# Paralelne strukture podataka bez međusobnog isključivanja

**Kalčićek, Eduard**

**Master's thesis / Diplomski rad**

**2018**

*Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj:* **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

*Permanent link / Trajna poveznica:* https://urn.nsk.hr/urn:nbn:hr:217:234918

**SVEUČILIŠTE U ZAGREBU**

**PRIRODOSLOVNO–MATEMATIČKI FAKULTET**

**MATEMATIČKI ODSJEK**

Eduard Kalčićek

# PARALELNE STRUKTURE PODATAKA BEZ MEđUSOBNOG ISKLJUčIVANJA

Diplomski rad

Voditelj rada:
Prof. dr. sc. Mladen Jurak

Zagreb, lipanj, 2018.

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

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

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

# Sadržaj

# Uvod

U višedretvenom programiranju potrebno je koristiti neki mehanizam zaštite struktura podataka kako bi im se na siguran način moglo pristupati iz više različitih dretvi. U prvom poglavlju ovoga rada opisan je slučaj i potencijalni problemi koji dolaze s neadekvatnim pisanjem programa namijenjenog za izvršavanje pomoću više dretvi. U drugom je poglavlju predložen jedan od načina izbjegavanja navedenog problema uz pomoć mutexa - tradicionalnog mehanizma za konstruiranje kritičnih odsječaka. Mutexi dolaze sa skupom problema zbog čega se u ovom radu proučava druga metoda: konstrukcija *lock-free* paralelnih struktura podataka. *Lock-free* algoritmi ne koriste lokote već uz pomoć tzv. *atomic* operacija, definiranih u C++ 2011 standardu, postižu siguran pristup podacima od strane više dretvi. U trećem se poglavlju proučava podrška višedretvenom programiranju uz uporabu `std::atomic` klase. Ovdje se također govori o redovima modifikacija uz pomoć kojih je moguće postići veću efikasnost pisanih programa. Glavni dio rada predstavlja četvrto poglavlje gdje je uz pomoć *atomic* operacija detaljno opisana paralelna implementacija *lock-free* strukture podataka stog te predstavljena razlika između brzine izvođenja programa uz uporabu različih memorijskih modela. Ondje su opisane i neke negativne karakteristike *lock-free* algoritama.

# Poglavlje 1

# Problemi s dijeljenjem podataka između dretvi

Problemi vezani s dijeljenjem podataka između različitih dretvi mogu nastati kao posljedica modificiranja podataka. Ukoliko se svi podaci kojima dretve pristupaju mogu samo čitati (*read-only*), tada nema takvih problema budući da podaci koje čita jedna dretva nisu efektirani od strane dretve koja čita te iste podatke. U radu se nadalje pretpostavlja da se radi s podacima koji se mogu i čitati i mijenjati (*read-and-write*). Nad podacima koji su dijeljeni od strane više dretvi postoji mogućnost za neprilike ukoliko jedna dretva počne mijenjati podatke, a druga dretva počne čitati iste. Problem s čitanjem podataka od strane jedne dretve *tijekom* modificiranja tih istih podataka od strane druge dretve može se prikazati primjerom:
Pretpostavimo da u programu koristimo dvostruko-vezanu listu u kojoj svaki čvor, osim prvog i zadnjeg, sadrži dva pokazivača - jedan na sljedeći čvor, te jedan na prethodni. Prilikom npr. brisanja čvora iz vezane liste, moguće je izgubiti konzistentnost između podataka ukoliko dretva pristupa podacima liste nakon što smo ažurirali jedan, a još nismo ažurirali drugi odgovarajući pokazivač u listi. Postupak brisanja čvora iz vezane liste

$$\cdots \rightleftarrows A \rightleftarrows B \rightleftarrows C \rightleftarrows \cdots$$

glasi:

1. Pronađi čvor koji je potrebno izbaciti iz liste ($B$).
2. Ažuriraj pokazivač čvora $A$ tako da pokazuje na čvor $C$.
3. Ažuriraj pokazivač čvora $C$ tako da pokazuje na čvor $A$.
4. Izbriši čvor $B$.

Između drugog i trećeg koraka pokazivači u jednom smjeru vezane liste nisu konzistentni s pokazivačima u drugom smjeru. Ukoliko jedna dretva čita podatke vezane liste dok druga obavlja brisanje elemenata, moguće je da dretva koja čita podatke vidi čvor u listi koji je djelomično uklonjen iz liste - budući da se samo jedna veza između čvorova promijenila. Dretva koja čita podatke s lijeva na desno će preskočiti čvor koji se trenutno briše, no dretva koja pokuša pobrisati čvor *C* iz liste može uzrokovati oštećenja podataka što može dovesti do rušenja programa (na sličan problem nailazimo i u slučaju ubacivanja elemenata u vezanu listu).

Ovakve operacija zahtjevaju pristup dvama različitim podacima pa se zbog toga moraju odvijati u odvojenim instrukcijama. To stvara potencijal da neka druga dretva pristupi podacima nakon što je obavljena samo jedna operacija. Ovakva neželjena ponašanja programa teško se pronalaze te teško dupliciraju budući da je vjerojatnost ponavljanja niska. Kada se broj brisanja (ubacivanja) elemenata u vezanu listu povećava, povećava se i vjerojatnost problematičnog izvršavanja programa.

# Poglavlje 2

# Zaštita podataka zaključavanjem

Jedan način izbjegavanja problema s dijeljenjem zajedničkih podataka jest korištenje lokota uz pomoć mutexa (engl. ***mut**ual **ex**clusion* - međusobno isključivanje). Ideja jest da dretva prije no što pristupi dijeljenim podacima zaključa mutex povezan s tim podacima, te otključa taj isti mutex nakon što završi s radom nad podacima. Bilo koja druga dretva koja s ciljem pristupa podacima pokuša zaključati mutex nakon što je prva dretva uspješno izvršila zaključavanje mutexa morati će čekati svoj red, tako dugo dok originalna dretva ne otključa mutex. S mutexima se čuvaju kritični odsječci - dijelovi kôda koje u bilo kojem trenutku može izvršavati najviše jedna dretva.

C++11 standard definira mehanizam zaštite podataka uz pomoć mutexa. Nad instancom klase `std::mutex` dostupne su funkcije članice `lock()` i `unlock()`.

Problem iz poglavlja 1 može se izbjeći pomoću `std::mutex` modificiranjem originalnog algoritma:

1. **Zaključaj mutex** povezan s vezanom listom.
2. Pronađi čvor koji je potrebno izbaciti iz liste ($B$).
3. Ažuriraj pokazivač čvora $A$ tako da pokazuje na čvor $C$.
4. Ažuriraj pokazivač čvora $C$ tako da pokazuje na čvor $A$.
5. Izbriši čvor $B$.
6. **Otključaj mutex** povezan s vezanom listom.

S gledišta neke druge dretve u procesu, modifikacija liste - u ovom slučaju brisanje čvora - ili nije započelo ili je već gotovo. Nema mogućnosti djelomično gotovih modifikacija nad podacima. Ostali dijelovi kôda koji obavljaju operacije nad vezanom listom također moraju biti zaključani istim mutexom.

U programu koji koristi dva ili više mutexa za određenu operaciju pojavljuje se problem zastoja (engl. *deadlock*). Zastoj je stanje prilikom kojeg dvije dretve međusobno čekaju da druga završi svoju operaciju, što rezultira time da nijedna ne nastavlja s radom. Obje dretve čekaju nešto što se nikada neće dogoditi (ukoliko nije prisutan mehanizam spriječavanja ovakvih situacija) te tako dolazi do stagnacije - nijedna dretva ne izvršava daljnje radnje. Takvo stanje moguće je prikazati primjerom: u programu su prisutne dvije dretve, `A` i `B`, te dva objekta nad kojima dretve obavljaju operacije, `X` i `Y`.

1. Dretva `A` obavlja operacije nad objektom `X`.
2. Dretve `A` i `B` pokušaju zaključati objekt `Y`.
3. Dretva `B` uspije zaključati objekt `Y` prije dretve `A`.
4. Dretva `B` treba pristup objektu `X`, no on je zaključan. S druge strane, dretva `A` treba pristup objektu `Y`, no i on je zaključan.

Drugi način izbjegavanja problema s dijeljenjem zajedničkih podataka u jeziku C++ jest uporaba *lock-free* struktura podataka uz pomoć klase `std::atomic`.

# Poglavlje 3

# C++ i std::atomic

Ukoliko slijed pristupa istoj memorijskoj lokaciji od strane više dretvi nije strogo određen, te barem jedna dretva mijenja podatke, tada u programu može doći do nedefiniranog ponašanja. U svrhu izbjegavanja nedefiniranog ponašanja, potrebno je za svaki par pristupa podacima u memoriji definirati željeni redoslijed. U ovome poglavlju to ćemo postići uz pomoć `std::atomic`.

*Atomic* operacija u C++-u definira se kao nedijeljiva operacija. Možemo ih smatrati kao jedinstvene instrukcije na razini stroja. Takva se operacija nad podacima smatraju trenutnima - one su ili izvršene ili nisu - proizvoljna dretva ne može opaziti takvu operaciju kao djelomično gotovu. Operacija čitanja *atomic* varijable vratiti će inicijalnu vrijednost varijable, ili vrijednost nakon neke modifikacije (Vidi 3.2). Operacije nad *atomic tipovima* vrlo su efikasne budući da su implementirane na niskoj razini.

Promotrimo višedretveni program u kojemu dretve povremeno inkrementiraju dijeljeni brojač. Instrukcija `brojac++` na prvi se pogled čini nedijeljiva, no nije *atomic*. Da bi inkrementirali brojač, procesor prvo pročita varijablu iz memorije i sprema ju u interni registar gdje ju povećava i konačno piše novu vrijednost nazad u memoriju. Ovakva operacija nije *atomic*. S ne-*atomic* operacijama u višedretvenom procesu moguća su sljedeća dva slučaja:

- **Operacija čitanja** Glavna dretva pročita jedan dio podataka iz memorije, druga dretva modificira te iste podatke, te zatim glavna dretva pročita ostatak objekta iz memorije. Pročitani podaci nisu niti podaci prije no što je druga dretva obavila modifikaciju, niti podaci nakon što je dretva obavila modifikaciju, već nešto između. Dolazimo do nedefiniranog ponašanja.

- **Operacija pisanja** Vrijednost podataka koje opaža druga dretva nije niti vrijednost prije, niti vrijednost nakon pisanja, već opet nešto između. Ponovno dolazimo do nedefiniranog ponašanja.

## 3.1 std::atomic<> predložak klase

U svrhu izbjegavanja navedenog, moguće je u programu koristiti `std::atomic`. Standardni *atomic* tipovi mogu se pronaći u `<atomic>` zaglavlju za što je pri kompilaciji potrebna zastavica `-std=c++11` (ili noviji C++14 ili C++17). `std::atomic` predložak može se instancirati nad bilo kojim tipom podataka kojeg je moguće trivijalno kopirati:

```
1  struct S{
2      int x, y;
3      char c;
4      unsigned long* pokazivac;
5  };
6  std::atomic<S> s; // specijalizacija
```

*Atomic* varijablom mogu postati pokazivači svih tipova - `std::atomic<T*>`. Također, svi sastavni tipovi jezika mogu biti *atomic* varijable:

- `char` tipovi: `char`, `char16_t`, `char32_t` i `wchar_t`,
- *signed* tipovi: `signed char`, `short`, `int`, `long` i `long long`,
- *unsigned* tipovi: `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long` i `unsigned long long`,
- *floating-point* tipovi: `float`, `double` i `long double`.

Sve operacije nad takvim tipovima podataka su *atomic* (i samo su te operacije *atomic* u smislu definicije jezika). Dostupne operacije nad `std::atomic<>` predloškom klase su:

- Čitanje: `load()`. Operacija čita i vraća trenutnu vrijednost *atomic* varijable.
- Pisanje: `store()`. Operacija vrši zamjenu trenutne vrijednosti *atomic* varijable s primljenom vrijednosti.
- Čitanje-modificiranje-pisanje: `exchange()`. Uz `exchange()`, dostupne su i operacije koje spremaju novu vrijednost ako je trenutna vrijednost jednaka očekivanoj vrijednosti: `compare_exchange_weak()` i `compare_exchange_strong()`. U ovu grupu spadaju još i specijalizirane funkcija članice: `fetch_add()` te `fetch_sub()`.

U tablici 3.1 prikazani su dostupni operatori nad *atomic* varijablama. Primjerice, `x*=2` nad `atomic<int>` tipu ne radi budući da `operator*=` nije definiran za taj tip podataka. Slično, inkrement `++` nad `atomic<double>` ne radi. U oba slučaja moguće je koristiti poziv funkcije `compare_exchange_strong()` za postizanje željenog rezultata.

| Operacija | atomic <bool> | atomic <T*> | atomic <sastavni-tip> | atomic <ostali-tipovi> |
|---|---|---|---|---|
| load() | ✓ | ✓ | ✓ | ✓ |
| store() | ✓ | ✓ | ✓ | ✓ |
| exchange() | ✓ | ✓ | ✓ | ✓ |
| compare_exchange_*() | ✓ | ✓ | ✓ | ✓ |
| fetch_add(), += | | ✓ | ✓ | |
| fetch_sub(), -= | | ✓ | ✓ | |
| fetch_or(), \|= | | | ✓ | |
| fetch_and(), &= | | | ✓ | |
| fetch_xor(), ^= | | | ✓ | |
| ++, -- | | ✓ | ✓ | |

Tablica 3.1: Dostupni operatori nad *atomic* tipovima podataka

Operacija `load()` označena je kao `const` budući da ne mijenja varijablu:

```
1  T load( std::memory_order order = std::memory_order_seq_cst ) const
       noexcept;
```

Deklaracija operacije `store()` glasi:

```
1  void store( T desired, std::memory_order order =
       std::memory_order_seq_cst ) noexcept;
```

Dodatno, svaka operacija nad *atomic* tipovima prima neobavezan argument reda modifikacije (Vidi 3.2).

U standardu su definirane i globalne verzije funkcija navedenih operacija s ciljem da jezik bude kompatibilan s jezikom C. *Atomic* objekt referenciran od strane funkcija članica u `std::atomic<>` predlošku klase je implicitan, no u globalnim verzijama tih funkcija potrebno je kao parametar poslati pokazivač na *atomic* varijablu. Globalne funkcije dolaze s prefiksom `atomic_` te su ekvivalentne odgovarajućim funkcijama članicama `std::atomic<>` predloška klase. Primjerice, instrukcija `std::atomic_store(&atomicVarijabla, vrijednost)` ekvivalentna je pozivu `atomicVarijabla.store(vrijednost)`. Dodatno, definirane su i funkcije sa sufiksom `_explicit` koje omogućuju specificiranje reda modifikacije. Tako primjerice dobivamo instrukciju `std::atomic_store_explicit(&atomicVarijabla, vrijednost, std::memory_order_release)`. Uporaba ovih funkcija biti će prikazana u odjeljku

4.1.1.

U sljedećem primjeru prikazane su uporabe funkcija predloška klase nad tipom podataka `std::atomic<int>`:

```
1  std::atomic<int> x{1};
2  int y = x.load();
3  x.store(2);
4  y = x.exchange(3);
5  int z = x.fetch_sub(4);
```

Funkcije koje obavljaju *čitanje-modificiranje-pisanje* mijenjaju vrijednost *atomic* varijable te vraćaju prijašnju vrijednost - onu prije modifikacije. Varijabla `y` nakon 4. linije poprima vrijednost 2, dok `z` nakon 5. linije poprima vrijednost 3.
Deklaracija funkcije `exchange()` glasi:

```
1  T exchange( T desired, std::memory_order order =
       std::memory_order_seq_cst ) noexcept;
```

Kako je spomenuto ranije, dostupne su i *compare / exchange* operacije `compare_exchange_weak()` i `compare_exchange_strong()` koje uspoređuju vrijednost *atomic* varijable s primljenom `expected` vrijednosti i spremaju `desired` vrijednost u varijablu ako su jednake. Ukoliko nisu jednake, `expected` se vrijednost mijenja na stvarnu vrijednost *atomic* varijable. Funkcije vraćaju `bool`: `true` ako je zamjena izvršena, `false` inače. Njihove deklaracije glase:

```
1  bool compare_exchange_weak( T& expected, T desired, std::memory_order
       order = std::memory_order_seq_cst ) noexcept;
2  bool compare_exchange_strong( T& expected, T desired,
       std::memory_order order = std::memory_order_seq_cst ) noexcept;
```

Parametri funkcija su:

- `expected` - referenca na vrijednost za koju se očekuje da ju poprima *atomic* varijabla.
- `desired` - vrijednost koja se sprema u *atomic* varijablu ako varijabla poprima očekivanu vrijednost.
- `order` - neobavezan red modifikacije.

`weak` verzija funkcije može biti neuspješna - ponaša se kao da je vrijednost *atomic* varijable različita od `expected`, čak i kada su jednake. Tada `compare_exchange_weak()` vraća `false`. Ovaj se slučaj može dogoditi na uređajima na kojima na niskoj razini ne postoji samostalna *compare-and-exchange* instrukcija. Drugim riječima, ukoliko

procesor ne može garantirati da je operacija obavljena na *atomic* način. Iz tog razloga `compare_exchange_weak()` koristimo u petlji:

```
1    bool expected = false;
2    extern atomic<bool> b; //Vrijednost postavljena na nekom drugom mjestu
         u programu
3    while(!b.compare_exchange_weak(expected, true) && !expected);
```

Petlja se izvršava tako dugo dok je `expected` još uvijek `false`.
`strong` verzija ove funkcije radi kako i očekujemo - vraća `false` samo onda kada je `*this != expected`. Sa `strong` nije potrebno koristiti petlju.

### 3.1.1 Aritmetika *atomic* pokazivača

Aritmetika nad *atomic* pokazivačima obuhvaća operatore += i -=, pre/post inkrement odnosno dekrement sa ++ i --, i funkcije `fetch_add()` i `fetch_sub()`. ++ operacija, na primjer, poveća *atomic* `T*` za `sizeof(T)`.

```
1    T niz[3];
2    std::atomic<T*> ptr(niz);
3    T* x = ptr.fetch_add(2);
4    assert(x == niz);
5    assert(ptr.load() == &niz[2]);
6    x = (ptr -= 1);
7    assert(&niz[1] == ptr.load());
8    assert(x == ptr.load());
```

`fetch_add(2)` u 3. liniji mijenja pokazivač `ptr` tako da pokazuje na treći element u nizu, te vraća pokazivač na prvi element. Sada `x` i `niz` pokazuju na istu memorijsku lokaciju. `ptr -= 1` smanjuje vrijednost na koju `ptr` pokazuje i vraća novu vrijednost koja se sprema u `x`.

## 3.2 Redovi modifikacija

Standard C++11 uz *atomic* operacije uveo je i `std::memory_order` (definirano u zaglavlju `<atomic>`). `std::memory_order` je enumeracija uz pomoć koje možemo specificirati redoslijed izvršavanja između *atomic* i ne-*atomic* operacija.

Svaki objekt u programu ima precizno definiran red modifikacija. Taj se red sastoji od svih promjena nad danim objektom u pojedinom izvršavanju programa. Prva takva promjena nad objektom je njegova inicijalizacija. Red modifikacije će varirati između većeg broja izvršavanja programa, no u pojedinačnom izvršavanju sve se dretve moraju

složiti oko redoslijeda modifikacija. Ukoliko objekt u programu nije *atomic* tipa, tada je programer zadužen za određivanje željenog redoslijeda modifikacija kako ne bi bilo dvosmislenosti između reda pristupa objektu od strane više dretvi. To je moguće postići npr. međusobnim isključivanjem - programer eksplicitno određuje kritične odsječke. U drugu ruku, ukoliko koristimo *atomic* varijable, tada je za navedeno zadužen kompajler.

Redoslijedi modifikacija su konstante koje se šalju kao drugi argument funkcijama članicama `std::atomic` klase. Razlikujemo šest različitih redoslijeda koji su podijeljeni u tri grupe:

1. *sequentially consistent*: `memory_order_seq_cst`,
2. *acquire-release*: `memory_order_consume`, `memory_order_acquire`, `memory_order_release`, i `memory_order_acq_rel`,
3. *relaxed*: `memory_order_relaxed`.

### 3.2.1 *Sequentially consistent* redoslijed

Sve dretve u programu vidjet će isti redoslijed modifikacija nad *atomic* objektima ukoliko koristimo *sequentially consistent* redoslijed. U tom slučaju, ponašanje višedretvenog programa biti će kao da se napisane operacije izvršavaju u određenom redoslijedu na **jednoj** dretvi. L. Lamport definirao[1] je *sequentially consistent* model kao

> *the result of any execution is the same as if the reads and writes occured in some order, and the operations of each individual processor appear in this sequence in the order specified by its program*

Ovaj red modifikacija jest zadan nad svim funkcijama *atomic* objekata kao što je vidljivo u npr. deklaraciji funkcije `exchange()` na stranici 9. Sljedeći program prikazuje uporabu `std::memory_order_seq_cst`:

```
1   #include <atomic>
2   #include <thread>
3   #include <cassert>
4
5   std::atomic<bool> x{false}, y{false};
6   std::atomic<int> z{0};
7
8   void pisi_x(){
9       x.store(true, std::memory_order_seq_cst);
10  }
```

[1] Preuzeto iz [5]

```
11    void pisi_y(){
12        y.store(true, std::memory_order_seq_cst);
13    }
14    void procitaj_x_y(){
15        while(!x.load(std::memory_order_seq_cst));
16        if(y.load(std::memory_order_seq_cst))
17            ++z;
18    }
19    void procitaj_y_x(){
20        while(!y.load(std::memory_order_seq_cst));
21        if(x.load(std::memory_order_seq_cst))
22            ++z;
23    }
24    int main(){
25        std::thread A(pisi_x), B(pisi_y), C(procitaj_x_y), D(procitaj_y_x);
26        A.join();
27        B.join();
28        C.join();
29        D.join();
30        assert(z.load() != 0);
31    }
```

Ako `y.load()` u 16. liniji vrati `false`, tada se `x.store()` izvršila prije `y.store()`. Dakle, `x.load()` u 21. liniji mora vratiti `true` pošto `while` petlja osigurava da je u tom trenutku `y == true`. Moguć je i drugi smjer: `x.load()` u 21. liniji vrati `false` te `y.load()` vrati `true`. U oba slučaja vrijedi `z == 1`.

Budući da ovaj red modifikacije zahtjeva globalnu sinkronizaciju između svih dretvi, u programu koji ga koristi može doći do smanjenja efikasnosti i povećanja potrošene memorije. Izvršavanje programa dodatno će usporiti zahtijevamo li sinkronizaciju u sistemu s više procesora, budući da je komunikacija između procesora prilično spora. Iz tog se razloga koriste i manje stroži redoslijedi modifikacija.

### 3.2.2 *Relaxed* redoslijed

U programu koji ne koristi *sequentially consistent* red modifikacije na *atomic* varijablama moguće su nesuglasice između dretvi oko redoslijeda događaja. Jedino što *relaxed* redoslijed garantira jest nemogućnost reorganiziranja redoslijeda pristupa *atomic* varijabli **unutar dretve**. Jednom kada dretva pročita vrijednost *atomic* varijable, sljedeća čitanja iste varijable ne mogu vratiti ranije verzije vrijednosti te iste varijable. U sljedećem primjeru prikazano je što se može desiti prilikom korištenja `std::memory_order_relaxed`:

```
1   std::atomic<bool> x{false}, y{false};
2   std::atomic<int> z{0};
3
4   void pisi_x_y(){
5       x.store(true, std::memory_order_relaxed);
6       y.store(true, std::memory_order_relaxed);
7   }
8   void procitaj_y_x(){
9       while(!y.load(std::memory_order_relaxed));
10      if(x.load(std::memory_order_relaxed))
11          ++z;
12  }
13  int main(){
14      std::thread A(pisi_x_y), B(procitaj_y_x);
15      A.join();
16      B.join();
17      assert(z.load() != 0);
18  }
```

`assert()` na 17. liniji sada može prekinuti izvršavanje programa budući da `x.load()` na 10. liniji može vratiti `false` čak i ako je vrijednost od `y` jednaka `true`. Iako je `x.store()` preciziran prije `y.store()` u funkciji `pisi_x_y()`, dretva `B` može vidjeti `store()` operacije u različitom redoslijedu.

Operacije s *relaxed* redoslijedom mogu biti reorganizirane od strane kompajlera ili hardvera (s ciljem optimizacije). Upravo se iz tog razloga ne preporuča uporaba *relaxed* redoslijeda. S ciljem sinkronizacije između dretvi potrebno je koristiti jedne od strožih redoslijeda modifikacija.

### 3.2.3 *Acquire-release* redoslijed

U *acquire-release* memorijskom modelu, `load()` i `store()` operacije sinkronizirane su kao i kod *sequentially consistent* modela, no ne postoji globalni redoslijed događaja. *Atomic* `load()` operacije koriste `std::memory_order_acquire`, dok `store()` operacije koriste `std::memory_order_release`. *Release* operacija sada je sinkronizirana s *acquire* operacijom koja čita zapisane podatke:

1. Nijedna instrukcija prije *release* `store()` **ne može** biti vidljiva drugim dretvama nakon te `store()` operacije. S druge strane, napisane instrukcije čitanja i pisanja nakon *release* `store()` **mogu** biti vidljive prije te `store()` operacije (kao da se u programu

nalaze prije te instrukcije). To predstavlja razliku između *acquire-release* i *sequentially consistent* modela.

2. Nijedna instrukcija nakon *acquire* `load()` **ne može** biti vidljiva drugim dretvama prije te `load()` operacije. Analogno, čitanja i pisanja prije *acquire* `load()` **mogu** biti vidljive nakon te `load()` operacije.

U programu sa stranice 12 promijenjen je memorijski model - umjesto `std::memory_order_relaxed` koristi se `std::memory_order_acquire` odnosno `std::memory_order_release`:

```
1    void pisi_x_y(){
2        x.store(true, std::memory_order_relaxed);
3        y.store(true, std::memory_order_release);
4    }
5    void procitaj_y_x(){
6        while(!y.load(std::memory_order_acquire));
7        if(x.load(std::memory_order_relaxed))
8            ++z;
9    }
```

Sada je `y.store()` sinkroniziran s `y.load()` pa dretva `B` ne može vidjeti `x.store()` **nakon** `y.store()`. U dretvi `A` redoslijed je jasan: prvo se mijenja varijabla `x`, a zatim `y`. Ukoliko dretva `B` čeka na promjenu varijable `y` sa navedenim memorijskim modelom, tada nisu moguće nesuglasice između reda događaja između dretvi. Povećanje varijable `z` sada se mora izvršiti pa `assert()` neće prekinuti izvršavanje programa. Naravno, ukoliko se u 6. liniji ne koristi petlja, već samo jedan `y.load()` i ukoliko je `y == false`, tada nemamo garanciju na `x` (u tom trenutku `x` može biti ili `true` ili `false`).

U svrhu postizanja željene sinkronizacije, *acquire* i *release* operacije moraju doći u paru nad **istom** *atomic* varijablom. Vrijednost spremljena s *release* redoslijedom mora biti čitana od strane *acquire* operacije.

*Acquire-release* model moguće je koristiti za sinkronizaciju tri ili više dretvi, kao što je prikazano u primjeru:

1. Dretva `A` izvrši operaciju `OP_1` te zatim release operaciju na *atomic* varijabli `x`.
2. Dretva `B` izvrši *acquire* operaciju na `x` te zatim *release* operaciju na *atomic* varijabli `y`.
3. Dretva `C` izvrši *acquire* operaciju nad `y` te zatim operaciju `OP_2`.

Sada se operacija `OP_1` izvršava prije `OP_2`. Ovaj se slučaj može pojednostaviti sa samo jednom varijablom (`sinkronizacija`) pomoću `std::memory_order_acq_rel`. Ideja jest

da kada prva dretva odradi svoj posao, ona objavljuje (*releases*) podatke. Sada te podatke može steći (*acquires*) sljedeća dretva koja nastavlja posao. Imamo garanciju da su svi podaci nad kojima se radilo prije `sinkronizacija.store(1, std::memory_order_release)` sada spremni za rad u drugoj dretvi nakon *acquire* `load()` - `while` petlja u 22. liniji:

```
1   std::atomic<short> sinkronizacija{0};
2
3   void dretva_A(){
4       std::cout << "Pocetak A" << std::endl;
5
6       // Simulacija OP_1()
7       std::cout << "Izvrsavanje OP_1()" << std::endl;
8       std::this_thread::sleep_for(std::chrono::milliseconds(5000));
9
10      sinkronizacija.store(1, std::memory_order_release);
11      std::cout << "Kraj A" << std::endl;
12  }
13  void dretva_B(){
14      std::cout << "Pocetak B" << std::endl;
15      short ocekivano = 1;
16      while(!sinkronizacija.compare_exchange_weak(ocekivano, 2,
            std::memory_order_acq_rel))
17          ocekivano = 1;
18      std::cout << "Kraj B" << std::endl;
19  }
20  void dretva_C(){
21      std::cout << "Pocetak C" << std::endl;
22      while(sinkronizacija.load(std::memory_order_acquire) != 2){
23          std::cout << "Dretva C: Cekam..." << std::endl;
24          std::this_thread::sleep_for(std::chrono::milliseconds(1000));
25      }
26      // Operacija OP_2()
27      std::cout << "Izvrsavanje OP_2()" << std::endl;
28
29      std::cout << "Kraj C" << std::endl;
30  }
```

U dretvi `B` potrebno je koristiti bilo koju *pročitaj-modificiraj* operaciju sa `std::memory_order_acq_rel` modelom, npr. `compare_exchange_weak()`. Jedan mogući ispis gornjeg programa glasi:

```
1   Pocetak A
```

```
2     Pocetak B
3     Pocetak C
4     Dretva C: Cekam...
5     Izvrsavanje OP_1()
6     Dretva C: Cekam...
7     Dretva C: Cekam...
8     Dretva C: Cekam...
9     Dretva C: Cekam...
10    Kraj A
11    Kraj B
12    Izvrsavanje OP_2()
13    Kraj C
```

`OP_2()` u dretvi `C` čeka na svoj red. Jednom u sekundi provjerava stanje varijable `sinkronizacija` te započinje tek nakon što `OP_1()` završi.

Navedena sinkronizacija postignuta je samo među dretvama koje koriste navedene *release* i *acquire* operacije. Druge dretve u procesu mogu vidjeti drugačiji redoslijed modifikacija. Sličnu *release-acquire* sinkronizaciju možemo postići koristeći mehanizme međusobnog isključivanja, npr. sa `std::mutex`. *Acquire* čitanje odgovara `lock()` funkciji iz klase `mutex`, dok *release* spremanje odgovara `unlock()` funkciji.

### 3.2.4 Sažetak

`std::memory_order_seq_cst` predstavlja najstroži red modifikacije, dok je *relaxed* najmanje strog. *Sequentially consistent* model jest dodijeljen izbor nad svim funkcijama članicama `std::atomic<>` predloška klase te tako programerima osigurava da će se izvršiti ono što su *htjeli* napisati. Greške proizvedene uporabom krivog memorijskog modela lako se dešavaju brzim pisanjem kôda te se teško dupliciraju. *Relaxed* model koji pruža najbrže izvršavanje može se koristiti prilikom npr. inkrementiranja zajedničkog *atomic* brojača od strane više dretvi. U tom slučaju nije nam bitan redoslijed - čak i ako se pojave nesuglasice - ukoliko nam je bitna samo vrijednost brojača nakon završetka svih dretvi. Želimo li sigurnost da neće biti nesuglasica između dretvi oko redoslijeda događaja, koristit ćemo najsporiji *sequentially consistent* model.

# Poglavlje 4

# *Lock-free* paralelne strukture podataka

Struktura podataka je *lock-free* ako dvije ili više dretvi na siguran način mogu pristupiti podacima te strukture istovremeno. Glavni razlog korištenja ovakvih struktura je osiguravanje maksimalne *istovremenosti*. S mehanizmima međusobnog isključivanja npr., postoji mogućnost blokiranja pristupa podacima dretvama - dretve čekaju svoj red. Takav princip spriječava istovremenost. Kod *lock-free* struktura u drugu ruku, ne postoji mogućnost blokiranja dretvi čekajući na dostupnost mutexa.

U ovom će se poglavlju promatrati *lock-free* implementacije strukture podataka stog, primijenit će se memorijski model (iz odjeljka 3.2) nad istima, usporedit će se efikasnost između dva promatrana memorijska modela te će biti opisane pozitivne i negativne strane *lock-free* struktura podataka.

## 4.1 Implementacija *lock-free* stoga

Implementacija strukture podataka stog postići će se uz pomoć vezane liste te jednog pokazivača koji identificira gornji element na stogu - onaj koji je zadnji stavljen na vrh stoga. Svaki element u listi sadržavat će pokazivač na podatke te pokazivač na sljedeći element. U prvom pokušaju sve će biti ostvareno sa zadanim *sequentially consistent* modelom, a kasnije u odjeljku 4.2 s ciljem postizanja boljih performansi biti će korišteni `std::memory_order_acq_rel` i `std::memory_order_relaxed`.

`push()` operacija dodaje novi element na vrh stoga, `pop()` operacija uklanja gornji element te `top()` vraća taj isti element, bez uklanjanja sa stoga. Brzo se dolazi do problema: vraćanja objekta po vrijednosti iz funkcije može baciti iznimku ukoliko dođe do greške prilikom kopiranja tog objekta. U tom je slučaju vrijednost objekta izgubljena. Ukoliko se koristi `std::shared_ptr<>` (i pripadna funkcija `std::make_shared()`) tada neće doći do iznimke.

`std::shared_ptr<>` je klasa koja sadrži i brojač koji ukazuje na broj pokazivača koji pokazuju na zajednički objekt. Upravo to može poslužiti kao indikator koliko dretvi trenutno obavlja `pop()` operaciju - brisanje elementa iz liste može se obaviti tek kada samo jedna dretva, ona koja obavlja brisanje, drži referencu na taj podatak. Naravno, u *lock-free* algoritmu, implementacija `std::shared_ptr<>` također mora biti *lock-free*. Prvi dio ovog odjeljka predstavljat će implementaciju stoga s pretpostavkom da to jest slučaj, dok će drugi dio biti pisan s ručnim brojanjem referenci na svaki element u stogu. C++11 standard definira globalnu funkciju `std::atomic_is_lock_free()` kojom je to moguće provjeriti:

```
1  std::shared_ptr<T> pokazivac;
2  std::cout << std::atomic_is_lock_free(&pokazivac);
```

Gornji segment može ispisati 0 ili 1, ovisno o implementaciji `std::shared_ptr<>` na danoj arhitekturi.

### 4.1.1 Slučaj kada std::shared_ptr<> je *lock-free*

Sljedeća klasa prikazuje implementaciju strukture podataka stog na platformama na kojima je `std::shared_ptr` *lock-free*:

```
1  template<typename T>
2  class lock_free_Stog{
3  private:
4      struct element{
5          std::shared_ptr<T> podatak;
6          std::shared_ptr<element> sljedeci;
7          element(const T &ulaz): podatak(std::make_shared<T> (ulaz)){}
8      };
9      std::shared_ptr<element> vrh;
10
11 public:
12     bool empty() const{
13         return std::atomic_load(&vrh) ? false : true;
14     }
15     void push(const T &ulaz){
16         const std::shared_ptr<element> noviElement =
               std::make_shared<element> (ulaz);
17         noviElement->sljedeci = vrh;
18         while(!std::atomic_compare_exchange_weak(&vrh,
               &noviElement->sljedeci, noviElement));
19     }
20     std::shared_ptr<T> top() const{
```

```
21          return std::atomic_load(&vrh) ? std::atomic_load(&vrh)->podatak
                : std::shared_ptr<T>();
22      }
23      void pop(){
24          std::shared_ptr<element> stariVrh = std::atomic_load(&vrh);
25          while(stariVrh && !std::atomic_compare_exchange_weak(&vrh,
                &stariVrh, stariVrh->sljedeci));
26      }
27  };
```

`push()` funkcija kreira novi element čiji pokazivač `sljedeci` postavlja na trenutnu vrijednost varijable `vrh`. U ovom trenutku novo kreirani element i varijabla `vrh` oba pokazuju na vrh stoga. U 18. se liniji u `while` petlji provjerava vrijednost varijable `vrh`. Ona je možda u međuvremenu promijenjena (između 17. i 18. linije) pa je stoga potrebno ponoviti instrukciju sa 17. linije. Upravo to izvršava `std::atomic_compare_exchange_weak()` operacija u slučaju da varijable `vrh` i `noviElement->sljedeci` nisu jednake. U tom je slučaju neka druga dretva promijenila varijablu `vrh` (ili preko svog poziva `push()` ili `pop()` operacije). Instrukcija `noviElement->sljedeci = vrh` sada se ponavlja sve dok dretva ne dobi neprekinut pristup varijabli `vrh` - iz tog se razloga koristi `while` petlja. U slučaju da nijedna druga dretva nije promijenila varijablu `vrh`, izvršit će se pridruživanje `vrh = noviElement` te će funkcija unutar `while` petlje vratiti `true`. Negacija osigurava prekid izvršavanja `while` petlje čim funkcija vrati `true`. Slična ideja koristi se i unutar `pop()` operacije.

Prilikom kreiranja novog elementa na samom početku funkcije `push()` nije potrebno koristiti *atomic* operacije budući da se alokacija odvija izvan strukture podataka. U to je vrijeme neka druga dretva slobodna napraviti modifikacije nad strukturom. Na kraju `pop()` funkcije nije potrebno očistiti memoriju elementa koji je upravo uklonjen - za to se brine `std::shared_ptr`. Objekt se briše s hrpe (engl. *heap*) čim više ne postoji (zajednički) pokazivač koji pokazuje na njega.

Sada je prolazak kroz strukturu moguće izvršiti pomoću primjerice `for` petlje:

```
1   for(; !S.empty(); S.pop()){
2       std::cout << *S.top() << " ";
3   }
```

### 4.1.2 Slučaj kada std::shared_ptr<> nije *lock-free*

Ukoliko na arhitekturi za koju se piše program ne postoji *lock-free* implementacija `std::shared_ptr`, potrebno je ručno provjeravati koji se elementi strukture podataka trenutno koriste prije nego što se izvrši brisanje elementa. Jedna mogućnost jest korištenje tzv. *hazard* pokazivača kao što je prikazano u [6], odjeljak 7.2.3. Takva implementacija

koristi dodatnu listu koja pamti elemente u uporabi. Ovdje će biti prikazana druga metoda - pamtit će se broj dretvi koje pristupaju svakom elementu.

Svaki element u listi sadržavat će dva brojača: unutarnji i vanjski. Zbroj te dvije varijable označavat će ukupan broj referenci na pojedini element u stogu. Nakon dodavanja novog elementa na vrh stoga, vrijednost unutarnjeg brojača elementa je nula, dok je vrijednost vanjskog brojača jednaka jedan. U tom je trenutku pokazivač `vrh` jedina vanjska referenca na novo ubačeni element. Vanjski brojač elementa povećava se sa svakim čitanjem pokazivača. Unutarnji se brojač smanjuje prilikom završetka operacije čitanja. Brisanje elementa sa stoga može se izvršiti tek kada je suma dva brojača jednaka nuli.

Implementacija[1] obuhvaća dvije strukture za svaki element: `element` i `brojac`. Prema konceptima standarda C++, strukturu `brojac` moguće je trivijalno kopirati pa varijabla `vrh` koja je tipa `brojac` može biti *atomic* varijabla:

```
1   template<typename T>
2   class lock_free_Stog{
3   private:
4       struct element;
5       struct brojac{
6           int vanjskiBrojac;
7           element* pokazivac = NULL;
8       };
9       struct element{
10          std::shared_ptr<T> podatak;
11          std::atomic<int> unutarnjiBrojac;
12          brojac sljedeci;
13          element(const T &ulaz): podatak(std::make_shared<T>(ulaz)),
                unutarnjiBrojac(0){}
14      };
15
16      std::atomic<brojac> vrh;
17
18      void povecajBrojac(brojac& stariBrojac){
19          brojac noviBrojac;
20          do{
21              noviBrojac = stariBrojac;
22              ++noviBrojac.vanjskiBrojac;
23          } while(!vrh.compare_exchange_strong(stariBrojac, noviBrojac));
24          stariBrojac.vanjskiBrojac = noviBrojac.vanjskiBrojac;
25      }
26
```

[1] Prilikom kompilacije potrebno je koristiti dodatnu zastavicu: `-march=native`

```
27      public:
28          bool empty() const{
29              return vrh.load().pokazivac ? false : true;
30          }
31
32          void push(T const& ulaz){
33              brojac noviElement;
34              noviElement.pokazivac = new element(ulaz);
35              noviElement.vanjskiBrojac = 1;
36              noviElement.pokazivac->sljedeci = vrh.load();
37              while(!vrh.compare_exchange_weak(noviElement.pokazivac->sljedeci,
                    noviElement));
38          }
39
40          std::shared_ptr<T> pop(){
41              brojac stariVrh = vrh.load();
42              while(true){
43                  povecajBrojac(stariVrh);
44                  element* p = stariVrh.pokazivac;
45                  if(!p)
46                      return std::shared_ptr<T>();
47                  if(vrh.compare_exchange_strong(stariVrh, p->sljedeci)){
48                      std::shared_ptr<T> vrati;
49                      vrati.swap(p->podatak);
50                      const int promjena = stariVrh.vanjskiBrojac - 2;
51                      if(p->unutarnjiBrojac.fetch_add(promjena) == -promjena)
52                          delete p;
53                      return vrati;
54                  }
55                  else if(p->unutarnjiBrojac.fetch_sub(1) == 1)
56                      delete p;
57              }
58          }
59
60          ~lock_free_Stog(){
61              while(!empty()){ pop(); }
62          }
63      };
```

#### push() operacija

S ciljem dodavanje novog elementa na vrh stoga konstruira se noviElement koji pokazuje na podatak tipa T. Ovdje se također postavljaju vrijednosti brojača. Trenutna vrijednost varijable vrh sprema se u novo alociranu strukturu, dok noviElement preuzima ulogu novog pokazivača na vrh stoga.

#### pop() operacija

pop() operacija sada obavlja dvije funkcije: briše element s vrha stoga te vraća taj isti element - kao operacija top().

U ovoj se funkciji obavlja operacija čitanja elementa pa je stoga najprije potrebno povećati vanjski brojač elementa kojeg pop() nastoji ukloniti. Za to je napisana pomoćna privatna funkcija povecajBrojac(). Ukoliko dretva pročita gornji element prije no što poveća brojač, neka bi druga dretva mogla u međuvremenu obaviti brisanje tog elementa. U tom bi slučaju originalna dretva koristila pokazivač na memorijski prostor koji više ne pripada procesu. Povećanjem brojača na početku operacije osigurano je da korišteni pokazivač ostane valjan sve do završetka operacije.

U 45. liniji se provjerava vrijednost pokazivača p dobivenog iz varijable stariVrh. Ako je p jednak NULL, tada je stog prazan pa pop() vraća novi std::shared_ptr.

Ukoliko compare_exchange_strong() u 47. liniji ne uspije izvršiti zamjenu (funkcija vraća bool), tada je druga dretva modificirala stog - ili je izbrisan element koji operacija pop() trenutno želi izbrisati, ili je na vrh stoga dodan novi element. U oba slučaja potrebno je dohvatiti ažuriranu vrijednost varijable vrh pomoću compare_exchange_strong(). Upravo je zbog toga potrebna while petlja u 42. liniji. Prije sljedećeg prolaza kroz petlju, potrebno je smanjiti unutarnji brojač elementa kojeg se pokušalo izbrisati. Ako je taj element već izbrisan (od strane druge dretve), unutarnji brojač biti će jednak jedan. U tom se slučaju može izvršiti brisanje elementa (56. linija).

U slučaju da druga dretva nije modificirala varijablu vrh, compare_exchange_strong() će izvršiti zamjenu p->sljedeci = vrh te vratiti true. U varijabli promjena sprema se vrijednost vanjskog brojača umanjenog za dva: element se briše sa stoga (za to minus jedan) te dretva koja obavlja brisanje više nema referencu na taj element (za to još jednom minus jedan). U 51. se liniji s fetch_add() unutarnjem brojaču dodaje vrijednost varijable promjena. Ako je zbroj brojača sada jednak nuli, prijašnja vrijednost unutarnjeg brojača jednaka je negativnoj vrijednost varijable promjena, pa se može izvršiti brisanje elementa.

## 4.2 Primjena memorijskog modela

S ciljem postizanja boljih performansi korištene strukture podataka, u ovom odjeljku napisane su modificirane operacije push() i pop() (i pripadna pomoćna privatna funkcija

`povecajBrojac()`) *lock-free* implementacije stoga kada `std::shared_ptr<>` nije *lock-free*. Umjesto zadanog *sequentially consistent* modela koriste se manje strože *acquire/release* i *relaxed* operacije.

Deklaracije `compare_exchange_weak()` i `compare_exchange_strong()` funkcija napisane u poglavlju 3 proširuju se tako da primaju još jedan parametar. Deklaracije sada glase:

```
1  bool compare_exchange_weak( T& expected, T desired, std::memory_order
       success, std::memory_order failure ) noexcept;
2  bool compare_exchange_strong( T& expected, T desired,
       std::memory_order success, std::memory_order failure ) noexcept;
```

Umjesto jednog memorijskog redoslijeda, funkcije sada primaju dva: prvi u slučaju uspješne zamjene vrijednosti *atomic* varijable te drugi u slučaju neuspjeha (slučaj kada je potrebno ponoviti operaciju).

### `push()` operacija

U svrhu postizanja sinkronizacije između dretvi od kojih jedna obavlja `push()`, a druga `pop()` operaciju, u 6. liniji sljedeće funkcije potrebno je koristiti `std::memory_order_release` redoslijed. U `push()` operaciji prvo se konstruira novi brojač i element te se zatim mijenja varijabla `vrh`. U `pop()` operaciji prvo se čita varijabla `vrh` te se zatim čita `pokazivac->sljedeci` elementa na koji `vrh` pokazuje. Ovdje je potrebna sinkronizacija: `push()` koristi `std::memory_order_release`, dok `pop()` koristi `std::memory_order_acquire` (6. linija programa u sljedećem odjeljku). U slučaju neuspjeha, ništa se nije promijenilo te se operacija mora ponoviti. Tada se može koristiti najslabiji memorijski redoslijed - `std::memory_order_relaxed`.

```
1  void push(T const& data){
2      brojac noviElement;
3      noviElement.pokazivac = new element(data);
4      noviElement.vanjskiBrojac = 1;
5      noviElement.pokazivac->sljedeci =
           vrh.load(std::memory_order_relaxed);
6      while(!vrh.compare_exchange_weak(noviElement.pokazivac->sljedeci,
           noviElement, std::memory_order_release,
           std::memory_order_relaxed));
7  }
```

### pop() operacija

Kako je već spomenuto, u privatnoj funkciji `povecajBrojac()` koristi se `std::memory_order_acquire` redoslijed modifikacija u slučaju uspjeha, odnosno `std::memory_order_relaxed` u slučaju neuspjeha.

Prilikom pristupa `p->podatak` u 19. liniji sljedećeg programa, potrebno je osigurati da taj pristup čitanja dretvi bude omogućen tek nakon operacije pisanja te varijable u `push()` funkciji. No, to je već osigurano pomoću *acquire* čitanja varijable u funkciji `povecajBrojac()`. Iz tog je razloga u 17. liniji moguće koristiti `std::memory_order_relaxed`.

Nadalje, potrebno je osigurati da se `swap()` događa prije `delete p` instrukcije. To se postiže uz pomoć `fetch_add()` i `fetch_sub()` funkcija. Jedna mogućnost jest korištenje *release* redoslijeda u zbrajanju te *acquire* redoslijeda u oduzimanju. No, `if` uvjet u 25. liniji neće uvijek vratiti `true` pa se u svrhu postizanja još bržeg izvršavanja, ovdje koristi `std::memory_order_relaxed` i dodatna `load()` operacija sa `std::memory_order_acquire`.

```
1   void povecajBrojac(brojac& stariBrojac){
2       brojac noviBrojac;
3       do{
4           noviBrojac = stariBrojac;
5           ++noviBrojac.vanjskiBrojac;
6       } while(!vrh.compare_exchange_strong(stariBrojac, noviBrojac,
            std::memory_order_acquire, std::memory_order_relaxed));
7       stariBrojac.vanjskiBrojac = noviBrojac.vanjskiBrojac;
8   }
9
10  std::shared_ptr<T> pop(){
11      brojac stariVrh = vrh.load(std::memory_order_relaxed);
12      while(true){
13          povecajBrojac(stariVrh);
14          element* p = stariVrh.pokazivac;
15          if(!p)
16              return std::shared_ptr<T>();
17          if(vrh.compare_exchange_strong(stariVrh, p->sljedeci,
                std::memory_order_relaxed)){
18              std::shared_ptr<T> vrati;
19              vrati.swap(p->podatak);
20              const int promjena = stariVrh.vanjskiBrojac - 2;
21              if(p->unutarnjiBrojac.fetch_add(promjena,
                    std::memory_order_release) == -promjena)
22                  delete p;
```

```
23              return vrati;
24          }
25          else if(p->unutarnjiBrojac.fetch_sub(1,
                std::memory_order_relaxed) == 1){
26              p->unutarnjiBrojac.load(std::memory_order_acquire);
27              delete p;
28          }
29      }
30  }
```

## 4.3 Efikasnost memorijskog modela

U ovom se odjeljku uspoređuje efikasnost implementacije koja koristi *sequentially consistent* model nad svim *atomic* varijablama u *lock-free* stogu (program iz odjeljka 4.1.2) u odnosu na implementaciju koja koristi manje strože operacije nad *atomic* varijablama (funkcije iz odjeljka 4.2). Proteklo vrijeme izvršavanja mjeri se sljedećim test programom:

```
1   #pragma optimize( "", off )
2   #define N (unsigned long long int)pow(10, 6)
3
4   std::vector<bool> vec(N);
5
6   void test(const unsigned int x){
7       std::chrono::steady_clock::time_point pocetak =
            std::chrono::steady_clock::now();
8       if(x==1)  // S1
9           for(unsigned long long int i=0; i<N; i++)
10              if(vec[i]) S1.push(true);
11              else S1.pop();
12      else      // S2
13          for(unsigned long long int i=0; i<N; i++)
14              if(vec[i]) S2.push(true);
15              else S2.pop();
16      std::chrono::steady_clock::time_point kraj =
            std::chrono::steady_clock::now();
17      std::cout << "Vrijeme trajanja (ID=" << std::this_thread::get_id()
            << "): " <<
            std::chrono::duration_cast<std::chrono::milliseconds>(kraj -
            pocetak).count() << " milisekundi." << std::endl;
18  }
```

```
19
20      void stvoriUzorak(){
21          std::default_random_engine generator;
22          std::uniform_int_distribution<int> distribucija(0, 1);
23          for(unsigned long long int i=0; i<N; i++)
24              vec[i] = distribucija(generator);
25      }
26
27      int main(){
28          stvoriUzorak();
29          std::thread A(&test, 1), B(&test, 1); // S1
30          A.join(); B.join();
31          std::thread C(&test, 2), D(&test, 2); // S2
32          C.join(); D.join();
33      }
```

Komandni redak za kompilaciju gornjeg programa sa svim zastavicama sada glasi:

```
1       $ g++ Efikasnost.cpp -o Efikasnost -std=c++11 -march=native -pthread
            -Wextra -pedantic
```

U programu za testiranje stvara se slučajan uzorak nula i jedinica varijabilne duljine pomoću uniformne distribucije. Slučajni se uzorak sprema u varijablu `vec` koja je duljine `N`. Varijabla `N` postavlja se na samom početku programa pomoću instrukcije `#define`. Ovisno o sadržaju varijable vec na $i-$tom mjestu, nad stogom se izvršava `push()`, `odnosno pop()` operacija.

U prvoj se liniji nalazi `#pragma optimize` instrukcija koja isključuje optimizacije od strane kompajlera. U programu se testiraju dvije implementacije iste strukture podataka pa je za što pravedniji rezultat potrebno isključiti potencijalne optimizacije kompajlera.

Mjerenje proteklog vremena postiže se uz pomoć `std::chrono::steady_clock` biblioteke, definirane C++11 standardom u zaglavlju `<chrono>`.

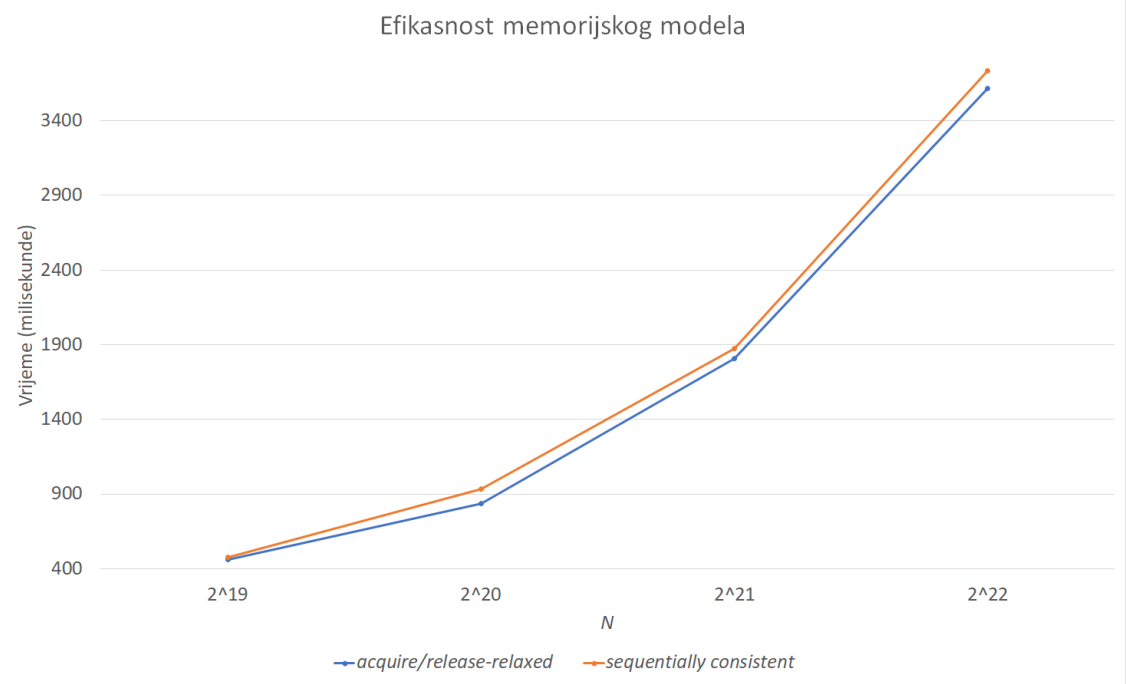
Za svaku implementaciju *lock-free* stoga potrebne su dvije dretve - `A` i `B` za objekt `S1` klase *sequentially consistent* implementacije, te dretve `C` i `D` za objekt `S2` klase *acquire/release-relaxed* implementacije. U bilo kojem trenutku izvršavanja programa dretve koje dolaze u paru istovremeno obavljaju `push()` / `pop()` operacije $N$ puta nad odgovarajućim objektom, simulirajući stvarni rad programa koji koristi strukturu podataka stog. Jedan mogući ispis programa za $N = 10^6$ glasi:

```
1       Vrijeme trajanja (ID=140613961316096): 8917 milisekundi.
2       Vrijeme trajanja (ID=140613952923392): 8974 milisekundi.
3       Vrijeme trajanja (ID=140613961316096): 8631 milisekundi.
```

```
4    Vrijeme trajanja (ID=140613952923392): 8706 milisekundi.
```

Za svaki par dretvi uzima se maksimum vremenskog trajanja `test()` funkcije. Dakle, nad objektom `S1` operacije završavaju za 8974 milisekundi, dok nad `S2` završavaju za 8706 milisekundi.

Uporabom odgovarajućeg memorijskog modela postiže se ubrzanje izvršavanja programa[2] za prosječnih 4%, kao što je vidljivo na slici 4.1. Graf je dobiven promjenom vrijednosti varijable `N` u drugoj liniji programa za testiranje.




Slika 4.1: Usporedba brzine izvršavanja *acquire/release-relaxed* i *sequentially consistent* modela

## 4.4 Pozitivne i negative strane *lock-free* struktura

U usporedbi *lock-free* struktura podataka u odnosu na strukture koje koriste međusobno isključivanje mogu se primijetiti neke pozitivne i neke negativne karakteristike:

- Ukoliko se iz bilo kojeg razloga prekine izvršavanje dretve koja trenutno drži lokot (tj. mutex) asociran s određenim podacima, tada staju i sve ostale dretve koje čekaju

[2]Program je pokrenut na osmojezgrenom procesoru AMD 8320 na 4.2GHz, x86-64 arhitektura

na svoj red za rad s tim istim podacima. Dolazi do zastoja cijelog procesa. Kod *lock-free* struktura u drugu ruku, ukoliko dođe do prekida izvršavanja jedne dretve tijekom izvršavanja operacije nad strukturom (npr. opisane `push()` ili `pop()` operacije), ostatak procesa, tj. ostale dretve, nastavljaju daljnji rad. Zastoj jedne dretve nema posljedice na ostale dretve u procesu.

- U *lock-free* algoritmima osigurano je da će neka dretva nastaviti svoj rad sa svakim sljedećim korakom[3]. Prilikom korištenja kritičnih odsječaka postoji mogućnost blokiranja dretvi i spriječavanja *istovremenosti*.

- Problem zastoja opisan u poglavlju 2 prilikom korištenja kritičnih odsječaka nije moguć kod *lock-free* algoritama. No, sada postoji mogućnost drugog problema: tzv. *livelock*. W. Stallings u knjizi [4] *livelock* definira kao situaciju u kojoj dva ili više procesa kontinuirano mijenjaju svoja stanja kao odgovor na promjenu stanja u drugim procesima bez obavljanja korisnog posla. U ovakvoj situaciji dvije dretve zahtijevaju ponovno pokretanje operacije jedna drugoj. Ovakvi tipovi "lokota" obično kratko traju budući da ovise o preciznom vremenskom izvršavanju operacija različitih dretvi.

- Procesor koji izvršava program mora sinkronizirati podatke između dretvi koje pristupaju istim *atomic* varijablama u *lock-free* algoritmima. Ovisno o tipu namjene strukture podataka, to može dovesti do ukupno sporijeg izvršavanja u odnosu na varijantu koja koristi međusobno isključivanje.

- Između korištenja *atomic* operacija i odgovarajućeg redoslijeda modifikacija nad varijablama, pisanje *lock-free* algoritama jest teže u odnosu na pisanje standardnih kritičnih odsječaka.

Algoritmi koji sadrže *čitanje-modificiranje-pisanje* operacije često sadržavaju petlje s ciljem promjene vrijednosti varijable (kao što je prikazano u poglavlju 3) zbog toga što neka druga dretva može u međuvremenu modificirati tu varijablu. Tada se operacija modifikacije mora ponoviti. Dretve će ponekad izvršiti suvišne poslove što u mnogim slučajevima pruža brže izvršavanje algoritma u odnosu na algoritme baziranim na kritičnim odsječcima.

[3]Mogućnost nastavka izvršavanja **svake** dretve sa svakim korakom izvršavanja programa moguće je postići pisanjem tzv. *wait-free* struktura podataka, no to je izvan područja ovoga rada

# Bibliografija


[1] *C++ reference*. `http://en.cppreference.com/w/cpp`.

[2] Alessandrini, Victor: *Shared Memory Application Programming. Concepts and strategies in multicore application programming*. Elsevier Inc., 2016.

[3] Pikus, Fedor (urednik): *C++ atomics, from basic to advanced. What do they really do?* CppCon. `https://cppcon.org/`.

[4] Stallings, William: *Operating Systems. Internals and Design Principles*. Pearson, 2018.

[5] Sutter, Herb (urednik): *atomic<> Weapons. The C++11 Memory Model and Modern Hardware*. C++ and Beyond 2012.

[6] Williams, Anthony: *C++ Concurrency in Action. Practical Multithreading*. Manning Publications Co., 2012.

# Sažetak

Tradicionalni mehanizmi višedretvenog programiranja obuhvaćaju međusobno isključivanje u svrhu zaštite podataka. U ovom je radu predstavljena metoda koja se ne oslanja na mutexe i kreiranje kritičnih odsječaka, već uz pomoć tzv. *atomic* operacija, standardiziranih u programskom jeziku C++ 2011. godine, postiže siguran pristup podacima od strane više različitih dretvi unutar procesa. Predstavljen je pripadni memorijski model koji služi kao osnova za konstrukciju *lock-free* algoritama. Opisani su redoslijedi modifikacija koji nam služe za sinkronizaciju podataka između dretvi. Detaljno su opisane paralelne implementacije *lock-free* strukture podataka stog te je prikazana razlika u efikasnosti izvođenja programa koji koriste različite memorijske modele. Navedene su i razlike u odnosu na ekvivalentne programe koji koriste mehanizme međusobnog isključivanja.

# Summary

Traditional mechanisms for multithreaded programming include mutual exclusion in order to protect data. This thesis presents a method which doesn't rely on mutexes nor creation of critical sections, but which with the help of so called *atomic* operations, standardized in programming language C++ 2011 revision, accomplishes safe data access for multiple threads inside a process. A memory model which serves as a basis for construction of *lock-free* algorithms was introduced. Modification orders are described which are used for data synchronization between multiple threads. Different implementations of data structure stack are described in detail, as well as difference in program efficiency which use different modification orders. The differences between *lock-free* algorithms and equivalent programs which use mechanisms for mutual exclusion are also presented.

# Životopis

Rođen sam 12. prosinca 1994. u Zagrebu, Hrvatska. Nakon osnovne škole, u istom gradu završavam X. gimnaziju "Ivan Supek", prirodoslovno-matematičkog usmjerenja. 2013. godine na Prirodoslovno-matematičkom fakultetu upisujem preddiplomski sveučilišni studij *Matematika; smjer: nastavnički*. Na preddiplomskom se studiju kroz izborne kolegije upoznajem s područjem računarstva te nakon prvostupničke diplome 2016. godine nastavljam diplomski studij na istom fakultetu na studiju *Računarstvo i matematika*.

Tijekom diplomskog studija ostvarujem stipendiju za izvrsnost grupe PBZ d.d.