C++ memorijski model i višedretveno programiranje bez zaključavanja

Batović, Šime

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Šime Batović

C++ MEMORIJSKI MODEL I VIŠEDRETVENO PROGRAMIRANJE BEZ ZAKLJUČAVANJA

Diplomski rad

Voditelj rada:
prof. dr. sc. Mladen Jurak

Zagreb, veljača, 2022.

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. _____

Za moju majku, koja mi je rekla da upišem arheologiju

Sadržaj

Uvod

Od njihove kreacije pa sve do početka 21. stoljeća, procesori su se sastojali od jedne procesorske jezgre, što je značilo da su se sve instrukcije programa koji se izvodi izvršavale sekvencijalno, a više različitih zadataka se izvodilo razmjenom procesorskog vremena. Razvojem tehnologije, pojavljuju se višejezgreni procesori koji znatno ubrzavaju računanje korištenjem više dretvi koje paralelno izvode različite operacije. No, sve dretve koje se izvršavaju koriste zajedničku memoriju preko koje komuniciraju dohvaćajući i zapisujući u nju podatke. Time je došlo do nove pojave koju nazivamo stanje natjecanja za resurse (*en. data race*). Do njega dolazi kada dvije dretve istovremeno žele dohvatiti isti podatak iz memorije, od kojih ga barem jedna dretva želi promijeniti. Takvo stanje je veoma opasno jer se program ponaša nepredvidljivo, odnosno ne znamo posljedično stanje memorije što često može dovesti do katastrofalnih posljedica. To je navelo Međunarodnu organizaciju za standardizaciju (ISO), koja je zadužena za održavanje i unapređenje standarda C++ programskog jezika, da 2011. godine objavi novi standard, poznatiji kao C++11, koji je kao jednu od najvažnijih novih značajki imao uvođenje memorijskog modela s podrškom za višedretveno programiranje.
Memorijski model opisuje ponašanje dretvi s obzirom na osnovne memorijske operacije, ponajviše operacije čitanje i pisanja u varijable potencijalno dostupne većem broju dretvi. Glavna pitanja na koja memorijski model daje odgovore su:

- **Nedjeljivost:** Koje memorijske operacije su nedjeljive (atomske)?
- **Vidljivost:** Kada će akcija pisanja jedne dretve biti vidljiva u akciji čitanja druge dretve?
- **Poredak:** Pod kojim uvjetima će niz memorijskih operacija jedne ili više dretvi biti vidljiv u istom poretku od strane ostlaih dretvi?

Postoje dva aspekta memorijskog modela: strukturalni aspekt, koji opisuje izgled i ponašanje stvari u memoriji te aspekt konkurentnosti (en. *concurrency*). Konkurentnost u računarstvu je mogućnost više različitih dijelova nekog programa ili algoritma da se izvršavaju u isto vrijeme, ali ne nužno u sekvencijalnom poretku, bez utjecaja na konačni

rezultat. Drugim riječima, konkurentnost označava isprepleteno izvršavanje više različitih zadataka, bez zahtjeva na broj zadataka koji se u nekom trenutku mora izvoditi.

Konkurentnost i paralelizam dva su povezana, ali ipak različita pojma. Paralelizam ne zahtjeva postojanje više zadataka i dok se dva zadatka mogu izvršavati konkurentno na jednojezgrenom procesoru, paralelizam zahtjeva barem dvije jezgre. Primjerice, kod procesora s jednom jezgrom, dva zadataka možemo izvoditi konkurentno na način da naizmjenično izvršavamo dijelove prvog i drugog zadatka. Na taj način istodobno izvršavamo dva zadatka, premda ne u istom vremenu. Paralelizam pak znači da se dijelovi jednog ili više zadataka izvršavaju istovremeno.

Jedna od najvećih mana višedretvenog programiranja je njegova složenost. Programer mora uložiti puno veći trud da bi sinkronizirao različite dretve te spriječio stanje natjecanja za resursima (*data race*). Najjednostavniji i najčešće korišteni postupak je uporaba lokota, odnosno mutexa. Mana lokota jest ta da, u slučaju da je lokot zauzet, a dretva ga želi preuzeti, dretva mora čekati, što troši vrijeme i može uvelike usporavati izvršavanje programa. Također, veoma opasna situacija koja se mora izbjeći je pojava stanja potpunog zastoja (*deadlock*). Zato su korisni neblokirajući algoritmi kod kojih greška ili obustava rada jedne dretve ne može prouzročiti prestanak rada druge dretve. Nadalje, neblokirajući algoritam je bez zaključavanja (*lock-free*) ako uvijek postoji garantiran napredak na razini sustava. Takvi algoritmi nisu nužno brži te mogu postojati čekanja među dretvama, ali eliminiraju mogućnost pojave potpunog zastoja.

U ovom radu ću opisati memorijski model jezika C++, atomske operacije i atomske varijable iz biblioteke `std::atomic`, njihovo korištenje te načine sinkronizacije između različitih programskih niti, s posebnim naglaskom na novosti uvedene u standardu C++20. Objasnit ću potrebu za različitim memorijskim uređajima (`memory_order_relaxed`, `memory_order_consume`, `memory_order_acquire`, `memory_order_release`, `memory_order_acq_rel`, `memory_order_seq_cst`). Diskutirat ću konstrukciju struktura podataka za asinhroni pristup bez zaključavanja (*lock free*) koje, uz već postojeće, koriste i novosti uvedene u standardu C++20 te ih usporediti s kodom koji koristi zaključavanje mutexima.

Poglavlje 1

C++ memorijski model i operacije na atomskim varijablama

U ovom poglavlju opisat ću strukturalni aspekt memorijskog modela programskog jezika C++, odnosno način na koji su objekti spremljeni u memoriji. Zatim dajem opis povezanosti strukturalnog aspekta i konkurentnosti te navodim što je red modifikacije i kako utječe na konkurentne zadatke. Naposljetku, opisujem atomske operacije i standardne atomske tipove iz `<atomic>` zaglavlja te kako promjena reda modifikacije može uvelike promijeniti slijed izvršavanja atomskih operacija.

1.1 Objekti i memorijske lokacije

Bajt (*en. byte*) je najmanja memorijska jedinica koju je moguće adresirati. Definira se kao kontinuirani niz bitova, dovoljno velik da sadrži bilo koji od 256 znakova iz UTF-8 kodiranja. S druge strane, memorijska lokacija je jedno od sljedećeg:

- Objekt skalarnog tipa; aritmetički tip, enumeracijski tip, pokazivač ili *std::nullptr_t*
- Najveći niz susjednih bit-polja; struktura podataka koja se sastoji od određenog broja susjednih memorijskih lokacija koje služe za držanje niza bitova, pohranjenih tako da se može adresirati bilo koji pojedinačni bit ili grupa bitova unutar skupa.

U C++ jeziku, svi podaci se sastoje od objekata. Svaki objekt je spremljen u nekoj memorijskoj lokaciji. Svaki objekt okupira *barem* jednu memorijsku lokaciju. U primjeru 1.1, preuzetog s [3], vidimo kako se objekt može zapisati u memoriji. Uočimo da su objekti `b` i `c` bit-polja koja dijele istu memorijsku lokaciju. Po pravilu, bit-polja veličine nula ne zauzimaju memorijsku lokaciju i ne smiju imati dodijeljen naziv, ali mogu služiti za separiranje sljedećeg bit-polja (u ovom slučaju bit-polje `d`) u zasebnu memorijsku lokaciju.

Listing 1.1: Primjer podjele strukture u memorijske lokacije

```
struct S {
    char a;      // memory location #1
    int b : 5;   // memory location #2
    int c : 11, // memory location #2 (continued)
          : 0,
        d : 8;   // memory location #3
    struct {
        int ee : 8; // memory location #4
    } e;
} obj; // The object 'obj' consists of 4 separate memory locations
```

Kao što smo naveli u uvodu, postoji uska povezanost između načina spremanja objekata u memoriju i višedretvenih programa. Ukoliko dvije dretve dohvaćaju objekte s različitih memorijskih lokacija, ili pak samo čitaju isti objekt, ne postoji mogućnost stanja natjecanja za resurse. S druge strane, ako dvije dretve žele mijenjati objekt na istoj memorijskoj lokaciji, mora postojati definiran poredak kojim dretve čitaju i mijenjaju sadržaj memorijske lokacije koji je jednak u svim dretvama programa. U suprotnom imamo nedefinirano ponašanje, što čini cijeli program nepredvidljivim i nesigurnim za korištenje te se mora izbjeći pod svaku cijenu. Postoji više načina uređivanja poretka operacija. Jedan od najčešćih je uporaba lokota (*mutexa*). U ovom radu opisat ćemo uređivanje poretka operacije korištenjem atomskih operacija i njihovih sinkronizacijskih svojstava. Prije opisa atomskih operacija i tipova, potreban nam je još pojam - **red modifikacije**.

1.1.1 Red modifikacije

Red modifikacije sastoji se od svih promjena danog objekta uzrokovanih pisanjem iz svih korištenih programskih dretvi, počevši s njegovom inicijalizacijom. Svaki objekt programa C++ ima svoj red modifikacije oko kojeg se u svakom izvršavanju programa sve dretve moraju složiti i kao rezultat vidjeti jedinstven red modifikacije svakog objekta. Ukoliko objekt nije *atomic* tipa, programer je sam zadužen za određivanje željenog redoslijeda modifikacija kako ne bi došlo do nedefiniranog ponašanja programa. S druge strane, ako je objekt *atomic* tipa, prevodilac sam određuje njegov red modifikacije.

Zbog optimizacije prevodioca i dohvaćanja odnosno spremanja podataka u memoriju, različite dretve ne moraju istovremeno vidjeti isto stanje objekta. Ono što je bitno jest da jednom kada dretva pročita neku promjenu iz reda modifikacije, kasnijim čitanjima može vidjeti samo istu ili neku od kasnijih promjena, a kasnija pisanja u taj objekt se moraju dogoditi kasnije u redu modifikacije. Također, ako se u istoj dretvi čitanje dogodi nakon pisanja, ono mora vidjeti zapisanu vrijednost ili neku kasniju vrijednost iz reda modifikacije. Više o poretku modifikacija među dretvama bit će opisano u poglavlju 1.4.

1.2 Atomski tipovi i operacije jezika C++

Atomska operacija je **nedjeljiva** operacija. To znači da u svakom trenutku, svaka dretva vidi tu operaciju kao nezapočetu ili već završenu, ne može ju vidjeti napola završenu. Kao rezultat toga, atomske operacije ne mogu prouzročiti stanje natjecanja za resurse. Ako atomski čitamo neki objekt, dohvatit ćemo ili inicijalnu vrijednost tog objekta ili neku vrijednost iz reda modifikacije. Da bi obavili atomsku operaciju, uglavnom moramo imati atomski tip podatka.

1.2.1 Standardni atomski tipovi

Svi standardni atomski tipovi mogu se pronaći u zaglavlju `<atomic>`. Sve operacije nad tim tipovima su atomske. Ključna stvar kod atomskih operacija jest ta da ne koriste lokote i time omogućuju programiranje bez zaključavanja. Ipak, prevodioci na nekim arhitekturama koriste lokote za operacije na nekim atomskim tipovima. Iz tog razloga gotovo svi atomski tipovi imaju funkciju članicu `is_lock_free()` koja za dan prevodioc i arhitekturu procesora, vraća istinu ako operacije ne koriste lokote. Kako danas mnogi programi moraju bit neovisni o arhitekturi, a svejedno zadržati pogodnosti koje nude atomske operacije, od standarda C++17, svi atomski tipovi imaju `static constexpr` varijablu članicu, `X::is_always_lock_free` koja je istinita ako i samo ako atomski tip `X` ne koristi zaključavanje lokotima ni za koju podržanu arhitekturu na kojoj korišteni prevodioc može raditi. Također su uvedeni makroi, npr. `ATOMIC_INT_LOCK_FREE`, koji mogu vratiti:

- 0 - ako navedeni atomski tip uvijek koristi lokote
- 1 - status korištenja lokota moguće je saznati jedino prilikom prevođenja programa korištenjem funkcije članice `X::is_always_lock_free`
- 2 - ako nikad ne koristi lokote

Jedini tip koji ne pruža metodu `is_lock_free()` je `std::atomic_flag`. Ovo je najjednostavniji atomski tip koji predstavlja logičku (*Boolean*) zastavu. Operacije nad njim **nikad** ne koriste zaključavanje lokotima. `std::atomic_flag` pruža minimalnu funkcionalnost: nakon inicijalizacije je očišćena te može biti postavljena korištenjem metode `test_and_set()` i očišćena korištenjem metode `clear()`.

Ostali atomski tipovi pružaju veći broj metoda i može ih se dohvatiti preko parametrizirane klase `std::atomic<>`. Za očekivati je da na većini popularnih arhitektura klase parametrizirane standardnim tipovima (poput `bool`, `int`, `long` itd.) ne koriste zaključavanje, premda to ne mora biti slučaj. Uz dohvaćanje preko parameterizirane klase, postoje predodređeni tipovi koji korespondiraju specijalizacijama klase standardnim integralnim tipovima. Njih možemo vidjeti u tablici 1.1

Tablica 1.1: Alternativni nazivi za standardne atomske tipove

Atomski tip	Specijalizacija
atomic_bool	std::atomic<bool>
atomic_bool	std::atomic<bool>
atomic_char	std::atomic<char>
atomic_schar	std::atomic<signed char>
atomic_uchar	std::atomic<unsigned char>
atomic_int	std::atomic<int>
atomic_uint	std::atomic<unsigned>
atomic_short	std::atomic<short>
atomic_ushort	std::atomic<unsigned short>
atomic_long	std::atomic<long>
atomic_ulong	std::atomic<unsigned long>
atomic_llong	std::atomic<long long>
atomic_ullong	std::atomic<unsigned long long>
atomic_char8_t[1]	std::atomic<char8_t>
atomic_char16_t	std::atomic<char16_t>
atomic_char32_t	std::atomic<char32_t>
atomic_wchar_t	std::atomic<wchar_t>

Standardna biblioteka jezika C++ također pruža niz *typedef* specijalizacija za atomske tipove koje odgovaraju definicijama tipova standardne biblioteke poput `std::size_t`. Cijelu listu možemo vidjeti u [2].

Operacije nad nekim atomskim tipom tipično se koriste kao funkcije članice atomskog objekta. Ipak, za svaku atomsku operaciju, uz atomsku funkciju članicu postoji i odgovarajuća slobodna funkcija koja prima atomski objekt kao argument te izvodi odgovarajuću operaciju nad njim. Nomenklatura je ista za sve operacije. U odnosu na operaciju članicu, slobodna funkcija koristi prefiks `atomic_`. Ako funkciji želimo predati i memorijski uređaj kao argument, tada moramo pozvati eksplicitnu verziju koja ima sufiks `_explicit`. Neeksplicitne verzije standardno koriste memorijski uređaj `std::memory_order_seq_cst`.

[1] Uvedeno u standardu C++20

Listing 1.2: "Primjer različitih načina pozivanja operacije *store*"

```
std::atomic<int> foo;
foo.store(21);
std::atomic_store(foo, 21);
std::atomic_store_explicit(foo, 21, std::memory_order_release);
```

Atomske tipove ne možemo konstruirati korištenjem konstruktora kopije niti konstruktora kopije premještanjem. Također ne možemo pridružiti jedan objekt istog tipa drugome. Naime, sve operacije nad atomskim tipovima moraju biti atomske, a premještanje i konstruktor kopije uključuju dva različita objekta. U slučaju konstruktora kopije, prvo moramo pročitati vrijednost jednog objekta i zapisati ju u drugu vrijednost. Ovo su dvije različite operacije nad dva različita objekta i njihova kombinacija ne može biti atomska.

U standardu C++20, uvedena je specijalizacija za *floating point* tipove *float*, *double* i *long double*. U odnosu na specijalizaciju integralnim tipovima, nisu podržane operacije `fetch_and`, `fetch_or` i `fetch_xor`, dok ostale metode imaju isto ponašanje. Također, operacije neće rezultirati nedefiniranim ponašanjem, čak i kada rezultat nije moguće reprezentirati *floating point* tipom.
Već smo spomenuli osnovni atomski tip `atomic_flag` koji ima veoma ograničenu funkcionalnost. Funkcionalnost koju pruža neki atomski tip bit će određena tipom kojim je specijaliziran. Tako atomske tipove možemo podijeliti u 5 grupa s obzirom na tip i metode koje su implementirane.

- `atomic_flag` - osnovni najjednostavniji atomski tip
- `atomic<bool>` - standardni *Boolean* tip
- `atomic<T*>` - specijalizacija tipom pokazivača
- `atomic<`*`integralni tip`*`>` - specijalizacija osnovnim integralnim tipom
- `atomic<`*`T`*`>` - T može biti bilo koji *TriviallyCopyable* tip koji je *CopyConstructible* i *CopyAssignable*

Grupe i njima dopuštene operacije dopuštene možemo vidjeti u tablici 1.2 Detalji operacija će biti opisani kasnije. Vidimo, dakle, da možemo kreirati atomsku varijablu gotovo bilo koje klase ili strukture koju sami kreiramo, no uz nešto ograničeniju funkcionalnost. Prije opisa funkcionalnosti pojedinih grupa atomskih tipova, dajemo kratki opis memorijskih uređaja i njihove uporabe u atomskim operacijama.

1.2.2 Memorijski uređaji

Atomske operacije se primarno koriste kako bi se sinkronizirali pristupi dijeljenoj memoriji u višedretvenom sustavu. U višedretvenom sustavu bez restrikcija na memorijski

uređaj, kada više dretvi istovremeno čita i zapisuje objekte u memoriju, jedna dretva može vidjeti različit poredak promjena objekata u memoriji od onog kojim je druga dretva zapisala te objekte. Razlog tome su hijerarhijska organizacija memorije te hardverske optimizacije koje prevodilac uvodi kako bi se program izvršavao brže i efikasnije. Svaka operacija nad atomskim tipom kao argument uzima memorijski uređaj. Memorijski uređaj specificira kako će pristupi memoriji, uključujući i ne-atomske pristupe, biti uređeni oko atomske operacije. Drugim riječima, memorijski uređaj određuje koliko je jaka veza dijeljenja podataka među dretvama. Naravno, što je veza jača, to su optimizacije prevodioca restriktivnije i program se izvršava sporije.

Postoji 6 različitih memorijskih uređaja. To su:

- `memory_order_relaxed`
- `memory_order_consume`
- `memory_order_acquire`
- `memory_order_release`
- `memory_order_acq_rel`
- `memory_order_seq_cst`

Ukoliko operaciji ne predamo memorijski uređaj kao argument, zadani memorijski uređaj će biti `memory_order_seq_cst`. Isto vrijedi i za slobodne atomske funkcije. To je najrestriktivniji uređaj koji uzrokuje jedinstven red modifikacije nad svim operacijama koje ga koriste. Samim time i svaka operacija koja ga koristi troši veći dio procesorskog vremena, što uzrokuje sporiji program. Ukoliko želimo koristiti atomske varijable za sinkronizaciju višedretvenog programa, a da pritom iskoristimo puni potencijal programiranja atomskim tipovima, moramo koristiti ostale, manje restriktivne memorijske uređaje.

Dopušten memorijski uređaj ovisi o kategoriji atomske operacije. Postoje 3 kategorije:

- *Store* operacije - mogu imati `memory_order_relaxed`, `memory_order_release` i `memory_order_seq_cst` uređaje.
- *Load* operacije - mogu imati `memory_order_relaxed`, `memory_order_consume`, `memory_order_acquire` i `memory_order_seq_cst` uređaje.
- *Read-modify-write* operacije - kao kombinacije prethodne dvije kateogrije, mogu imati bilo koji od mogućih 6 memorijskih uređaja.

Ovo je bilo osnovno što treba znati o memorijskim uređajima prije prelaska na detaljniji opis atomskih tipova i operacija. Kada se javlja potreba za pojedinim memorijskim uređajem i koje posljedice on ima za memoriju, bit će opisano u poglavlju 1.4

1.2.3 Operacije nad `std::atomic_flag`

`std::atomic_flag` je, kao što smo prije naveli, najjednostavniji atomski tip. Operacije nad njom su uvijek *lock-free*. Predstavlja boolean zastavu koja može biti u dva stanja: postavljena (*set*) i očišćena (*clear*). Objekt tipa `std::atomic_flag` se do uvoda standarda C++20 uvijek morao inicijalizirati s vrijednosti `ATOMIC_FLAG_INIT`, što postavlja zastavicu u *clear* stanje. U standardu C++20, očišćeno stanje zastavice se postavlja u konstruktoru, a vrijednost `ATOMIC_FLAG_INIT` je označena kao zastarjela.

Nakon što smo inicijalizirali zastavicu, postoje 3 operacije koje možemo primijeniti: uništiti ju pozivanjem destruktora, očistiti ju pozivom metode `clear()` ili ju postaviti i dobiti nazad vrijednost koju je prethodno sadržavala pozivom metode `test_and_set()`. Operacija `clear()` pripada kategoriji *store* operacija, a `test_and_set()` kategoriji *Read-modify-write* operacija te su time određeni dopušteni memorijski uređaji. Kao ni bilo koji drugi atomski tip, objekt tipa `std::atomic_flag` ne možemo konstruirati koristeći konstruktor kopije niti konstruktor kopije premještanjem. Također mu ne možemo pridružiti drugi objekt istog tipa.

`std::atomic_flag` je, zbog svoje jednostavnosti, dobar objekt za konstrukciju jednostavnog lokota. Inicijalno, zastavica je očišćena što znači da je lokot otključan. Možemo ga zaključati kontinuiranim pozivima metodi `test_and_set()` dok ne dobijemo povratnu vrijednost *false*. Naime, jer smo dobili vrijednost *false*, znamo da je lokot bio otključan pa ga možemo preuzeti i ponovno zaključati. Lokot zatim možemo jednostavno otključati pozivom metode `clear()`. Ovakvu implementaciju možemo vidjeti u sljedećem primjeru.

Listing 1.3: Implementacija lokota pomoću `atomic_flag` objekta

```
std::atomic_flag lock = ATOMIC_FLAG_INIT;

void lock() {
    while(lock.test_and_set(std::memory_order_acquire));
}
void unlock() {
    lock.clear(std::memory_order_release);
}
```

U prethodnom primjeru smo koristili standardne memorijske uređaje za ovakve operacije. Detaljan opis zašto se koriste i kako ovako korišteni memorijski uređaji pružaju dovoljan uvjet za rad lokota će biti dan kasnije.

Ovakav lokot nije idealan jer ukoliko je lokot zaključan, a mi ga želimo dohvatiti, metoda `lock()` će zapeti u petlji i neprestano trošiti procesorsko vrijeme. `atomic_flag` je, možemo reći, već zastarjeli atomski tip koji se jako rijetko može vidjeti u svakodnevnoj uporabi. Čak i za ovako jednostavni lokot, efikasnije je koristiti `atomic<bool>` objekt zbog mogućnosti dohvata vrijednosti bez njene modifikacije.

1.2.4 Operacije nad `std::atomic<bool>`

Atomski tip `std::atomic<bool>` može pružiti puno više od `std::atomic_flag`, koji je suviše jednostavan za stvarnu upotrebu. Za početak, može se konstruirati iz već postojećeg ne-atomskog *bool* tipa, što znači da se može inicijalizirati na *true* ili *false*. Kao što smo prije naveli, svi atomski tipovi osim `atomic_flag` imaju metodu `is_lock_free()` koja vraća *true* ukoliko zadani atomski tip ne koristi zaključavanje lokotima na trenutno korištenoj arhitekturi. `std::atomic<bool>`, kao i svi ostali atomski tipovi osim `atomic_flag`, koristi metode `load()` i `store()` za dohvaćanje odnosno zapisivanje vrijednosti. `load()` predstavlja nemodificirajuću *load* operaciju, a `store()` standardnu *store* operaciju. Umjesto funkcije `test_and_set()`, koristi se metoda `exchange()` koja dopušta programeru da zapiše željenu vijednost i kao povratnu vrijednost dobije prethodno sadržanu vrijednost. `exchange()` je također *read-modify-write* operacija. Sve metode kao argument primaju memorijski uređaj.

Listing 1.4: Primjer rada sa `std::atomic<bool>`

```
std::atomic<bool> a, b(true);                    // b = true
a = false;                                       // a = false
bool c = a.load(std::memory_order_acquire);      // c = false
b.store(false);                                  // b = false
c = a.exchange(true, std::memory_order_acq_rel); // c = false, a = true
```

Još jedna stvar koju uvodi `std::atomic<bool>`, a prisutna je i u ostalim složenim tipovima, jest dodjeljivanje vrijednosti pridruživanjem. Ne možemo pridružiti drugu atomsku varijablu zbog zabrane kopiranja, već proizvoljnu ne-atomsku vrijednost. Pridruživanje je ekvivalentno pozivanju metode `store()` sa zadanom vrijednošću. Za razliku od ostalih operatora pridruživanja, pridruživanje atomskim tipovima ne vraća referencu, već kopiju spremljene vrijednosti. Naime, kad bi se vraćala referenca na atomski objekt, bilo koji drugi objekt koji ovisi o rezultatu pridruživanja bi morao izvršiti eksplicitni *load* vrijednosti čime bi mogao dohvatiti vrijednost koju je spremila neka druga dretva.

`std::atomic<bool>`, kao i svi ostali složeni atomski tipovi, koriste još jednu *read-modify-write* operaciju koja je specifična za atomske tipove i predstavlja temelj za pisanje *lock-free* struktura podataka, a to je tzv. **compare-exchange** operacija. Ona dolazi u dvije varijante: `compare_exchange_strong()` i `compare_exchange_weak()`. Kao argumente primaju barem dvije vrijednosti: referencu na varijablu koja sadrži vrijednost koju očekujemo te vrijednost koju želimo zapisati. Punu definiciju možemo vidjeti u 1.5, analogno je i za `compare_exchange_weak()`. Compare-exchange operacija uspoređuje trenutnu vrijednost atomske varijable s očekivanom i ukoliko su jednake, zapisuje željenu vrijednost i vraća *true* kao povratnu vrijednost. Ukoliko vrijednosti nisu jednake ili zapisivanje ne uspije, varijabla s očekivanom vrijednosti je ažurirana vrijednošću atomske varijable (razlog slanja reference na varijablu) i operacija vraća *false*.

Listing 1.5: Puna definicija operacije `compare_exchange_strong()`

```
bool compare_exchange_strong( T& expected, T desired,
                              std::memory_order success,
                              std::memory_order failure )
```

U novouvdenom standardnu C++20, dosadašnje uspoređivanje objektne reprezentacije zamijenjeno je vrijednosnom reprezentacijom. Za objekt tipa T, **objektna reprezentacija** je niz `sizeof(T)` objekata tipa `unsigned char` (ili `std::byte`), počevši od iste adrese. **Vrijednosna reprezentacija** objekta je skup bitova koji drže vrijednost tipa T. Za detalje pogledati [5].

Dok operacija `compare_exchange_strong()` vraća *false* ako i samo ako stvarna vrijednost nije bila jednaka očekivanoj, `compare_exchange_weak()` može vratiti *false* čak i kada to nije slučaj. Tada kažemo da operacija spuriozno nije uspjela. U slučaju spurioznog neuspjeha, vrijednost atomske varijable ostaje nepromijenjena, a operacija vraća *false*. Šanse da se to dogodi su male, ali ipak postoje kod nekih platformi gdje procesor ne može garantirati nedjeljivost compare-exchange operacije. Na primjer, u sustavu gdje ima više dretvi nego procesora, dretva koja vrši operaciju može biti zaustavljena i zamijenjena, a da slijed instrukcija compare-exchange operacije nije završio. Iz tog razloga, `compare_exchange_weak()` operacije se najčešće koristi u petlji da bi se osiguralo od spurioznog neuspjeha.

Listing 1.6: Primjer korištenja `compare_exchange_weak()`

```
bool expected=false;
extern atomic<bool> b;        // Defined outside
while(!b.compare_exchange_weak(expected,true) && !expected);
```

Dakle, vrtimo petlju sve dok je `expected==false` kako bi se izbjegao spuriozni neuspjeh.

Činjenica da se takav neuspjeh može dogoditi, možda nas navodi da uvijek koristimo operaciju `compare_exchange_strong()`, no ona u sebi ima ugrađenu provjeru spurioznog neuspjeha, što na nekim platformama može biti osjetno skuplje nego korištenje `compare_exchange_weak()`. Također, u nekim slučajevima, programer može i dopustiti spuriozni neuspjeh čime je korištenje *strong* verzije nepotrebno.

Još jedna neobičnost compare-exchange operacije, a koju smo mogli vidjeti u njenoj definiciji, jest korištenje dva memorijska uređaja kao parametre. Prvi zadani memorijski uređaj se koristi u slučaju uspjeha operacija, a drugi u slučaju neuspjeha. U slučaju neuspjeha, operacija ne sprema iznos u memoriju, tako da kao drugi parametar ne možemo imati `memory_order_release` ili `memory_order_acq_rel` uređaje. Također ne možemo imati stroži memorijski uređaj u slučaju neuspjega nego u slučaju uspjeha. Ako operaciji zadamo samo jedan memorijski uređaj kao parametar, isti će bit korišten i za neuspjeh, ali će `memory_order_release` postati `memory_order_relaxed`, a `memory_order_acq_rel` `memory_order_acquire`. Ako pak uopće ne zadamo memorijski uređaj, kao i kod ostalih atomskih operacija, koristit će se `memory_order_seq_cst`.

1.2.5 Operacije nad `std::atomic<T*>`

`std::atomic<T*>` predstavlja atomski tip pokazivača na neki tip `T`. Sučelje je isto kao kod `std::atomic<bool>`, dakle zadržane su analogne operacije, ali naravno uz povratni tip *T** umjesto *bool*. Nove operacije koje podržava `std::atomic<T*>` su:

- `fetch_add()` i `fetch_sub()` - primaju argument tipa `std::ptrdiff_t` koji predstavlja cjelobrojni tip s predznakom koji je rezultat oduzimanja dva pokazivača i vrše atomsko zbrajanje i oduzimanje. Ovo su *read-modify-write* operacije tako da su svi memorijski uređaji dozvoljeni. Kao povratnu vrijednost vraćaju originalnu vrijednost prije izvršavanja operacije. Dakle povratni tip je `T*`, a ne referenca na `std::atomic<T*>` objekt tako da pozivajući kod ovisi o prethodnoj vrijednosti kao što je bilo opisano u prethodnom odjeljku.

- Operatori += i -= - Automatski zamjenjuje trenutnu vrijednost s rezultatom zbrajanja odnosno oduzimanja trenutne vrijednosti i argumenta. Imaju isto ponašanje kao i `fetch_add()` i `fetch_sub()` operacije, ali za razliku od njih ne primaju memorijski uređaj kao drugi argument, već kao zadano koriste `memory_order_seq_cst` uređaj te vraćaju novu, a ne originalnu vrijednost. Naravno, povratni tip je opet `T*`, a ne referenca na objekt.

- Operatori ++ i `--` - post i pred-inkrement operatori koji imaju isto ponašanje kao += i -= operatori. Pred-inkrement vraća vrijednost atomske varijable nakon modifikacije, a post-inkrement vrijednost prije operacije.

S obzirom da radimo s pokazivačima, rezultat može biti nedefinirana adresa, ali operacije inače ne mogu imati nedefinirano ponašanje. Tip T mora predstavljati neki objektni tip. Primjenu prethodnih operacija možemo vidjeti u sljedećem primjeru.

```
class C{}                       // Neka proizvoljna klasa
C array[10];
std::atomic<C*> p(array);
auto x = p.fetch_add(5);        // x je tipa C*
assert(x == array);
assert(p.load() == &array[5]);
x = p--;
assert(x == &array[5]);
assert(p.load() == &array[4]);
x = --p;
assert(x == &array[3]);
```

1.2.6 Operacije nad standardnim atomskim integralnim tipovima

Uz operacije opisane u poglavljima 1.2.4 i 1.2.5, standardni atomski tipovi poput `std::atomic<int>` ili `std::atomic<long long>` sadrže metode `fetch_and()`, `fetch_or()` i `fetch_xor()` koje provode standardne *bitwise* operacije te njima korespondirajuće operatore &=, |= i ^=. Ponašanja su ista kao što je bilo opisano za operacije `fetch_add()` i `fetch_sub()` i operatore += i -= u prethodnom odjeljku. U ovu skupinu atomskih tipova spadaju i novouvedene specijalizacije *floating point* tipova *float, double* i *long double*.

Vidimo da nisu definirani operatori množenja, dijeljenja i bitnog posmaka, no njih možemo po potrebi možemo simulirati koristeći compare-exchange operaciju u petlji.

Standard C++20 uveo je mnogo novosti u `std::atomic` biblioteku koje ćemo sljedeće navesti.

1.3 Novosti standarda C++20

Već smo naveli neke promjene koje je uveo standard C++20, poput podrške za *floating-point* tipove *float*, *double* i *long double*, te nove tipove poput `atomic_char8_t`, no uvedeno je još mnogo novosti u `<atomic>` biblioteci. Iznijet ćemo najvažnije promjene, ostale se mogu pronaći u [2] i [1].

1.3.1 Sinkronizirajuće operacije

C++20 uvodi tri nove metode koje služe za olakšanu višedretvenu sinkronizaciju, a čije je korištenje omogućeno nad svim atomskim tipovima, čak i `std::atomic_flag`. To su:

- `wait(T old, std::memory_order order=std::memory_order::seq_cst)` - operacija atomskog čekanja. U pozadini, neprestano izvodi sljedeći niz operacija:
  - Usporedi vrijednosnu reprezentaciju od `this->load(order)` i `old`.
    - Ako su iste, blokiraj dretvu dok `*this` nije obaviješten operacijom `notify_one()` ili `notify_all()`, ili dretva bude odblokirana spuriozno.
    - Inače *return*.

  Funkcija garantirano vrši *return* tek kada je promijenjena vrijednost, čak i kada je dretva bila odblokirana spuriozno.

- `notify_one()` - Obavijesti *barem jednu* dretvu koja čeka na atomski objekt.

- `notify_all()` - Obavijesti *sve* dretve koje čekaju na atomski objekt.

Primijetimo da operacije `notify_one()` i `notify_all()` ne primaju memorijski uređaj kao parametar. Razlog zašto su ove operacije uvedene jest taj što je ovakav oblik detektiranja promjene vrijednosti atomskog tipa puno efektivniji od korištenja atomskih lokota poput onog napisanog u primjeru 1.3. Jedan jednostavan oblik sinkronizacije dvije dretve možemo vidjeti u sljedećem primjeru.

```
std::vector<int> myVec{};
std::atomic_flag atomicFlag{};

void prepareWork() {
    myVec.insert(myVec.end(), {0, 1, 0, 3});
    std::cout << "Sender: Data prepared." << std::endl;
    atomicFlag.test_and_set();
    atomicFlag.notify_one();                                    // (1)
}

void completeWork() {
    std::cout << "Worker: Waiting for data." << std::endl;
    atomicFlag.wait(false);                                     // (2)
    myVec[2] = 2;
    std::cout << "Waiter: Complete the work." << std::endl;
}

int main() {
    std::thread t1(prepareWork);
    std::thread t2(completeWork);
    t1.join();
    t2.join();
    for (auto i: myVec) std::cout << i << " ";      // 0 1 2 3
}
```

Dretva koja priprema posao puni vektor vrijednostima, postavlja zastavicu i u `(1)` obaviještava jednu dretvu da se dogodila promjena zastavice. Dretva koja završava posao u početku čeka sve dok je zastavica postavljena na *false* `(2)`. Nakon što je obaviještena o promjeni zastavice, provjerava da ona više nije postavljena na *false* te nastavlja s radom. Na taj način sinkronizirali smo rad dvije dretve te po završetku obje dretve, dobivamo ispis `0 1 2 3`, kakav smo i željeli.

1.3.2 Atomske specijalizacije `std::shared_ptr` i `std::weak_ptr`

Kod višedretvenog programiranja bez zaključavanja, korištenje pametnih pokazivača može uvelike olakšati pisanje sigurnog programa. `std::shared_ptr` je jedini ne-atomski tip na kojeg možemo primijeniti atomske operacije. Naime, ima svojstvo minimalne nedjeljivosti: povećanje i smanjenje brojača referenci su atomske operacije te postoji garancija da će resurs na koji pametni pokazivač pokazuje biti uništen točno jednom. No s

druge strane, dohvaćanje tih resursa nije atomska operacije. Ako više dretvi dohvaća isti `std::shared_ptr` objekt bez međusobne sinkronizacije i bilo koja od tih dretvi koristi nekonstantnu metodu članicu tog objekta, dogodit će se stanje natjecanja za resursima. Ista stvar vrijedi i za `std::weak_ptr`.

```
std::shared_ptr<int> ptr = std::make_shared<int>(100);

for (auto i= 0; i<10; i++){
   std::thread([&ptr, i]{                          (1)
     ptr= std::make_shared<int>(i);                (2)
   }).detach();
}
```

U prethodnom primjeru, lambda funkcija prima pokazivač `ptr` po referenci `(1)`. Zbog toga, promjena objekta u `(2)` uzrokuje stanje natjecanja za resurse što znači da program ima nedefinirano ponašanje.

Kao rješenje ovog problema, standard C++20 uvodi tipove `std::atomic<std::shared_ptr<T>>` i `std::atomic<std::weak_ptr<T>>`. Dohvaćanje njihovih resursa je atomska operacija pa ne uzrokuje natjecanje za resursima što je čini sigurnom za korištenje u višedretvenim programima. Atomske specijalizacije objekata `std::shared_ptr` i `std::weak_ptr` pružaju istu funkcionalnost kao `std::atomic<bool>`: dopuštene su operacije poput `load()`, `store()`, `wait()`, `notify_one/all()` i *compare-exchange* operacija, no operacije poput `fetch_ add()` ili inkrementa(++) i dekrementa(--) nisu implementirane.

Atomski pametni pokazivači su izrazito korisni pri pisanju struktura podataka s podrškom za višedretveno programiranje. Primjerice, možemo definirati stog na sljedeći način:

```
template<typename T> class concurrent_stack {
    struct Node {
        T t;
        shared_ptr<Node> next;
    };
    atomic_shared_ptr<Node> head;
    concurrent_stack( concurrent_stack & ) = delete;
    void operator=( concurrent_stack& ) = delete;

    ...              // Definicije konstruktora i preostalih varijabli
};
```

Primijetimo da ne smijemo dopustiti korištenje konstruktora kopije, ni operator pridruživanja jer to ne mogu biti atomske operacije te stoga mogu prouzročiti nedefinirano ponašanje. Operacije ubacivanja elementa i izbacivanja početnog elementa tada su vrlo jednostavne, a ipak sigurne za korištenje u višedretvenom programu.

```
void push_front( T t ) {
    auto p = make_shared<Node>();
    p->t = t;
```

```
    p->next = head;          // in C++11: atomic_load(&head)
    while( !head.compare_exchange_weak(p->next, p) );
}
void pop_front() {
   auto p = head.load();
   while( p && !head.compare_exchange_weak(p, p->next) );
}
```

1.3.3 Atomska referenca `std::atomic_ref`

Još jedna velika novost standarda C++20 bila je uvođenje novog atomskog tipa; atomske reference odnosno `std::atomic_ref`. Ona primjenjuje atomske operacije na objekt koji referencira. Za vrijeme životnog vijeka atomske reference, objekt koji referencira se smatra atomskim. Također, životni vijek referenciranog objekta mora biti dulji od životnog vijeka atomske reference. Sve dok postoji atomska referenca na objekt, sva dohvaćanja i pisanja tog objekta moraju se izvršavati kroz tu referencu. Štoviše, ne smije postojati atomska referenca na niti jedan pod-objekt objekta na koji već postoji atomska referenca. Atomska referenca pruža isti skup dopuštenih metoda kao objekt kojim je specijalizirana.

`std::atomic_ref` je *CopyConstructible*, odnosno može se konstruirati iz lijeve vrijednosti (*lvalue*) korištenjem konstruktora kopije. Ukoliko koristimo konstantnu atomsku referencu, tj. `const std::atomic_ref`, svejedno je moguće promijeniti referenciranu vrijednost. Objekt `std::atomic_ref` možemo instancirati bilo kojim *TriviallyCopyable* tipom T (uključujući i *bool*). Dakle, sljedeći primjer je valjan.

```
struct Counters {
    int a;
    int b;
};
Counter counter;
std::atomic_ref<Counters> cnt(counter);
```

Pokažimo sada na jednostavnom primjeru mogućnosti koje donosi `std::atomic_ref`. Zamislimo da imamo strukturu `ExpensiveToCopy` čije je kopiranje preskupo te ga nastojimo izbjeći.

Listing 1.7: Primjer potrebe za `std::atomic_ref`

```
struct ExpensiveToCopy {
    ...
    int counter{};
};

void increase_count(ExpensiveToCopy& exp) {              // (2)
    std::vector<std::thread> v;
    std::atomic<int> counter{exp.counter};               // (3)
```

```
    for (int n = 0; n < 10; ++n) {                          // (4)
        v.emplace_back([&counter] {
            for (int i = 0; i < 10; ++i) { ++counter; }
        });
    }
    for (auto& t : v) t.join();
}

int main() {
    ExpensiveToCopy exp;                                    // (1)
    increase_count(exp);
    std::cout << "exp.counter:" << exp.counter << '\n'; // (5)
}
```

Objekt `exp (1)` predstavlja strukturu čije kopiranje želimo izbjeći. Zato funkcija `increase_count (2)` prima referencu na objekt `exp`. Ona incijalizira `atomic<int>` objekt s `exp.counter (3)`. Zatim kreira 10 dretvi `(4)`, od kojih svaka izvodi lambda izraz koji prima `counter` po referenci te ga inkrementira. Na kraju, s obzirom da `increase_count` stvara 10 dretvi koje inkrementiraju brojač 10 puta, očakivana vrijednost u ispisu `(5)` bi bila 100. No, pokrenemo li ovaj program, dobivamo ispis:

```
exp.counter: 0
```

Naime, incijalizacija `std::atomic<int> counter{exp.counter};` stvara kopiju objekta `exp.count` te inkrementiranje atomskog objekta ne utječe na referencirani objekt. Kod ovakvih situacija, jednostavno rješenje se krije u korištenju atomske reference. Napišemo li funkciju `increase_count()` na sljedeći način;

```
void increase_count(ExpensiveToCopy& exp) {
    std::vector<std::thread> v;
    std::atomic_ref<int> counter{exp.counter};
    ...
}
```

tada kreiramo atomsku referencu na objekt `exp.counter` koja je sigurna za višedretveno korištenje te konačno u ispisu dobivamo:

```
exp.counter: 100
```

Ovaj problem smo mogli riješiti i drugačijom definicijom strukture *ExpensiveToCopy*, definiranjem samog brojača atomskim tipom.

```
struct ExpensiveToCopy {
    ...
    std::atomic<int> counter{};
};
```

Ovo je zasigurno valjano rješenje. Lambda izrazu tada samo proslijedimo `exp.counter` i rezultat će biti korektan. No postoji razlog zašto ovakav pristup nije prihvatljiv. Svako dohvaćanje brojača tada bi bilo sinkronizirano, što nije jeftino te bi bilo poželjno izbjeći ako se objekt često modificira. S druge strane, atomska referenca na dopušta da eksplicitno kontroliramo kada želimo atomski pristup objektu. Ukoliko ga uglavnom želimo čitati, tada nema potrebe za definiranjem člana kao atomskog.

S ovim smo opisali osnovne atomske tipove zaglavlja <atomic>. Konačan popis svih dopuštenih metoda po grupama atomskih tipova možemo vidjeti u teblici 1.2. Iduće dajemo detaljan opis memorijskih uređaja navedenih u poglavlju 1.2.2, diskutiramo potrebu za njihovim korištenjem i prezentiramo njihov utjecaj na memoriju.

Tablica 1.2: Operacije dopuštene atomskim tipovima

Operacija	`atomic_flag`	`atomic<bool>`	`atomic<T*>`	`atomic<integralni tip>`	`atomic<ostali>`
`test_and_set`	●				
`clear`	●				
`is_lock_free`		●	●	●	●
`load`		●	●	●	●
`store`		●	●	●	●
`exchange`		●	●	●	●
`compare_exchange_weak`, `compare_exchange_strong`		●	●	●	●
`fetch_add`, `+=`			●	●	
`fetch_sub`, `-=`			●	●	
`fetch_and`, `&=`				●	
`fetch_or`, `|=`				●	
`fetch_xor`, `^=`				●	
`++`, `--`			●	●	
`wait()`	●	●	●	●	●
`notify_one()`	●	●	●	●	●
`notify_all()`	●	●	●	●	●

1.4 Memorijski uređaji

Ključ višedretvenog *lock-free* programiranja nalazi se u ispravnom i pravovremenom korištenju različitih memorijskih uređaja prilikom svake atomske operacije kako bi stvorili jedinstven i ispravan red modifikacije svakog objekta. Već smo napomenuli da je uređaj `memory_order_seq_cst` najrestriktivniji memorijski uređaj koji uzrokuje sekvencijalno izvođenje operacija nad atomskim tipom, no njegovo korištenje može biti skupo i želi se izbjeći prilikom pisanja programa. Ključ, dakle, stoji u pravilnom izmjenjivanju manje restriktivnih memorijskih uređaja nad atomskim operacijama s ciljem ostvarivanja ispravnih relacija memorijskih modela *sinkroniziran-s* i *dogodio-se-prije*. Navedimo razliku između ovih dviju relacija.

Odgovarajuće označena atomska operacija pisanja W nad objektom `x` je u u relaciji ***sinkroniziran-s*** s odgovarajuće označenom atomskom operacijom čitanja nad objektom `x` koja čita vrijednost zapisanu operacijom W ili nekom kasnijom atomskom operacijom pisanja nad `x` u istoj dretvi ili nizom atomskih *read-modify-write* operacija nad `x` bilo koje dretve, gdje je prva pročitana vrijednost u nizu bila ona zapisana operacijom W. Očito relaciju *sinkroniziran-s* možemo imati samo nad atomskim operacijama te je ona relacija parcijalnog uređaja. Pod "odgovarajuće označenom" operacijom, pretpostavlja se na ispravno zadan memorijski uređaj. Detalji će biti jasniji nakon opisa pojedinih memorijskih uređaja i njihovih korištenja.

S druge strane, relacija ***dogodio-se-prije*** je relacija nad svim operacijama programa. Također, postoji i relacija ***dogodio-se-strogo-prije***. Za slučaj jedne dretve, situacija je jednostavna jer imamo uvijek jedinstven niz operacije: ako se operacije A dogodila u nizu prije operacije B, tada je A u relaciji *dogodio-se-prije* i relaciji *dogodio-se-strogo-prije* s operacijom B. Radi jednostavnosti, reći ćemo da se operacija A dogodila prije odnosno strogo prije operacije B. Specijalan slučaj kada ne postoji relacija *dogodio-se-prije* među operacijama u istoj dretvi jest kada se one dogode u istom izrazu.

Listing 1.8: Slučaj kada je poredak evaluacije nespecificiran

```
#include <stdio.h>

void print_elements(int a, int b) {
    std::cout << a << "," << b;
}
int get_int() {
    static int val = 0;
    return val++;
}
int main() {
    print_elements(get_int(), get_int());    // Poredak poziva funkcija
    return 0;                                // get_int() nespecificiran
}
```

Ovo je poznato ponašanje jezika C++, poredak evaluacije operacija u istom izrazu je nespecificiran. U prethodnom primjeru, ne znamo hoće li se izvesti funkcija `print_elements(0,1)` ili `print_elements(1,0)`. Jedino što znamo jest da će se te operacije dogoditi prije operacije u sljedećem izrazu.

Što se tiče slučaja s više dretvi, relacija *dogodio-se-prije* usko je povezana s relacijom *sinkroniziran-s*. Ukoliko je operacija A sinkronizirana s operacijom B, tada se A dogodila prije B. Znamo da je *sinkroniziran-s* relacija nad atomskim tipovima tako da moramo iskoristiti tranzitivnost. Ako je operacija A sinkronizirana s operacijom B, a operacija B se dogodila prije operacije C, tada se operacija A dogodila prije operacije C. Analogno, ukoliko se operacija A dogodila prije operacije B, a operacije B je sinkronizirana s operacijom C, tada se operacija A dogodila prije operacije C. Ovo nam znači da ukoliko napravimo niz promjena unutar jedne dretve, dovoljna je *sinkroniziran-s* relacija da bi te promjene bile vidljive u drugoj dretvi. Iz tog razloga kažemo da se atomske varijable uglavnom koriste kao sinkronizacijski elementi.

Za relaciju *dogodio-se-strogo-prije* vrijede ista pravila kao i za relaciju *dogodio-se-prije*. Jedina razlika je u tome operacije označene memorijskim uređajem `memory_order_consume` sudjeluju u *dogodio-se-prije* relacijama, ali ne i *dogodio-se-strogo-prije*. S obzirom da se `memory_order_consume` ne bi smio koristiti (standard C++17 eksplicitno preporuča da se ne koristi), nadalje će se spominjat samo relacija *dogodio-se-prije*.

Sad kad smo naveli osnovne relacije memorijskih modela, možemo konačno opisati same memorijske uređaje i njihovo korištenje. U poglavlju 1.2.2 već smo naveli 6 postojećih memorijskih uređaja. Svaki od njih može imati različite vremenske troškove na pojedinim CPU arhitekturama. Procesori koji koriste x86 ili x86-64 arhitekture poput Intelovih ili AMD-ovih procesora koji se koriste u većini osobnih računala, ne zahtijevaju dodatne instrukcije za *acquire-release* operacije, osim onih za osiguravanje nedjeljivost, stoga njihov dodatan trošak na ovim arhitekturama ne mora biti suviše značajan.

1.4.1 Sekvencijalno-konzistentan uređaj

Sekvencijalno-konzistentan uređaj, odnosno `std::memory_order_seq_cst`, je najrestriktivniji memorijsku uređaj, ali i najjednostavniji za razumjeti te je zato postavljen kao zadani. Kao što mu ime govori, on implicira da je ponašanje programa konzistentno s jednostavnim sekvencijalnim pogledom na svijet. On uzrokuje jedinstven totalan red modifikacije svakog atomskog objekta u višedretvenom programu koji je njime označen. Sve operacije koje koriste `std::memory_order_seq_cst` će se dogoditi jednom kada su se svi pristupi memoriji koji mogu imati vidljive posljedice na preostale dretve već dogodili. Drugim riječima, prevodilac će obaviti svu sinkronizaciju umjesto programera te će se sve atomske operacije označene ovim uređajem ponašati kao u jednodretvenom programu. Jedna mana jest ta da ne možemo preurediti poredak operacija. Ukoliko imamo jednu operaciju prije

druge u jednoj dretvi, takav isti poredak će biti vidljiv svim dretvama.

U sekvencijalno-konzistentnom uređaju, sekvencijalno-konzistentna *store* operacija se sinkronizira s sekvencijalno-konzistentnom *load* operacijom iste varijable koja čita tu zapisanu vrijednost. Štoviše, bilo koja sekvencijalno-konzistentna operacija nakon te *load* operacije mora se pojaviti i nakon *store* operacije. Takav jedinstven poredak operacija je, kao što smo prethodno naveli, vidljiv svim dretvama programa.

Napomenimo samo jednu veoma bitnu stvar: čim u igru uđu operacije koje **nisu označene s** `memory_order_seq_cst`, sva garancija sekvencijalne konzistentnosti je **izgubljena**. Drugim riječima, jednom kada se izvrši atomska operacija koja nije označena s `memory_order_seq_cst`, poredak operacija ne mora više biti jedinstven te dretve mogu vidjeti različit red modifikacije nekih objekata.

Sekvencijalna konzistentnost se koristi kada imamo veći broj dretvi koje sve moraju vidjeti neki niz operacija u istom poretku. Kao što smo ranije napomenuli, na nekim arhitekturama procesora, nepotrebno korištenje sekvencijalno konzistentnih operacija može postati suviše skupo te se treba izbjeći. Ipak, današnji Intelovi i AMD-ovi procesori nude sekvencijalnu konzistentnost relativno jeftino.

Demonstraciju sekvencijalne konzistentnosti možemo vidjeti u sljedećem primjeru. Sva dohvaćanja i pisanja su označena sa `std::memory_order_seq_cst`.

Listing 1.9: Primjer utjecaja sekvencijalne konzistentnosti

```
#include <thread>
#include <atomic>
#include <cassert>

std::atomic<bool> x = {false};
std::atomic<bool> y = {false};
std::atomic<int> z = {0};
void write_x() {
    x.store(true, std::memory_order_seq_cst);    // (1)
}
void write_y() {
    y.store(true, std::memory_order_seq_cst);    // (2)
}
void read_x_then_y() {
    while (!x.load(std::memory_order_seq_cst)); // (3)
    if (y.load(std::memory_order_seq_cst))       // (4)
        ++z;
}
void read_y_then_x() {
    while (!y.load(std::memory_order_seq_cst));
    if (x.load(std::memory_order_seq_cst))       // (5)
        ++z;
}
int main() {
```

```
    std::thread a(write_x);
    std::thread b(write_y);
    std::thread c(read_x_then_y);
    std::thread d(read_y_then_x);
    a.join(); b.join(); c.join(); d.join();
    assert(z.load() != 0);                    // (6) will never happen
}
```

Dakle, imamo 4 dretve, dvije koje zapisuju vrijednosti te dvije koje ih čitaju. `assert` u `(6)` neće nikad baciti iznimku jer se zapisavanja u `(1)` ili `(2)` moraju prvo dogoditi, premda nije specificirano u kojem poretku. Jer je sekvencijalno dohvaćanje sinkronizirano sa sekvencijalnim pisanjem, *while* petlje će se izvršavati sve dok dretve pisači ne zapišu vrijednosti. Ukoliko `load` u `(4)` vrati *false*, tada će `load` u `(5)` sigurno vratiti *true* jer postoji jedinstven poredak svih atomskih operacija među svim dretvama, a u `(3)` je osigurano da `load` varijable `x` vraća *true*. Tada će `z.load()` u `(6)` vratiti 1. Analogno će se dogoditi ako `load` u `(5)` vrati *false* zbog simetričnosti koda. Također, moguće je da `z.load()` vrati 2, ali nikada 0.

1.4.2 Relaksirani uređaj

Relaksirani memorijski uređaj, odnosno `std::memory_order_relaxed`, predstavlja suprotnost sekvencijalno-konzistentnom uređaju, tj. `memory_order_seq_cst`. Za razliku od njega, čije postizanje može biti veoma skupo po procesorskom vremenu, relaksirani uređaj predstavlja najjednostavniji i najjeftiniji memorijski uređaj što se tiče uštede procesorskog vremena. Operacije nad atomskim tipovima označene s `std::memory_order_relaxed` ne sudjeluju u *sinkroniziran-s* relacijama. Dakle, to ne mogu biti sinkronizacijske operacije. Ipak, operacije unutar jedne dretve sudjeluju u relaciji *dogodio-se-prije*. U slučaju više dretvi, ne postoje nikakvi preduvjeti na poredak atomskih operacija koji preostale dretve vide, osim toga da kada dretva vidi neku vrijednost atomske varijable, kasnijim čitanjem može vidjeti jedino tu istu vrijednost ili vrijednost zapisanu nekim kasnijim pisanjem. Jedina garancija koju imamo jest nedjeljivost atomskih operacija. Za demonstraciju, pogledajmo sljedeći primjer sa stranice [4].

```
// x i y su std::atomic<int> objekti inicijalizirani na 0
// Thread 1:
r1 = y.load(std::memory_order_relaxed); // A
x.store(r1, std::memory_order_relaxed); // B
// Thread 2:
r2 = x.load(std::memory_order_relaxed); // C
y.store(42, std::memory_order_relaxed); // D
```

Ovaj primjer može producirati vrijednost `r1 == r2 == 42`. Iako se operacija A dogodila prije B te se C dogodila prije D, ništa ne sprječava da se D nalazi prije A u redu modifikacije

od `y` i B prije C u redu modifikacije od `x`. Tada bi zapisana vrijednost 42 u D bila vidljiva u A te bi spremanje vrijednosti `r1 = 42` bilo vidljivo u C.

Ako u primjeru 1.9 sva pojavljivanja uređaja `std::memory_order_seq_cst` zamijenimo s `std::memory_order_relaxed`, tada `assert` u `(6)` može baciti iznimku. Naime, kako svaka dretva može vidjeti različit poredak operacija, ništa ne sprječava da dretva `c` vidi operaciju `(1)` prije `(2)`, a dretva `d` vidi `(2)` prije `(1)`. Tada se u oba slučaja inkrement neće dogoditi te će ostati `z == 0`.

Za razliku od sekvencijalno-konzistentnog uređaja, koji uvijek producira isti poredak atomskih operacija među dretvama, relaksirani uređaj, ali i svi preostali memorijski uređaji, ne moraju producirati isti poredak operacija. Većim brojem izvođenja istog programa, možemo dobiti različite rezultate. Uzrok tome su optimizacije koje provodi prevodilac kako bi se operacije izvršile što brže i efikasnije. Različite priručne memorije procesora i interni *buffer-i* mogu držati različite vrijednosti za istu memoriju. Kao posljedica, i ovo je vrlo važno za shvatiti, dretve se ne moraju složiti oko poretka događaja. Jedini zahtjev kojeg se moraju držati jest da se sve dretve slože oko reda modifikacije svake dretve. Dakle, globalno gledajući dretve mogu vidjeti različit poredak operacija nad skupom svih varijabli, ali moraju vidjeti isti poredak promjena svakog pojedinačnog objekta.

Za bolje razumijevanje načina rada relaksiranog uređaja, zamislimo da svaka dretva višedretvenog programa ima svoj lokalni popis operacija koje su se u njoj dogodile i koji je samo njoj vidljiv. Također, postoji globalan popis stanja svih zajedničkih varijabli koji je vidljiv svim dretvama. Svaka operacija ima oznaku kada se dogodila. Ukoliko je atomska operacija bila označena s `memory_order_seq_cst`, dretva će obaviti operaciju i nakon toga objaviti lokalni popis promjena dane atomske varijable, ali i svih ostalih ne-atomskih zajedničkih varijabli. Taj popis će se nadodati u globalan popis i bit će vidljiv svim ostalim dretvama. Ukoliko, pak, je atomska operacija označena s `std::memory_order_relaxed`, njom uzrokovana promjena će se zapisati u lokalni popis, ali se neće objaviti i stoga neće biti vidljiva preostalim dretvama. Kada neka dretva pokuša dohvatiti vrijednost atomske varijable, ona će dohvatiti vrijednost s globalnog popisa ili svog lokalnog, ovisno koja vrijednost je kasnije zapisana. Objavljivanje ovih popisa nije trivijalno i zauzima veći dio procesorskog vremena pa ga prevodilac pokušava izbjeći. Zbog tog razloga su sekvencijalno-konzistentne operacije skuplje za izvesti od preostalih. Ipak, nakon nekog vremena, prevodilac će objaviti lokalan popis dretve, iako to nije bilo uzrokovano nekom atomskom operacijom. Kada će se to dogoditi, nama ipak ne može biti poznato. Iz tog razloga mogu postojati različiti rezultati istog programa koji koristi operacije koje nisu označene uređajem `std::memory_order_seq_cst`. Prevodilac interno izvršava objave lokalnih popisa u ovisnosti o procesorskom vremenu koje ima na raspolaganju.

Ovaj primjer naravno nije doslovna reprezentacija stvarnog rada, ali predstavlja približan princip rada s atomskim varijablama. Još jedan primjer koji demonstrira način rada relaksiranog uređaja možemo pogledati u [7].

1.4.3 *Acquire-release* uređaji

Acquire-release memorijski uređaji predstavljaju kompromis između relaksiranog i sekvencijalno-konzistentnog uređaja. Ne postoji totalan uređaj nad redoslijedom operacija, ali omogućena je sinkronizacija. Pod *acquire-release* uređaje spadaju `std::memory_order_acquire` za atomske *load* operacije, `std::memory_order_release` za *store* operacije i `std::memory_order_acq_rel` koji predstavlja kombinaciju prethodna dva uređaja te se koristi za *read-modify-write* operacije, premda je kod njih dopušten bilo koji od postojećih uređaja. Sinkronizacija se postiže između dretve koja radi *release* i dretve koja radi *acquire*, naravno nad istom atomskom varijablom. Te dvije operacije su stoga u relaciji *sinkroniziran-s* koju smo opisali na početku ovog poglavlja. Različite dretve i dalje mogu vidjeti različit poredak operacija, ali je on ipak restriktivniji nego u slučaju relaksiranog uređaja.

Ukoliko je atomska *store* operacija u dretvi A bila označena s `std::memory_order_release`, a korespondirajuća *load* operacija nad istom varijablom u dretvi B s `std::memory_order_acquire`, sve modifikacije objekata koje su se u dretvi A dogodile prije *store* operacije, bile te modifikacije nad ne-atomskim objektima ili nad atomskim objektima s relaksiranim poretkom, biti će vidljive dretvi B nakon što obavi *load* operaciju. Uočimo da će operacije *store* i *load* biti sinkronizirane jedino ako *load* dohvati vrijednost zapisanu *store* operacijom. Ukoliko pročita neku drugu vrijednost, do sinkronizacije neće doći i promjene koje su bila napravljene u dretvi A ne moraju biti vidljive dretvi B. Ipak, ispravnom kombinacijom atomskih operacija s *acquire-release* uređajima, programer može sinkronizirati dvije dretve i operacije među njima. Situaciju gdje jedna dretva mijenja objekt, a druga čita tu promjenu možemo vidjeti u sljedećem primjeru.

```
std::atomic<std::string*> ptr;
int data;

void producer() {
    std::string* p  = new std::string("Hello");
    data = 42;
    ptr.store(p, std::memory_order_release);
}

void consumer() {
    std::string* p2;
    while (!(p2 = ptr.load(std::memory_order_acquire)))
        ;
    assert(*p2 == "Hello");      // never fires
    assert(data == 42);          // never fires
}

int main() {
    std::thread t1(producer);
```

```
    std::thread t2(consumer);
    t1.join(); t2.join();
}
```

Dretva `t1` mijenja *string* na koji pokazuje atomski pokazivač `ptr` te koristi `std::memory_order_release` kako bi obavijestila drugu dretvu o toj promjeni. Dretva `t2` u petlji čeka da prethodna promjena bude vidljiva te ju dohvaća koristeći `memory_order_acquire`. Jer se *acquire* operacija sinkronizira s *release* operacijom, pridruživanje u petlji će postati istinito tek nakon što se obavi *release* operacija. Također, jer operacijom označenom s `std::memory_order_release` sve prethodne promjene koja su se dogodile prije te operacije postaju vidljive dretvi koja dohvaća tu vrijednost, dretva `t1` će vidjeti i vrijednost `data == 42`.

Služeći se primjerom iz prethodnog odjeljka s globalnim i lokalnim popisima, operacija označena s `std::memory_order_release` odgovara objavi lokalnog popisa promjena, dok operacija označena s `std::memory_order_acquire` odgovara dohvaćanju tog popisa. Lokalni popis neće biti objavljen direktno u globalni popis, već će biti označen tako da bude vidljiv samo onoj dretvi koja obavi *acquire* operaciju nad istom atomskom varijablom. Preostale dretve ne moraju vidjeti te promjene.

S obzirom da je relacija *sinkroniziran-s* relacija parcijalnog uređaja, ona je **tranzitivna**, što znači da korištenjem *acquire-release* uređaja možemo sinkronizirati veći broj dretvi. Ovo svojstvo nam omogućuje pisanje kompleksnog višedretvenog programa bez korištenja sekvencijalno-konzistentnog uređaja. Ipak, moramo zapamtiti da ne moraju sve dretve vidjeti isti poredak operacija. Korištenje tranzitivnosti među 3 dretve možemo vidjeti u sljedećem primjeru.

```
std::vector<int> data;
std::atomic<int> flag = {0};

void thread_1() {
    data.push_back(42);
    flag.store(1, std::memory_order_release);
}

void thread_2() {
    int expected=1;
    while (!flag.compare_exchange_weak(expected, 2,
                                std::memory_order_acq_rel)) {
        expected = 1;
    }
}

void thread_3() {
    while (flag.load(std::memory_order_acquire) < 2)
        ;
```

```
    assert(data.at(0) == 42); // will never fire
}

int main() {
    std::thread a(thread_1);
    std::thread b(thread_2);
    std::thread c(thread_3);
    a.join(); b.join(); c.join();
}
```

Prva dretva zapisuje vrijednost 42 u vektor te sprema vrijednost 1 u `flag` korištenjem `std::memory_order_release` uređaja. Druga dretva u petlji čeka da zapisana vrijednost postane 1 te ju zatim mijenja u 2. S obzirom da je `compare_exchange_strong()` *read-modify-write* operacija, koristimo `std::memory_order_acq_rel` uređaj kako bi *acquire* operacijom dohvatili vrijednost i *release* operacijom zapisali novu vrijednost. Također, moramo svaki put postaviti `expected = 1` jer `compare_exchange_strong()` u slučaju neuspjeha u `expected` zapisuje vrijednost koju je pročitala, u ovom slučaju to može biti 0. Treća dretva dohvaća u petlji vrijednost zastave *acquire* operacijom i provjerava da nije manja od 2 kako se ne bi dohvatila vrijednost zapisana u prvoj dretvi. Jednom kada je dohvaćena tražena vrijednost, znači da se obavila sinkronizacija s drugom dretvom, a koja je bila sinkronizirana s prvom dretvom, tako da će treća dretva svaki put vidjeti vrijednost 42 u vektoru te `assert` neće nikada baciti iznimku.

Već smo napomenuli da ukoliko koristimo bilo koji memorijski uređaj osim `std::memory_order_seq_cst`, gubimo bilo kakvu nadu o sekvencijalnoj konzistentnosti. Ako pak koristimo sekvencijalno-konzistentni uređaj u kombinaciji s *acquire-release* uređajima, tada se *load* operacije označena s `std::memory_order_seq_cst` ponaša kao *acquire* operacija, *store* se ponaša kao *release*, a *read-modify-write* operacije kao njihova kombinacija.

1.4.4 *Release-consume* uređaj

U uvodu u ovo poglavlje naveli smo da operacije označene s `memory_order_consume` ne sudjeluju u *dogodio-se-prije* relacijama te da standard C++17 eksplicitno preporuča da se ne koristi. Ipak, radi cjelovitosti opisujemo njegovo korištenje i djelovanje na memoriju.

`memory_order_consume` je specijalni slučaj uređaja `memory_order_acquire` koji ograničava sinkronizaciju među dretvama te se kao takav koristi u *load* operacijama te korespondira sa *store* operacijom označenom s `std::memory_order_release`. Opišimo ga formalnije. Za operacije A i B, takve da se A dogodila prije B, kažemo da B podatkovno ovisi o A ako B ovisi o rezultatu operacije A. Ova relacija je očito tranzitivna. Ako je događaj A spremanje nekog rezultata, a B čitanje tog rezultata, očito B podatkovno ovisi o A. Neka su A i B operacije u različitim dretvama te neka je A *store* operacija označena s `std::memory_order_release`, `std::memory_order_acq_rel` ili `std::memory_order`

_seq_cst, a B *load* operacija označena s std::memory_order_consume koja čita rezultat operacije A. Tada se operacija B sinkronizira samo s operacijom A te svim operacijama o kojima A podatkovno ovisi. Ukoliko se neka operacija dogodila u istoj dretvi prije operacije A, a A ne ovisi podatkovno o njoj, tada nema garancije da će operacijom B rezultat te operacije biti vidljiv.

Tu leži razlika s uređajem std::memory_order_acquire. Njegovim korištenjem osiguravamo da će svi događaji koji su se dogodili u toj dretvi prije operacije A biti vidljivi nakon dohvaćanja u B. std::memory_order_consume je bio zamišljen da se koristi na pokazivačima na objekte, kada je dovoljno znati samo stanje tog objekta.

Listing 1.10: Primjer korištenja std::memory_order_consume

```
struct Foo{
    int x,y;
}
std::atomic<Foo*> ptr;
int data;

void producer() {
    Foo* p  = new Foo;
    Foo.x = 57;
    data = 42;                                          // (1)
    ptr.store(p, std::memory_order_release);            // (2)
}

void consumer() {
    Foo* p2;
    while (!(p2 = ptr.load(std::memory_order_consume))) //(3)
        ;
    assert(Foo->x == 57); // (4) never fires: *p2 carries
                          //      dependency from ptr
    assert(data == 42);   // (5) may or may not fire: data does
}                         //      not carry dependency from ptr

int main() {
    std::thread t1(producer);
    std::thread t2(consumer);
    t1.join(); t2.join();
}
```

Očito je spremanje u (1) u relaciji *dogodio-se-prije* s spremanjem u (2), koje je označeno s std::memory_order_release. S obzirom da je dohvaćanje u (3) označeno s std::memory_order_consume i izvršava se u petlji, ono će se dogoditi nakon otpuštanja u (2). Dretva t2 će dakle sigurno vidjeti operacije o kojima (2) podatkovno ovisi, u ovom slučaju to je samo spremanje Foo.x = 57. Zato assert u (4) neće nikada baciti iznimknu, ali u (5) može. Da smo koristili *acquire* operaciju umjesto *consume*, imali bi garanciju da

ni (5) neće baciti iznimku. S obzirom da današnji procesori nemaju razlike u performansama operacija *acquire* i *consume*, korištenje uređaja `std::memory_order_consume` nema previše smisla i zato se u standardu C++17 preporuča ne koristiti ga.

Napomenimo još da postoji i opcija "ubijanja" podatkovne ovisnosti korištenjem funkcije `std::kill_dependency()`. Na primjer, ako imamo globalno *read-only* polje i koristimo `std::memory_order_consume` pri dohvatu indexa u to polje, korištenjem `std::kill_dependency()` javljamo prevodiocu da ne mora pročitati sadržaj polja kako bi ostvario sinkronizaciju. S obzirom da se koristi skupa `std::memory_order_consume`, uporaba `std::kill_dependency()` je nepotrebna.

1.4.5 Ograde

Ograde su operacije koje ne modificiraju memoriju, već uređuju memorijski poredak operacija na temelju zadanog memorijskog uređaja. One su globalne operacije koje utječu na poredak ostalih atomskih operacija unutar dretve u kojoj su pozvane. Često ih se zove i memorijske barijere (en. *memory barriers*). Uglavnom ih se koristi za uređivanje poretka neatomskih i atomskih operacija označenih s `std::memory::order::relaxed`. Uvode se relacije *dogodio-se-prije* i *sinkroniziran-s* između operacija gdje ih prije nije bilo. Operacija koja predstavlja ogradu definirana je kao

```
extern "C" void atomic_thread_fence( std::memory_order order ) noexcept;
```

Postoje 3 tipa ograda:

- Puna ograda - predstavlja `atomic_thread_fence()` s argumentima `std::memory_order_acq_rel` ili `std::memory_order_seq_cst` i zabranjuje preuređivanje redoslijeda između bilo koje dvije operacije osim kada su to *store* - *load*, tj. kada *load* slijedi nakon *store* operacije.
- *Acquire* ograda - `std::atomic_thread_fence(std::memory_order_acquire)`. Ne dopušta da se poredak *read* operacije koja se dogodila prije ograde preuredi s nekom atomskom operacijom koja se dogodila nakon ograde.
- *Release* ograda - `std::atomic_thread_fence(std::memory_order_release)`. Ne dopušta da se poredak *write* operacije koja se dogodila nakon ograde preuredi s nekom atomskom operacijom koja se dogodila prije ograde.

U idućem primjeru ilustriramo kako se korištenjem atomskih ograda može osigurati jedinstven poredak relaksiranih atomskih operacija kao i neatomskih operacija.

Listing 1.11: Uređivanje poretka operacija korištenjem ograda

```
#include <atomic>
#include <thread>
```

```
#include <assert.h>
std::atomic<bool> x,y;
std::atomic<int> z;

void write_x_then_y() {
    x.store(true, std::memory_order_relaxed);                    // (1)
    std::atomic_thread_fence(std::memory_order_release);         // (2)
    y.store(true, std::memory_order_relaxed);                    // (3)
}
void read_y_then_x() {
    while(!y.load(std::memory_order_relaxed))                    // (4)
        ;
    std::atomic_thread_fence(std::memory_order_acquire);         // (5)
    if(x.load(std::memory_order_relaxed))                        // (6)
        ++z;
}

int main() {
    x = false;
    y = false;
    z = 0;
    std::thread t1(write_x_then_y);
    std::thread t2(read_y_then_x);
    t1.join();
    t2.join();
    assert(z.load() != 0);                    // Will never fire (7)
}
```

U primjeru imamo dvije dretve, `t1` i `t2`. *Release* ograda `(2)` u dretvi `t1` se sinkronizira s *acquire* ogradom `(5)` u dretvi `t2`. Zbog toga operacija spremanja `(1)` će biti sinkronizirana s `(6)` te će se izvršiti `++z;` i `assert` u `(7)` neće nikada baciti iznimku. Iako ćemo, zbog dohvaćanja u *while* petlji, dohvaćanjem vrijednosti od `y` u `(4)` vidjeti vrijednost zapisanu u `(3)`, u odsustvu ograda ne bi postojala garancija da će *assert* u `(7)` proći jer, zbog relaksiranog uređaja, nema garancije da će dretva `t2` vidjeti isti poredak operacija kakav je bio u dretvi `t1`.

U prethodnom primjeru, isti efekt bi se postigao da smo umjesto korištenja *acquire-release* ograda koristili `std::memory_order_release` u `(3)` i `std::memory_order_acquire` u `(4)`. Ograde koristimo kada želimo sinkronizirati operacije, ali nam nije potrebna dodatna vrijednost koju treba pamtiti.

Postoji još jedna vrsta atomske ograde a to je, u punoj definiciji

```
extern "C" void atomic_signal_fence(std::memory_order order) noexcept;
```

Ona, slično kao i `std::atomic_thread_fence`, uspostavlja poredak operacija po uzoru na zadani memorijski uređaj, ali bez dodatnih CPU instrukcije za uspostavljanje memorijskog poretka, tj. ne dopušta preuređivanje poretka instrukcija za vrijeme kompajliranja.

Kao rezultat, ne može uzrokovati međudretvenu sinkronizaciju. Iz tog razloga, rijetko se koristi. Slučajevi kada bi njeno korištenje bilo od koristi jest kada bi imali dvije operacije koje zauzimaju mnogo procesorskog vremena te ne želimo dopustiti prevodilacu da preuredi njihov poredak.

Poglavlje 2

Implementacija *lock-free* struktura podataka

U ovom poglavlju ću opisati implementaciju *lock-free* struktura podataka: stoga i reda. U implementaciji ću se oslanjati prvenstveno na novosti uvede u standardu C++20, posebice na atomski pametni pokazivač `std::atomic<std::shared_ptr<T>>`. Demonstrirat ću kako njegovim korištenjem možemo uvelike olakšati pisanje *lock-free* koda, bez većeg gubitka brzine. Štoviše, u nekim slučajevima je moguće ostvariti i veću brzinu.

Navedenu implementaciju ću usporediti s *lock-free* strukturama podataka baziranim na standardu C++11 koje možemo pronaći u [7] i koje, radi ostvarivanja sigurnosti korištenja u višedretvenom sustavu, koriste brojanje referenci koje je uvelike kompleksnije i zahtjeva veće znanje i uloženo vrijeme od strane programera. Također, usporedit ću i s implementacijama koje koriste lokote (mutex-e) kako bi se osigurala višedretvena sigurnost operacija `push()` i `pop()` nad strukturama `std::stack` i `std::queue`.

S obzirom da većina postojećih prevodioca još ne sadrže podršku za novosti uvede u standardu C++20, za implementaciju i testiranje koristim *Microsoft Visual Studio 2022* i najnoviju verziju njegovog prevodioca s obzirom da trenutno jedini pruža podršku za atomske pametne pokazivače. Napomenimo da ova implementacija pametnih pokazivača još uvijek nije konačna. Naime, naredba `std::atomic<std::shared_ptr<T>>::is_lock_free()` vraća *false* za bilo koji tip *T*, no s vremenom bi se implementacija trebala usavršiti.

Struktura

Obje strukture podataka koje ću implementirati, stog i red, imaju istu unutarnju strukturu. Implementirane su kao vezana lista, što nam omogućuje da se obje klase sastoje samo od pokazivača na početni element u slučaju stoga, odnosno na početni i zadnji element u slučaju reda. Jedine metode koje korisnik može pozvati su:

- `void push(T const& data)` - dodaje novu vrijednost u vezanu listu
- `std::shared_ptr<T> pop()` - izbacuje vrijednost iz liste i vraća pokazivač na nju

Elemente prikazujem kao čvorove tipa `node` koji sadrže pokazivač na zadanu vrijednost te atomski pokazivač na sljedeći čvor.

Listing 2.1: Definicija čvora vezane liste

```
struct node
{
    std::shared_ptr<T> data;
    std::atomic<std::shared_ptr<node>> next;
    node(T const& data_) :
        data(std::make_shared<T>(data_)) {}
};
```

S obzirom da čvorove želimo konstruirati samo kada imamo vrijednost koju treba ubaciti u vezanu listu, jedini potreban konstruktor prima konstantnu vrijednost elementa te sprema pokazivač na nju. Razlog zašto spremamo pokazivač, a ne samu vrijednost jest sigurnost i jednostavnost implementacije. Ukoliko spremamo vrijednost kao objekt u čvoru, prilikom brisanja čvora i vraćanja vrijednosti morali bismo vratiti kopiju te vrijednosti. Ako se pak prilikom kopiranja vrijednosti dogodi iznimka, vrijednost će biti izgubljena. S druge strane, držanje pokazivača na vrijednost omogućava nam da jednostavno vratimo `nullptr` ukoliko je lista prazna te na taj način pozivom metode `pop()` znamo je li struktura podataka prazna, bez potrebe za bacanjem iznimke.

Uz pokazivač na pripadnu vrijednost, čvor mora sadržavati i pokazivač na sljedeći čvor. Primijetimo kako deklariranjem tog pokazivača kao atomskog, ostvarujemo sigurnost od gubljenja elementa prilikom zamjene sljedećim. Na primjer, u stogu, kod implementacije `pop()` funkcije, moramo zamijeniti pokazivač na početni element pokazivačem na njegovog sljedbenika. Kada pokazivač `next` ne bi bio atomski te više dretvi istovremeno pozove `pop()`, riskiramo nedefinirano ponašanje jer više dretvi može istovremeno pokušat dohvatiti isti pokazivač. U međuvremenu, jedna od dretvi izbriše taj element što rezultira nedefiniranim ponašanjem jer dretva želi dohvatiti nepostojeći objekt.

Sad kad smo opisali detalje koji su zajednički objema strukturama, slijede opisi metoda `push()` i `pop()` za stog, odnosno red, te usporedba rezultata ove implementacije s onom iz [7] te implementacijom koja koristi lokote (*mutexe*).

2.1 Stog

Za razliku od reda, stog koristi samo jedan pokazivač kako bi znao početak stoga. Njega deklariramo kao:

```
std::atomic<std::shared_ptr<node>> head;
```

Sve ubacivanja i izbacivanja elemenata se vrše na početku vezane liste. Implementacija funkcije `push()` je jednostavna i vrlo slična onoj iz [7] gdje nemamo pametne atomske pokazivače.

Listing 2.2: Implementacija `push()` funkcije

```
void push(T const& data)
{
    std::shared_ptr<node> new_node = std::make_shared<node>(data);
    new_node->next = head.load(std::memory_order_relaxed);
    std::shared_ptr<node> new_node_ptr =
                    new_node->next.load(std::memory_order_relaxed);
    while (!head.compare_exchange_weak(new_node_ptr,
        new_node,
        std::memory_order_release,
        std::memory_order_relaxed)) {
        new_node->next.store(new_node_ptr, std::memory_order_relaxed);
    }
}
```

Kreiramo novi čvor koji sadrži pokazivač na zadani element. Vrijednost koja se trenutno nalazi u `head` želimo postaviti u `next` pokazivač novog elementa te ažurirati `head`. Zato prvo moramo postaviti `new_node->next` na `head`. S obzirom da se ova naredba nalazi u nizu prije *while* petlje u kojoj postavljamo `head`, dovoljno je koristiti `std::memory_order_relaxed`. Sada nailazimo na problem kod ovakve definicije samog čvora. Kako je `next` atomski tip, a u `head.compare_exchange_weak()` prvi element se prima po referenci i mi tu očekujemo upravo `new_node->next`, moramo deklarirati novu varijablu koja će držati pokazivač koji se trenutno nalazi u `new_node->next` i nju predati kao prvi argument.

Iz razloga navedenih u poglavlju 1.2, funkciji ne možemo predati referencu na atomski objekt. Kako se u slučaju neuspjeha prvi argument, tj. očekivana vrijednost, ažurira novom vrijednošću, moramo svaki put ponovo postaviti `new_node->next`. Ovo ozbiljno utječe na brzinu izvođenja `push()` funkcije, što se može vidjeti na slici 2.1.

Što se tiče potrebnih memorijskih uređaja u *compare-exchange* funkciji, znamo da `push()` u jednoj dretvi mora biti sinkroniziran s `pop()` u drugoj dretvi. Kako bi postigli sinkroniziranost, dovoljno je koristiti `std::memory_order_release` kako bi dretva koja radi `pop()` vidjela ovu promjenu. Ukoliko *compare-exchange* ne uspije, ništa se ne mijenja pa je dovoljan `std::memory_order_relaxed`. Također, napomenimo da uvijek koristimo *weak* verziju *compare-exchange* funkcije jer se ona poziva u petlji te će u slučaju spurioznog neuspjeha koji je rijedak, vrijednost biti postavljena u idućoj iteraciji.

Pokažimo sada `pop()` funkciju. Ukoliko prvo pogledamo implementaciju *lock-free* stoga Anthnoy Williamsa u [7], vidimo da je ona uvelike kompliciranija od `push()` funkcije. Naime, u slučaju kada nemamo pametne atomske pokazivače, moramo se sami po-

brinuti o brisanju čvorova kako bi izbjegli curenje memorije. To se može raditi brojanjem referenci na pojedini čvor, te brisanjem čvora u `pop()` funkciji jednom kada je čvor izbačen iz liste i nema drugih referenci na njega. Takav pristup je korektan, ali ipak puno kompliciraniji od onog koji nam omogućava korištenje atomskih pokazivača.

Listing 2.3: Implementacija `pop()` funkcije

```
std::shared_ptr<T> pop()
{
    std::shared_ptr<node> old_head = head.load(
                                          std::memory_order_relaxed);
    while (old_head &&
           !head.compare_exchange_weak(old_head,
               old_head->next.load(std::memory_order_relaxed)));
    if (old_head) {
        old_head->next.store(nullptr,
                             std::memory_order_relaxed);
        return old_head->data;
    }
    return nullptr;
}
```

Prvo dohvaćamo trenutnu vrijednost `head` pokazivača koju želimo izbaciti. Dovoljan nam je `std::memory_order_relaxed`. Zatim vrtimo *while* petlju sve dok `old_head` nije `nullptr` (znači da stog nije prazan) i `head.compare_exchange_weak()` ne zamijeni vrijednost pokazivača i vrati *true*. Uočimo da sada imamo suprotan slučaj od onog u `push()` funkciji. Očekujemo vrijednost od `old_head` i želimo je zamijeniti vrijednošću `next` pokazivača. Kako se drugi parametar na prima po referenci, već po vrijednosti, nema potrebe za deklariranjem nove varijable kao u `push()`. Kao memorijski uređaj nam je potreban `std::memory_order_acq_rel` jer se želimo dohvatiti promjene koje je napravila druga dretva pozivanjem `push()` ili `pop()` te se isto tako sinkronizirati s drugom dretvom koja može izvršiti `pop()`. Kako se u ovoj situaciji `std::memory_order_seq_cst` ponaša isto kao `std::memory_order_acq_rel`, nema potrebe za eksplicitnim zadavanjem memorijskog uređaja.

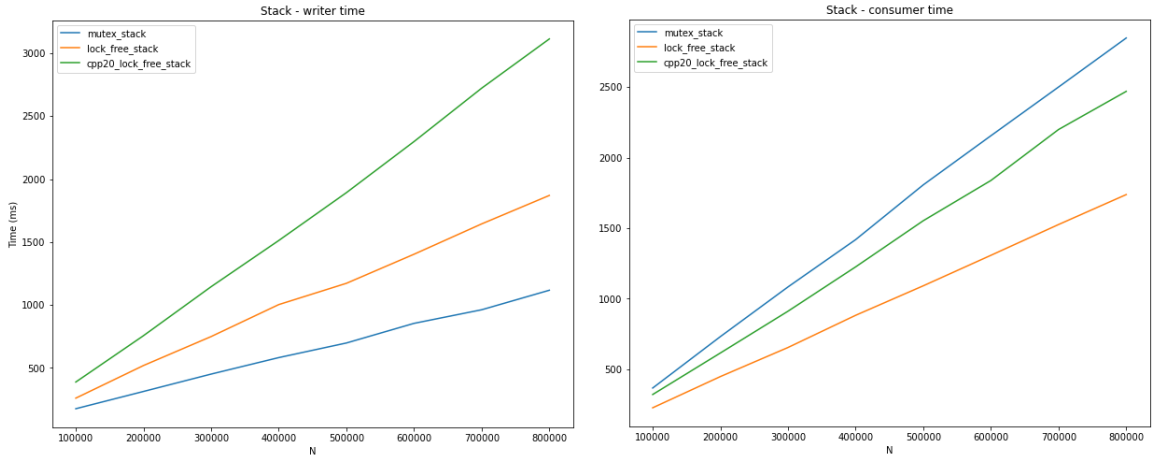
Konačno, ukoliko smo izašli iz *while* petlje i `old_head` nije `NULL`, znači da smo uspješno zamijenili vrijednost idućom. Tada postavljamo vrijednost `next` pokazivača na `NULL` kako bi čvor postao spreman za destrukciju te vraćamo pokazivač na izbačenu vrijednost. U suprotnom, znači da je stog prazan i nismo izbacili element pa vraćamo `nullptr`.

Upravo u `pop()` funkciji se vidi velika prednost korištenja pametnih atomskih pokazivača uvedenih u standardu C++20: jednostavnost. Gdje smo prethodno morali brojiti reference na svaki čvor u listi, sada je dovoljno da postavimmo `next` na `nullptr`, čime čvor postao neovisan te će se prevodilac sam pobrinuti o njegovom uništenju jednom kada `pop()` završi.

Testiranje

Sav kod je pisan i testiran korištenjem alata Microsoft Visual Studio 2022. Svi testovi su rađeni na istom računalu pod istim uvjetima. Računalo koristi procesor Intel Core i7-6700HQ s radnim taktom od 2.60 GHz. Kao što smo već napomenuli, testirat ćemo 3 strukture podataka. One su:

- Stog koji za sinkronizaciju koristi lokote. U legendi, njegov naziv je `mutex_stack` te je obojen plavom bojom.
- *Lock-free* stog Anthony Williams-a baziran na brojenju referenci. U legendi, njegov naziv je `lock_free_stack` te je obojen narančastom bojom.
- *Lock-free* stog koji implementiramo i koji koristi novosti standarda C++20. U legendi, njegov naziv je `cpp20_lock_free_stack` te je obojen zelenom bojom.



Slika 2.1: Usporedba rezultata testiranja stoga.

Testiramo ćemo na sljedeći način. Imamo 3 dretve koje će paralelno ubacivati svaka po N elemenata u stog koristeći `push()`. Svi elementi bit će tipa `double`. Prije nego dretve započnu, započinje mjerenje vremena. Jednom kada svaka dretva ubaci N elemenata, zaustavljamo vrijeme. Dakle, nakon uspješnog ubacivanja elemenata, stog treba sadržavati $3N$ elemenata. Ukoliko stog nije siguran za višedretveno korištenje, doći će do gubljenja elemenata te će njihov ukupan broj biti manji od $3N$.

Jednom kada dretve završe, ispisujemo rezultat mjerenja u milisekundama te na isti način testiramo brzinu `pop()` funkcije: deklariramo 3 dretve koje paralelno zovu `pop()` sve dok stog nije prazan. Svaka dretva pamti broj elemenata koji je izbacila, odnosno koliko puta je uspješno zvala `pop()`. Ukupan broj izbačenih elemenata mora biti $3N$.

Ovakav test provodimo za $N = 100000, 200000, \ldots, 800000$. Rezultate možemo vidjeti na slici 2.1. Lijeva slika prikazuje vrijeme potrebno za ubacivanje $3N$ elemenata, a desna za njihovo izbacivanje. x os označava N, dok y os označava vrijeme ubacivanja odnosno izbacivanja $3N$ elemenata. Možemo vidjeti koliko na efikasnost `push()` funkcije utječe problem koji smo prije naveli. Stog koji koristi lokote iznenađujuće brzo ubacuje elemente, ali ih zato jako sporo izbacuje. S obzirom da u `pop()` funkciji nemamo problem kao u `push()`, rezultati su nešto bliži onima verzije koja koristi brojanje referenci.

2.2 Red

Implementacija reda nešto je kompliciranija od stoga. Više nemamo samo jedan pokazivač koji moramo pazit, već dva. `push()` funkcija djeluje na kraju reda, tj. repu (`tail`), dok `pop()` djeluje na početku, tj. glavi (`head`). Osnovna struktura je ipak slična onoj u slučaju stoga. Čvorovi su prikazani strukturom `node` opisanoj u listingu 2.1, dok su pokazivači deklarirani kao:

```
std::atomic<std::shared_ptr<node>> head;
std::atomic<std::shared_ptr<node>> tail;
```

Ukoliko pogledamo implementaciju Anthony Williamsa u [7], možemo vidjeti da koristi 3 strukture samo za prikaz čvorova u vezanoj listi. Također, koristi dvostruko brojanje referenci, unutarnje i vanjsko, te još nekoliko funkcija koje se brinu za povećanje odnosno smanjivanje tih brojača. Implementacija koja koristi pametne atomske pokazivače je ipak puno jednostavnija. Funkcija `push()` je implementirana na sljedeći način:

Listing 2.4: Implementacija `push()` funkcije

```
void push(T const& data)
{
    std::shared_ptr<node> new_node = std::make_shared<node>(data);
    new_node->next.store(nullptr, std::memory_order_relaxed);
    std::shared_ptr<node> old_tail = tail.load(
                                          std::memory_order_relaxed);
    while (1) {
        if (tail.compare_exchange_weak(old_tail, new_node)) {
            if (old_tail)
                old_tail->next.store(new_node,
                                     std::memory_order_relaxed);
            else
                head.store(tail.load(std::memory_order_relaxed),
                           std::memory_order_release);
            break;
        }
    }
}
```

Implementacija je slična onoj u slučaju stoga. Stvaramo novi pokazivač na čvor sa zadanom vrijednošću. Kako idući element novog čvora ne postoji, postavljamo `new_node->next = nullptr`. Iduće dohvaćamo trenutnu vrijednost od `tail` te ulazimo u petlju. Koristili smo `std::memory_order_relaxed` jer ove naredbe prethode petlji u kojoj se zove *compare-exchange* funkcija. U petlji pokušavamo zamijeniti vrijednost pokazivača `tail`. Očekivana vrijednost je `old_tail` koju smo prethodno dohvatili. Ovdje nema potrebe za deklariranjem nove varijable koja će držati pokazivač kao u slučaju stoga jer ne očekujemo vrijednost `next` pokazivača. Samim time, nema gubitka efikasnosti.

Petlja se vrti sve dok `compare_exchange_weak()` ne uspije. Tada moramo provjeriti koja je bila vrijednost starog repa. Ukoliko je ona bila `null`, znači da je red bio prazan pa moramo namjestiti da `head` i `tail` pokazuju na isti element. U suprotnom, red nije bio prazan pa je dovoljno namjestiti pokazivač starog repa na novo-ubačeni čvor.

Primjetimo da smo u `compare_exchange_weak()` koristili zadani memorijski uređaj, `std::memory_order_seq_cst`. Rekli smo u poglavlju 1.4.1 da sekvencijalna konzistentnost pada u vodu kada koristimo ostale memorijske uređaje. U ovom slučaju, zadani memorijski uređaj zapravo postaje `std::memory_order_acq_rel` koji zapravo želimo. S obzirom da `push()` mijenja rep reda, ona se sinkronizira s drugom `push()` funkcijom. Stoga želimo da *read* operacija koristi memorijski uređaj `std::memory_order_acquire`, a *store* operacija `std::memory_order_release`. Jedini slučaj kada `push()` sinkronizira s `pop()` jest kada je red bio prazan i zato koristimo `std::memory_order_release` pri zapisivanju nove vrijednosti glave.

Pogledajmo sada `pop()` funkciju.

Listing 2.5: Implementacija `pop()` funkcije

```
std::shared_ptr<T> pop()
{
    std::shared_ptr<node> old_head =
                        head.load(std::memory_order_relaxed);
    while (old_head && !head.compare_exchange_weak(old_head,
                        old_head->next.load(std::memory_order_relaxed)));
    if (old_head) {
        old_head->next.store(nullptr, std::memory_order_relaxed);
        return old_head->data;
    }
    tail.store(nullptr, std::memory_order_release);
    return nullptr;
}
```

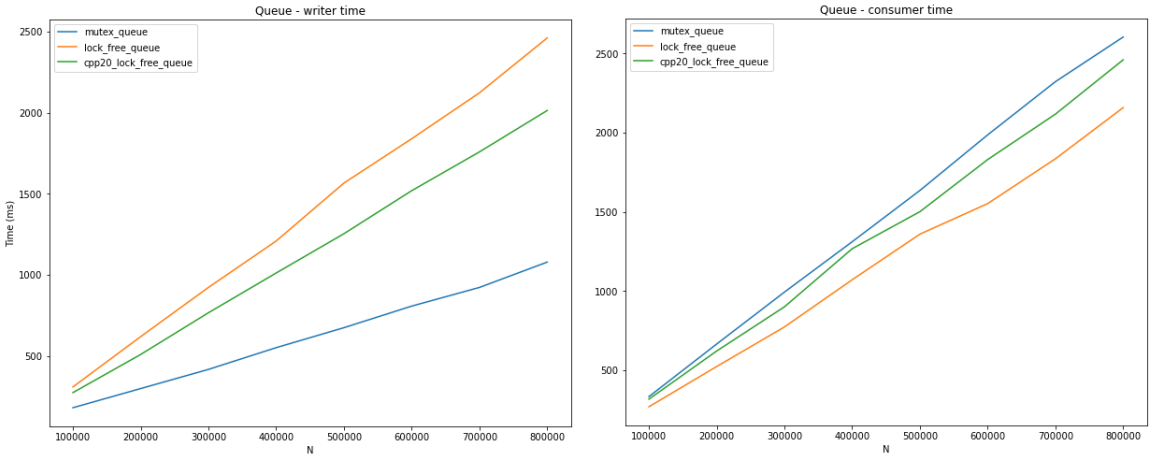
Implementacija je veoma slična onoj u slučaju stoga. Jedina razlike su što u slučaju da nismo uspjeli izbaciti element, dakle kad je red prazan, moramo postaviti `tail` na `null`. Inače bi `tail` i dalje držao pokazivač na čvor koji treba biti izbrisan. Također, iz istog razloga kao u `push()` funkciji, u *compare-exchange* operaciji koristimo zadani memorijski uređaj te pri spremanju novog repa koristimo `std::memory_order_release`.

Testiranje

Testiranje reda provodimo na isti način kao u slučaju stoga. Dakle, testiramo 3 strukture:

- Red koji za sinkronizaciju koristi lokote. U legendi, njegov naziv je `mutex_queue` te je obojen plavom bojom.
- *Lock-free* red Anthony Williams-a baziran na brojenju referenci. U legendi, njegov naziv je `lock_free_queue` te je obojen narančastom bojom.
- *Lock-free* red koji implementiramo i koji koristi novosti standarda C++20. U legendi, njegov naziv je `cpp20_lock_free_queue` te je obojen zelenom bojom.

Rezultati su vidljivi na slici 2.2. Lijevi graf prikazuje rezultate testiranja `push()`, a desni `pop()` funkcije. Ovi rezultati su drugačiji nego u slučaju stoga. Red koji koristi lokote opet



Slika 2.2: Usporedba rezultata testiranja reda.

najbrže izvodi `push()`, a najsporije `pop()` funkciju. Ipak, korištenjem pametnih atomskih pokazivača, dobili smo red koji brže ubacuje elemente i tek malo sporije ih izbacuje, prvenstveno jer Anthony Williams u svojoj implementaciji koristi dretvu koja čeka u petlji da bi pomogao drugoj dretvi koja izvodi `push()`. Ipak, razlika u vremenima postaje osjetna tek nakon što moramo izbaciti više od milijun elemenata iz reda. Dakle, korištenjem atomskih pokazivača, uspjeli smo dobiti sveukupno brži red uz puno manje kompleksan kod koji koristi desetak puta manje linija koda.

Zaključak

Višedretveno programiranje bez zaključavanja korištenjem zaglavlja `<atomic>` predstavlja dobru i korisnu alternativu programiranju korištenjem lokota. Iako često nećemo dobiti na brzini, činjenica da je program *lock-free* može biti od velike koristi u sustavima s velikim brojem korisnika i većom šansom za pojavom potpunog zastoja. Veliki broj programera izbjegava korištenje atomskih tipova upravo zbog znanja potrebnog da bi ih ispravno koristili. Moramo razumjeti potrebu za korištenjem pojedinog memorijskog uređaja te odrediti veze među operacijama kako bi ih ispravno znali postaviti. Uz sve to, veoma bitni tipovi poput pokazivača i referenci do nedavno nisu imali atomske specijalizacije (i još uvijek nemaju usavršenu) pa su programeri morali smišljati nove kompleksne metode poput brojenja referenci kako bi program ostao siguran za višedretveno korištenje. Uvođenjem standarda C++20, koji sa sobom donosi tipove poput `std::atomic_ref` i atomske specijalizacije `std::shared_ptr`, pisanje *lock-free* koda postaje puno jednostavnije. Ukoliko možemo izbjeći dodatne operacije kakve smo imali kod implementacije stoga, možemo dobiti i sveukupno efikasniji program, što smo mogli vidjeti u slučaju reda. Iako implementacija pametnih atomskih pokazivača još nije usavršena pa koristi lokote za sinkronizaciju, jednom kad se usavrši, predstavaljat će ozbiljnu alternativu korištenju lokota u strukturama podataka za asinhroni pristup.

Bibliografija

[1] *Programming Languages — C++, ISO/IEC JTC1 SC22 WG21 N 4860*, Mar 2020, https://isocpp.org/files/papers/N4860.pdf, - web lokacija.

[2] *Atomic library*, https://en.cppreference.com/w/cpp/atomic/atomic, (siječanj 2022.).

[3] *Memory model*, https://en.cppreference.com/w/cpp/language/memory_model, (siječanj 2022.).

[4] *Memory order*, https://en.cppreference.com/w/cpp/atomic/memory_order, (siječanj 2022.).

[5] *Object*, https://en.cppreference.com/w/cpp/language/object, (siječanj 2022.).

[6] Andrei Alexandrescu, Hans Boehm, Kevlin Henney, Doug Lea i Bill Pugh, *Memory model for multithreaded C++*, (2005).

[7] Anthony Williams, *C++ Concurrency in Action, Second Edition*, Manning Publications, 2019.

Sažetak

Pri pisanju višedretvenog programa, programeri najčešće koriste lokote odnosno međusobno isključivanje za zapisivanje i dohvaćanje podataka iz zajedničke memorije. U ovom radu predstavljena je alternativna metoda koja ne koristi lokote, već se oslanja na atomske tipove i operacije nad njima kako bi ostvarila sinkronizaciju među dretvama. Atomske operacije su po definiciji nedjeljive stoga ne mogu prouzročiti nedefinirano ponašanje. Štoviše, omogućuju preuređivanje atomskih i neatomskih operacija s obzirom na zadani memorijski uređaj. Atomski tipovi su uvedeni u standardu C++11 2011. godine. Uz atomske tipove i operacije, tim standardnom predstavljen je i memorijski model jezika C++ koji opisuje ponašanje dretvi s obzirom na memoriju i predstavlja podlogu za pisanje *lock-free* programa. Uvođenjem standarda C++20, dobivena je podrška za mnoge nove atomske tipove koji uvelike olakšavaju korištenje atomskih operacija i pisanje struktura podataka za asinhorni pristup.

Sam rad se sastoji od dva poglavlja. U prvom dajemo opis memorijskog modela i zaglavlja `<atomic>`. Prvo opisujemo standardne atomske tipove uvedene standardnom C++11 i operacije koje oni podržavaju. Dajemo detaljan opis djelovanja svake operacije i primjere kada se koristi. Zatim dajemo opis najbitnijh tipova uvedenih standardom C++20, njihovog korištenja i prednosti koje dovode u odnosu na postojeće tipove. Nakon opisa tipova i operacija, dajemo pregled memorijskih uređaja. Opisujemo kako svaki memorijski uređaj djeluje na poredak pisanja i čitanja iz memorije u odnosu na atomsku operaciju u kojoj je zadan te dajemo primjere situacija u kojima se koriste.

U drugom poglavlju dajemo pregled implementacije dvije *lock-free* strukture podataka: stoga i reda. U njihovoj implementaciji, oslanjamo se na pametne atomske pokazivače uvedene u standardu C++20 te opisujemo kako njihovim korištenjem možemo uvelike olakšati pisanje *lock-free* koda. Dajemo detaljan opis glavnih funkcije svake strukture te konačno uspoređujemo brzinu rada dane strukture s implementacijama koje koriste mutexe i koje se oslanjaju na atomske operacije uvedene u standardu C++11.

Summary

While writing multithreaded programs, programmers usually use locks, that is, mutual exclusion, to read and write data. This thesis presents an alternative method that does not use locks but relies on atomic types and operations to achieve synchronization. Atomic operations are, by definition, inseparable operations, which means they cannot cause data race or undefined behavior. Furthermore, they can establish inter-thread synchronization and order non-atomic memory accesses as specified by the given memory order. Atomic types were introduced in C++11 standard in 2011. Along with atomic types and operations, this standard introduced a memory model to the C++ language which describes the behavior of threads with respect to basic memory operations and serves as a basis for writing lock-free programs. With the recent C++20 standard, support was added for many new atomic features and types which greatly facilitates the usage of atomics while writing lock-free data structures.

This paper is comprised of two chapters. In the first chapter, we give a description of the C++ memory model and `<atomic>` header. First, we describe standard atomic types introduced in 2011. and operations it supports. We give a detailed explanation of the effect of each operation and examples of how to use it. Next, we describe the main novelties introduced to the atomic library in standard C++20, their usage, and the advantages it has over the existing types. After this, we give an overview of the memory orders. We describe how each memory order affects memory accesses, including reads and writes to the memory, and their order around an atomic operation while giving examples of the situations in which they are used.

In the second chapter, we give an overview of the implementation of two lock-free data structures: stack and queue. In their implementation, we rely heavily on smart atomic pointers introduced in the C++20 standard. We describe how by using them we can greatly facilitate the writing of lock-free code. We give a detailed description of the main functions and finally, we compare the operating speed of that data structure with one implementation which uses mutexes and one which relies on atomics from the C++11 standard.

Životopis

Rođen sam 11. kolovoza 1997. u Zadru. Nakon završene Osnovne škole Petra Preradovića u Zadru, 2012. godine upisujem prirodoslovno-matematičku Gimnaziju Jurja Barakovića. Njenim završetkom, 2016. godine godine upisujem preddiplomski studij Matematika na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta u Zagrebu, gdje sam 2019. godine stekao titulu sveučilišnog prvostupnika matematike, *univ. bacc. math.*, nakon čega sam nastavio s diplomskim studijem Računarstva i matematike. Tijekom druge godine diplomskog studija, započinjem s radom kao robotički inženjer pripravnik u kompaniji Gideon Brothers, gdje sam član tima zaduženog za lokalizaciju robota i mapiranje prostora.