

Operativni Sistemi - Konkurentno programiranje 3

Veljko Petrović

Mart, 2024

Semafori

Sinhronizacija pomoću semafora

- Sinhronizacija niti može da se zasnije i na ideji saobraćajnog semafora koji reguliše ulazak vozova u stanicu sa jednim kolosekom.
- Kada se jedan voz nalazi na staničnom koloseku, pred semaforom se moraju zaustaviti svi vozovi koji treba da dođu na stanični kolosek.
- Po analogiji sa saobraćajnim semaforom prolaz niti kroz (softversku) kritičnu sekciju bi regulisao (softverski) semafor.

Sinhronizacija pomoću semafora

- Sinhronizacija niti, koju omogućuje semafor, se zasniva na zaustavljanju aktivnosti niti, kao i na omogućavanju nastavljanja njihove aktivnosti.
- Ulazak niti u kritičnu sekciju zavisi od stanja semafora.
- Kada stanje semafora dozvoli ulazak niti u kritičnu sekciju, pri ulasku se semafor prevodi u stanje koje onemogućuje ulazak druge niti u kritičnu sekciju.
- Ako se takva nit pojavi, njena aktivnost se zaustavlja pred kritičnom sekcijom.
- Pri izlasku niti iz kritične sekcije semafor se prevodi u stanje koje dozvoljava novi ulazak u kritičnu sekciju i ujedno omogućuje nastavak aktivnosti niti koja najduže čeka na ulaz u kritičnu sekciju (ako takva nit postoji).

Semafor simuliran sa propusnicom

```
1 class Semaphore {
2     mutex mx;
3     int state;
4     condition_variable queue;
5 public:
6     Semaphore(int value = 1) : state(value) {};
7     void stop();
8     void resume();
9 };
```

Semafor simuliran sa propusnicom

```
10 void Semaphore::stop(){
11     unique_lock<mutex> lock(mx);
12     while(state < 1)
13         queue.wait(lock);
14     state--;
15 }
16 void Semaphore::resume(){
17     unique_lock<mutex> lock(mx);
18     state++;
19     queue.notify_one();
20 }
```

Semafor u C++ biblioteci

- C++ biblioteka, od standarda iz 2020 podržava semafore
- Nalaze se u zaglavlju `<semaphore>`
- Klasa je `counting_semaphore` i šablonska je gde je (jedini) parametar maksimalna numerička vrednost ugrađenog brojača
- `stop` i `resume` opcije se zovu `acquire` i `release` ali je centralna ideja ista

Vrste i upotreba semafora

- Semafor čije stanje ne može preći vrednost 1 se zove binarni semafor.
- Ako se njegovo stanje inicijalizuje na vrednost 1, tada su njegove operacije `stop()` i `resume()` slične operacijama `lock()` i `unlock()` klase `mutex`.
- Ako se njegovo stanje inicijalizuje na vrednost 0, tada su njegove operacije `stop()` i `resume()` slične operacijama `wait()` i `notify_one()` klase `condition_variable`

- Operacije klase `condition_variable` su namenjene za ostvarenje uslovne sinhronizacije u okviru kritičnih sekcija u kojima je međusobna isključivost ostvarena pomoću operacija klase `mutex`.

Raspodeljeni binarni semafor

- Međutim, upotreba operacija binarnog semafora (sa stanjem inicijalizovanim na vrednost 0) po uzoru na operacije klase `condition_variable` u okviru kritičnih sekcija u kojima je međusobna isključivost ostvarena pomoću operacija drugog binarnog semafora (sa stanjem inicijalizovanim na vrednost 1) izaziva mrtvu petlju.
- Zato se uvodi posebna vrsta binarnog semafora, nazvana raspodeljeni binarni semafor (split binary semaphore).

Raspodeljeni binarni semafor

- Realizuje se uz pomoć više binarnih semafora za koje važi ograničenje da suma njihovih stanja ne može preći vrednost 1.
- Pomoću raspodeljenog binarnog semafora se ostvaruje uslovna sinhronizacija tako što se na ulazu u svaku kritičnu sekciju poziva operacija `stop()` jednog od njegovih binarnih semafora, a na izlazu iz nje operacija `resume()` tog ili nekog od preostalih binarnih semafora.
- Na taj način najviše jedna nit se može nalaziti najviše u jednoj od pomenutih kritičnih sekcija, jer su stanja svih semafora manja od vrednosti 1 za vreme njenog boravka u dotičnoj kritičnoj sekciji.

Message_box sa semaforima

```

1  #include "sem.hh"
2
3  template<class MESSAGE>
4  class Message_box {
5      MESSAGE content;
6      Semaphore empty;
7      Semaphore full;
8  public:
9      Message_box() : full(0) {};
10     void send(const MESSAGE* message);
11     MESSAGE receive();
12 };

```

Message_box sa semaforima

```
13 template<class MESSAGE>
14 void Message_box<MESSAGE>::send(const MESSAGE* message){
15     empty.stop();
16     content = *message;
17     full.resume();
18 }
```

Message_box sa semaforima

```
19 template<class MESSAGE>
20 MESSAGE Message_box<MESSAGE>::receive(){
21     MESSAGE message;
22     full.stop();
23     message = content;
24     empty.resume();
25     return message;
26 }
```

Generalni semafor - primer sa slobodnim baferima

- Semafor, čije stanje može sadržati vrednost veću od 1, se naziva generalni semafor (general semaphore).
- On omogućuje ostvarenje uslovne sinhronizacije prilikom rukovanja resursima.
- Pozitivno stanje generalnog semafora može predstavljati broj slobodnih primeraka nekog resursa.
- Zahvaljujući tome, zauzimanje primerka resursa se može opisati pomoću operacije stop(), a njegovo oslobađanje pomoću operacije resume() generalnog semafora.

Generalni semafor - primer sa slobodnim baferima

- Ova klasa sadrži binarni semafor mex i generalni semafor list_member_count.
- Binarni semafor omogućuje međusobnu isključivost prilikom uvezivanja i izvezivanja slobodnog bafera.
- Generalni semafor omogućuje uslovnu sinhronizaciju, jer njegovo stanje pokazuje broj slobodnih bafera (ono je na početku inicijalizovano na 0).

Generalni semafor i slobodni baferi

```
1  #include "sem.hh"
2
3  struct List_member {
4      List_member* next;
5      char buffer[512];
6  };
7
8  class List {
9      List_member* first;
10     Semaphore list_member_count;
11     Semaphore mex;
12 public:
13     List () : first(0), list_member_count(0) {};
14     void link(List_member* member);
15     List_member* unlink();
16 };
```

Generalni semafor i slobodni baferi

```
17 void List::link(List_member* member){
18     mex.stop();
19     member->next=first;
20     first=member;
21     mex.resume();
22     list_member_count.resume();
23 }
```

Generalni semafor i slobodni baferi

```
24 List_member* List::unlink(){
25     List_member* unlinked;
26     list_member_count.stop();
27     mex.stop();
28     unlinked=first;
29     first=first->next;
30     mex.resume();
31     return unlinked;
32 }
```

Rešenje problema pet filozofa pomoću semafora

- Rešenje problema pet filozofa pomoću semafora sprečava pojavu mrtve petlje, jer za parne filozofe zauzima prvo levu, a za

neparne filozofe zauzima prvo desnu viljušku.

- Viljuške predstavljaju binarni semafori forks[5], koji omogućuju uslovnu sinhronizaciju prilikom zauzimanja viljuški.
- Međusobnu isključivost omogućuje binarni semafor mex.
- Stanja filozofa su promenjena, jer nije moguća mrtva petlja, pa nije bitno koju viljušku filozof čeka.

Pet filozofa i semafori

```
1  #include<thread>
2  #include<iostream>
3
4  using namespace std;
5  using namespace chrono;
6  using namespace this_thread;
7
8  #include "sem.hh"
9
10 int mod5(int a){
11     return (a > 4 ? 0 : a);
12 }
```

Pet filozofa i semafori

```
13 enum Philosopher_state { THINKING = 'T', HUNGRY = 'H', EATING = 'E' };
14 Philosopher_state philosopher_state[5];
15 Semaphore forks[5];
16 Semaphore mex;
17
18 int philosopher_identity(0);
19 const milliseconds THINKING_PERIOD(10);
20 const milliseconds EATING_PERIOD(10);
```

Pet filozofa i semafori

```
21 void show(){
22     for(int j = 0; j < 5; j++) {
23         cout << '(' << (char)(j+'0') << ':'
24             << (char)(philosopher_state[j]) << ") ";
25     }
26     cout << endl;
27 }
```

Pet filozofa i semafori

```
28 void thread_philosopher(){
29     mex.stop();
30     int pi = philosopher_identity++;
31     philosopher_state[pi] = THINKING;
32     mex.resume();
33     for(;;) {
34         sleep_for(THINKING_PERIOD);
35         mex.stop();
36         philosopher_state[pi] = HUNGRY;
37         show();
38         mex.resume();
```

Pet filozofa i semafori

```
39         forks[pi%2 == 0 ? pi : mod5(pi+1)].stop();
40         forks[pi%2 == 0 ? mod5(pi+1) : pi].stop();
41         mex.stop();
42         philosopher_state[pi] = EATING;
43         show();
44         mex.resume();
45         sleep_for(EATING_PERIOD);
46         mex.stop();
47         philosopher_state[pi] = THINKING;
48         show();
49         mex.resume();
50         forks[pi].resume();
51         forks[mod5(pi+1)].resume();
52     }
53 }
```

Pet filozofa i semafori

```
54 int main(){
55     cout << endl << "DINING PHILOSOPHERS" << endl;
56     thread philosopher0(thread_philosopher);
57     thread philosopher1(thread_philosopher);
58     thread philosopher2(thread_philosopher);
59     thread philosopher3(thread_philosopher);
60     thread philosopher4(thread_philosopher);
61     philosopher0.join();
62     philosopher1.join();
63     philosopher2.join();
64     philosopher3.join();
```

```

65     philosopher4.join();
66 }

```

Čitanje i pisanje i semafori

```

1  #include<thread>
2  #include<iostream>
3
4  using namespace std;
5  using namespace chrono;
6  using namespace this_thread;
7
8  #include "sem.hh"
9
10 const int ACCOUNTS_NUMBER = 10;
11 const int INITIAL_AMOUNT = 100;

```

Čitanje i pisanje i semafori

```

12 class Bank {
13     Semaphore mex;
14     int accounts[ACCOUNTS_NUMBER];
15     short readers_number;
16     short writers_number;
17     short readers_delayed_number;
18     short writers_delayed_number;
19     Semaphore readers;
20     Semaphore writers;
21     void show();
22     void reader_begin();
23     void reader_end();
24     void writer_begin();
25     void writer_end();

```

Čitanje i pisanje i semafori

```

26 public:
27     Bank();
28     void audit();
29     void transaction(unsigned source, unsigned destination);
30 };

```


Čitanje i pisanje i semafori

```
31 Bank::Bank() : mex(1), readers(0), writers(0){
32     for(int i = 0; i < ACCOUNTS_NUMBER; i++)
33         accounts[i] = INITIAL_AMOUNT;
34     readers_number = 0;
35     writers_number = 0;
36     readers_delayed_number = 0;
37     writers_delayed_number = 0;
38 }
```

Čitanje i pisanje i semafori

```
39 void Bank::show(){
40     cout << "RN: " << readers_number << " RDN: "
41         << readers_delayed_number
42         << " WN: " << writers_number << " WDN: "
43         << writers_delayed_number << endl;
44 }
```

Čitanje i pisanje i semafori

```
45 void Bank::reader_begin(){
46     mex.stop();
47     if((writers_number > 0) || (writers_delayed_number > 0)) {
48         readers_delayed_number++;
49         show();
50         mex.resume();
51         readers.stop();
52     }
```

Čitanje i pisanje i semafori

```
53     readers_number++;
54     show();
55     if(readers_delayed_number > 0) {
56         readers_delayed_number--;
57         show();
58         readers.resume();
59     } else
60         mex.resume();
61 }
```

Čitanje i pisanje i semafori

```
62 void Bank::reader_end(){
63     mex.stop();
64     readers_number--;
65     show();
66     if((readers_number == 0) && (writers_delayed_number > 0)) {
67         writers_delayed_number--;
68         show();
69         writers.resume();
70     } else
71         mex.resume();
72 }
```

Čitanje i pisanje i semafori

```
73 void Bank::writer_begin(){
74     mex.stop();
75     if((readers_number > 0) || (writers_number > 0)) {
76         writers_delayed_number++;
77         show();
78         mex.resume();
79         writers.stop();
80     }
81     writers_number++;
82     show();
83     mex.resume();
84 }
```

Čitanje i pisanje i semafori

```
86 void Bank::writer_end(){
87     mex.stop();
88     writers_number--;
89     show();
90     if(writers_delayed_number > 0) {
91         writers_delayed_number--;
92         show();
93         writers.resume();
94     }
```

Čitanje i pisanje i semafori

```
95 const milliseconds READING_PERIOD(1);
96
```

```

97 void Bank::audit(){
98     int sum = 0;
99     reader_begin();
100    sleep_for(READING_PERIOD);
101    for(unsigned i = 0; i < ACCOUNTS_NUMBER; i++)
102        sum += accounts[i];
103    reader_end();
104    if(sum != ACCOUNTS_NUMBER*INITIAL_AMOUNT) {
105        mex.stop();
106        cout << " audit error " << endl;
107        mex.resume();
108    }
109 }

```

Čitanje i pisanje i semafori

```

110
111 const milliseconds WRITING_PERIOD(1);
112
113 void Bank::transaction(unsigned source, unsigned destination){
114     int amount;
115     writer_begin();
116     sleep_for(WRITING_PERIOD);
117     amount = accounts[source];
118     accounts[source] -= amount;
119     accounts[destination] += amount;
120     writer_end();
121 }

```

Čitanje i pisanje i semafori

```

122 Bank bank;
123
124 void thread_reader(){
125     bank.audit();
126 }
127 void thread_writer0to1(){
128     bank.transaction(0, 1);
129 }
130 void thread_writer1to0(){
131     bank.transaction(1, 0);
132 }

```

Čitanje i pisanje i semafori

```
134 int main(){
135     cout << endl << "READERS AND WRITERS" << endl;
136     thread reader0(thread_reader);
137     thread reader1(thread_reader);
138     thread writer0(thread_writer0to1);
139     thread reader2(thread_reader);
140     thread writer1(thread_writer1to0);
141     reader0.join();
142     reader1.join();
143     writer0.join();
144     reader2.join();
145     writer1.join();
146 }
```

Konkurentno programiranje bez zaključavanja

Atomici i programiranje bez zaključavanja

- Posle svog ovog truda da napravimo i propusnicu i semafor i sve ovo, zašto za ime sveta bi hteli da sada *odustanemo* od svega toga?
- Kao i obično kada je u pitanju c++ egzotika za ovim posežemo kada nam treba još performansi: svo zaključavanje (propusnice, semafori, sinhronizacija, inače) su *spore*.
- Komparativno govoreći, za većinu tema ovo je više nego zadovoljavajuće
- Šta je alternativa?

Hardver i portabilnost

- Setite se kada smo pričali o compare-and-swap i hardverskoj podršci?
- Ima još dosta takvih instrukcija koje omogućavaju da se rade operacije atomički
- Takođe imaju načini da se zatraži da instrukcije poštuju određene *memorijske modele*.
- Ovo naravno zavisi jako od arhitekture na kojoj ovo koristite.

Memorijski modeli

- Normalno procesor i kompajler su u zaveri da se određenim instrukcijama menja redosled ne bi li se povećale performanse.
- Sve se vrti oko procesa koji se zove **pipelining**
- Mi možemo zatražiti da se na to ponašanje nametnu ograničenja od limitiranih do toga da zatražimo 'sekvencijalnu konzistentnost' što znači da se izvršavanje dešava tačno kako mi kažemo.

Kako koristiti hardversku podršku?

- C++ pruža zaglavlje <atomic> koje sadrži tipove koji označavaju da su određeni primitivni tipovi (int, recimo) atomički
- To znači da se nad njima mogu pozivati isključivo *atomičke operacije*
- Umesto da slobodno pristupamo memoriji moramo da je učitamo sa .load() (i tako dobijemo povratnu vrednost) ili da smeštamo vrednosti sa .store()
- Iza kulisa, kompajler pretvara ove naše zahteve u kompleksne pozive instrukcija koje su same po sebi atomičke, a ako nema podršku, vara tako što koristi iste mutekse itd. koje bi i mi koristili.

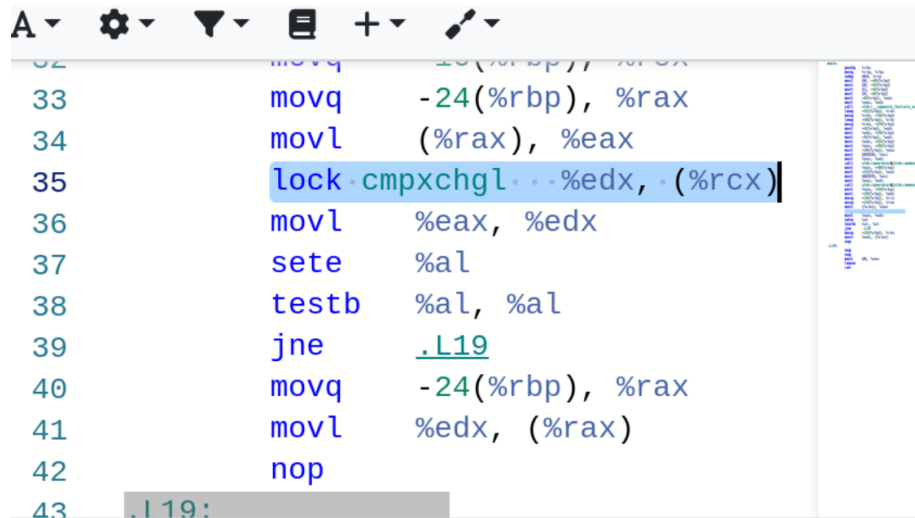
Trivijalan primer

```
#include <atomic>

using namespace std;

int main(){
    int z = 0;
    atomic<int> a(0);
    a.compare_exchange_strong(z, 1);
    return 0;
}
```

Izlaz kompajlera



```
32      movq    -24(%rbp), %rax
33      movq    -24(%rbp), %rax
34      movl    (%rax), %eax
35      lock cmpxchgl %edx, (%rcx)
36      movl    %eax, %edx
37      sete    %al
38      testb   %al, %al
39      jne     .L19
40      movq    -24(%rbp), %rax
41      movl    %edx, (%rax)
42      nop
43      .L19:
```

Zar to nismo negde videli?

- Da li je ova instrukcija poznata?
- I to čak sa *lock* prefiksom, kao što smo pričali.
- Naravno u zavisnosti od toga za koji kompajler ovo radimo, ovo može da iza kulisa ima ili nema stvarne atomičke instrukcije

Tipovi atomičkih operacija

- Možda ste primetili neobično ime 'compare_exchange_strong'
- To je zato što ova operacija dolazi u dve forme: snažna i slaba
- Snažna uvek radi ali je nešto sporija
- Slaba ponekad može da zakaže, ali samo tako što vrati rezultat da stvari koje se porede nisu iste kada jesu.
- Slaba operacija se preferira u petljama budući da će petlja da garantuje da će konverigrati tačnom rezultatu, a performanse su bolje.

Primer upotrebe rukovanje baferima

```
1  #include <iostream>
2  #include <thread>
3  #include <atomic>
4  #include <cassert>
5
6  using namespace std;
```

Primer upotrebe rukovanje baferima

```
7 struct List_member {
8     List_member* next;
9     char buffer[512];
10 };
11
12 class List {
13     atomic<List_member*> first;
```

Primer upotrebe rukovanje baferima

```
14 public:
15     List () {
16         first.store(nullptr);
17     };
18     void link(List_member* member);
19     List_member* unlink();
20 };
```

Primer upotrebe rukovanje baferima

```
21 void List::link(List_member* member){
22     member->next = first.load();
23     while(!first.compare_exchange_weak(member->next, member));
24 }
```

Primer upotrebe rukovanje baferima

```
25 List_member* List::unlink(){
26     List_member* unlinked;
27     unlinked=first.load();
28     List_member* next = nullptr;
29     do{
30         if(unlinked == nullptr) return nullptr;
```

Primer upotrebe rukovanje baferima

```
31         next = unlinked->next;
32     }while(!first.compare_exchange_weak(unlinked, next));
33     return unlinked;
34 }
35 void f(List* baferi){
36     List_member* x = nullptr;
```

Primer upotrebe rukovanje baferima

```
37     for(int i = 0; i < 10000; i++){
38         if(x == nullptr){
39             x = baferi->unlink();
40         }else{
41             baferi->link(x);
42             x = nullptr;
43         }
```

Primer upotrebe rukovanje baferima

```
44     }
45
46     if(x != nullptr){
47         baferi->link(x);
48     }
49 }
```

Primer upotrebe rukovanje baferima

```
50 int main(){
51     List l;
52     List_member* b1 = new List_member();
53     List_member* b2 = new List_member();
54     List_member* b3 = new List_member();
55     List_member* b4 = new List_member();
56     List_member* b5 = new List_member();
```

Primer upotrebe rukovanje baferima

```
57     l.link(b1);
58     l.link(b2);
59     l.link(b3);
60     l.link(b4);
61     l.link(b5);
62     thread t1(f, &l);
63     thread t2(f, &l);
```

Primer upotrebe rukovanje baferima

```
64     t1.join();
65     t2.join();
66     List_member* p;
67     assert((nullptr != l.unlink()) && "U klasi nema dovoljno bafera posle testa");
```



```

68     assert((nullptr != l.unlink()) && "U klasi nema dovoljno bafera posle testa");
69     assert((nullptr != l.unlink()) && "U klasi nema dovoljno bafera posle testa");
70     assert((nullptr != l.unlink()) && "U klasi nema dovoljno bafera posle testa");

```

Primer upotrebe rukovanje baferima

```

71     assert((nullptr != l.unlink()) && "U klasi nema dovoljno bafera posle testa");
72     assert((nullptr == l.unlink()) && "U klasi ima sufficit bafera posle testa");
73     delete b1;
74     delete b2;
75     delete b3;
76     delete b4;
77     delete b5;

```

Primer upotrebe rukovanje baferima

```

78     return 0;
79 }

```

Pobeda?

- Deluje kao da je sve OK, zar ne?
- Nije.
- Probajte slobodno da napravite treću nit i videćete da se manifestuju problemi koje je teško opisati rečima
- Potpun kaos, štetno preplitanje, užas.
- Što? ABA

ABA

- ABA je noćna mora koja vas može proganjati kada radite atomske operacije
- U pitanju je sekvenca promena koja ostavi onu promenljivu sa kojom atomski operišete u naizgled dobrom stanju koje maskira, u stvari, nekonzistentno stanje.
- Hajde da to razumemo u kontekstu baš ove naše klase.

ABA

1. Nit 1 počne unlink operaciju i vidi na vrhu A i B ('vidimo' dve stvari jer imamo i next)
2. Nit 2 počne unlink operaciju i završi je vrativši A
3. Nit 2 počne link operaciju i ubaci u listu neko D.
4. Nit 2 počne link operaciju i ubaci u listu ono isto A koje je izvadila.

5. Nit 1 vidi na vrhu A i zaključi da nije došlo do desinhronizacije i završi svoj unlink što znači da smo čvor D preskočili.

ABA

- Naravno ABA može da uradi i druge probleme
- U stvari, može da uradi praktično sve što može i štetno preplitanje
- Ne javlja se uvek, naravno, ali bilo kada imamo stanje koje može da izgleda isto spolja a nije, u opasnosti smo

Rešenje?

- Ubacimo polje za određivanje verzije
- Sada kada radimo `compare_and_swap` operaciju sve što treba da uradimo jeste da je uradimo nad dve vrednosti istovremeno.
- Jedna vrednost je pokazivač, druga je ista kao pokazivač u dimenzijama ali sadrži samo monotono rastuć broj verzije promene što garantuje detekciju ABA problema (osim ako nemamo overflow, ali na 64-bitnim arhitekturama to nije veliki rizik)
- Da li to uopšte može?

DWCAS

- Proizvođači procesora su nam to omogućili
- Generički termin za ovo je Double Word Compare and Swap
- Intel to zove `cmpxchg16b` i dostupno je na novim procesorima (Pogledajte da li se u `/proc/cpuinfo` nalazi flag `cx16`)
- Onda nam samo treba da napravimo umesto pokazivača struct koji je pokazivač + verzija i da ubedimo kompajler da emituje ovu korisnu instrukciju i sve je lako

Lako?

- Nikako.
- Prvo, ima teškoća u implementaciji ovoga kako valja ali sa tim se, uz pažnju, da izboriti.
- Kompajleri će vam prirediti mnogo gori problem: kako stvari stoje GCC, recimo, jednostavno odbija da emituje DWCAS instrukciju maltene šta god vi radili.
- Jedini kompajler na kome sam uspeo da proizvedem korektan kod je `clang` a i on zahteva poseban tretman – struct koji koris-

tite za DWCAS mora biti eksplicitno poravnan u memoriji na 16 bajtova što obično nije potrebno.

Verzija sa DWCAS

```
1 #include <iostream>
2 #include <thread>
3 #include <atomic>
4 #include <cassert>
5 #include <cstdint>
6
7 using namespace std;
```

Verzija sa DWCAS

```
8 struct List_member {
9     List_member* next;
10    char buffer[512];
11 };
12
13 struct alignas(2 * sizeof(void*)) p_aba {
```

Verzija sa DWCAS

```
14     uintptr_t aba;
15     List_member* p;
16 };
17
18 class List {
19     atomic<p_aba> first;
20 public:
```

Verzija sa DWCAS

```
21     List () {
22         first.store({0, nullptr});
23     };
24     void link(List_member* member);
25     List_member* unlink();
26 };
```

Verzija sa DWCAS

```
27 void List::link(List_member* member){
28     p_aba found = first.load();
```

```

29     p_aba next;
30     do{
31         member->next = found.p;
32         next.aba = found.aba + 1;

```

Verzija sa DWCAS

```

33         next.p = member;
34     }while(!first.compare_exchange_weak(found, next));
35 }
36 List_member* List::unlink(){
37     p_aba found = first.load();

```

Verzija sa DWCAS

```

38     p_aba novi;
39     do{
40         if(found.p == nullptr) return nullptr;
41         novi.p = found.p->next;
42         novi.aba = found.aba + 1;
43     }while(!first.compare_exchange_weak(found, novi));
44     return found.p;

```

Verzija sa DWCAS

```

45 }
46
47 void f(List* baferi){
48     List_member* x = nullptr;
49     for(int i = 0; i < 10000; i++){
50         if(x == nullptr){
51             x = baferi->unlink();

```

Verzija sa DWCAS

```

52     }else{
53         baferi->link(x);
54         x = nullptr;
55     }
56 }
57
58 if(x != nullptr){

```

Verzija sa DWCAS

```
59     baferi->link(x);
60 }
61 }
62
63 int main(){
64     List l;
65     List_member* b1 = new List_member();
```

Verzija sa DWCAS

```
66     List_member* b2 = new List_member();
67     List_member* b3 = new List_member();
68     List_member* b4 = new List_member();
69     List_member* b5 = new List_member();
70     l.link(b1);
71     l.link(b2);
72     l.link(b3);
```

Verzija sa DWCAS

```
73     l.link(b4);
74     l.link(b5);
75     thread t1(f, &l);
76     thread t2(f, &l);
77     thread t3(f, &l);
78     t1.join();
79     t2.join();
```

Verzija sa DWCAS

```
80     t3.join();
81     List_member* p;
82     assert((nullptr != l.unlink()) && "U klasi nema dovoljno bafera posle testa");
83     assert((nullptr != l.unlink()) && "U klasi nema dovoljno bafera posle testa");
84     assert((nullptr != l.unlink()) && "U klasi nema dovoljno bafera posle testa");
85     assert((nullptr != l.unlink()) && "U klasi nema dovoljno bafera posle testa");
86     assert((nullptr != l.unlink()) && "U klasi nema dovoljno bafera posle testa");
```

Verzija sa DWCAS

```
87     assert((nullptr == l.unlink()) && "U klasi ima sufficit bafera posle testa");
88     delete b1;
89     delete b2;
```

```

90     delete b3;
91     delete b4;
92     delete b5;
93     return 0;
94 }

```

Kako se ovo kompajlira?

```
clang++ -O3 -pthread -std=c++20 -mcx16 -o lld lockless_double.cpp
```

Da li stvarno emituje instrukciju?

```
objdump -d lld|grep cmpxchg16b
```

Rezultat

```

40142a:    f0 48 0f c7 0c 24    lock cmpxchg16b (%rsp)
40143a:    f0 48 0f c7 0c 24    lock cmpxchg16b (%rsp)
40144a:    f0 48 0f c7 0c 24    lock cmpxchg16b (%rsp)
40145a:    f0 48 0f c7 0c 24    lock cmpxchg16b (%rsp)
40146b:    f0 48 0f c7 0c 24    lock cmpxchg16b (%rsp)
40147b:    f0 48 0f c7 0c 24    lock cmpxchg16b (%rsp)
40149b:    f0 48 0f c7 0c 24    lock cmpxchg16b (%rsp)
4014ab:    f0 48 0f c7 0c 24    lock cmpxchg16b (%rsp)
4014cb:    f0 48 0f c7 0c 24    lock cmpxchg16b (%rsp)
4015e0:    f0 48 0f c7 0c 24    lock cmpxchg16b (%rsp)
401600:    f0 48 0f c7 0c 24    lock cmpxchg16b (%rsp)
401610:    f0 48 0f c7 0c 24    lock cmpxchg16b (%rsp)
401630:    f0 48 0f c7 0c 24    lock cmpxchg16b (%rsp)
401640:    f0 48 0f c7 0c 24    lock cmpxchg16b (%rsp)
401660:    f0 48 0f c7 0c 24    lock cmpxchg16b (%rsp)
401670:    f0 48 0f c7 0c 24    lock cmpxchg16b (%rsp)
401690:    f0 48 0f c7 0c 24    lock cmpxchg16b (%rsp)
4016a0:    f0 48 0f c7 0c 24    lock cmpxchg16b (%rsp)

```

Šta možemo da naučimo iz ovoga?

- Atomsko programiranje je opasno - kao što ime i sugeriše.
- Problemi su jako izazovni i kompleksni za razumevanje i zavise od hardvera koji nikako nije univerzalan
- Debagovanje je efektivno nemoguće
- Dokumentacija je vrlo štura i, recimo, primer dat na cppreference (inače vrlo pouzdan izvor) sadrži ABA problem ako se implementira.

Šta možemo da naučimo iz ovoga?

- Treba se baviti ovim samo ako ste *veoma* sigurni da znate šta radite, i ako ste uvereni da će posledice biti vredne truda.
- Naročito morate se postarati, čak i pre nego počnete, da su sledeće stvari tačne:
 - Performanse su apsolutno kritične
 - Vreme koje je neophodno za lock-ovanje u nekoj strukturi je važan faktor usporenja
 - Greška u implementaciji neće imati pogubne posledice, tj. ono što programirate sme da se sruši a da se ne desi kataklizma.

Šta možemo da naučimo iz ovoga?

- Čak i tada treba dvaput razmisliti: u zavisnosti od neverovatno suptilnih razlika u tome kako upravljate memorijom može se desiti da dobijete brži kod na x86_64 platformi (dobra uređenost) i kod koji je sporiji nego da smo koristili mutex na ARM platformi zato što je ona vrlo sklona, arhitektonski govoreći, out-of-order egzekuciji.
- Dalje, ako je to moguće, valja koristiti gotove implementacije kao što su one u standardnoj biblioteci ili u bibliotekama kao što je Folly i Abseil

Šta možemo da naučimo iz ovoga?

- Nemojte podceniti težinu rada na ovom nivou: pravljenje nečega tako prostog kao što je jednostruko linkovana lista koja je bez lock-ova a *korektna* je bio ozbiljan istraživački projekat.
- Ne samo to, eksperti i dalje znaju da se zbune: bag vezan za baš ove probleme je vrebao u Linux kernelu **deset godina**.

Pitanja

Pitanja

- Šta karakteriše semafor?
- Koje operacije su vezane za semafor?
- Kako semafor obezbeđuje sinhronizaciju međusobne isključivosti?
- Kako se obično implementira semafor?
- U čemu se semafori razlikuju od isključivih regiona?
- Koji semafori postoje?

- Šta karakteriše binarni semafor?

Pitanja

- Šta karakteriše raspodeljeni binarni semafor?
- Šta karakteriše generalni semafor?
- Šta omogućuje raspodeljeni binarni semafor?
- Šta omogućuje binarni semafor?
- Šta omogućuje generalni semafor?
- Koje su prednosti i mane semafora?

Pitanja

- Šta je lockless programiranje / programiranje bez zaključavanja?
- Šta je ABA problem?
- Šta je DWCAS i zašto je potreban?
- Šta su mane a šta prednosti programiranja bez zaključavanja / lockless programiranja?