Objektno orijentirano programiranje C++

Predavanje 11 - višedretveno programiranje

Marko Živković

Prirodoslovno-matematički fakultet
Matematički odsjek

9. lipnja 2026.

Kreiranje višedretvenih programa u C++-u

Višedretveni programi imaju mogućnost korištenja više logičkih i/ili fizičkih procesora računala koji istovremeno (paralelno) izvršavaju instrukcije. Time se ubrzava izvođenje koda.

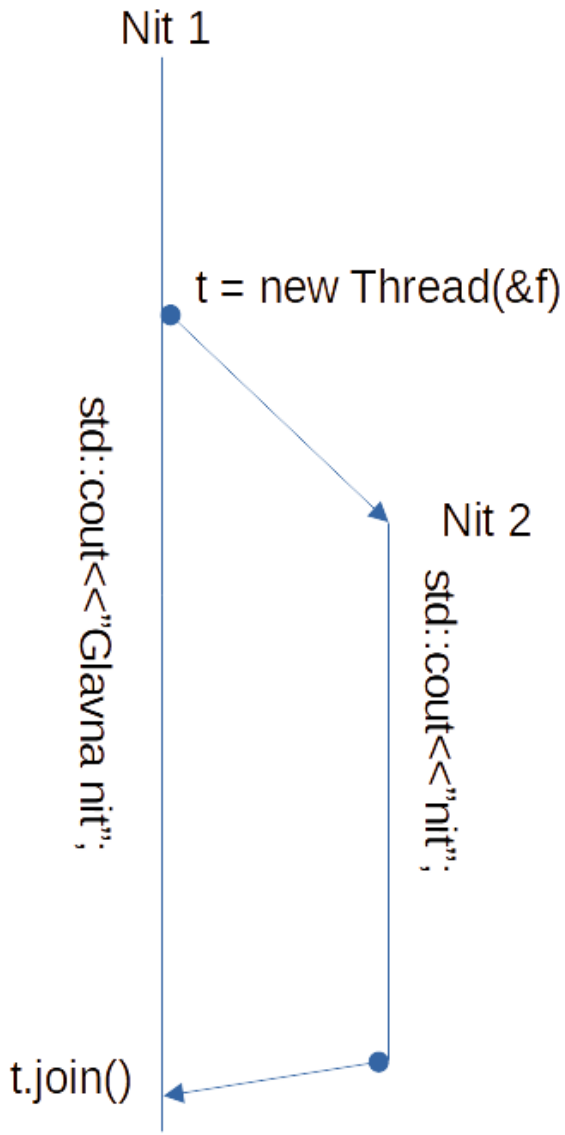
Osnovna klasa koja omogućava stvaranje nove dretve (niti) koja može izvršavati naredbe je klasa `std::thread`.

```
1  //defaultni konstruktor
2  thread() noexcept; //ne povezuje objekt s dretvom
3  //move konstruktor
4  thread( thread&& other ) noexcept;
5  //konstruktor, povezuje objekt s dretvom, pocinje
6  //izvoditi naredbe funkcije f s argumentima Args
7  template< class Function, class... Args >
8  explicit thread( Function&& f, Args&&... args );
9  //copy konstruktor je onemogucen
10 thread( const thread& ) = delete;
```

Pokretanje pomoćne dretve i čekanje na završetak

```
1  #include <iostream>
2  #include <thread>
3
4  void funkcija_dretve() {
5      std::cout << "dretva" << std::endl; }
6
7  int main() {
8      //konstruiranje i pokretanje
9      std::thread t(&funkcija_dretve);
10     //moze:  &f, f i *f
11     //ispis u glavnoj dretvi
12     std::cout << "glavna dretva\n";
13     //glavna dretva ceka zavrsetak nove dretve
14     t.join();
15 }
```

Pokretanje pomoćne dretve i čekanje na završetak



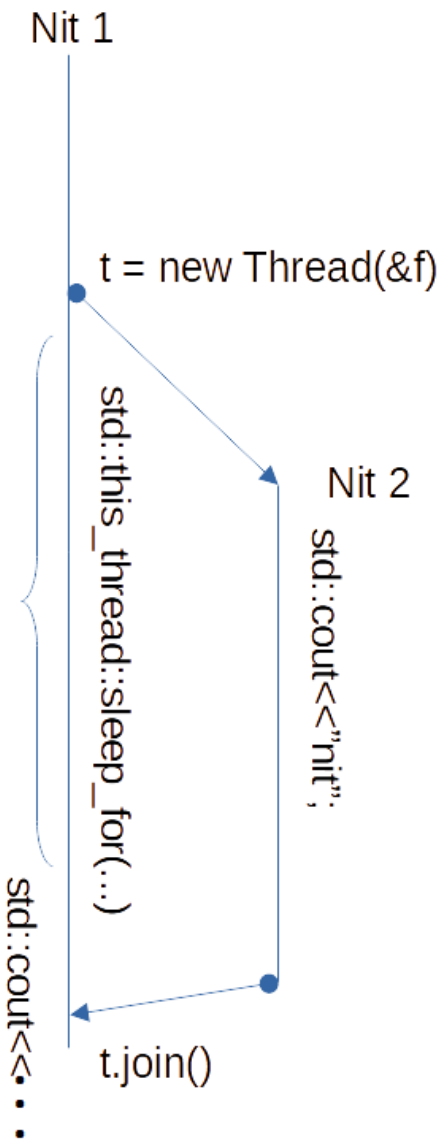
Odnos dretve prema glavnoj dretvi

- Ako varijabla `std::thread t` drži stvarnu dretvu) ona je "joinable".
- Funkcija `bool std::thread::joinable()` provjerava to stanje.
- Dok je `t` joinable ne smije se uništiti (pozvati destruktor). To ruši cijeli program, ne samo dretvu!
- Iz tog stanja `t` može izaći na dva načina:
  - `t.join()` - Glavna dretva čeka da dretva `t` završi. Ako je dretva već završila program samo nastavi.
  - `t.detach()` - Glavna dretva oslobađa dretvu koja nastavlja raditi neovisno. Varijabla `t` više nije vezana za dretvu.
- Unutar C++-u nije moguće direktno zaustaviti dretvu iz druge dretve! Njen se završetak mora ili pričekati (`join`) ili je se pustiti da sama radi (`detach`).
- Završetkom programa OS zaustavlja sve dretve programa (on to može).

Pauziranje izvođenja dretve na određeno vrijeme

```
1  void funkcija_dretve(){
2      std::cout << "dretva" << std::endl;
3  }
4
5  int main() {
6      std::thread t(&funkcija_dretve);
7      //zaustavimo izvrsavanje glavne niti 1s
8      std::this_thread::sleep_for
9                  (std::chrono::seconds(1));
10     //ispis u glavnoj niti
11     std::cout << "glavna dretva\n";
12     //glavna nit ceka zavrsetak nove niti
13     t.join();
14 }
```

Pauziranje izvođenja dretve na određeno vrijeme



Nezavisne dretve

```
1  void funkcija_dretve() {
2      std::this_thread::sleep_for
3          (std::chrono::seconds(1));
4      std::cout << "dretva" << std::endl;
5  }
6
7  int main() {
8      std::thread t(&funkcija_dretve);
9      std::cout << "glavna dretva\n";
10     t.detach();
11     std::this_thread::sleep_for
12         (std::chrono::seconds(2));
13 }
```

Pozivi dretvi s parametrima

```
1  void funkcija_dretve1(int &x) {
2      for(int i=0;i<1000;i++)
3      std::cout << "dretva " <<x++<<std::endl;
4  }
5  void funkcija_dretve2(int x) {
6      for(int i=0;i<1000;i++)
7      std::cout << "dretva " <<x++<<std::endl;
8  }
9
10 int main() {
11     int a = 10;
12     std::thread t(funkcija_dretve1, std::ref(a));
13     t.join();
14     std::thread t1(funkcija_dretve2, a);
15     std::cout << "glavna dretva\n";
16     t1.join();
17 }
```

Pokretanje dretve korištenjem lambda izraza

```
1  #include <iostream>
2  #include <thread>
3
4  int main() {
5      int sum = 0;
6      std::cout << "Main: " << sum << std::endl;
7
8      auto f = [&](){
9          for(int i=0;i<100;i++) sum++; };
10
11     std::thread t(f);
12     t.join();
13
14     std::cout << "Main: " << sum << std::endl;
15 }
```

Pokretanje dretve korištenjem funkcijskog objekta

```
1  #include<iostream>
2  #include<thread>
3
4  class sumator {
5      int *sum;
6
7  public:
8      sumator(int *s) {
9          sum = s;
10     }
11
12     void operator()(int n) {
13         for(int i=0; i<n; i++) *sum+=i;
14     }
15 };
```

Pokretanje niti korištenjem funkcijskog objekta

```
1  int main() {
2      int sum = 0;
3      std::cout << "Main: " << sum << std::endl;
4      std::thread t(sumator(&sum), 100);
5
6      t.join();
7
8      std::cout << "Main: " << sum << std::endl;
9  }
```

Premještanje dretve

Moguće je koristiti funkciju `move` za premještanje dretve:

```
1 std::thread t([]{std::cout << std::this_thread
2                                 ::get_id();});
3 std::thread t1([]{std::cout << std::this_thread
4                                  ::get_id();});
5
6 t.join();
7 t = std::move(t1);
8 t.join();
```

`t1` se može premjestiti u `t` tek kada `t` više nije *joinable*. Nakon premještanja varijabla `t` preuzima dretvu od `t1` i `t1` nije više *joinable*.

Korištenje klasa za ispravan rad s dretvama

Preporučljivo je konstruirati klasu koja će paziti da se nad kreiranim dretvama pozove funkcija `join`. Taj poziv možemo napraviti u destruktoru klase (poziva se u trenutku uništavanja objekta ili izlaska iz dosega).

```
1  class klasaDretve {
2      std::thread t;
3  public:
4      explicit klasaDretve(std::thread t_):
5                                  t(std::move(t_)) {
6      if (!t.joinable()) throw
7              std::logic_error("Nema dretve!"); }
8      ~klasaDretve() { t.join(); }
9      klasaDretve(const klasaDretve&) = delete;
10     klasaDretve& operator=(const klasaDretve&)
11         = delete;
12 };
```

Takvu klasu koristimo:

```
12 int main(){
13   klasaDretve t(std::thread([]{std::cout <<
14                   std::this_thread::get_id() <<
15                   std::endl;}));
16 }
```

Utrke pri pristupu podacima, kritični odsječak

Utrke pri pristupu podacima nastaju kada **dvije ili više dretvi** pri izvršavanju naredbi koriste **dijeljene objekte** (iz dosega dohvatljivog svim dretvama) i **barem jedna dretva vrši pisanje** (izmjenu vrijednosti objekta).

Pojava utrka pri pristupu podacima neće uzrokovati nikakvu formalnu grešku, međutim njihova prisutnost uzrokuje nedeterminističko ponašanje programa, što pobija svrhu tog programa. Problem je u tome što ne znamo hoće li se prije dogoditi pisanje vrijednosti u objekt od strane dretve koja vrši pisanje ili će prije dretve koje čitaju vrijednosti izvršiti učitavanje.

Napomena: korištenje `std::cout` garantirano ispiše znak u konzolu čak i kada više dretvi istovremeno vrši pisanje, međutim poredak ispisa **nije određen**.

Utrke pri pristupu podacima, kritični odsječak

```
1  #include <iostream>
2  #include <thread>
3
4  int main() {
5      int brojac = 0;
6      std::thread t([&](){brojac = 1;});
7
8      std::cout << "Brojac: " << brojac
9          << std::endl;
10     t.join();
11 }
```

Gornji program **nedeterministički** ispisuje ili vrijednost 0 ili vrijednost 1.

Utrke pri pristupu podacima, kritični odsječak

Rješenje: uvesti poredak u višedretvena izvršavanja kod kojih bi potencijalno mogli dobiti **nedeterministički** rezultat.

Poredak uvodimo zaštitom dijeljenog resursa mutex-om (*mutal exclusion*, klasa `std::mutex`).
U primjeru s brojačem, uvodimo poredak na način da zaključamo dijeljenu varijablu `brojacu` glavnoj dretvi prije nego što pokrenemo pomoćnu dretvu, tada pomoćna dretva čeka s izvođenjem dok ne otključamo mutex, te napravi inkrement brojača. Sada je ispis programa uvijek `Brojac: 0`.

Dio koda zaštičen sinkronizacijskim mehanizmima (npr. mutex-om, lokotom) se zove **kritični odsječak**.

Utrke pri pristupu podacima, kritični odsječak

```
1  std::mutex mutex;
2
3  int main() {
4      int brojac = 0;
5      mutex.lock();
6      std::thread t([&](){
7          mutex.lock();
8          brojac = 1;
9          mutex.unlock();});
10     std::cout<<"Brojac: "<<brojac<<std::endl;
11     mutex.unlock();
12     t.join();
13 }
```

Potpuni zastoj

```
1  void zastoj(std::mutex &a, std::mutex &b) {
2    a.lock();
3    std::cout << "Dohvacamo prvi mutex iz niti " <<
4              std::this_thread::get_id()<< std::endl;
5    std::this_thread::sleep_for(
6                       std::chrono::milliseconds(1));
7    b.lock();
8    std::cout << "Dohvacamo drugi mutex iz niti "<<
9              std::this_thread::get_id()<< std::endl;
10   a.unlock();
11   b.unlock();
12 }
```

Potpuni zastoj

```
17 int main() {
18   std::mutex l1;
19   std::mutex l2;
20
21   std::thread t1([&]{zastoj(l1,l2);});
22   std::thread t2([&]{zastoj(l2,l1);});
23   t1.join();
24   t2.join();
25 }
```

Ispis programa:

```
Dohvacamo prvi mutex iz niti 2/3
Dohvacamo prvi mutex iz niti 3/2
```

Nakon navedenog ispisa, program će zauvijek stati.

Zašto?

Lokoti

Lokoti su klase koje se brinu za oslobađanje resursa - pridruženog mutex-a, kada dealociramo odgovarajuću varijablu lokota ili ona iziđe iz dosega.
Obradit ćemo dvije vrste lokota:

- `std::lock_guard` - jednostavan lokot koji automatski otključa mutex nakon dealokacije ili izlaska pridružene varijable iz dosega.
- `std::unique_lock` - kompleksnija vrsta lokota koja omogućava: a) stvaranje bez pridruženog (zaključanog) lokota, b) eksplicitno i ponovljeno postavljanje i otpuštanje povezanog mutex-a, c) premiještanje mutex-a, d) pokušaj zaključavanja mutex-a i e) zaključavanje povezanog mutex-a uz vremensko čekanje ako je lokot zauzet.

Lokoti

```
{
  std::mutex m,
  std::lock_guard<std::mutex> lockGuard(m);
  ...zasticeni dio koda...
}//automatsko oslobadanje mutex-a m
```

```
void sink(std::mutex &a, std::mutex &b) {
  std::unique_lock<std::mutex>
    l1(a, std::defer_lock);
  std::cout << "Nit: " << std::this_thread::
    get_id() << " prvi mutex" <<  std::endl;
  std::this_thread::sleep_for
    (std::chrono::milliseconds(1));
```

Lokoti

```
11    std::unique_lock<std::mutex>
12      l2(b,std::defer_lock);
13    std::cout << "Nit: " << std::this_thread::
14      get_id() << " drugi mutex" <<  std::endl;
15
16    std::cout << "Nit: " << std::this_thread::
17      get_id() << " provjerila oba mutex-a" <<
18                                        std::endl;
19    std::lock(l1,l2);
20    std::cout<<"Kriticni odsjecak, nit: "<<
21      std::this_thread::get_id()<<std::endl;
22  }
```

Lokoti

```
25 int main() {
26   std::cout << std::endl;
27   std::mutex m1;
28   std::mutex m2;
29   std::thread t1([&]{sink(m1,m2);});
30   std::thread t2([&]{sink(m2,m1);});
31   t1.join();
32   t2.join();
33 }
```

Korektna inicijalizacija podataka u višedretvenom okruženju

Inicijalizaciju podataka u višedretvenom okruženju možemo napraviti na korektan način tako da:

- Koristimo konstantne izraze,
- Koristimo funkciju `std::call_once` u kombinaciji sa zastavicom `std::once_flag`,
- Koristimo statičke varijable s dosegom bloka

Korektna inicijalizacija konstantnim izrazima

Konstantni izrazi se inicijaliziraju tijekom prevođenja stoga su sigurni kod izvođenja u višedretvenom okruženju. Izraze činimo konstantnim korištenjem ključne riječi `constexpr` ispred izraza.

Korisnički tipovi mogu biti konstantni izrazi uz sljedeća ograničenja:

- Ne smiju imati virtualnih baznih klasa,
- Konstruktor mora biti `delete` ili `default`. U suprotnom smije koristiti samo jako trivijalne konstrukcije (uz prazne naredbe),
- Ne statički objekti članovi moraju biti inicijalizirani, konstruktori tih objekata moraju biti `constexpr`,
- Metode članice moraju biti `constexpr`

Korektna inicijalizacija konstantnim izrazima

Primjer strukture koja zadovoljava svojstva:

```
1  struct MojInt {
2    constexpr MojInt(int v): vrijednost(v){}
3    constexpr int dohvati(){ return vrijednost; }
4  private:
5    double vrijednost;
6  };
7
8  constexpr MojInt i(12);
9  std::cout << i.dohvati() << std::endl;
```

Korektna inicijalizacija uz std::call_once i std::once_flag

Korištenjem kombinacije std::call_once i std::once_flag osiguravamo da se poziv funkcije dogodi samo jednom bez obzira na broj registracija poziva.

```
1  #include <iostream>
2  #include <thread>
3  #include <mutex>
4
5  std::once_flag zastavica;
6
7  void izvrsi() {
8      std::call_once(zastavica, [](){
9      std::cout << "Izvrsavanje." << std::endl; });
10 }
```

Korektna inicijalizacija uz std::call_once i std::once_flag

```
1  int main() {
2    std::cout << std::endl;
3
4    std::thread t1(izvrsi);
5    std::thread t2(izvrsi);
6    std::thread t3(izvrsi);
7
8    t1.join();
9    t2.join();
10   t3.join();
11
12   std::cout << std::endl;
13 }
```

Ispis: Izvrsavanje.

Korektna inicijalizacija statičkim varijablama s dosegom bloka

Statičke varijable s dosegom bloka kreiraju se samo jednom i spremaju se u memoriju do kraja izvođenja programa. Sve instance tipa dijele statičke varijable članice klase.

```
1  class Podaci {
2  static int x,y;
3    Podaci() = default;
4    ~Podaci() = default;
5    Podaci(const Podaci&) = delete;
6    Podaci& operator=(const Podaci&) = delete;
7  public:
8    static Podaci& dohvati() {
9      static Podaci instanca;
10     return instanca;
11   }
12   static int getX(){return x;}
13   static int getX(){return y;}
14 };
```

Korektna inicijalizacija statičkim varijablama s dosegom bloka

```
14 int Podaci::x = 5;
15 int Podaci::y = 10;
16
17 int main() {
18   std::cout<<Podaci::dohvati().dohvatiX()<<"␣"<<
19         Podaci::dohvati().dohvatiY()<<std::endl;
20
21   std::thread t([](){
22     std::cout<<Podaci::dohvati().dohvatiX()<<"␣"
23       <<Podaci::dohvati().dohvatiY()<<std::endl;
24   });
25   t.join();
26 }
```

Ispis:

```
5 10
5 10
```

Podaci lokalni za dretvu izvršavanja

Podaci lokalni za dretvu izvršavanja se stvaraju za svaku dretvu i ekskluzivno pripadaju dretvi. Kreiraju se kod prvog korištenja i traju dok se izvršava dretva.

```
1  std::mutex lokot;
2  thread_local std::string s("Dretva: ");
3
4  void koja() {
5    std::ostringstream ss;
6    ss << std::this_thread::get_id();
7    std::string s2 = ss.str();
8    s+=s2;
```

Podaci lokalni za dretvu izvršavanja

```
15    std::lock_guard<std::mutex> guard(lokot);
16    std::cout << s << std::endl;
17    std::cout << "&s: " << &s << std::endl;
18    std::cout << std::endl;
19  }
20
21  int main() {
22    std::cout << std::endl;
23    std::thread t1(koja);
24    std::thread t2(koja);
25    std::thread t3(koja);
26    std::thread t4(koja);
27    t1.join(); t2.join(); t3.join(); t4.join();
28  }
```

Primijetimo, lokalni string *s* ima drugačiju adresu u svakoj niti (odnosno svaka nit ima svoju kopiju stringa)!

Varijable uvjeta

Varijable uvjeta su sinkronizacijski mehanizam koji koristi `std::mutex` da bi blokirao jednu ili više dretvi dok neka druga dretva ne modificira dijeljenu varijablu (uvjet) i obavijesti varijablu uvjeta (`condition_variable`).

Dretva koja pokušava promijeniti dijeljenu varijablu mora:

- Dohvatiti `std::mutex`, uglavnom koristeći `std::lock_guard`,
- Modificirati dijeljenu varijablu dok posjeduje lokot,
- Pozvati `notify_one` ili `notify_all` na `std::condition_variable` (može se napraviti nakon otpuštanja lokota).

Varijable uvjeta

Dretva koja čeka `std::condition_variable` mora:

- Dohvatiti `std::unique_lock<std::mutex>` nad mutex-om korištenim za zaštitu dijeljene varijable:
  `std::unique_lock<std::mutex> lokot(mutex);`,
- pozvati
  `wait(lock, predikat);`
  na `std::condition_variable`. Taj poziv automatski otvara mutex, čeka buđenje dretve, provjerava predikat i ako je zadovoljen zatvara mutex i nastavlja.
- Moguće je lažno buđenje dretve (*spurious wakeup*)! To nije bug nego optimizacija OS. Zato treba imati još jednu provjeru kroz predikat koji je obično lambda tipa:
  `[]{return spreman;}`
  gdje je `spreman` globalna `bool` varijabla.

Varijable uvjeta

- `std::condition_variable` radi samo s `std::unique_lock<std::mutex>`.
- `std::condition_variable_any` omogućava varijablu uvjeta koja radi sa `std::shared_lock`.
- Klasa `std::condition_variable` ne može se kopirati konstruktorom kopije ili operatorom pridruživanja, ne može se niti preuzimati vlasništvo nad objektom `move` konstruktorom ili `move` operatorom pridruživanja.

Varijable uvjeta

```
1  #include <iostream>
2  #include <string>
3  #include <thread>
4  #include <mutex>
5  #include <condition_variable>
6
7  std::mutex lokot;
8  std::condition_variable uvjetna;
9  std::string podaci;
10 bool spreman = false;
11 bool obraden = false;
```

Varijable uvjeta

```
12 void radnik() {
13     // Cekaj podatke od main-a
14     std::unique_lock<std::mutex> lk(lokot);
15     uvjetna.wait(lk, []{return spreman;});
16
17     // radnik posjeduje lokot.
18     std::cout << "Obradujem\n";
19     podaci += "␣nakon␣obrade";
20
21     // Salji podatke main-u
22     obraden = true;
23     std::cout << "Obrada␣gotova\n";
24
25     // otkljucaj, dojavi
26     lk.unlock();
27     uvjetna.notify_one();
28 }
```

Varijable uvjeta

```
28 int main() {
29     std::thread radna(radnik);
30     podaci = "Neki primjer";
31     {// posalji podatke radniku
32         std::lock_guard<std::mutex> lk(lokot);
33         spreman = true;
34         std::cout << "main() podaci spremni\n";
35     }
36     uvjetna.notify_one();
37
38     {// cekaj radnika
39         std::unique_lock<std::mutex> lk(lokot);
40         uvjetna.wait(lk, []{return obraden;});
41     }
42     std::cout << "main, podaci = " << podaci;
43     radna.join();
44 }
```