UNIVERZITET U NOVOM SADU
FAKULTET TEHNIČKIH NAUKA
KATEDRA ZA PRIMENJENE RAČUNARSKE NAUKE

Paralelno računarstvo

Računarske vežbe

Letnji semestar 2020/2021.
Studijski program: Informacioni inženjering

Obećanja (*engl. Promises*)

Obećanja

- C++11 standard nudi mehanizam za rukovanje povratnim vrednostima koji je dizajniran tako da ispunjava karakteristike asinhronog izvršavanja
- Programeri mogu da definišu takozvana obećanja (*engl. promises*) koji će biti ispunjeni u budućnosti.
- Implementirana su kroz par objekata (p,f), gde je p (*engl .promise*) obećanje i preko njega postavljamo stanje objekta. Sa druge strane f (*engl. future*) je objekat preko koga pristupamo vrednosti obećanja p, nakon što je obećanje ispunjeno.
- Vrednost objekta p se može postaviti samo jednom i naziva se ispunjenje obećanja.

Obećanja

- Kroz mehanizam obećanja ostvarujemo uzročno-posledičnu vezu između objekata p i f koja se može koristiti za sinhronizaciju između glavne niti i pokrenute niti
- Obećanja su zapravo apstrakcija nad primitivama za sinhronizaciju

Obećanja

Upotreba objekata p i f:

- Prvo napravimo obećanje p koje ima svoj tip T **std::promise<T> p;**
- Napravimo objekat f koji je povezan sa obećanjem p

  std::future<T> f = p.get_future();
- Pošaljemo referencu objekta p niti (**std::promise<T> && p**) . P se mora kopirati kroz **std::move()** funkciju.
- Obećanje se ispunjava u telu pokrenute niti **p.set_value(some_value);**
- Na kraju kroz objekat f se isčitava povratna vrednost u glavnoj nit **f.get()**
- Glavna nit je blokirana sve dok obećanje p ne bude ispunjeno kroz objekat f.

Zadatak - Obećanja

- Implementirati zadatak koji izračunava prvih n elemenata fibonacijevog niza. Svaki element niza se računa koristeći posebnu nit. Sinhronizaciju niti i prebacivanje rešenja u glavnu nit imlementirati koristeći obećanja.

Uprošćeno asinhrono izvršavanje

- C++11 nudi olakšani mehanizam za rad sa asinhronim funkcijama i obećanjima kroz **std::async**
- Komanda **std::async** izvršava zadatak upotrebom glavne niti ili kreiranjem nove niti
- Kreiranje zadatka i objekta f je veoma jednostavno

  auto future = std::async(fibo, id);

Uprošćeno asinhrono izvršavanje

Stvari na koje treba obratiti pažnju pri korišćenju **std::async** :

- Pri naivnom pozivu **std::async** nemamo garancije da će se napraviti nova nit. Moguće je da postojeća nit izvrši zadatak
- Izvršavanje zadatka može biti zauvek odloženo ukoliko ne pristupimo vrednosti f objekta, kroz **future.get()**

Uprošćeno asinhrono izvršavanje

Moguće je bliže specificirati **std::async** comandi kako želimo da se ponaša:

- **std::launch::async** pokreće nit i odma izvršava zadatak
- **std::launch::deferred** izvršava zadatak na lenji način tj. zadatak se izvršava na istoj niti pri pozivu metode **future.get()**

Zadatak – std::async

- Implementirati zadatak koji izračunava prvih n elemenata fibonačijevog niza. Pri implementaciji koristiti std::async
- Obratiti paznju da loše napisan kod može dovesti do sekvencijalno izvršavanja, iako se koristi std::lunch::async

```
for (uint64_t id = 0; id < num_threads; id++) {
auto future = std::async(std::launch::async, fibo, id);
} // ← Ovde se poziva destruktor futur objekta
```

Zadatak – dodatni

- Implementirati funkciju koji izračunava **n-ti** elemenata fibonačijevog niza **rekurzivno**. Implementirati algoritam tako da je svaki rekurzivni poziv nova nit. Pri implementaciji koristiti std::async.
- Kakve su performanse ovako impelentiranog resenja?

Atomics

Atomics

- Do sada smo implementirali algoritme koji su na neki način zaobilazili trku do podataka (*engl. race condition*) ili su koristili mutex za zaključavanje kritičnih sekcija koda
- Problem je u tome što se neki problemi ne mogu rešiti bez međusobnog isključivanja, a zaključavanje je skupa operacija koje zahtevaju puno procesorskog vremena

Atomics

- Čest primer korišćenja mutex objekta jeste prosto povećavanje brojača
- Ukoliko pri povećavanju brojača koristimo mutex većina procesorskog vremena će se potrošiti na zaključavanje i otključavanje kritičnih sekcija ili čekanje, što je ekstremno neefikasno.
- Kako bi se rešio ovaj problem u C++11 standardu se uvodi koncept atomičnih tipova podataka koji se mogu bezbedno koristiti u konkurentnom okruženju bez potrebe za korišćenjem skupih zaključavanja.

Atomics

- Moderni CPU i GPU podržavaju napredne hardverske metode za atomičan rad sa 32 i 64-bitnim integer vrednostima
- Takve instrukcije omogućavaju atomično **uvećavanje/umanjivanje vrednosti** promenljive ili **zamenu vrednosti dve promenljive** bez izazivanja prekida.

  std::atomic<T> variable

Zadatak - Atomics

- Implementirati funkciju za atomično povećavanje brojača. Više niti izvršava istu funkciju. Niti izvršavaju operaciju povećavanja brojača ciklično.
- Uporediti performanse napisane funkcije sa performansama funkcije koja koristi mutex.

Atomics

- Atomične promenljive funkcionišu na očekivani način u konkurentnom okruženju zato što je hardverski onemogućeno pravljenje prekida pri izvršavanju instrukcija nad njima.
- Ukoliko se sa njima radi kroz dve ili više instrukcije, može doći do prekida i nepredviđenog ponašanja.

```
std::atomic<uint64_t> atomic (0);
{ // više instrukcija izvršava kod
// ekvivalentno sa value=atomic.load()
value = atomic; // 1. operacija
value++; // 2. operacija
//ekvivalentno sa atomic.store(value)
atomic = value; // 3. operacija
}
```

- Da li bi **atomic++;** rešilo problem?

Atomics

Operacije koje se mogu bezbedno koristiti sa atomičnim promenljivim:

- operator++
- operator--
- operator+=
- operator-=
- operator&=
- operator|=
- operatorˆ=

Atomics

- Funkcije poput max i min nije moguće izračunati upotrebom ovih operacija
- Zato postoje CAS (*engl. compare and swap*) operacija
- Postoje **compare_exchange_strong()** koja je potpuno sigurna i **compare_exchange_weak()** koja pruža bolje performanse ali se može desiti da se ne izvrši svaki put.
- Izvršavanjem **compare_exchange_weak()** u petlji dobijamo dobre performanse i sigurnost

Atomics

atomic.compare_exchange_weak(T& expected, T desired);

CAS operacija nad std::atomic<T> prihvata dva argumenta:

- Referencu na vrednost za koju se očekuje da se nalazi u atomičnoj promenljivoj
- Vrednost koja će se smestiti u atomiču promenljivu ukoliko je tačan uslov **atomic.load() == expected**, tj. ukoliko se u atomic promenljivoj nalazi očekivana (*engl. expected*) vrednost

CAS izvršava tri koraka na atomičan način:

- Poredi vrednost **expected** sa vrednošću koja se nalazi u promenljivoj atomic
- Ukoliko su vrednosti jednake, postaviti vrednost **desired** kao novu vrednost atomične promenljive
- Vrati *true* ukoliko je postavka **desired** vrednosti u prethodnoj stavci uspešna, u suprotnom vrati *false*

Zadatak - Atomics

- Implementirati funkciju za računanje maksimalnog element koristeći CAS operaciju. Već implementiranu funkciju incorect_max prekopirati i izmeniti tako da računa max na pravilan način.