Objektno orijentirano programiranje C++

Predavanje 11 - višedretveno programiranje

Matej Mihelčić

Prirodoslovno-matematički fakultet
Matematički odsjek

26. svibnja 2023.

Višenitni programi imaju mogućnost korištenja više logičkih i/ili fizičkih procesora računala koji istovremeno (paralelno) izvršavaju instrukcije. Time se ubrzava izvođenje koda (u istoj jedinici vremena izvedemo puno više instrukcija).

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

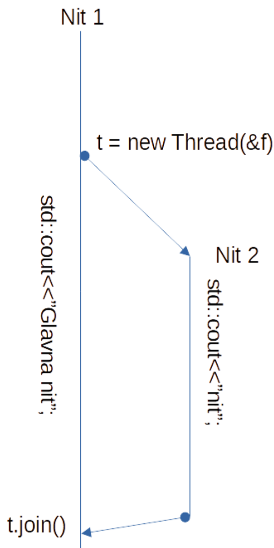
```
//defaultni konstruktor
thread() noexcept; //ne povezuje objekt s niti
//move konstruktor
thread( thread&& other ) noexcept;
//konstruktor, povezuje objekt s niti, pocinje
//izvoditi naredbe funkcije f s argumentima Args
template< class Function, class... Args >
explicit thread( Function&& f, Args&&... args );
//copy konstruktor je onemogucen
thread( const thread& ) = delete;
```

Pokretanje pomoćne niti i čekanje na završetak

Onovni način korištenja niti:

```
1  #include <iostream>
2  #include <thread>
3
4  void funkcija_niti(){
5      std::cout << "nit"<<std::endl; }
6
7  int main(void){
8      //konstruiranje i pokretanje
9      std::thread t(&funkcija_niti);
10 //moze:  &f, f i *f
11     //ispis u glavnoj niti
12     std::cout << "glavna nit\n";
13     //glavna nit ceka zavrsetak nove niti
14     t.join();
15     return 0;}
```

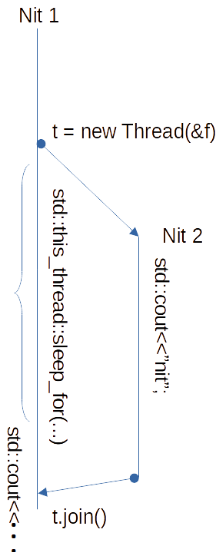
Pokretanje pomoćne niti i čekanje na završetak



Pauziranje izvođenja niti na određeno vrijeme

```
1  #include <iostream>
2  #include <thread>
3  #include <chrono>
4
5  void funkcija_niti(){
6        std::cout << "nit"<<std::endl; }
7
8  int main(void){
9      std::thread t(&funkcija_niti);
10 //zaustavimo izvrsavanje glavne niti 1s
11    std::this_thread::sleep_for
12                (std::chrono::seconds(1));
13     //ispis u glavnoj niti
14     std::cout << "glavna nit\n";
15     //glavna nit ceka zavrsetak nove niti
16     t.join();
17     return 0;}
```

Pauziranje izvođenja niti na određeno vrijeme



Problemi bez `join`

Ukoliko ne koristimo naredbu `join` i glavna nit završi izvođenje prije neke druge niti:

- Ispisi koji su se još trebali dogoditi u pomoćnoj niti se neće izvršiti.
- Ukoliko pomoćna nit koristi podatke kreirane u glavnoj niti (preko reference ili pokazivača) doći će do grešaka pri izvođenju (kao da radimo s pokazivačem nakon poziva funkcije `free/delete`).

Pozivi dretvi s parametrima

```
1  void funkcija_niti1(int &x){
2      for(int i=0;i<1000;i++)
3      std::cout << "nit␣"<<x++<<std::endl; }
4  void funkcija_niti2(int x){
5      for(int i=0;i<1000;i++)
6      std::cout << "nit␣"<<x++<<std::endl; }
7
8  int main(void){
9      int a = 10;
10     //poziv funkcije i prosljedivanje reference
11     std::thread t(funkcija_niti1,std::ref(a));
12      t.join();
13     //poziv i prosljedivanje vrijednosti
14     std::thread t1(funkcija_niti2,a);
15     std::cout << "glavna␣nit\n";
16     t1.join();
17     return 0; }
```

Nezavisne dretve

```
int main(void){
    std::thread t(*funkcija_niti2,1000);
     std::cout << "glavna nit\n";
  //t je nezavisna dretva, izvodi se i nakon
  //zavrsetka izvodenja glavne niti
  t.detach();
  //ispis potencijalno nece biti potpun zbog
  //zavrsetka izvodenja glavne dretve
     return 0;
}
```

Pokretanje niti korištenjem lambda izraza

```
#include <iostream>
#include <thread>

int main(void){
    int sum = 0;
    std::cout<<"Main: "<<sum<<std::endl;

    auto f = [&](){
        for(int i=0;i<100;i++) sum++; };

    std::thread t(f);
    t.join();

    std::cout<<"Main: "<<sum<<std::endl;

    return 0; }
```

Pokretanje niti korištenjem funkcijskog objekta

```
#include<iostream>
#include<thread>

class sumator {
    int *sum;

public:
    sumator(int *s){
        sum = s; }

    void operator()(int n){
        for(int i=0;i<n;i++)
            *sum+=i; }
};
```

Pokretanje niti korištenjem funkcijskog objekta

```
1 int main(void){
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     return 0; }
```

Preuzimanje vlasništva nad objektom niti

Moguće je koristiti funkciju `move` za preuzimanje vlasništva nad objektom niti. Pozivom naredbe:

```
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= std::move(t1);
```

nit t preuzima vlasništvo nad objektom niti t_1, stoga se poziva destruktor nad objektom koji je prethodno pripadao niti t (izvođenje niti se prekida).

Preuzimanje vlasništva nad objektom niti

Ispravno preuzimanje vlasništva se radi:

```
std::thread t([]{std::cout << std::this_thread
                                    ::get_id();});
std::thread t1([]{std::cout << std::this_thread
                                    ::get_id();});

t.join();
t= std::move(t1);
t.join();
```

Nakon izvođenja koda, nit t_1 više ne možemo čekati iz glavnog programa (nije *joinable*). Odnosno, poziv: `std::cout « std::boolalpha « "t1.joinable(): " « t1.joinable() « std::endl;` će vratiti `t1.joinable(): false`.

Korištenje klasa za ispravan rad s nitima

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

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

Korištenje klasa za ispravan rad s nitima

Takvu klasu koristimo:

```
12 int main(void){
13   klasaNiti t(std::thread([]{std::cout <<
14               std::this_thread::get_id() <<
15               std::endl;}));
16     return 0;
17   }
```

Utrke pri pristupu podacima, kritični odsječak

Utrke pri pristupu podacima nastaju kada **dvije ili više niti** pri izvršavanju naredbi koriste **dijeljene objekte** (iz dosega dohvatljivog svim nitima) i **barem jedna nit 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 niti koja vrši pisanje ili će prije niti koje čitaju vrijednosti izvršiti učitavanje.

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

Utrke pri pristupu podacima, kritični odsječak

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

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

Utrke pri pristupu podacima, kritični odsječak

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

Poredak uvodimo zaštitom dijeljenog resursa mutex-om (klasa `std::mutex`).
U primjeru s brojačem, uvodimo poredak na način da zaključamo dijeljenu varijablu `brojac` od strane glavne niti prije nego što pokrenemo pomoćnu nit, tada pomoćna nit 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**.

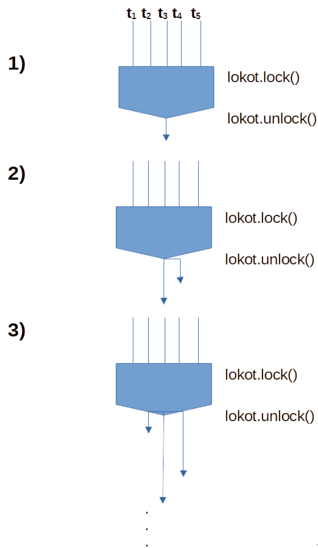
Utrke pri pristupu podacima, kritični odsječak

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex lokot;

int main(void){
    int brojac = 0;
    lokot.lock();
    std::thread t([&](){
        lokot.lock();
        brojac = 1;
        lokot.unlock();});
    std::cout<<"Brojac: "<<brojac<<std::endl;
    lokot.unlock();
    t.join();
    return 0; }
```

Utrke pri pristupu podacima, kritični odsječak



Potpuni zastoj

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

Potpuni zastoj

```
17 int main(void){
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   return 0; }
```

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. Lokoti slijede `RAII` proceduru.
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 ukoliko je lokot zauzet.

Lokoti

```
1  {
2    std::mutex m,
3    std::lock_guard<std::mutex> lockGuard(m);
4    ...zasticeni dio koda...
5  }//automatsko oslobadanje mutex-a m
```

```
1  #include <iostream>
2  #include <chrono>
3  #include <mutex>
4  #include <thread>
5
6  void sink(std::mutex &a, std::mutex &b){
7    std::unique_lock<std::mutex>
8                              l1(a,std::defer_lock);
9    std::cout << "Nit: " << std::this_thread::
10       get_id() << " prvi mutex" <<  std::endl;
```

Lokoti

```
11    std::this_thread::sleep_for
12            (std::chrono::milliseconds(1));
13
14    std::unique_lock<std::mutex>
15                     l2(b,std::defer_lock);
16    std::cout << "Nit: " << std::this_thread::
17     get_id() << " drugi mutex" <<  std::endl;
18
19    std::cout << "Nit: " << std::this_thread::
20     get_id() << " provjerila oba mutex-a" <<
21                                 std::endl;
22    std::lock(l1,l2);
23    std::cout<<"Kriticni odsjecak, nit: "<<
24         std::this_thread::get_id()<<std::endl; }
```

Lokoti

```
25 int main(void){
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   return 0; }
```

Mogućnosti kod konstrukcije objekta tipa `std::unique_lock`:

- `defer_lock_t` - nemoj preuzeti vlasništvo nad mutex-om,
- `try_to_lock_t` - pokušaj dohvatiti vlasništvo nad mutex-om bez blokiranja,
- `adopt_lock_t` - pretpostavi da nit pozivatelj već ima vlasništvo nad mutexom.

Korektna inicijalizacija podataka u višenitnom okruženju

Inicijalizaciju podataka u višenitnom 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 s obzirom na izvođenje u višenitnom okruženju. Izraze činimo konstantnim korištenjem ključne riječi `constexpr` ispred izraza.

Korisnički tipovi mogu biti konstanti 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 constexpr MojInt i(12);
8 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

```
int main(void){
  std::cout << std::endl;

  std::thread t1(izvrsi);
  std::thread t2(izvrsi);
  std::thread t3(izvrsi);
  std::thread t4(izvrsi);

  t1.join();
  t2.join();
  t3.join();
  t4.join();

  std::cout << std::endl;
  return 0; }
```

Rezultat izvođenja: `Izvrsavanje.`

Korektna inicijalizacija statičkim varijablama s dosegom bloka

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

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

Korektna inicijalizacija statičkim varijablama s dosegom bloka

```
14 int Podaci::x = 5;
15 int Podaci::y = 10;
16
17 int main(void){
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   return 0; }
```

Ispis:

5 10

5 10

Podaci lokalni za nit izvršavanja

Podaci lokalni za nit izvršavanja se stvaraju za svaku nit i ekskluzivno pripadaju niti. Kreiraju se kod prvog korištenja i traju dok se izvršava nit.

```
1  #include <iostream>
2  #include <string>
3  #include <mutex>
4  #include <thread>
5  #include <sstream>
6
7  std::mutex lokot;
8  thread_local std::string s("Nit: ");
9
10 void koja(){
11   std::ostringstream ss;
12   ss << std::this_thread::get_id();
13   std::string s2 = ss.str();
14   s+=s2;
```

Podaci lokalni za nit 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   int main(void){
21     std::cout << std::endl;
22     std::thread t1(koja);
23     std::thread t2(koja);
24     std::thread t3(koja);
25     std::thread t4(koja);
26     t1.join(); t2.join(); t3.join(); t4.join();
27     return 0;}
```

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 niti dok neka druga nit ne modificira dijeljenu varijablu (uvjet) i obavijesti varijablu uvjeta (`condition_variable`).

Nit 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

Nit koja čeka std::condition_variable mora:

- Dohvatiti std::unique_lock<std::mutex> nad mutex-om korištenim za zaštitu dijeljene varijable,
- Učini jedno od sljedećeg:
  - Provjeriti uvjet u slučaju da je modificiran ili dojavljen,
  - Pozvati wait, wait_for ili wait_until na std::condition_variable (atomski oslobađa mutex i obustavlja izvođenje dretve dok nije dojavljena varijabla uvjeta, isteklo vrijeme, ili se dogodilo lažno buđenje, tada automatski dohvaća mutex prije vraćanja),
  - Provjeriti uvjet i nastaviti čekati ukoliko nije zadovoljen.

  Može se koristiti i predefinirana verzija funkcija wait, wait_for ili wait_until koja će izvoditi gornja 3 koraka.

std::condition_variable radi samo s std::unique_lock<std::mutex>.

Varijable uvjeta

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

```
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;
```

```
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 int main(void){
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     return 0; }
```