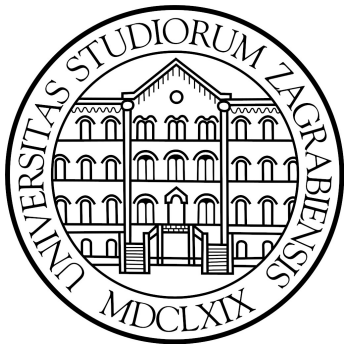
Objektno orijentirano programiranje C++

Predavanje 12 - višedretveno programiranje

Matej Mihelčić

Prirodoslovno-matematički fakultet
Matematički odsjek

02. lipnja 2023.



std::future i std::promise

Tipovi std::future i std::promise su definirani u zaglavlju future. Oni u paru omogućavaju slanje podataka između dvije niti.

```
1  #include <vector>
2  #include <thread>
3  #include <future>
4  #include <numeric>
5  #include <iostream>
6  #include <chrono>
7
8  void sumiraj(std::vector<int>::iterator first,
9                  std::vector<int>::iterator last,
10                 std::promise<int> promiseSuma){
11     int sum = std::accumulate(first, last, 0);
12     promiseSuma.set_value(sum); }
13 // Dojavi pripadnom future-u
```

std::future i std::promise

```
14 void zadatak(std::promise<void> dojavi){
15     std::this_thread::sleep_for(
16                     std::chrono::seconds(1));
17     dojavi.set_value(); }
18
19 int main(void){
20  std::vector<int> brojevi = {2, 4, 6, 8, 10, 12};
21  std::promise<int> promiseSum;
22  std::future<int> futureSum =
23                       promiseSum.get_future();
24  std::thread sumator(sumiraj, brojevi.begin(),
25        brojevi.end(), std::move(promiseSum));
26
27  std::cout << "rez=" << futureSum.get() << '\n';
28  sumator.join();
```

```
29  std::promise<void> dojavi;
30  std::future<void> dojavi_future =
31                                  dojavi.get_future();
32  std::thread radnik(zadatak, std::move(dojavi));
33  dojavi_future.wait();
34  radnik.join(); }
```

get() metoda objekta tipa std::future omogućava čekanje na rezultat koji šalje povezani objekt tipa std::promise iz niti radnika. Ovaj mehanizam je puno generalniji od običnog korištenja join jer omogućava proslijeđivanje vrijednosti, iznimaka i notifikacija (slično korištenju varijabli uvjeta) iz niti radnika.

`std::future` se ne smije koristiti bez sinkronizacije ukoliko više niti istovremeno čeka rezultat ili dojavu od strane niti radnika.

Postoje dvije vrste objekata kojima možemo čekati na podatke ili obavijest niti, `std::future<>` koji omogućava čekanje na događaj jedne niti, dok više instanci `std::shared_future<>` mogu biti povezani s istim događajem. Ukoliko je više objekata tipa `std::shared_future<>` povezano s istim događajem, oni će svi dobiti podatke/obavijest u isto vrijeme, te mogu pristupati dohvaćenim podacima. Specijalizacije `std::future<void>` i `std::shared_future<void>` se koriste bez podataka za slanje obavijesti (npr. zadatak je izvršen, podaci su spremni itd.).

Zadaci

Zadaci se stvaraju pomoću poziva parametrizirane funkcije `std::async`. Parametrizirana funkcija `std::async` ima iste parametre kao i konstruktor klase `std::thread`. Kao prvi parametar joj možemo proslijediti funkciju, lambda izraz ili funkcijski objekt.

`std::async` kao povratnu vrijednost vraća objekt tipa `std::future`, preko kojeg nit koja je pozvala zadatak može dohvatiti rezultat/obavijest.

```
1  int rez;
2  std::thread t([&]{rez= 12;});
3  t.join();
4  std::cout << rez << std::endl;
5
6  auto fut=std::async([]{return 12;});
7  std::cout << fut.get() << std::endl;
```

Zadaci

Standardno, prevodioc sam odlučuje hoće li `std::async` kreirati novu nit 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 niti. Standardna postavka `std::launch::deferred | std::launch::async` označava da prevodioc može izabrati jednu od dvije opcije. Ukoliko je izvođenje funkcije odgođeno, može se dogoditi da se nikada ne izvrši.

Zadaci

```
1  //pokrecemo zadatak z6 u novoj niti
2  auto z6=std::async(std::launch::async,A(),3);
3  //pokrecemo z7 tek kada naidemo na naredbu
4  //wait() ili get()
5  auto z7=std::async(std::launch::deferred,baz,
6                     std::ref(x));
7  //prevodioc bira kako izvrsiti zadatak z8
8  auto z8=std::async(
9     std::launch::deferred | std::launch::async,
10    baz,std::ref(x));
11 //prevodioc bira kako izvrsiti zadatak z9
12 auto z9=std::async(baz,std::ref(x));
13 //izvrsavamo zadatak z7
14 z7.wait();
```

Prosljeđivanje iznimke u zadacima

```
1  double korjen(double x){
2      if(x<0)
3      {
4          throw std::out_of_range("x<0");
5      }
6      return sqrt(x); }
7
8  double x=square_root(-2);
```

Ukoliko funkcijski poziv izvršavan koristeći `std::async` prijavi iznimku, ta iznimka se sprema u `future` umjesto povratne vrijednosti, `future` postaje spreman za korištenje i poziv funkcije `get()` ponovo prijavljuje spremljenu iznimku. Standard ne garantira da će biti vraćen isti objekt iznimke, moguće je dobivanje i kopije.

Prosljeđivanje iznimke u zadacima

Iznimka se može spremiti i koristeći `promise`:

```
1 extern std::promise<double> pr;
2 try{
3     pr.set_value(izracunaj());}
4 catch(...){
5     pr.set_exception(
6                 std::current_exception());}
```

Gornji kod koristi `std::current_exception()` za dohvaćanje prijavljene iznimke. Alternativa je korištenje `std::make_exception_ptr()` za direktno spremanje iznimke bez prijavljivanja:

```
1 some_promise.set_exception(
2       std::make_exception_ptr(
3             std::logic_error("foo "));
```

Prosljeđivanje iznimke u zadacima

Iznimku možemo spremiti u `future` i uništavanjem pridruženog objekta tipa `std::promise` ili `std::packaged_task`. Destruktor objekta `std::promise` ili `std::packaged_task` će spremiti iznimku tipa `std::future_error` s kodom `std::future_errc::broken_promise` ukoliko `future` nije spreman. Ukoliko prevodioc ne spremi ništa u `future` kod takvog postupka, niti mogu zauvjek čekati na `wait`-u.

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 }
```

QuickSort izlaz i provjera

Izlaz programa:

```
Vrijeme(s): 21 s //sekvencijalni QS
1
Velicina: 10000000
Vrijeme(s): 15 s //paralelni QS
1
Velicina: 10000000
Isti!

Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz.
```

`std::packaged_task`

`std::packaged_task` omogućava omatanje funkcija, lambda izraza ili drugih funkcijskih objekata, te omogućava da se ti objekti pozovu asinkrono. Povratna vrijednost objekata tipa `std::packaged_task` ili iznimka se spremaju u dijeljeno stanje kojem se može pristupiti kroz objekt tipa `std::future`.

Višenitno računanje skalarnog umnoška dva vektora

```
1  #include <algorithm>
2  #include <future>
3  #include <iostream>
4  #include <thread>
5  #include <deque>
6  #include <vector>
7
8  class Skalarni{
9  public:
10   Skalarni(int b, int e, std::vector<int> &v1,
11            std::vector<int> &v2): beg(b),end(e),
12            prvi(v1), drugi(v2){}
```

Višenitno računanje skalarnog umnoška dva vektora

```
13    int operator()(){
14      long long int sum{0};
15      for (int i= beg; i < end; ++i )
16                sum += prvi[i]*drugi[i];
17      return sum; }
18
19  private:
20      int poc, kraj;
21      std::vector<int> &prvi, &drugi;
22  };
23
24  static const unsigned int PBN= 4;
25  static const unsigned int duljina= 20;
```

Višenitno računanje skalarnog umnoška dva vektora

```
26 int main(void){
27   unsigned int niti= std::thread::
28                          hardware_concurrency();
29   unsigned int BN= (niti != 0)? niti : PBN;
30
31   std::vector<int> v1, v2;
32
33   for(int i=0;i<duljina;i++){
34    v1.push_back(i+1);
35    v2.push_back(i+2); }
36
37   std::vector<Skalarni> skalarni;
38   for ( unsigned int i= 0; i < BN; ++i){
39    int begin= (i*duljina)/BN;
40    int end= (i+1)*duljina/BN;
41    skalarni.push_back(Skalarni(begin,end,v1,v2));
42   }
```

Višenitno računanje skalarnog umnoška dva vektora

```
43   std::deque<std::packaged_task<int()>> zadaci;
44   for ( unsigned int i= 0; i < BN; ++i){
45     std::packaged_task<int()> s(skalarni[i]);
46     zadaci.push_back(std::move(s)); }
47
48   std::vector< std::future<int>> rezultati;
49   for ( unsigned int i= 0; i < BN; ++i){
50     rezultati.push_back(zadaci[i].get_future());}
51
52   while ( not zadaci.empty() ){
53     std::packaged_task<int()> zadatak=
54                             std::move(zadaci.front());
55     zadaci.pop_front();
56     std::thread zadatakNit(std::move(zadatak));
57     zadatakNit.detach(); }
```

Višenitno računanje skalarnog umnoška dva vektora

```
58    int sum= 0;
59    for ( unsigned int i= 0; i < hwConcurr; ++i){
60      sum += rezultati[i].get(); }
61
62    std::cout << "skalarni = " << sum << std::endl;
63
64    std::cout << std::endl;
65     return 0; }
```

Ispis programa:

```
skalarni = 3080
```

Memorijske barijere

Operacije pisanja (P) i čitanja (Č) se mogu generalno javiti u 4 različita poretka: 1) PP, 2) PČ, 3) ČP, 4) ČČ. Ukoliko operacije čitanja i pianja radimo u dvije različite niti, bez sinkronizacijskih mehanizama, naredbe u paru se mogu izvršiti u proizvoljnom redosljedu.

Postavljanje memorijske barijere između para operacija osigurava da će se one izvršiti u točno određenom redosljedu. Postoje tri vrste memorijske barijere:

- Puna memorijska barijera `std::atomic_thread_fence()` - onemogućava izmjenu poretka dvije operacije osim operacija tipa PČ.
- Barijera dohvaćanja (eng. *acquire fence*) `std::atomic_thread_fence(std::meemory_order_acquire)` - onemogućava da se operacije čitanja prije barijere izmjene s operacijama čitanja ili pisanja nakon barijere.

Memorijske barijere

- Barijera otpuštanja (eng. *release fence*) `std:memory_thread_fence(std::memory_order_relase)` - onemogućava izmjenu operacija čitanja ili pisanja prije barijere s operacijama pisanja nakon barijere.

```
1  #include <iostream>
2  #include <atomic>
3  #include <thread>
4
5  std::atomic<bool> postavljen(false);
6  int a;
7
8  void izracun(){
9    a = 100;
10   atomic_thread_fence(std::memory_order_release);
11   postavljen.store(true,
12                    std::memory_order_relaxed); }
```

Memorijske barijere

```
13 void procitaj(){
14   while(!postavljen.load(
15             std::memory_order_relaxed))
16         ;
17
18   atomic_thread_fence(std::memory_order_acquire);
19   std::cout << a << '\n';  }
20
21 int main(void){
22     std::thread t1 (izracun);
23     std::thread t2 (procitaj);
24
25     t1.join(); t2.join();
26     return 0;
27 }
```