Paralelne strukture podataka bazirane na međusobnom isključivanju

Kuzmić, Mislav

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Mislav Kuzmić

PARALELNE STRUKTURE PODATAKA BAZIRANE NA MEĐUSOBNOM ISKLJUČIVANJU

Diplomski rad

Voditelj rada:
Mladen Jurak

Zagreb, 2018.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

Cilj ovog rada je proučiti višedretvenost, te obratiti posebnu pažnju na strukture podataka i mehanizme koje, kao podršku višedretvenom programiranju, pruža programski jezik C++. Rad će se primarno baviti standardom iz 2011. godine, a dotaknut će se i podrške koju pružaju standardi programskog jezika C++ iz 2014. (C++14) i 2017. (C++17) godine, tj. razlike u odnosu na standard iz 2011. godine (C++11).

Višedredvenost je način programiranja koji, u novije doba, postaje neizbježan alat, odnosno način programiranja, i pridonosi značajna poboljšanja u performansama softvera napisanih višedretvenim programiranjem u odnosu na one napisanim korištenjem starijih metoda sekvencijalnog programiranja. Svakako treba pripaziti s pojmom višedretvenosti. Višedretvenost se često može zamijeniti s pojmom paralelnosti, pojmom koji označava odvijanje više od jedne radnje istovremeno. Naravno, krajnji cilj višedretvenog programiranja je postizanje paralelizma unutar aplikacije u kojoj se višedretvnost koristi, međutim paralelnost je moguće postići i drugim metodama, kao što su metode distribuiranog programiranja. Kod distribuiranog programiranja paralelizam se postiže podijelom posla između više procesa, koji mogu biti pokrenuti na različitim računalima. Za razliku od distribuiranog programiranja, višedretveno programiranje ograničava se na paralelizam unutar jednog procesa, dakle mogućnost da se više radnji odvija u isto vrijeme unutar jednog procesa. Ovaj rad bavi se isključivo višedretvenim programiranjem, odnosno podrškom za višedretveno programiranje koje pruža programski jezik C++, tako da se tema distribuiranog programiranja (te slične teme) više neće spominjati.

Višedretveno se programiranje proučava već duži niz godina, te se također vrlo rano počinje koristit u znanstvenom računanju na jačim višeprocesorskim računalima. S napretkom tehnologije višedretveno programiranje počinje se koristiti i za programe pisane za osobna računala radi povećanja performansi i kvalitete spomenutih programa. U najranijim slučajevima korištenja višedretvenog programiranja za osobna računala procesori su se najčešće sastojali od jedne jezgre. U tom slučaju višedretvenost je bila samo prividna, svakoj dretvi dano je određeno procesorko vrijeme, nakon čega bi se aktivnost

prebacila na drugu dretvu i u takvim intervalima dobila bi se ta prividna višedretvenost. Svakako i takav pristup višedretvenom programiranju dao je bolje rezultate u odnosu na sekvencijalno. Nadalje, s razvojem tehnologije pojavili su se višejezgreni procesori, koji se koriste i danas, na kojima je višedretvnost postala puno bliža onoj definiciji paralelnog računanja. Naime, svaka jezgra sada može odrađivati posao jedne, pa i više, dretvi istovremeno čime se postiže željeni paralelizam. Međutim, najbolji primjer paralelnog izvršavanja korištenjem višedretvenog programiranja i dalje predstavljaju višeprocesorska računala, kod kojih svaki procesor može odrađivati posao jedne dretve. Ovime su predstavljeni načini na koje se višedretvenost može odraditi na razini hardvera, ali bitnija tema, pa tako i tema ovog rada, je način na koji se softverski može dobiti višedretveni način rada i na što sve treba pripaziti kod višedretvenog pristupa programiranju.

Do sada je rečeno samo o pozitivni stvarima vezanim u višedretveno programiranje. Da bi se stekao potpuni dojam višedretvenog programiranja, treba se dotaknuti i određenih problema, koji se eventualno mogu pojaviti prilikom razvoja programa. Naime, unatoč tome što višedretveno programiranje doprinosi performansama softvera, lako se može dogoditi da se postigne upravo suprotno. Neispravno korištenje višedretvenih metoda može doprinjeti usporavanju rada softvera, kao npr. u slučaju kada se pokrene prevelik broj dretvi unutar nekog procesa (bit će jasnije o čemu je riječ nakog uvoda sljedećeg poglavlja, koje će biti posvećeno dretvama). Višedretveno programiranje može imati i puno gore posljedice ukoliko se višedretvenom razvoju ne pristupi na ispravan način, a može doći i do prestanka rada softvera ili čak nepredviđenom radu u najgorem slučaju. U narednim poglavljima ovog rada bit će opisane metode i mehanizmi kojima se takve neželjene posljedice mogu izbjeći na efikasan način.

U sljedećem poglavlju bit će dane osnovne napomene vezane uz pojam dretve, što je to dretva, kako se ona ponaša u pogledu memorijskog modela računala i na koji način se dretve mogu podijeliti na različite tipove. Zatim će biti opisan način kreiranja dretve u programskog jeziku C++, te osnove za kontrolu više dretvi, a poglavlje završava jednostavnim primjerom višedretvenog koda napisanog u C++ programskom jeziku.

Poglavlje 1

Dretve

Dretva je najmanja cjelina jednog procesa, a proces se može sastojati od više dretvi. Dretva predstavlja jednu nezavisnu kontrolu toka unutar procesa. Dretve, odnosno višedretvenost, izuzetno su bitne za operacijske sustave. Svaki od danas modernijih operacijskih sustava u svom radu koristi višedretven način rada. Operacijski sustav također je djelomično odgovoran za upravljanje dretvama. Time dolazimo do podjele drevi. Postoje tzv. dretve na razini jezgre operacijskog sustava i korisničke dretve, a oba tipa dretvi bit će opisana u sljedećim podpoglavljima.

1.1 Sistemske dretve

Sistemske dretve pod potpunim su nadzorom jezgre operacijskog sustava i općenito su sigurnije za upotrebu, s obzirom da za ispravan rad nema potrebe za ikakvim dodatnim postupcima nad sistemskim dretvama. Jezgra operacijskog sustava rezervirat će sve potrebne resurse za kreiranje i ispravan rad dretve, a odgovorna je i za međusobnu komunikaciju dretvi, te zaštitu od gubitaka podataka ili neželjenih i nepredvidivih modifikacija nad podacima. Kasnije će biti opisani mogući problemi i poslijedice koje mogu nastati prilikom pogrešnog načina međusobne komunikacije, koje mogu nastati korištenjem korisničkih dretvi. Također, bit će dane metode i mehanizmi potrebni za ispravno i odgovorno upravljanje dretvama, kao što je ranije spomenuto.

Svaki proces sadrži barem jednu sistemsku dretvu. Naime, glavni tok svakog procesa odvija se upravo unutar takvog tipa dretve, a isti je naknadno moguće razgranati na više tokova korištenjem višedretvenog programiranja tako da se kreiraju ili sistemske ili korisničke dretve, a moguća je i kombinacija tipova. Više o tome bit će spomenuto kada će se govoriti o modelima.

Prednost sistemske dretve je razina sigurnosti i jednostavnosti koju pruža činjenica da je sva kontrola prepuštena jezgri operacijskog sustava. To podrazumijeva mogućnost jezgre da simultano upravlja dretvama unutar nekog procesa na više aktivnih procesa, te u slučaju pojave blokade nad dretvama unutar nekog procesa, jezgra će osigurati ispravno upravljanje nove dretve za isti proces. Međutim, kreiranje sistemskih dretvi zahtjeva više vremena i puno je složeniji proces u odnosu na proces kreiranja korisničke dretve, a prebačaj kontrole nad dretvom između dretvi unutar istog procesa zahtjeva administratorski način rada.

1.2 Korisničke dretve

Drugi tip dretve su korisničke dretve. Tu se radi o dretvama koje su programski kreirane unutar koda korištenjem posebne biblioteke (eng. *thread library*) koja omogućava kreiranje i upravljanje nad dretvama. Jezgra operacijskog sustava sada nema nikakav nadzor nad dretvama, štoviše jezgra nema niti informaciju o postojanju ovog tipa dretvi, i svi mehanizmi za ispravan rad prepušteni su programeru na brigu. To naravno znači da je sada programer odgovoran za uspostavu ispravne komunikacije među dretvama i da pogramer mora napraviti metode koje će se pobrinuti za zaštitu dijeljenih podataka i ostalih resursa. Naravno, glavna dretva procesa, ona u kojoj se odrađuje posao glavnog toka kontrole pokrenutog funkcijom `main()`, i dalje je sistemska dretva. Jezgra operacijskog sustava u svakom trenutku mora imati nadzor nad svakim pokrenutim procesom, kako bi sustav ispravno radio.

Prednost korisničih dretvi je mogućnost potpunog nadzora i kontrole nad dretvama, te jednostavnost i cijena kreiranja korisničkih dretvi. Također, aplikacija nema potrebu prelaska u administratorski način rada, što smanjuje mogućnost nepredvidivih akcija koje proces može pokrenuti unutar sustava. Spomenuta sloboda koju korisničke dretve omogućavaju dolaze uz brigu o ispravnom načinu rada svake dretve i pisanju mehanizama za usklađivanje i komunikaciju nad dretvama (o čemu će biti puno više rečeno u kasnijim poglavljima). Dakle, kod rada s dretvama potrebno je dobro i pažljivo proučiti potrebe za dretvama i odlučiti koji tip dretve je pogodniji za korištenje u određenoj situaciji.

Unutar iste aplikacije moguće je i kombinirati sistemske i korisničke dretve, a postoje i modeli koji omogućavaju lakše i efikasnije korištenje različitih tipova dretvi unutar aplikacije. Kombinacijom dretvi unutar aplikacije dobiva se mogućnost iskorištavanja prednosti i jednog i drugog tipa. Korištenjem više sistemskih dretvi postiže se mogućnost prave paralelizacije. Također, korištenje više sistemskih dretvi osigurava da pojava blo-

kade na nekoj dretvi neće prouzročiti blokadu cijelog procesa, dio procesa koji svoj posao odrađuje na drugim granama sistemskih dretvi nastavit će s radom. Ukoliko se pojavi potreba za većom kontrolom nad nekom dretvom, može se kreirati user level dretva.

Tri su tipa modela koji koriste gore navedeni pristup višedretvenom programiranju, a to su:

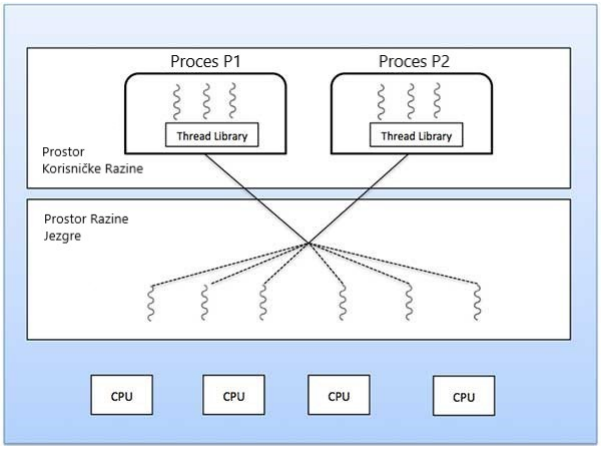
- više prema više model
- više prema jedan model
- jedan prema jedan model

U nastavku slijede kratki opisi navedenih modela.

1.3 Više prema više model

U modelu više prema više broj sistemskih dretvi koje granamo na user level dretve je manji ili jednak broju user level dretvi.

Slika 1.1: Primjer modela više prema više



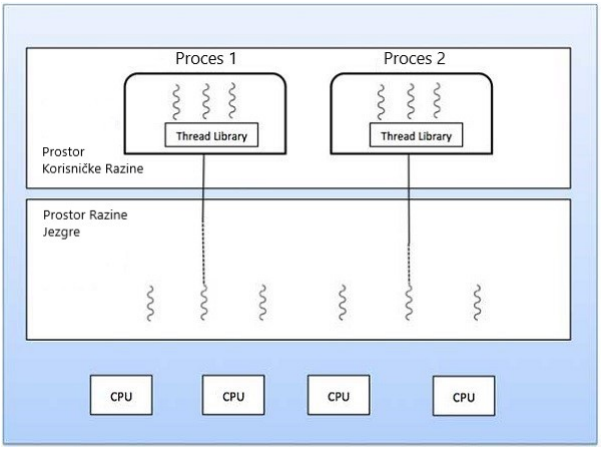
Kod ovakvog modela moguće je kreirati proizvoljno mnogo korisničkih dretvi, dok pripadne sistemske dretve neometano odrađuju svoj posao, paralelno s ostalim dretvama. Ovaj model je najbliži definiciji paralelnog izvršavanja, te u slučaju blokade neke dretve, jezgra može jednostavno kreirati novu dretvu.

1.4 Više prema jedan model

U modelu više prema jedan sve korisničke dretve povezane su točno jednom sistemskom. Tada je sva kontrola nad dretvama predana biblioteci za dretve na korisničkoj razini, te u slučaju pojave blokade na jednoj dretvi, cijeli sustav postaje blokiran.

U slučaju kada operacijski sustav ne podržava način na koji je implementirana biblioteka za rad s korisničkim dretvama, koriste se sistemske dretve prema opisu modela više prema jedan.

Slika 1.2: Primjer modela više prema jedan



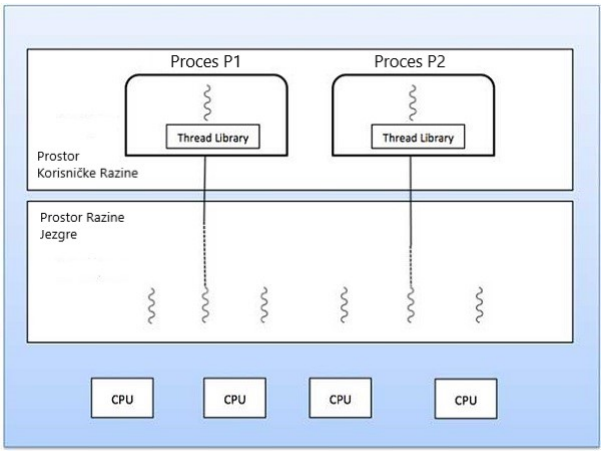
Kod ovakvog modela moguće je kreirati proizvoljno mnogo korisničkih dretvi, dok pripadne sistemske dretve neometano odrađuju svoj posao, paralelno s ostalim dretvama. Ovaj model najbliži je definiciji paralelnog izvršavanja, te u slučaju blokade jezgra može jednostavno kreirati novu dretvu.

1.5 Jedan prema jedan model

Ovaj model povezuje točno jednu korisničku dretvu sa točno jednom sistemskom dretvom i pruža veću razinu paralelnosti nego prethodno opisani model više prema jedan, jer omogućava paralelno poretanje više dretvi na višeprocesorskim računalima. U slučaju pojave blokade, omogućava da se pokrene nova dretva.

Mana ovog modela je to što u slučaju kreiranja korisničke dretve potrebno je kreirati i sistemsku drevu, što je, kako je ranije naglašeno, skupa operacija.

Slika 1.3: Primjer modela jedan prema jedan



1.6 Uvod u višedretvenu podršku, C++11

U ovoj sekciji bit će dane uvodne napomene za korištenje podrške za višedretveno programiranje koje pruža programski jezik C++, preciznije standard iz 2011. godine. Programski jezik C++11 koristi biblioteku <**thread**> za kreiranje i upravljanje dretvama. Glavna klasa koja se koristi za rad s dretvama unutar spomenute biblioteke je `std::thread` klasa. Svaki program napisan u programskom jeziku C++ sarži barem jednu dretvu, dretvu koja će se u daljenjem tekstu nazivati glavnom dretvom. Glavna dretva programa sadrži tok kontrol u kojem se pokreće funkcija `main()`, a ona se dalje može razgranati na nove dretve. Svaka novo kreirana dretva zapravo je, iz pogleda operacijskog sustava, identična glavnoj dretvi programa u kojoj je izvršava funkcija `main()`. Svaka dretva predstavlja nezavisan tok kontrole unutar kojeg se pokreću naredbe programa na isti način na koji se one pokreću u glavnoj dretvi. Jedina razlika, koja na neki način čini glavnu dretvu posebnom je činjenica da se glavni tok procesa odvija upravo unutar glavne

dretve koja je kreirana s funkcijom `main()`. Glavna dretva počinje sa svojim radom u trenutku kada se pokrene proces i završava onda kada proces više nije aktivan. Svakao treba napomenuti da, unatoč tome, dretve kreirane u glavnoj dretvi (pa tako i bilo kojoj drugoj dretvi) mogu nastaviti s radom i nakon što je dretva u kojoj su kreirane završila s radom i moguće je već uništena. No prije nego što nastavim sa spomenutom mogučnošću rada dretve neovisno o roditeljskoj dretvi, treba reći kako se uopće dretva kreira unutar programskom jezika C++.

U programskom jeziku C++ dretvu kreiramo prosljeđivanjem određenih parametara konstruktoru klase `std::thread`. Potpis konstruktora klase `std::thread` je

```
template<class Function, clas... args>
    explicit thread(Function&& f, Args&&... args),
```

gdje argument `Function&& f` predstavlja refencu na glavnu funkciju toka nove dretve, a `Args&&... args` su argumenti funkcije. Jedini obavezni parametar konstruktora klase `std::thread` referenca na funkciju. Prosljeđena funkcija predstavlja početak i kraj novo nastale kontrole toka, dakle ona igra ulogu funkcije `main()` iz glavnog toka kontrole. Drugi argument nije obavezan, on predstavlja argumente je potrebno prosljediti funkciji toka ukoliko ih ona koristi. Treba napomenuti kako prvi argument koji prosljeđujemo konstruktoru klase `std::thread` ne mora nužno biti referenca na funkciju, već to može biti bilo koji *callable* tip programskog jezika C++. To znači da glavni dio kontrole toka novo kreirane dretve može biti i klasa kod koje je preopterećen operator funkcijskog poziva `operator()`. Kod prosljeđivanja takve klase konstruktoru dretve treba pripaziti na način na koji se kreira objekt. Ako se konstruktoru prosljedi privremeni objekt umjesto definiranog objekta, C++ kompajler može kreiranje nove dretve shvatiti kao definiciju nove funkcije. Za bolje razumijevanje dan je sljedeći primjer:

```
class Klasa
{
    public:
        void operator()() const
        {
            ...
        }
};

Klasa klasa;
```

```
std::thread dretva1( klasa );
std::thread dretva2( Klasa() );
```

Drugi poziv (`std::thread dretva2(Klasa())`) C++ kompajler shvaća kao definiciju nove funkcije `dretva2` koja prima jedan parametar tipa pokazivač na funkciju koja ne prima niti jedan parametar i vraća objekt tipa `Klasa`, te kao povratnu vrijednost vraća objekt tipa `std::thread`. Kako bi se izbjegao ovakav „problem" C++11 pruža nekoliko mogućih načina na koji se može kreirati dretva. Jedan način je već dan u gornjem primjeru, u liniji iznad problematičnog dijela, gdje je objekt klase `Klasa` najprije definiran, a zatim proslijeđen konstruktoru klase `std::thread`. Također, moguće je argument konstruktora klase `std::thread` „zapakirati" unutar dodatnih zagrada ili koristiti inicijalizator. Slijede primjeri:

```
std::thread dretva2( (Klasa()) );
std::thread dretva2 { Klasa() };
```

U prvom primjeru se dodatnim zagradama spriječava da kompajler parsira liniji kao definiciju funkcije, dok se korištenjem inicijalizacijskih zagrada `{}` eksplicitno deklarira varijabla klase `std::thread`. Još jedan način na koji se dretva može jednostavno kreirati je korištenje lambda izraza, koji su dobili podršku u C++jeziku u standardu C++11, na sljedeći način:

```
std::thread dretva2( []( ... ){ ... } );
```

Sada, nakon što je pokazano kao kreirati dretvu, vraćamo se na mogućnost rada dretve u pozadini, neovisno o roditeljskoj dretvi. Naime, u programskom jeziku C++, prije završetka trenutne kontrole toka, potrebno je definirati način na koji će se završiti posao ostalih dretvi kreiranih u tom toku. U suprotnom, po izlazu iz kontrole toka, za svaku nedefiniranu dretvu poziva se funkcija `std::terminate()`, koja automatski zaustavlja rad dretve i uništava ju. Taj mehanizam je potreban kako ne bi došlo do neželjenih i nepredvidivih radnji kreirane dretve.

Dvije su mogućnosti kako možemo definirati rad dretve unutar neke kontrole toka. Prvi način je pozivanjem metode `join()` klase `std::thread`, koja kotroli toka daje do znanja da je potrebno pričekati dretvu da obavi svoj dio posla prije nego što kontrola toka završi sa svojim radom. To je ujedno i najjednostavniji način za upravljanje dretvama. Drugi način je poziv metode `detach()` klase `std::thread`, kojom se daje doznanja kontroli toka da će novo kreirana dretva nastaviti sa svojim radom i nakon što kontrola toka

bude prekinuta. Na taj način se kreiraju tzv. *background* dretve, a metodu `detach()` može se, u pravilu, pozvati odmah nakon kreiranja dretve koja bi svoj posao trebala odraditi u pozadini.

Za kraj ovog uvodnog poglavlja reći ćemo i nešto o prosljeđivanju parametara novo kreiranim dretvama, te će biti dan kratak primjer koji bi trebao obuhvatiti do sada spomenute mogućnosti podrške programskog jezika C++ za višedretveno programiranje. Spomenuto je već da se argumenti funkcije toka prosljeđuju kroz drugi parametar konstruktora klase `std::thread`. Ono što pritom treba imati na umu je činjenica da se argumenti **kopiraju** u internu memoriju dretve, čak i onda kada je to pointer ili referenca. Prvi mogući problem sa spomenutim kopiranjem koji se može pojaviti je oslanjanje na implicitnu konverzaciju nekog tipa u pointer ili referencu. Pretpostavimo da imamo sljedeći primjer:

```
void funkcija(double x, std::string const& s);

void foo(double y)
{
    char spremnik[255];
    sprintf(spremnik, "%i", y);
    std::thread dretva(funkcija, 3, spremnik);
    dretva.detach();
}
```

Sada je vrlo vjerojatno da će funkcija foo završtiti sa svojim radom prije nego li se dovrši odgovarajuća implicitna konverzija u liniji

```
std::thread dretva(funkcija, 3, spremnik),
```

što će rezultirati greškom. Kako bi se takav problem izbjegao preporuča se korištenje eksplicitne konverzije kada je to potrebno. Tada gornju liniju koda treba zamijeniti sa

```
std::thread dretva(funkcija, 3, std::string(spremnik))
```

Time je osigurano da će se potrebna konverzija sigurno dogoditi prije nego dođe do poziva funkcije `foo()` i gore navedeni problem je izbjegnut. Promotrio sada drugi slučaj. Pretpostavimo da funkcija koji prosljeđujemo konstruktoru klase `std::thread` uzima referencu na neki tip, te očekujemo da će dretva promijeniti prosljeđeni parametar, kao u sljedećem primjeru:

```
void funkcija(int a, int& b);

void foo(int x)
{
    int y = 10;
    std::thread dretva(funkcija, x, y);
    printf("x: %d, y: %d", x, y);
    dretva.join();
}
```

Konstruktor klase `std::thread` ne zna da funkcija koja mu je prosljeđena kao parametar prima referencu na tip i samo kopira lokalnu vrijednost parametra `int y`, što rezultira time da se nova vrijednost parametra koja mu je pridodana u dretvi `dretva` odbacuje pri povratku iz nje, te se rad funkcije foo nastavlja sa originalnom lokalnom vrijednošću. U gore navedenom primjeru dovoljno je zamijeniti liniju

```
std::thread dretva(funkcija, x, y)
```

sa linijom

```
std::thread dretva(funkcija, x, std::ref(y))
```

kako bi konstruktoru dali doznanja da je prosljeđeni paramtera referenca na tip. Osim navedenog primjera, dretvu možemo kreirati prosljeđivanjem konstruktoru klase `std::thread` članicu neke klase. Tada se kao drugi parametar konstruktora klase `std::thread` prosljeđuje adresa instance, odnosno objekta, klase čija članica je prosljeđena kao funkcija dretve

```
class Klasa
{
public:
    void funkcija();
};

Klasa instanca;
std::thread dretva(&Klasa::funkcija, &instanca);
```

Ukoliko bi prosljeđena funkcija imala dodatne parametre, oni bi bili navedeni odmah nakon reference na instancu klase u redoslijedu u kojem se pojavljuju u funkciji klase. Kao zadnji bitan slučaj navodi se problem tipova koji se ne mogu kopirati, ali se mogu premještati (*eng. movable*). Takve parametre potrebno je prosljediti korištenjem `std::move` mehanizma, na isti način kako je bilo prikazano za slučaj sa `std::ref`.

Na kraju ovog poglavlja slijedi kratki primjer korištenja višedretvenih metoda do sada opisanih u ovom radu. Na početku primjera dretve će biti kreirane na sve, do sada, opisane načine, odraditi neki osnovni posao kroz koji se može vidjeti kako dretve odrađuju posao paralelno. U drugom dijelu je napravljen primjer pokretanja dretve prosljeđivanjem konstruktoru klase `std::thread` metodu klase, zajedno s instancom te klase. Također u drugom dijelu se vidi jedan način dohvaćanja neke povratne vrijednosti kroz referencu na tip, nakon čega opet slijedi odgovarajući ispis. Slijedi primjer:

```
#include <iostream>
#include <thread>
#include <chrono>
#include <utility>
#include <string>

struct Tocka{
    Tocka(int x = 0, int y = 0){
        this->x = x;    this->y = y;
    }
    int x;
    int y;
};

void f1(){
    for(int i = 0; i < 3; ++i){
        std::cout << "Thread 1" << std::endl;
        std::this_thread::sleep_for
            (std::chrono::milliseconds(1000));
    }
}
```

```
void funkcija(int threadId, Tocka& t, std::string str){
    for(int i = 0; i < 5; ++i){
        std::cout << "Dretva " << threadId << std::endl;
        t.x += i;   t.y += -i;
        std::this_thread::sleep_for
            (std::chrono::milliseconds(300));
    }
    std::cout << "Izlaz iz dretve "
        << threadId << ", " << str
        << ": (" << t.x << ", " << t.y << ")" << std::endl;
}

class KlasaFunkcija{
public:
    void operator()(){
        for(int i = 0; i < 3; ++i){
            std::cout << "Thread 3" << std::endl;
            std::this_thread::sleep_for
                (std::chrono::milliseconds(2000));
        }
    }
};

class Klasa{
public:
    void funkcija(int threadId, Tocka& t, std::string str){
        for(int i = 0; i < 5; ++i){
            std::cout << "Dretva " << threadId << std::endl;
            t.x += 1;   t.y += -5;
            std::this_thread::sleep_for
                (std::chrono::milliseconds(250));
        }

        std::cout << "Izlaz iz dretve "
            << threadId << ", " << str
            << ": (" << t.x << ", " << t.y << ")" << std::endl;
    }
};
```

```
int main(void){
    std::thread thread1 = std::thread(f1);
    std::thread thread2 = std::thread(
        []{
            for(int i = 0; i < 5; ++i)
            {
                std::cout << "Thread 2" << std::endl;
                std::this_thread::sleep_for(
                    std::chrono::milliseconds(500)
                );
            }
        });
    std::thread thread3 = std::thread((KlasaFunkcija()));
    thread1.join();
    thread2.join();
    thread3.join();

    std::cout << std::endl
        << "---------------------- drugi dio ----------------------"
        << std::endl;
    Tocka t1, t2;
    thread1 = std::thread
        (funkcija, 1, std::ref(t1), "t1");
    thread2 = std::thread
        (&Klasa::funkcija, Klasa(), 2, std::ref(t2), "t2");

    thread1.join();
    thread2.join();

    std::cout << "Nakon rada dretve: " << std::endl;
    std::cout
        << "\t t1 = (" << t1.x << ", " << t1.y << ")"
        << std::endl;
    std::cout
        << "\t t2 = (" << t2.x << ", " << t2.y << ")"
        << std::endl;

    return 0;
}
```

Slijedi ispis gornjeg koda:

```
Thread 1
Thread 3
Thread 2
Thread 2
Thread 1
Thread 2
Thread 2
Thread 3
Thread 2
Thread 1
Thread 3

---------------------- drugi dio ----------------------
Dretva 1
Dretva 2
Dretva 2
Dretva 1
Dretva 2
Dretva 1
Dretva 2
Dretva 1
Dretva 2
Dretva 1
Izlaz iz dretve 2, t2: (5, -25)
Izlaz iz dretve 1, t1: (10, -10)
Nakon rada dretve:
         t1 = (10, -10)
         t2 = (5, -25)
```

U sljedećem poglavlju rad se počinje baviti problemima koji se mogu pojaviti tijekom pisanja programa korištenjem višedretvenog programiranja, te načina na koji se ti problemi mogu izbjeći. Poglavlje započinje kratkim uvodom, u kojem će biti navedene dodatne napomene uz upravljanje dretvama, a zatim započinje sa načinom na koji se koriste *mutex* mehanizmi.

Poglavlje 2

Dijeljenje podataka između dretvi

Prije nego krenem sa samim dijeljenjem podataka između dvije ili više dretvi, treba navesti još par napomena upravljanju dretvama. Mnoge tipove u jeziku C++ ne možemo kopirati, ali na njima možemo koristiti `std::move` metodu. Jedan od tih tipova je i do sada spomenuta klasa `std::thread` za kontrolu nad dretvama. Ta činjenica je izuzetno bitna ako je potrebno prebaciti vlasništvo (*eng. ownership*) dretve s jednog objekta na drugi, kao povratni tip funkcije ili kao njen ulazni parametar. Za takav prebačaj vlasništva potrebno je koristiti spomenutu metodu `std::move`. Slijedi primjer, koji je uzet iz [2]

```
void funkcija1();
void funkcija2();
std::thread dretva1(funkcija1)
std::thread dretva2 = std::move(dretva1);
dretva1 = std::thread(funkcija2);
std::thread dretva3;
dretva3 = std::move(dretva2);
dretva1 = std::move(dretva3);
```

Slijedi kratak opis gornjeg primjer, a detaljniji se može pronaći u [2]. Dretva `dretva1` kreirana je prosljeđivanjem funkcije `funkcija1` konstruktoru klase `std::thread`, te je zatim vlasništvo te dretve prebačeno s objekta `dretva1` na objekt `dretva2`, a `dretva1` ostaje prazna. Druga dretva kreirana je u objektu `dretva1`, te je za njen glavni tok prosljeđena funkcija `funkcija2`. Vlasništvo prve dretve je zatim predano objektu `dretva3`. Problem nastaje u zadnjoj liniji primjera, kada se vlasništvo prve dretve pokušava prebaciti na objekt `dretva1`. Treba primjetiti kako objekt `dretva1` kontrolira rad dretve čiji tok predstavlja funkcija `funkcija2`, tj. dretva koja je druga kreirana. Ranije je spomenuto kako

je za svaku kreiranu dretvu potrebno definirati treba li kontrola toka pričekati završetak rada dretve ili ju odbaciti kao dretvu koja će svoj posao odrađivati u pozadini neovisno o drugim dretvama. U našem primjeru nigdje se ne pojavljuje poziv funkcijama `detach()` i `join()`, te u tom slučaju proces završava s radom i poziva `std::terminate()`, kako bi sve ostalo u skladu sa implementacijom koju programski jezik C++ pruža za višedretveni rad. Kao nadopunu gornjem primjeru slijedi kod koji riješava gornji primjer.

```
void funkcija1(int n){
    for(int n = 0; i < 3; ++i){
        std::cout << "Dretva " << n << std::endl;
        std::this_thread::sleep_for
            (std::chrono::milliseconds(2000));
    }
}

void funkcija2(int n){
    for(int i = 0; i < 5; ++i){
        std::cout << "Dretva " << n std::endl;
        std::this_thread::sleep_for
            (std::chrono::milliseconds(500));
    }
}

int main(void){
    std::thread dretva1(funkcija1, 1);
    std::thread dretva2 = std::move(dretva1);
    dretva1 = std::thread(funkcija2, 2);
    std::thread dretva3 = std::move(dretva2);
    dretva1.detach();
    dretva1 = std::move(dretva3);
    dretva1.join();

    return 0;
}
```

Naravno prava korist ove mogućnosti prenošenja vlasništva dretve je mogućnost da funkcija kao povratni tip vraća kreiranu instancu klase `std::thread` ili ju prima kao argument. To se opet postiže korištenjem `std::move` operatora, kao što se može vidjeti u ovom primjeru:

```
void f(std::thread f);
std::thread g(){
    void funkcija();
    f(std::thread(funkcija));
    std::thread t(funkcija);
    f(std::move(t));

    return std::thread(funkcija);
}
```

Kod razvijanja višedretvenih aplikacija, treba imati na umu ograničenja koja na višedretvenost postavlja hardver računala na kojem će se aplikacija pokretati. Naime, broj dretvi koje neko računalno može izvršavati u punom smislu pojma paralelnog računanja je ograničeno (u najvećoj mjeri) procesorom. Ovisno o snazi procesora, neka računala mogu istovremeno izvršavati veći broj dretvi od računala sa slabijim procesorima. C++ programski jezik u tu svrhu pruža funkciju

```
std::thread::hardware_concurrency(),
```

koja vraća broj dretvi koje se stvarno mogu izvršavati paralelno.

Za kraj ovog uvodnog dijela dajem par napomena vezanih uz identifikaciju kreiranih dretvi. Programski jezik C++ identificira dretve pomoću tipa `std::thread::id`, a za dohvaćanje člana `std::thread::id` poziva se metoda `get_id()`. Tip `std::thread::id` implementira totalni uređaj, a objekt klase `std::thread` koji nije asociran niti sa jednom dretvom vraća *not any thread* pri pozivu metode `get_id()`. Identifikator dretve moguće je dobiti i pozivom funkcije `std::this_thread::get_id()`, koja vraća identifikator kontrole toka unutar kojeg je funkcija pozvana.

Identifikator dretve također se može pospremiti u strukturu podataka pridruženu nekoj dretvi. Na taj način postiže se potpuna kontrola na identifikatorima dretve jer tip `std::thread::id` u sebi sadrži identifikator njemu dodijeljen od strane operacijskog sustava.

Više detalja o do sada napisanom može se naći u [2]. U nastavku ovog rada bit će opisani mogući problemi koji nastaju neispravnim rukovanjem dretvama, te mehanizmi i metode kojima se ti problemi mogu izbjeći, odnosno kojima se oni riješavaju.

2.1 Problemi pri dijeljenju podataka kod višedretvenog programiranja

Višedretven način programiranja pruža jedno izuzetno korisno svojstvo, jednostavan način dijeljenja podataka u paralelnom izvršavanju. Naravno, kod takvog svojstva javljaju se i određeni problemi. Cilj ovog podpoglavlja je objasniti o kakvim problemima se tu radi, te gdje i kako se oni mogu pojaviti u kodu.

Mnogi problemi kod dijeljenja podatak između dretvi pojavljuju se kod modifikacije podataka. Naime, ako dodatno kreirane dretve samo čitaju dijeljene podatke tada smo sigurni da se nikakav problem neće pojaviti, svaka dretva će u nekom trenuku pristupiti podatku i preuzeti njegovu vrijdnost neovisno o ostalim dretvama. Međutim ukoliko je potrebno promijeniti dijeljeni podatak, može doći problema. Pogledajmo sljedeći, naizgled bezazleni, primjer:

```
void nekaFunkcija(){
    while(brojac < 100){
        ...
        brojac++;
        ...
    }
}
```

Dana funkcija `nekaFunkcija()` odrađuje neki posao, zatim povećava brojač i onda nastavlja s poslom koji se ponavlja dokle god je brojac strogo manji od 100. Kod sekvencijalnog programiranja, kada imamo samo jednu kontrolu toka, ovaj primjer ne predstavlja nikakav problem i čini se potpuno trivijalnim. Međutim, kada bi dodali već samo jednu dodatnu dretvu, koja bi odrađivala isti posao, a uz pretpostavku da obje kontrole toka dijele isti brojac (npr. ako želimo ubrzati isti posao tako da ga podijelimo na dvije dretve), vrlo lako se može dogoditi da jedna dretva poveća vrijednost varijable `brojac` na 100, ali netom nakon što je druga dretva već krenula sa sljedećim korakom petlje **`while`**. Tada dolazi do problema, petlja **`while`** odradit će barem jedan korak više nego što bi trebala i program se više ne ponaša na očekivan način. Pogledajmo još jedan primjer:

```
void nekaDrugaFunkcija(){
    N = -N;
    if(N > 0){
        ...
    }
    else{
        ...
    }
}
```

Opet imamo naizgled jednostavan primjer. Pri ulasku u funkciju varijabla `N` promijeni predznak, te se zatim odradi neki posao, ovisno o predznaku varijable `N`. Problem opet može nastati pri korištenju višedretvenog programiranja. Dvije dretve mogu istovremeno pristupiti podatku i tada će opet doći do istog problema opisanog u prethodnom primjeru.

Navedeni problem nazivamo stanje nadmetanja (*eng. race condition*). Radi se o problemu istovremenog pristupa dijeljenom resursu iz više dretvi, pri čemu dretve modificiraju spomenuti resurs, te daljnje odvijanje kontrole toka ovisi o modificiranoj vrijednosti. Tada nastupa borba za pristupom dijeljenom podatku i dolazi do mogućnosti pojave probema. Problem stanja nadmetanja ujedno predstavlja najveći i najbitniji problem kod višedretvenog programiranja, na koji je potrebno posebno istaknuti pažnju. Katkada se višedretveno programiranje svodi upravo na rješavanje spomenutog problema.

Novije verzije C++ biblioteka pružaju kod koji je sam po sebi siguran za uporabu u višedretvenom programiranju (*eng. thread safe*), međutim kako je ranije spomenuto, u puno slučajeva potrebno je napraviti potrebne radnje sinkronizacije kako bi se dijeljeni resursi vlastitog koda napravili sigurnima za višedretveni pristup. Nekoliko je metoda na koje se dijeljeni resursi mogu zaštiti od neželjenog istovremenog pristupa iz više dretvi. One se sve uglavnom baziraju na jednom od dva principa **međusobnom isključivanju** ili **sinkronizacijskim događajima** (*eng. events*). Prva spomenuta metoda zasniva se na ograničavanju dretvi da određene blokove koda odrađuju jedna po jedna. Te blokove nazivamo *kritičnim produčjima* i za dretvu koja odrađuje dio koda koji predstavlja kritično područje kažemo da se ona **nalazi u kritičnom području**. Ako ne neka dretva nalazi u kritičnom području, tada ostale dretve čekaju dok na njih ne dođe vijeme, a kriterij po kojem se dretvama dozvoljava ulazak u kritično područje ovisi o izvedbi algoritma za zaštitu podataka. Druga metoda za zaštitu podataka zasniva se na čekanju dretve dok se ne dogodi određeni događaj kako bi ona pristupila zaštičenim podacima.

Sada će, ukratko, biti navedene neke metode kojima je moguće zaštiti podatke za višedretveno programiranje, a nakon toga rad se počinje detaljnije baviti nekim najkorištenijim metodama, kao što je metoda zaštite podataka korištenjem mutexa, što će biti i prva detaljnije opisana metoda. Najjednostavnija metoda je zapakirati dijeljene resurse unutar posebnih struktura podataka, s implementiranim mehanizmima koji osiguravaju da više dretvi ne može istovremeno modificirati njihove članove. Druga opcija je dizajnirati strukture podataka unutar koda koje sadrže dijeljene resurse tako da se modifikacije nad tim resursima odrađuju u tzv. serijama nedijeljivih promjena. Takav pristup višedretvenom programiranju nazivamo ***lock-free*** programiranje i uglavnom se svodi na teže i kompleksnije načine pisanja koda. Lock-free pristup izlazi iz dohvata ovog rada, a više detalja moguće je pronaći u [2], poglavlje 7. Prije nego što se rad počne baviti mutex metodama za zaštitu podataka, bit će opisan još jedan način na koji se podaci mogu osigurati za paralelni pristup. Radi se pristupu kakav se koristi kod baza podataka, a zasnovan je na **transakcijama**. Sve modifikacije nad dijeljenim resursima odrađuju se unutar transakcije, te se potom sve promjene unutar jedne transakcije potvrđuju (*eng. commit*). U slučaju da se potvrda ne može odraditi jer su dijeljeni resursi modificirani od strane druge dretve, transakcija se ponavlja. Ova metoda poznata je pod imenom ***software transactional memory (STM)***.

2.2 Međusobno isključivanje (*MUTEX*)

Spomenuto je da je jedna od najjednostavniji metoda za zaštitu dijeljenih resursa kod višedretvenog programiranja međusobno isključivanje, ili skraćeno **mutex** (*eng. mutual exclusion*). Dio koda u kojem se nalaze dijeljeni resursi naziva se kritično područje. Ti resursi ne smiju biti dostupni više od jednoj dretvi u isto vrijeme. Mutex (skraćeno od *mutual exclusion*) predstavlja dijeljeni resurs koji može biti u vlasništvu **točno** jedne dretve (tada kažemo da je dretva preuzela vlasništvo, *eng. ownership*, nad mutexom). Da bi prisupila kritičnom području, dretva mora preuzeti vlasništvo nad mutexom, tj. postaviti lokot nad kritičnim područjem. Ukoliko je dretva zatražila pristup kritičnom području kada je mutex u vlasništvu neke druge dretve, tada dretva čeka u redu dok ne dobije vlasništvo nad mutexom.

Dakle, kao mehanizam za zaštitu kritičnih područja unutar koda mutexi su relativno efikasni, a razne biblioteke za rad s višedretvenim programiranjem pružaju različite vrste mutexa koje programerima omogućavaju da dodatno poboljšaju način na koji mutex štiti kritično područje od utjecaja više od jedne dretve. Time se također poboljšavaju performanse aplikacije koja koristi mutexe. U nastavku će biti nabrojani neki tipovi mutexa, te će biti dan kratak opis svakog:

- standardni i *spin* mutexi
- pošteni i nepošteni mutexi (*eng fair/unfair*)
- rekurzivni mutex

Rečeno je već da svaka dretva koja želi ući u kritično područje mora posjedovati vlasništvo nad mutexom, a ukoliko mutex nije slobodan (u vlasništvu je neke druge dretve) dretva u pitanju mora pričekati da se mutex oslobodi i dretva neće izaći iz funkcije dokle god nije dobila zatraženi mutex. Tu se pojavljuje razlika između standardnog i spin mutexa. Standardni mutex implementiran je tako da dretva koja čeka na dohvaćanje vlasništva nad mutexom oslobađa procesorsko vrijeme (*idle wait*), kako bi se resursi računala za to vrijeme mogli predati nekoj drugoj dretvi. Kada mutex postane slobodan, dretva ponovo postaje aktivna i nastavlja sa svojim radom. S druge strane kod spin mutexa dretva nikada neće osloboditi procesorsko vrijeme u potpunosti (*busy wait*), već će ostati izvršavati neki "beskorisni" posao (poput poziva metodi `std::thread::sleep_for()`). I jedan i drugi pristup imaju svojih prednosti i svojih mana. Standardni mutex oslobađa resurse računala i daje ih na raspolaganje ostalim dretvama, što kod dužih čekanja ostvaruje bolje performanse. Međutim kod kraćih čekanja spin mutex pruža bolje performanse jer ponovna aktivacija dretve je relativno skupa operacija. Sama implementacija mutexa ovisi o korištenoj biblioteci, a najčešće se radi o standardnom mutexu.

Razlika između poštenih i nepoštenih mutexa je u redoslijedu u kojem dretva, koja se nalazi u redu čekanja za mutexom, dobiva vlasništvo nad mutexom. Kod poštenih mutexa dretve dobivaju vlasništvo u onom redoslijedu u kojem su prijavljene u red čekanja (čak i u slučaju kada su na neki način blokirane). Nepošteni mutex implemetiran je tako da vlasništvo daje dretvi koja je prva spremna za rad, što često može dati bolje performanse zbog brzine izvođenja. Međutim, u velikom broju slučajeva bitan je redoslijed u kojem će se određene operacije izvršiti, za što su bolji i pouzdaniji pošteni mutexi.

Glavna razlika između običnih i rekurzivnih mutexa je mogućnost poziva metode za zaključavanje mutexa više puta u slučaju rekurzivnog mutexa. Naime, običan mutex implementiran je tako da jednom kada neka dretva pozove metodu za zaključavanjem mutexa, za pristup određenim podacima, ponovni poziv istoj metodi rezultira greškom *undefined behaviour*. Rekurzivni mutex dopušta dretvi koja ima vlasništvo nad njim da ponovo pozove metodu za zaključavanje mutexa. U tom slučaju, naravno, dretva mora jednak broj puta pozvati i metodu za otključavanje mutexa prije nego što druga dretva dobije vlasništvo nad mutexom. Kako i sam naziv mutexa indicira, jedna od upotreba ovog tipa mutexa je kod rekurzivnih funkcija, gdje je u slučaju višedretvenog programiranja moguća potreba za pozivanjem metode za zaključavanje mutexa više puta, zbog rekuzivnog poziva funk-

cije. Rekurzivni mutex se također koristi u slučaju postojanja zaštićenih privatnih podataka neke klase koja se koristi u dijelu programa koji je napisan višedretvenim načinom rada. Tada članice koje pristupaju tom privatnom podatku moraju moći pozvati metodu za zaključavanjem mutexa neovisno o ostalim članicama i odraditi svoj posao, naravno unutar jedne dretve. Kako bi bolje ilustrirao navedenu upotrebu rekurzivnog mutexa, dan je kratak primjer:

```
class Klasa{
private:
    int nekaVarijabla;

public:
    funkcija1();
    funkcija2();
};
```

Pretpostavimo sada da je funkcija `funkcija1()` implemetirana tako da najprije postavi lokot na mutex za pristup varijabli `nekaVarijabla`, odradi neki posao s njom i zatim napravi poziv za funkciju `funkcija2()`. Funkcija `funkcija2()` također treba pristupiti privatnoj varijabli `nekaVarijabla` za što je potrebno postaviti lokot na mutex, jer je varijabla `nekaVarijabla` zaštićen podatak. Kada se nebi koristio rekurzivni mutex, dobili ranije spomenutu grešku *undefined behaviour*. Sa korištenjem rekurzivnog mutexa, metoda će normalno postaviti lokot na mutex, odraditi svoj posao, te zatim otključati mutex i u nekom trenutnku nastavit će se rad funkcije `funkcija1()`. Funkcija `funkcija1()` prije završetka svoga rada također mora maknuti lokot sa mutexa, kako bi se podatak u potpunosti oslobodio za pristup iz drugi dretvi.

Više detalja o razim tipovima mutexa može se pronaći u [1], a u nastavku ovog poglavlja bit će opisan konkretan način na koji se koriste mutexi u programskom jeziku C++.

2.3 Korištenje mutexa u programskom jeziku C++

Programski jezik C++ pruža podršku za korištenje mutexa kroz biblioteku `<mutex>`, koja sadrži klasu `std::mutex`. Instanciranjem klase `std::mutex` kreira se objekt kroz koji se koriste članice za zaštitu podataka za sigurno korištenje u višedretvenom programiranju. Pozivom metode `lock()` postavlja se lokot na mutex, a pozivom metode `unlock()` se postavljeni lokot otklanja i dopušta se pristup sljedećoj dretvi u redu čekanja. Osim

spomenute metode `lock()`, programski jezik C++ pruža i dodatnu metodu `try_lock()` unutar klase `std::mutex`. Razlika između spomenutih metoda je u načinu na koji se dretva ponaša kao odgovor na rezultat metode. Ako se pozove metoda `lock()` dretva ostaje blokirana dok ne dobije pristup zatraženom kritičnom području, dok u slučaju poziva metode `try_lock()` dretva nastavlja s radom, bez pristupa kritičnom području.

Ipak, ne preporuča se korištenje mutexa kroz gore navedene metode. Naime, kada bi se koristile isključivo gore navedene metode, članice klase `std::mutex`, trebalo bi posvetiti posebnu pažnju da se svaki postavljeni lokot na dani mutex i oslobodi u svim mogućim izlazima iz kritičnog područja. To uključuje izlaze iz svih mogućih grananja, kao i dio koda u koji se ulazi u slučaju da bude bačena iznimka (slučaj kada se koriste **`try`** i **`catch`** blokovi). Umjesto toga, biblioteka <mutex> pruža klasu `std::lock_guard`, koja implementira tzv. *scope-based* omotač za mutex. To je klasa koja automatski postavlja lokot na mutex koji se objektu te klase prosljedi kao parametera konstruktora prilikom kreiranja objekta. Klasa `std::lock_guard` sama se brine za otklanjanje postavljenog lokota na neki mutex, lokot se automatki otklanja kod uništavanja objeka klase `std::lock_guard`. Slijedi primjer korištenja spomenute klase:

```
std::list<int> lista;
std::mutex lokot;

void dodaj_u_listu(int n){
    std::lock_guard omotac(lokot);
    lista.push_back(n);
}
```

U prethodnom primjeru, dretva koja želi dodati element u globalnu listu `lista`, poziva funkciju `dodaj_u_listu` u kojoj se najprije kreira objet `omotac` klase `std::lock_guard` kako bi se lista zaštitila od isovremenog pristupa iz više dretvi. Zatim se novi element dodaje u listu i dretva izlazi iz funkcije. Treba primjetiti kako se nigdje ne pojavljuje dio koda koji otklanja lokot sa korištene liste. Naime, kako je ranije napomenuto, objekt `omotac` klase `std::lock_guard` automatski otklanja postavljeni lokot prilikom poziva destruktora. Slijedi malo upotpunjeniji primjer korištenja klase `std::lock_guard`:

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

std::mutex lokot;
int suma = 0;

void zbroji(int dretvaId, int n){
    std::lock_guard<std::mutex> omotac(lokot);
    suma += n;

    std::cout
        << "dretva" << dretvaId << ": "
        << "suma: " << suma << std::endl;
}

int main(void){
    std::thread dretva1(zbroji, 1, 3);
    std::thread dretva2(zbroji, 2, 4);

    dretva1.join();
    dretva2.join();

    std::cout
        << "suma = " << suma
        << std::endl;
    return 0;
}
```

Klasa `std::lock_guard` pruža veću sigurnost u odnosu na *ručno* korištenje mutexa pozivajući se direktno na metode implementirane u klasi `std::mutex`, kao što je prikazano u prethodna dva primjera. Međutim, klasa `std::lock_guard` ima svojih mana i postoje slučajevi na koje treba paziti čak i kada nju koristimo. Jedan primjer slučaja kada niti klasa `std::lock_guard` ne daje potpunu zaštitu je neispravno prosljeđivanje pokazivača i referenci kroz kod. Pretpostavimo da unutar koda radimo sa pokazivačem na podatke koji se nalaze u kritičnom području, tj. one za koje ne želimo dopustiti pristup iz više dretvi istovremeno. Ako bi postojala metoda ili funkcija u kodu koja vrača takav pokazivač, tada bi se moglo dogoditi da dvije dretve istovremeno dobiju pristup podacima kritičnog područja, jer bi jedna od tih dviju dretvi imala izravan pristup podacima kroz po-

kazivač (s obzirom da dretva radi kroz pokazivač, zaobilazi se mehanizam mutexa). Kako bi izbjegli takve slučajeve potrebno je pažljivo dizajnirati strukture podataka koje se koriste u višedretvenom programiranju, a posebno one koje se koriste unutar kritičnog područja.

Dodatni problem koji se može pojaviti kod korištenja mutexa je odluka o veličini kritičnog područja. Odabir premalog područja rezultira očitim poslijedcima. Dvije dretve mogu istovremeno pristupiti nekom podatku u isto vrijeme (iako se to nije smijelo dogoditi), jer podatak nije uključen u kritično područje. A rezultat toga je naravno stanje nadmetanja. S druge strane, možda malo manje očita pojava je korištenje prevelikog kritičnog područja, tj. zaštita prevelike količine podataka nepotrebno. Problem koji nastaje u toj situaciji je gubitak performanski programa (dolazi do toga da dretve moraju čekati u redu na podatke kojima bi u normalnim okolnostima smijele pristupiti istovremeno), što je suprotno onome što pokušavamo dobiti višedretvenim pristupom.

2.4 Deadlock

U ovom potpoglavlju objasnit će se pojam potpuni zastoj (*eng. deadlock*), te posljedice i mogući načini da se potpuni zastoj izbjegne. Naime, na samom kraju prethodnog poglavlja opisani su mogući problemi korištenja mutexa, a jedan o njih je odabir granica kritičnog područja. Kako bi se ispravno odabrale optimalne granice za kritično područe, često je potrebno koristiti više od jednog mutexa za zaštitu podataka unutar iste strukture. To nas dovodi do pojma potpunog zastoja. Pretpostavimo da dvije dretve trebaju postaviti lokote na dva mutexa kako bi u potpunosti odradile svoj posao. Ako bi svaka dretva postavila lokot na jedan od dva mutexa (naravno ne na isti, jer to se ne može dogoditi) dogodila bi se situacija da dretve čekaju jedna na drugu da oslobode svaka svoj mutex, a to se nikada neće dogoditi jer obje dretve trebaju drugi mutex da bi završile svoj posao. Tada dolazi do potpunog zastoja i program ne može nastaviti s radom.

Najjednostavniji način za izbjegavanje pojave potpunog zastoja je postavljanje pravila redoslijeda po kojem se na povezane mutexe mora postaviti lokot, a kojeg se moraju pridržavati sve dretve unutar jednog procesa. Postavljanje ovakvog strogog redoslijeda postavljanja lokota na mutexe ima svojih mana. Dovoljno je pogledati primjer kada se dva mutexa koriste za zaštitu različitih instanci iste klase, te neka je potrebno prenijeti podatke iz jedne instance u drugu iz više dretvi. Tada je dovoljno zamijeniti redoslijed parametara (instanci klase) i može se dogoditi da dvije dretve zaštite različite instance u isto vrijeme, te dolazi do potpunog zastoja. U tu svrhu programski jezik C++ pruža funkciju `std::lock` koja se brine o tome da su svi lokoti na mutexe postavljeni ispravno, tako da se izbjegne pojava potpunog zastoja. Funkciji `std::lock` dovoljno je prosljediti sve instance klase `std::mutex` kako bi se svi potrebni lokoti postavili bez pojave potpunog zastoja. Slijedi

kratak primjer korištenja funkcije `std::lock`:

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


class Klasa{
private:
    int vrijednost;
    std::mutex lokot;

public:
    Klasa(int n) : vrijednost(n) { }

    friend void postaviVrijednost
        (int dretvaId, Klasa& izvor, Klasa& destinacija);
};

void postaviVrijednost(int dretvaId, Klasa& izvor, Klasa& destinacija){
    if (&izvor == &destinacija)
        return;

    std::lock(izvor.lokot, destinacija.lokot);

    std::lock_guard<std::mutex>
        lokot1(izvor.lokot, std::adopt_lock);
    std::lock_guard<std::mutex>
        lokot2(destinacija.lokot, std::adopt_lock);

    std::cout << "dretva" << dretvaId << ":"
        << "\n\tprije: izvor = " << izvor.vrijednost
        << ", destinacija = " << destinacija.vrijednost;

    destinacija.vrijednost = izvor.vrijednost;
```

```
        std::cout
            << "\n\tposlije: izvor = " << izvor.vrijednost
            << ", destinacija = " << destinacija.vrijednost
            << std::endl;
    }

    int main(void){
        Klasa instanca1(5);
        Klasa instanca2(10);
        Klasa instanca3(15);

        std::thread dretva1
            (
                postaviVrijednost,
                1,
                std::ref(instanca1),
                std::ref(instanca2)
            );
        std::thread dretva2
            (
                postaviVrijednost,
                2,
                std::ref(instanca2),
                std::ref(instanca3)
            );
        std::thread dretva3
            (
                postaviVrijednost,
                3,
                std::ref(instanca3),
                std::ref(instanca1)
            );

        dretva1.join();
        dretva2.join();
        dretva3.join();

        return 0;
    }
```

Prethodni primjer daje sljedeći ispis:

```
dretva1:
    prije: izvor = 5, destinacija = 10
    poslije: izvor = 5, destinacija = 5
dretva3:
    prije: izvor = 15, destinacija = 5
    poslije: izvor = 15, destinacija = 15
dretva2:
    prije: izvor = 5, destinacija = 15
    poslije: izvor = 5, destinacija = 5
```

Umijesto funkcije `std::lock`, programski jezik C++ pruža i fleksibilniju opciju korištenjem klase `std::unique_lock`. Radi se o klasi koja radi na sličnom principu kao klasa `std::unique_pointer`, te osigurava ispravno postavljanje lokota na mutexe bez opasnosti od efekta potpunog zastoja. Međutim, fleksibilnost klase `std::unique_lock` zahtjeva veću memoriju i u pravilu radi sporije nego korištenje funkcije `std::lock`. Više detalja o spomenutoj klasi, kao i dodatne napomene za izbjegavanje pojave efekta potpunog zastoja, može se pronaći u [2].

Poglavlje 3

Sinkronizacija događajima

U prethodnom poglavlju objašnjen je mehanizam sinkronizacije dretvi korištenjem mutex objekata. Puno detaljniji opis rada s mutex mehanizmima može se pronaći u [1]. Mutex objekti u najopćenitijem slučaju omogućavaju dretvi da odradi svoj posao neovisno o ostalim dretvama, u slučaju kada joj je potreban pristup podacima koje dijeli s ostalim dretvama unutar procesa. Međutim, u višedretvenom programiranju jako je česta potreba *suraďivanja* dviju ili više dretvi, npr. da bi dretva mogla svoj posao odraditi do kraja treba pričekati da neka druga dretva odradi svoj posao. Dobar primjer iz prakse upravo navedenog je upotreba dretve koja u pozadini priperma podatke i sprema ih u neku bazu podataka, dok ostatak programa čita i prikazuje podatke. Međutim, dio koda koji se brine za prikaz podataka ne može prikazati potrebne podatke dok dretva koja radi u pozadini ne završi svoj posao, tj. dok ti podaci ne budu dostavljeni dretvi za prikaz podataka (*eng. UI thread*).

Dakle, radi se o drugačijem mehanizmu sinkronizacije dretvi koje je temeljeno na **događajima** (*eng. events*). Događaj se može definirati kao bilo koju boolean varijablu, koja se naziva **predikat**, takvu da mijenja vrijednosti iz istine (*eng. true*) u laž (*eng. false*) ili obratno. Općenito kažemo da se neki događaj dogodio ako predikat promijeni stanje. U ovom potpoglavlju bit će opisane metode za sinkronizaciju koja se bazira na događajima, tj. promatrat će se slučajevi kada dretva čeka da se dogodi neki događaj. Preciznije, promatrat će se mehanizmi koje pruža programski jezik C++ , a to su **kondicionalne varijable** i objekte koji se nazivaju ***futures***.

U prethodnom poglavlju, kada se govorilo o mutex objektima, spomenuto je da postoji dio vremena koje dretva provede ne radeći ništa korisno. Također je spomenuto da postoje dva bitno različita načina na koje dretva može čekati druge dretve da odrade svoj posao, a radi se o *spin* ili *idle* čekanju. Ista je situacija i kod sinkronizacije događajima. Idle

čekanje je prirodan mehanizam za sinkronizaciju, dok se spin čekanje postiže dodatnim programskim kodom (i izlazi iz opsega kojeg obuhvaća ovaj rad, a više detalja može se pronaći u [1]).

Idle čekanje radi na sljedećem principu. Dretva 1 odrađuje svoj posao dok ne dođe do dijela koda koji ispituje istinitost predikata. U slučaju da je vrijednost predikata postavljena na `false` dretva 1 prelazi u stanje čekanja pozivajući blokadu funkcije koja predstavlja kontrolu toka dretve. Time je izlaz iz funkcije onemogućen do trenutka uništavanja ili ponovnog buđenja dretve. Dretva 1 se budi u slučaju da dretva 2 tijekom svog rada promijeni vrijednost predikata, kojeg dretva 1 čeka, na vrijednost `true`, te tada dretva 1 nastavlja sa svojim poslom. Ovaj proces ponovno je opisan jer je najbliži načinu sinkronizacije više dretvi pomoću **kondicionalne varijable**. S ovom uvodnom napomenom započinje sljedeće potpoglavlje, koje će detaljnije obraditi pojam kondicionalne varijble.

3.1 C++ i sinkronizacija kondicionalnim varijablama

Programski jezik C++ pruža dvije implementacije kondicionalnih varijabli:

```
std::condition_variable
std::conditional_variable_any
```

Obje implementacije definirane su u biblioteci <condition_variable>. Također treba napomenuti kako obje implementacije zahtjevaju korištenje mutexa za uspostavu sinkronizacije. Razlika u implementacijama je u fleksibilnosti koju pojedina implementacija pruža. Klasa `std::condition_variable` ograničena je na rad isključivo s klasom `std::mutex`, dok `std::condition_variable_any` pruža veću fleksibilnost tako što omogućava rad s bilo kojim objektom koji implementira mutex mehanizam, tj. mehanizam međusobnog isključivanja. Kao i do sada, veća fleksibilnost koju klasa `std::condition_variable_any` pruža ima svojih ograničenja, korištenje te klase zauzima veći memorijski prostor i općenito troši više resursa operacijskog sustava.

Uloga kondicionalnih varijabli je jednostavna uspostava veze između dretvi koje zahtjevaju međusobnu komunikaciju, a ponašaju se kao satovi koji bude dretve u slučaju da se dogodi neki događaj potaknut radom neke druge dretve. Kondicionalna varijabla pritom ne zna ništa o prirodi samog događaja, a veza između događaja i kondicionalne varijable prepuštena je na odgovornost programeru koji piše višedretveni kod. Tri su elementa ovakvog sinkronizacijskog mehanizma usko povezana, a to su **predikat** čija promjena stanja daje do znanja kondicionalnoj varijabli da se dogodio čekani događaj, **mutex**

u svrhu zaštite spomenutog predikata, te sama **kondicionalna varijabla**. Dobra programerska praksa je definirati spomenute elemente na istom mijestu, kako bi se lakše pratilo njihovo ponašanje u kodu.

Kako bi bolje razumijeli sinkronizaciju korištenjem kondicionalnih varijabli, potrebno je razumijeti protokol čekanja. Najprije je dan pseudokod pomoću kojeg će se lakše predočiti način na koji je protokol realiziran:

```
------------------------------
mutex.zakljucaj
while(!predikat)
    dretva.cekaj(mutex, uvjet)
mutex.otkljucaj
------------------------------
```

Dretva koju je potrebno sinkronizirati pomoću kondicionalne varijable (u nastavku predikat) prvo treba zaključati mutex koji spriječava dohvaćanje predikata od strane više dretvi. Tada slijedi provjera stanja predikata, te ukoliko vrijednost nije zadovoljena dretva ulazi u stanje čekanja. Bitno je dati detalje implementacije funkcije koja dretvu postavlja u stanje čekanja, jer je upravo ta implementacija bitna za razumijevanje rada protokola čekanja. Zanimljiva je činjenica da se funkciji čekanja prosljeđuje mutex kojim se štiti predikat od utjecaja ostalih dretvi. Naime, funkcija čekaj na početku svoga rada otključava mutex i time se ostalim dretvama dozvoljava pristup sinkronizacijskom predikatu. To je potrebno kako bi dretva na koju se čeka mogla promijeniti stanje predikata i obavijestiti o tome dretvu koja čeka da se taj događaj ostvari. Prije izlaza iz funkcije čekanja (kada se ispuni uvjet), mutex se ponovo zaključava i dretva ponovno postaje aktivna, te dolazi do ponovne provjere uvjeta. Treba napomenuti kako je ponovna provjera stanja predikata bitna, jer postoji šansa da neka druga dretva pristupi predikatu (i moguće promijeni vrijednost predikata) u onom kratkom vremenu prije nego što dretva koja čeka ponovo zaključa mutex i spriječi daljnji pristup predikatu. To je upravo razlog zašto se ispitivanje stanja predikata mora odraditi pomoću petlje **`while`**, a ne korištenjem kontrole grananja **`if`**. Ako je stanje predikata sada zadovoljavajuće, mutex se otključava i dretva nastavlja sa svojim radom. Preostalo je primjetiti kako je nužno da upravo funkcija, koja dretvu postavlja u stanje čekanja, odrađuje otključavanje mutexa. Naime, kada bi sama dretva otključala mutex prije poziva funkcije za čekanje, moglo bi se dogoditi da neka druga dretva u međuvremenu ispuni uvjet i pošalje potrebnu obavijest. Kako dretva koja čeka na ispunjenje uvjeta nije ranije bila prebačena u stanje čekanja, ona neće primiti obavijest o ispunjenju uvjeta, te ostaje u stanju čekanja (iako nebi trebala).

Na gore opisan način realizirane su implementacije svih poznatijih biblioteka za sinkronizaciju dretvi pomoću kondicionalnih varijbli, pa isto vrijedi i za biblioteku koju koristi programski jezik C++. U programskom jeziku C++, dretva ulazi u stanje čekanja pozivom na jednu od članica `wait` ili `wait_for`, klase `std::condition_variable`. Postoji također i metoda `wait_until`, ali ona će biti izostavljena iz ovog rada. Metoda `wait` postavlja dretvu u stanje bezuvjetnog čekanja, tj. dretva postaje blokirana do trenutka ispunjenja uvjeta. Za razliku od metode `wait`, metoda `wait_for` dretvu postavlja u stanje čekanja na određeno vrijeme. Ukoliko unutar predviđenog perioda uvjet ne bude ispunjen, dretva vraća vrijednost `timeout`. U nastavku će biti dan primjer korištenja metode `wait`, dok kao dobar primjer korištenja metode `wait_for` treba navesti primjer slanja upita za konekciju na bazu podataka koja se koristi unutar aplikacije. Ako nakon postavljenog vremena čekanja konekcija nije uspostavljena, dretva šalje grešku `timeout`.

Preostaje navesti metode `notify_one` i `notify_all`, kojima dretva na koju se čeka šalje obavijest da je uvjet čekanja ispunjen. Razlika je u broju dretvi koje će biti obaviještene. Pozivom na metodu `notify_all` obavijest o ispunjenju uvjeta šalje se svim dretvama koje čekaju, dok metodom `notify_one` obavijest dobiva samo jedna dretva, pri čemu programer nema kontrolu na time koja dretva će biti obaviještena, već je to prepušteno operacijskom sustavu. Zato treba dodatno pripaziti na to koju metodu je potrebno iskoristiti u kodu. U nastavku slijedi konkretan primjer korištenja kondicionalnih varijabli u svrhu sinkronizacije:

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

#define THREAD_No 4

std::mutex m;
std::mutex coutMutex;
std::condition_variable cv;
int dretvaGotova = 0;

void funkcija(int dretvaId){
    for (int i = 0; i < 3; ++i){
        std::lock_guard<std::mutex>
            coutLokot(coutMutex);
```

```
            std::cout
                << "Dretva" << dretvaId
                << " obavlja posao..."
                << std::endl;
            std::this_thread::sleep_for
                (std::chrono::milliseconds(1000));
        }

        std::unique_lock<std::mutex> lokot(m);
        dretvaGotova++;

        {
            std::lock_guard<std::mutex>
                coutLokot(coutMutex);
            std::cout
                << "Dretva" << dretvaId
                << " zavrsava s poslom..."
                << std::endl;
        }

        cv.notify_one();
    }

    int main(void){
        std::thread dretve[THREAD_No];
        for(int i = 0; i < THREAD_No; ++i){
            dretve[i] = std::thread(funkcija, i+1);
            dretve[i].detach();
        }

        std::unique_lock<std::mutex> lokot(m);

        {
            std::lock_guard<std::mutex> coutLokot(coutMutex);
            std::cout
                << "main ceka..."
                <<std::endl;
        }
```

```
        cv.wait(lokot, []{return dretvaGotova == THREAD_No;});
        std::cout
            << "uvjet zadovoljen, "
            << "main vise ne ceka..."
            << std::endl;

        return 0;
    }
```

Prije nego se rad počne baviti sljedećim mehanizmom za sinkronizaciju, bit će dane posljednje napomene za korištenje kondicionalnih varijabli. U zadnjem primjeru prikazano je korištenje kondicionalnih varijabli u svrhu sinkronizacije više dretvi koje posao rade u pozadini, dakle bez poziva metode `join`. Pri tome je korištena metoda `notify_one` da bi se obavijestila glavna dretva (ona čija kontrola toka je predstavljena funkcijom `main`). U tom slučaju jedna dretva čekala je notifikaciju više dretvi, te je korištenje metode `notify_one` bila sigurna opcija. Drugi mogući slučaj bio bi primjer kada više dretvi čeka da se ispuni isti uvjet. Tada je potrebno pozvati metodu `notify_all`, kako bi sve dretve bile obaviještene o odgovarajućem događaju. Dakle, kao što je ranije spomenuto, treba s oprezom pristupiti ovom tipu sinkronizacije, pogotovo kod odabira koju metodu za obaviještavanje je bolje pozvati.

Ako je unaprijed poznato da će dretva čekati da kondicionalna varijabla postigne traženu vrijednost samo jednom tjekom izvršavanja programa, tada je najvjerojatnije korištenje kondicionalne varijable nepotrebno. U tom slučaju bolji odabir predstavlja mehanizam kojim će se baviti iduće potpoglavlje, a radi se mehanizmu koje se naziva *futures*.

3.2 Sinkronizacija čekanjem događaja koji se ostvaruje točno jednom

Kondicionalne varijable predstavljaju jak mehanizam za sinkronizaciju više dretvi pomoću događaja. Međutim, često se pojavljuje slučaj kada je inicijalizacija i općenito korištenje takvog mehanizma nepotrebna, a to je upravo slučaj kada je poznato da će se određeni događaj ostvariti **točno** jednom unutar procesa. U tu svrhu programski jezik C++ predstavlja objekte koje nazivamo **futures**. Ti objekti predstavljeni su klasom `std::future<T>`, koja je definirana u biblioteci `<future>`. Radi se o parametriziranoj klasi, čiji parametar predstavlja tip povratne vrijednosti dretve na čiju povratnu vrijednost se čeka. Dakle korištenjem klase `std::future<T>` moguće je ostvariti razmjenu podataka

između više dretvi, kao i dohvaćanje povratne vrijednosti iz neke dretve (što programski jezik C++ ne pruža klasičnim korištenjem objekata tipa `std::thread`).

Osim klase `std::future`, programski jezik C++ pruža klasu za rad s više dretvi, a radi se o klasi `std::shared_future`. Radi se o istom principu kao kod klasa `std::unique_ptr` i `std::shared_ptr`. Klasa `std::future` također se može koristiti kod sinkronizacije više dretvi, međutim u tom slučaju potrebno je koristiti neki oblik mehanizma međusobnog isključivanja za zaštitu objekta klase `std::future`, te je općenito jednostavnije postići željeni efekt koriteći objekte klase `std::shared_future` koji dijele iste podatke, ali se mogu koristiti iz više dretvi bez potrebe za dodatnim sinkronizacijskim mehanizmom.

U nastavku slijedi vrlo jednostavan primjer u kojem se može vidjeti kako prebaciti povratnu vrijednost funkcije, čije računanje se želi odraditi u pozadini. Spomenuto je već da programski jezik C++ ne pruža prirodan način za dohvaćanje povratne vrijednosti korištenjem klasičnog objekta klase `std::thread`. U tu svrhu implementirana je funkcija `std::async`, koja je također definirana u biblioteci <future>. Funkcija `std::async` koristi se na identičan način kao i klasa `std::thread`, te podržava iste mehanzime. Razlika je u tome što kao povratni tip, funkcija `std::async` kreira objekt klase `std::future<T>`, s parametrom T koji označava tip povratne vrijednosti funkcije čiji poziv se odvija asinkrono. Ukoliko je ne postoji povratni tip, kao parametar `T` prosljeđuje se tip `void`. Neka je dan sljedeći primjer:

```
#include <iostream>
#include <future>
#include <chrono>

int zbroji(int a, int b){
    std::this_thread::sleep_for
        (std::chrono::milliseconds(3000));
    return a + b;
}
int main(void){
    std::future<int> vrijednost
        = std::async(zbroji, 2, 8);
    std::cout
        << "Hello world!"
        << std::endl;
    std::cout
```

```
        << "Povratna vrijednost: "
        << vrijednost.get()
        << std::endl;
    return 0;
}
```

Program kreće s radom s pozivom funkcije `std::async` koja kreira objekt tipa `std::future<int>`, koja u ovom slučaju kao tip povratne vrijednosti prima **int**. Također se može primjetiti sličnost s korištenjem objekta tipa `std::thread`. Funkciji `std::async` prosljeđen je pointer na funkciju čije izvršavanje se odvija asinkrono, te su zatim nabrojani parametri koji se prosljeđuju funkciji `zbroji`. Funkcija `zbroji` simulira duži rad pozivom funkcije `std::this_thread::sleep_for`, a zatim vraća zbroj prosljeđenih parametara.Tijekom izvođenja asinkrone funkcije, glavna dretva programa nastavlja s poslom do poziva metode `get`, članice klase `std::future`, kada `get` blokira dretvu dok vrijednsot nije spremna. Pomoću metode `get` dohvaća se povratna vrijednost, te se ispisuje na ekran.

Upravo pokazani način definiranja i korištenja `std::future` objekata samo je jedan od tri moguća načina. Programski jezik C++ pruža klasu `std::packaged_task<T>`, pomoću koje je moguće asocirati zadatak (*task*) sa `std::future` objektom. Parametar predloška klase `std::packaged_task` predstavlja potpis funkcije koja će se odraditi unutar zadatka, a kojemu se ta funkcija proslijeđuje kao argument na isti način na koji se funkcija proslijeđuje objektu tipa `std::thread`. Instanca klase `std::packaged_task` predstavlja objekt kojeg je moguće pozvati na isti način kao bilo koju funkciju, pri čemu se izvršava funkcija predana konstruktoru klase. Povratnu vrijednost izvšene funkcije dohvaća se pomoću metode `get_future`, članice klase `std::packaged_task`. Slijedi primjer iz kojeg se može vidjeti korištenje upravo opisane klase:

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

std::mutex m;

std::string napraviString(char* poljeZnakova, int duljinaPolja){
    {
        std::lock_guard<std::mutex> lokot(m);
        std::cout
```

```
            << "Dretva zapocinje spajati znakove!"
            << std::endl;
    }

    std::string str("");
    for(int i = 0; i < duljinaPolja; ++i)
        str += *(poljeZnakova + i);




    {
        std::lock_guard<std::mutex> lokot(m);
        std::cout
            << "Dretva zavrsava spajati znakove!"
            << std::endl;
    }

    return str;
}

int zbrojiNiz(int* poljeBrojeva, int duljinaPolja){
    {
        std::lock_guard<std::mutex> lokot(m);
        std::cout
            << "Dretva zapocinje zbrajati brojeve!"
            << std::endl;
    }

    int vrijednost = 0;
    for(int i = 0; i < duljinaPolja; ++i)
        vrijednost += *(poljeBrojeva + i);

    {
        std::lock_guard<std::mutex> lokot(m);
        std::cout
            << "Dretva zavrsava zbrajati brojeve!"
            << std::endl;
    }
```

```
    return vrijednost;
}

int main(void){
    std::packaged_task<std::string(char*,int)>
        task1(napraviString);
    std::future<std::string>
        rezultat1 = task1.get_future();
    char poljeZnakova[] = {'s','t','a','r',' ','w','a','r','s'};
    std::thread
        dretva1(std::move(task1), poljeZnakova, 9);

    int poljeBrojeva[] = {1, 2, 3, 4, 5};
    std::packaged_task<int()>
        task2(std::bind(zbrojiNiz, poljeBrojeva, 5));
    std::future<int>
        rezultat2 = task2.get_future();
    task2();

    dretva1.join();
    std::cout
        << "Spojeni znakovi: " << rezultat1.get()
        << std::endl;
    std::cout
        << "Zbroj brojeva: " << rezultat2.get()
        << std::endl;

    return 0;
}
```

U primjeru se pozivaju dvije jednostavne metode korištenjem `std::package_task` objekta. Iz primjera se lako mogu vidjeti dva načina na koja se može pokrenuti zadatak. Jedan način je direktno pozvati zadatak preko operatora poziva (u primjeru `task2()`) ili premještanjem zadatka u drugu dretvu, koja automatki pozove odgovarajuću funkciju. Dakle, klasu `std::packaged_task` moguće je premještati. Jedan mogući ispis gornjeg primjer je:

```
Dretva zapocinje zbrajati brojeve!
Dretva zapocinje spajati znakove!
Dretva zavrsava zbrajati brojeve!
Dretva zavrsava spajati znakove!
Spojeni znakovi: star wars
Zbroj brojeva: 15
```

Klasa `std::packaged_task` uglavnom se koristi kada je neki zadatak moguće podijeliti na više manjih podzadataka, koje je onda moguće proslijediti upravitelju zadataka (*eng. task scheduler*). Time se postiže paralelno izvršavanje kreiranih podzadataka.

Treći način na koji je moguće asocirati objekt klase `std::future` sa asinkronim rezultatom je pomoću klase `std::promise<T>`. Klasa `std::promise` potrebna je u slučajevima kada se zadatak ne može predstaviti kao jednostavan poziv funkcije ili kada rezultat dolazi iz više izvora. Parametar predloška klase `std::promise` ponovno označava tip povratne vrijednosti. Povratnu vrijednost moguće je dohvatiti na isti način kao i kod `std::packaged_task` klase, pozivom metode `get_future`. Metoda `get_future` vraća objekt tipa `std::future`. Slijedi jednostavan primjer korištenja klase `std::promise`:

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

std::promise<int> p;
std::mutex m;

void funkcija(){
    {
        std::lock_guard<std::mutex> lokot(m);
        std::cout
            << "Dretva zapocinje s radom..."
            << std::endl;
    }


    std::this_thread::sleep_for
        (std::chrono::microseconds(5000));
```

```
        {
            std::lock_guard<std::mutex> lokot(m);
            std::cout
                << "Dretva zavrsava s radom..."
                << std::endl;
        }


        p.set_value(10);
    }

    int main(void){
        std::future<int> future = p.get_future();

        std::thread dretva(funkcija);
        dretva.detach();
        {
            std::lock_guard<std::mutex> lokot(m);
            std::cout
                << "Funkcija main ceka..."
                << std::endl;
        }

        future.get();
        std::cout
            << "Funkcija main nastavlja..."
            << std::endl;

        return 0;
    }
```

Kao što se može vidjeti iz primjera, klasa `std::promise` radi na sljedeći način. Dretva koja čeka da se obavi neki posao kreira objekt tipa `std::future` pozivom na metodu `get_future` objekta tipa `std::promise` pomoću kojeg se postiže sinkronizacija. Pozivom metode `get` ostvaruje se blokada dretve, sve dok dretva koja obavlja posao ne pozove metodu `set_value`, kojom postavlja vrijednost objekta tipa `std::future` i prebacuje ga u spremno stanje. Time dretva koja čeka nastavlja s radom.

Iako klasa `std::future` izvršava svu potrebnu sinkronizaciju za slanje podataka

između više dretvi, treba napomenuti da sam objekt tipa `std::future` također zahtjeva određenu razinu sinkronizacije. Naime, kada se objekt koristi u više dretvi istovremeno, može doći do borbe za prioritetom.

Tom napomenom završava sekcija koja se bavi sinkronizacijom dretvi, a time je odrađena tema kojom se bavi ovaj rad. Višedretvenost će kao neizbježan alat svakog programera ostati predmet promatranja dugo vremena. Više informacija o podršci koju programski jezik C++ pruža za rad s višedretvenim aplikacijama može se pronaći u [2] i [1].

Bibliografija

[1] V. Alessandrini, *Shared Memory Application Programming, Concepts and strategies in multicore application programming*, Elsevier, 2016.

[2] A. Williams, *C++ Concurency in Action, Practical Multithreading*, Manning, 2012.

Sažetak

Višedretveno programiranje je posebna vrsta programiranja koja zahtjeva pažljivo kreirane strukture podataka. Razlog tome je potreba za sigurnim pristupom iz više dretvi istovremeno podacima koji se nalaze u strukturama podataka koje se koriste za višedretveno programiranje. Cilj ovog rada je proučiti podršku višedretvenom programiranju koju pruža programski jezik C++ , preciznije standard iz 2011. godine. U radu će biti opisani mehanizmi za pristup zajedničkim resursima koji se baziraju na međusobnom isključivanju. Također će biti proučeni sinkronizacijski mehanizmi za rad s dijeljenom memorijom. Prvo će biti opisani mehanizmi koju su implementirani korištenjem kondicionalnih varijabli, a rad će biti zaključen mehanizmima koji za sinkronizaciju koriste događaje koji se unutar jednog procesa dogode isključivo jednom, te omogućuju razmjenu podataka između više dretvi. Svaki opisani mehanizam bit će dodatno opisan konkretnom izvedbom u programskom jeziku C++ .

Životopis

Mislav Kuzmić rođen je 5.5.1994. u Zagrebu, Republika Hrvatska. Odrastao je u Sesvetama gdje nakon završene osnovne škole 2009. upisuje Srednju školu u Sesvetama te nakon četiri godine srednjoškolskog školovanja završava smjer Tehničar za računarstvo. S obzirom na odličan uspjeh te posebnu zainteresiranost za matematiku i računarstvo odlučuje se na upis na Prirodoslovno-matematički fakultet u Zagrebu - Matematički odsjek. 2013. godine uspješno završava preddiplomskii studij te stiče zvanje Sveučilišnog prvostupnika matematike (Univ.Bacc.Match). Svoje usavršavanje nastavlja na istom fakultetu i upisuje Diplomski studij Računartstvo i matematika.