now();
30 rezultat1 = pQuickSort(lista1,&bNiti);
31 kraj = std::chrono::steady_clock::now();
32
33 std::cout << "Vrijeme (s): "
34         << std::chrono::duration_cast
35         <std::chrono::seconds>
36         (kraj - pocetak).count()
37         << " sec"<<std::endl;
38
39 std::cout<<provjera(rezultat1)<<std::endl;
40 std::cout<<"Velicina: "<<rezultat1.size()<<
41                                     std::endl;
```

# QuickSort glavni program

```
42 int isti = 1;
43 std::list<int>::iterator it1;
44 it1 = rezultat1.begin();
45 for(it = rezultat.begin();it!=rezultat.end();
46                                          it++){
47       if(*it!=*it1){
48         isti = 0;
49         break;
50       }
51       it1++; }
52
53 if(isti) std::cout<<"Isti!"<<std::endl;
54 else std::cout<<"Razliciti"<<std::endl;
55
56     return 0; }
```

# QuickSort izlaz i provjera

```
1  template<typename T>
2  bool provjera(std::list<T> ulaz){
3      typename std::list<T>::iterator it,it1;
4
5      it = ulaz.begin();
6      it1 = it;
7      ++it1;
8
9      while(it1!=ulaz.end()){
10         if(*it>*it1) return false;
11         ++it; ++it1;
12     }
13
14     return true;
15 }
```