Operativni Sistemi - Konkurentno programiranje 2

Veljko Petrović

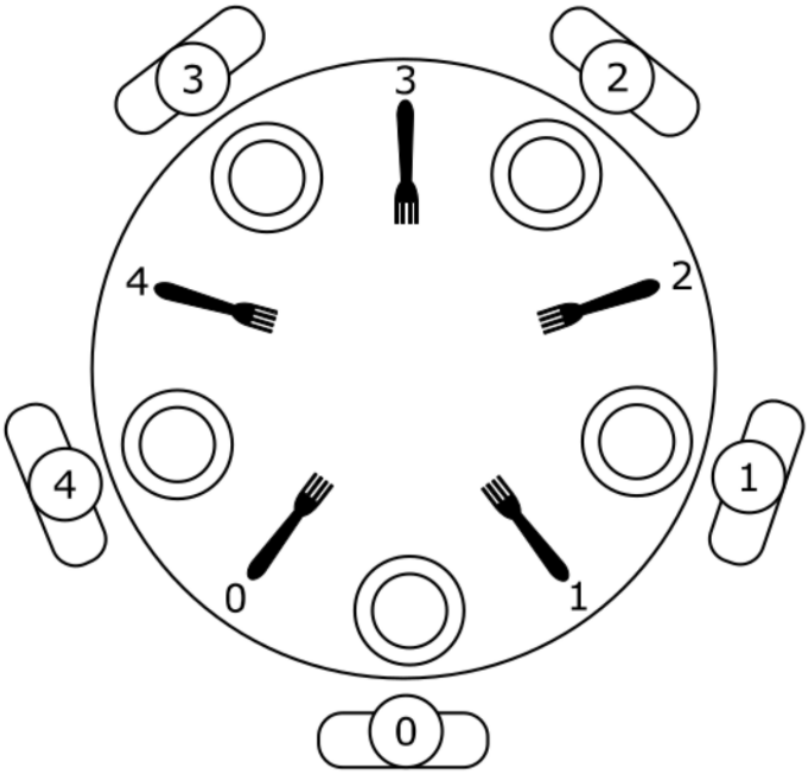
Mart, 2023

Problem Pet Filozofa

Postavka

- Zauzimanje više primeraka resursa iste vrste, neophodnih za aktivnost svake niti iz neke grupe niti, predstavlja tipičan problem konkurentnog programiranja.
- On se, u literaturi, ilustruje primerom problema pet filozofa (dining philosophers).
- Svaki od njih provodi život razmišljajući u svojoj sobi i jedući u zajedničkoj trpezariji.
- U njoj se nalazi pet stolica oko okruglog stola sa pet tanjira i pet viljuški između njih.

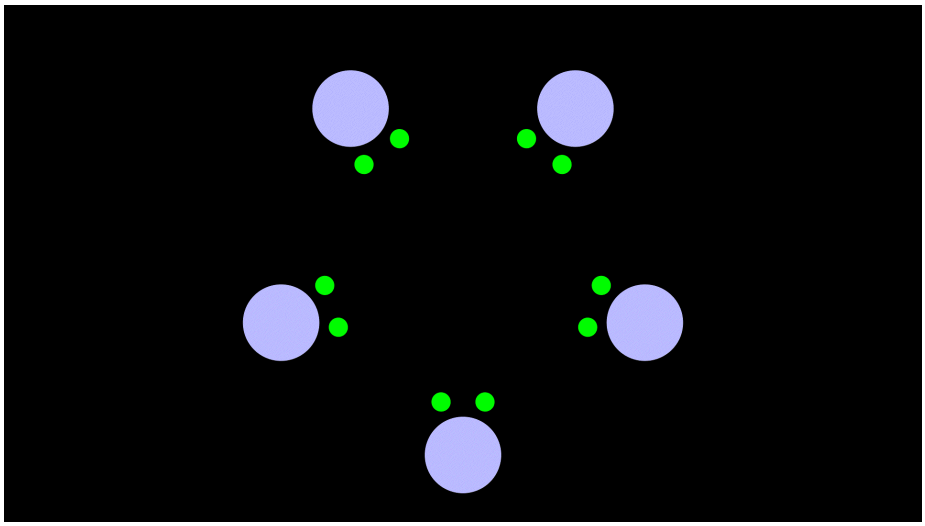
Postavka



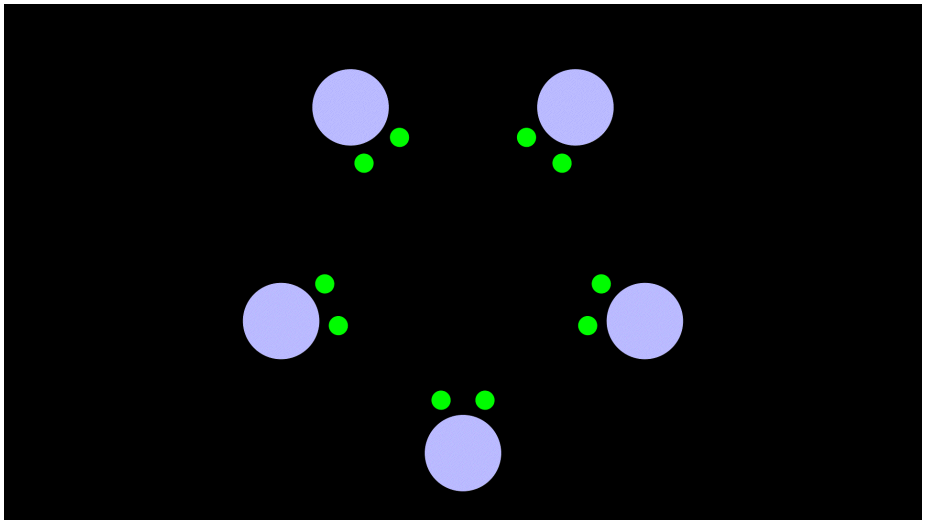
Postavka

- Pošto se, po želji filozofa, u trpezariji služe uvek špagete, svakom filozofu su za jelo potrebne dve viljuške.
- Ako svi filozofi istovremeno ogladne, uđu u trpezariju, sednu na svoje mesto za stolom i uzmu viljušku levo od sebe, tada nastupa mrtva petlja (deadlock), s kobnim ishodom po život filozofa.

Normalno ponašanje



Mrtva petlja



Simulator

- Ponašanje svakog filozofa opisuje funkcija `thread_philosopher()`.
- Razmišljanje filozofa se predstavlja kao odlaganje aktivnosti niti koja reprezentuje filozofa.
- Trajanje ovog odlaganja određuje konstanta `THINKING_PERIOD`.
- Na sličan način se predstavlja jedenje filozofa.

- Trajanje obroka filozofa određuje konstanta `EATING_PERIOD`.

Simulator

- Uzimanje viljuške pre jela i njeno vraćanje posle jela, opisuju operacije `take_fork()` i `release_fork()` klase `Dining_table`.
- Svaki filozof uzima viljuške jednu po jednu samo ako su one slobodne.
- U suprotnom, filozof čeka da svaka viljuška postane raspoloživa.
- Prilikom vraćanja viljušaka filozof oslobađa viljuške jednu po jednu i omogući nastavak aktivnosti svojih suseda.

Simulator

- Operacije `take_fork()` i `release_fork()` klase `Dining_table`, dopuštaju pojavu mrtve petlje.
- Da bi se ona sigurno desila uvedena su odlaganja aktivnosti niti, koja reprezentuje filozofa, u trajanju koje određuje konstanta `MEANTIME`.

Simulator

- Polje `fork_available` deljene klase `Dining_table` omogućuje očekivanje ispunjenja uslova da je viljuška raspoloživa, kao i objavljivanje ispunjenosti ovog uslova.
- Klasa `Dining_table` sadrži i polja `philosopher_state` i `fork_state`.
- Prvo od njih izražava stanja filozofa (`THINKING`, `WAITING_LEFT_FORK`, `HOLDING_ONE_FORK`, `WAITING_RIGHT_FORK`, `EATING`), a drugo stanja viljuški (`FREE`, `BUSY`).

Simulator

- Operacija `show()` klase `Dining_table` omogućuje prikazivanje svake promene stanja filozofa.
- Za svakog filozofa se u zagradama navode njegova numerička oznaka i njegovo stanje.
- Funkcija `mod5()` podržava modulo aritmetiku.
- U funkciji `main()` se kreiraju niti filozofi, a zatim se sačeka kraj njihove aktivnosti.
- Svaka od ovih niti preuzme svoj identitet (`Dining_table::take_identity()`): 0, 1, 2, 3 i 4 koji omogućuje razlikovanje filozofa.

Simulator kod

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

using namespace std;
using namespace chrono;
using namespace this_thread;

int mod5(int a)
{
    return (a > 4 ? 0 : a);
}

enum Philosopher_state {THINKING = 'T',
                    WAITING_LEFT_FORK = 'L',
                    HOLDING_ONE_FORK = 'O',
                    WAITING_RIGHT_FORK = 'R',
                    EATING = 'E'};

enum Fork_state {FREE, BUSY};
```

Simulator kod

```
class Dining_table {
    mutex mx;
    int philosopher_identity;
    Philosopher_state philosopher_state[5];
    Fork_state fork_state[5];
    condition_variable fork_available[5];
    void show();
public:
    Dining_table();
    int take_identity();
    void take_fork(int fork, int philosopher, Philosopher_state waiting_state,
            Philosopher_state next_state);
    void release_fork(int fork, int philosopher, Philosopher_state next_state);
};
```

Simulator kod

```
Dining_table::Dining_table(){
    philosopher_identity=0;
    for(int i = 0; i < 5; i++) {
        philosopher_state[i] = THINKING;
        fork_state[i] = FREE;
```

```
    }
}

void Dining_table::show(){
    for(int i = 0; i < 5; i++) {
        cout << '(' << (char)(i+'0') << ':'
             << (char)philosopher_state[i] << ") ";
    }
    cout << endl;
}

int Dining_table::take_identity(){
    unique_lock<mutex> lock(mx);
    return philosopher_identity++;
}
```

Simulator kod

```
void Dining_table::take_fork(int fork, int philosopher,
                             Philosopher_state waiting_state,
                             Philosopher_state next_state){
    unique_lock<mutex> lock(mx);
    if(fork_state[fork] == BUSY) {
        philosopher_state[philosopher] = waiting_state;
        show();
        do {fork_available[fork].wait(lock);}while(fork_state[fork] == BUSY);
    }
    fork_state[fork] = BUSY;
    philosopher_state[philosopher] = next_state;
    show();
}

void Dining_table::release_fork(int fork, int philosopher,
                                Philosopher_state next_state){
    unique_lock<mutex> lock(mx);
    fork_state[fork] = FREE;
    philosopher_state[philosopher] = next_state;
    show();
    fork_available[fork].notify_one();
}
```

Simulator kod

```
Dining_table dining_table;
const milliseconds THINKING_PERIOD(10);
```

```
const milliseconds MEANTIME(5);
const milliseconds EATING_PERIOD(10);
void thread_philosopher(){
    int philosopher = dining_table.take_identity();
    int fork = philosopher;
    for(;;) {
        sleep_for(THINKING_PERIOD);
        dining_table.take_fork(fork, philosopher,
                WAITING_LEFT_FORK, HOLDING_ONE_FORK);
        sleep_for(MEANTIME);
        dining_table.take_fork(mod5(fork+1), philosopher,
                WAITING_RIGHT_FORK, EATING);
        sleep_for(EATING_PERIOD);
        dining_table.release_fork(fork, philosopher, HOLDING_ONE_FORK);
        sleep_for(MEANTIME);
        dining_table.release_fork(mod5(fork+1), philosopher, THINKING);
    }
}
```

Simulator kod

```
int main(){
    cout << endl << "DINING PHILOSOPHERS" << endl;
    thread philosopher0(thread_philosopher);
    thread philosopher1(thread_philosopher);
    thread philosopher2(thread_philosopher);
    thread philosopher3(thread_philosopher);
    thread philosopher4(thread_philosopher);
    philosopher0.join();
    philosopher1.join();
    philosopher2.join();
    philosopher3.join();
    philosopher4.join();
}
```

Rezultat simulacije

```
(0:T) (1:T) (2:O) (3:T) (4:T)
(0:O) (1:T) (2:O) (3:T) (4:T)
(0:O) (1:T) (2:O) (3:O) (4:T)
(0:O) (1:O) (2:O) (3:O) (4:T)
(0:O) (1:O) (2:O) (3:O) (4:O)
(0:O) (1:O) (2:R) (3:O) (4:O)
(0:R) (1:O) (2:R) (3:O) (4:O)
(0:R) (1:O) (2:R) (3:R) (4:O)
(0:R) (1:R) (2:R) (3:R) (4:O)
(0:R) (1:R) (2:R) (3:R) (4:R)
```

Rešenje?

- Postoje mnoga rešenja ovog problema
- Najlakši način jeste da se uvede *konobar* odnosno centralni sinhronizacioni entitet koji služi da se garantuje da se viljuške uzimaju odjednom, odn. ili obe, ili nijedna.
- Alternative jeste da se razbije cikličnost tako što se primeti da imamo simetriju: uvek uzimamo levu pa desnu. Šta ako makar jedan filozof uvek prvo uzme prvo desnu pa levu ili parni filozofi uzimeju uvek levu pa desnu?

Rešenje

- Još jedan problem koji rešava problem stohastički za relativno retke petlje jeste onaj koji se koristi u određenim verzijama Ethernet
- Umesto da čekamo zauvek (`.wait`) koristimo čekanje sa `wait_for` ili `wait_until` što omogućava da se čeka do ispunjenja uslova *ili* dok ne istakne neko vreme.
- Naravno ako čekamo svi isto vreme možemo da upadnemo u petlju gde se svi ‘sudarimo’ i onda čekamo neko (isto) vreme,

pa se opet ‘sudarimo’ i tako u beskonačnost
- To se zove i ‘živa’ petlja odn. ‘livelock’

Rešenje

- Umesto da čekamo uvek isto vreme, mi oslobodimo ono što smo zauzeli, čekamo nasumično vreme, i probamo opet
- Ovo razbije simetriju i omogući da se sistem eventualno odglavi

Mrtva petlja

Pet filozofa i mrtve petlje

- Pet filozofa je namerno upečatljiv primer mrtve petlje, ali ona može da nastane na raznim mestima
- Svaki višenitni program je ranjiv
- Takođe svaki operativni sistem koji zaključava fajlove za pristup
- Ako se steknu uslovi, onda gdegod da imamo zaključavanje i višestruko izvršavanje, mrtva petlja je moguća

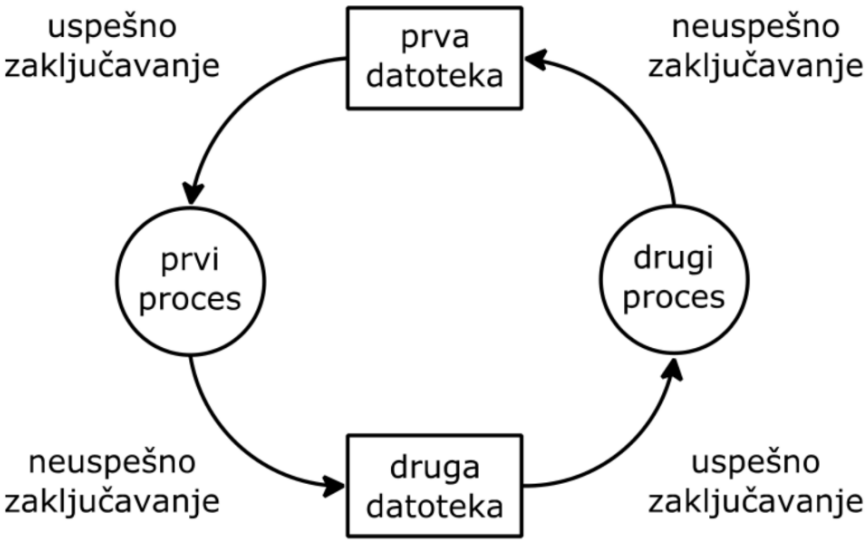
Uslovi za pojavu mrtve petlje

- Mrtva petlja je problematična pojava trajnog zaustavljanja aktivnosti međusobno zavisnih procesa.
- Na primer, ona se javi kada dva procesa žele da u režimu međusobne isključivosti pristupaju dvema datotekama.
- Ako prvi od njih zaključa prvu datoteku, a zatim drugi od njih zaključa drugu datoteku, tada nema mogućnosti za nastavak aktivnosti tih procesa, bez obzira da li je sistemska operacija zaključavanja blokirajuća ili ne.

Uslovi za pojavu mrtve petlje

- U slučaju blokirajućih sistemskih operacija zaključavanja, pokušaj prvog procesa da zaključa drugu datoteku dovodi do trajnog zaustavljanja njegove aktivnosti.
- Isto se dešava sa drugim procesom prilikom njegovog pokušaja da zaključa prvu datoteku.
- Zaustavljanje aktivnosti ova dva procesa je trajno, jer je svaki od njih zauzeo datoteku koja treba onom drugom procesu za nastavak njegove aktivnosti i nema nameru da tu datoteku oslobodi.

Uslovi za pojavu mrtve petlje



Uslovi za pojavu mrtve petlje

- U slučaju neblokirajuće sistemske operacije zaključavanja, procesi upadaju u beskonačnu petlju (starvation), pokušavajući da zaključaju datoteku, koju je zaključao drugi proces.
- Ovakav oblik međusobne zavisnosti procesa se naziva i živa petlja (livelock).
- Ona se, po svom ishodu, suštinski ne razlikuje od mrtve petlje.

Uslovi za pojavu mrtve petlje

- Pojava mrtve petlje je vezana za zauzimanje resursa, kao što su, na primer, prethodno pomenute datoteke.
- Pri tome, za pojavu mrtve petlje je potrebno da budu ispunjena četiri uslova.

Uslovi za pojavu mrtve petlje

1. zauzimani resursi se koriste u režimu međusobne isključivosti
2. resursi se zauzimaju jedan za drugim, tako da proces, nakon zauzimanja izvesnog broja resursa, mora da čeka da zauzme preostale resurse
3. resurse oslobađaju samo procesi koji su ih zauzeli

4. postoji cirkularna međuzavisnost procesa (prvi proces čeka oslobađanje resursa koga drži drugi proces, a on čeka oslobađanje resursa koga drži treći proces, i tako redom do poslednjeg procesa iz lanca procesa, koji čeka oslobađanje resursa koga drži prvi proces).

Tretiranje mrtve petlje

- Postoje četiri pristupa tretiranja problema mrtve petlje:
- sprečavanje (prevention) pojave mrtve petlje (onemogućavanjem važenja nekog od četiri neophodna uslova za njenu pojavu)
- izbegavanje (avoidance) pojave mrtve petlje
- otkrivanje (detection) pojave mrtve petlje i oporavak (recovery) od nje
- ignorisanje pojave mrtve petlje.

Tretiranje mrtve petlje

- Kod sprečavanja pojave mrtve petlje, važenje prvog uslova obično nije moguće sprečiti, jer se resursi najčešće koriste u režimu međusobne isključivosti.
- Važenje drugog uslova se može sprečiti, ako se unapred zna koliko treba resursa i ako se oni svi zauzmu pre korišćenja.
- Pri tome, neuspeh u zauzimanju bilo kog resursa dovodi do oslobađanja prethodno zauzetih resursa, što je moguće učiniti bez posledica, jer nije započelo njihovo korišćenje.

Tretiranje mrtve petlje

- Važenje trećeg uslova se obično ne može sprečiti, jer najčešće ne postoji način da se zauzeti resurs privremeno oduzme procesu.
- I konačno, važenje četvrtog uslova se može sprečiti, ako se resursi uvek zauzimaju u unapred određenom redosledu, koji isključuje mogućnost cirkularne međuzavisnosti procesa (primer pozitivnog rešenja 5 filozofa).

Tretiranje mrtve petlje

- Izbegavanje pojave mrtve petlje zahteva poznavanje podataka:
  - o maksimalno mogućim zahtevima za resursima
  - ukupno postavljenim zahtevima za resursima
  - stanju resursa
- Podrazumeva se da se udovoljava samo onim zahtevima za koje se proverom ustanovi da, nakon njihovog ispunjavanja, postoji

redosled zauzimanja i oslobađanja resursa u kome se mogu zadovoljiti maksimalno mogući zahtevi svih procesa.

Tretiranje mrtve petlje

- Praktična vrednost ovoga pristupa nije velika, jer se obično unapred ne znaju maksimalno mogući zathevi procesa za resursima, a to je neophodno za proveru da li se može udovoljiti pojedinim zahtevima.
- Sem toga, ovakva provera je komplikovana, a to znači i neefikasna.

Tretiranje mrtve petlje

- Otkrivanje pojave mrtve petlje se zasniva na sličnom pristupu kao i izbegavanje pojave mrtve petlje.
- U ovom slučaju se proverava da li postoji proces, čijim zahtevima se ne može udovoljiti ni za jedan redosled zauzimanja i oslobađanja resursa.
- Pored komplikovanosti ovakve provere, problem je i šta učiniti, kada se i otkrije pojava mrtve petlje.

Tretiranje mrtve petlje

- Ako se resursi ne mogu privremeno oduzeti od procesa, preostaje jedino uništavanje procesa, radi oslobađanja resursa koje oni drže.
- Međutim, to nije uvek prihvatljiv zahvat.
- Zbog toga ni ovaj pristup nema veliki praktični značaj.

Tretiranje mrtve petlje

- Ignorisanje pojave mrtve petlje je pristup koji je najčešće primenjen u praksi, jer se ovaj problem ne javlja tako često da bi se isplatilo da ga rešava operativni sistem.
- Prema tome, kada se mrtva petlja javi, na korisniku je da se suoči sa ovim problemom i da ga reši na način, koji je primeren datim okolnostima.

Problem čitanja i pisanja

Postavka

- Problem čitanja i pisanja (readers-writers problem) se može objasniti na primeru kao što je rukovanje bankovnim računima.
- Bankovni računi pripadaju komitentima banke i sadrže ukupan iznos novčanih sredstava svakog od komitenata.
- U najjednostavnijem slučaju, rukovanje bankovnim računima se svodi: na prenos sredstava (s jednog računa na drugi) i na proveru (stanja svih) računa.

Postavka

- Prenos sredstava obuhvata četiri koraka:
  - Čitanje stanja računa s koga se prenose sredstva
  - Pisanje novog stanja na ovaj račun. Novo stanje se dobije umanjivanjem pročitanog stanja za prenošeni iznos
  - Čitanje stanja računa na koji se prenose sredstva.
  - Pisanje novog stanja na račun na koji se prenose sredstva. Novo stanje se dobije uvećavanjem pročitanog stanja za prenošeni iznos

Postavka

- Uz pretpostavku da su prenosi sredstava mogući samo između posmatranih bankovnih računa, ukupna suma njihovih stanja je nepromenljiva.
- Prema tome, provera računa se svodi na čitanja, jedno za drugim, stanja svih računa, radi njihovog sumiranja.
- Ispravnost prenosa sredstava zavisi od očuvanja konzistentnosti stanja svih računa, za šta je neophodna međusobna isključivost raznih prenosa sredstava. U suprotnom, moguće su razne greške.

Postavka

- Iz prethodne analize sledi:
  - Da je za ispravnost prenosa bitno da prenosi budu međusobno isključivi.
  - Da je za ispravnost provera bitno da provere i prenosi budu međusobno isključivi.
- Pošto prenosi sadrže pisanja, a provere samo čitanja, sledi da operacije sa pisanjem moraju biti međusobno isključive, kao što moraju biti međusobno isključive operacije sa pisanjem i operacije sa čitanjem.

- Za operacije koje sadrže samo čitanja međusobna isključivost nije potrebna.

Kod

```
#include<thread>
#include<iostream>
using namespace std;
using namespace chrono;
using namespace this_thread;
const unsigned ACCOUNTS_NUMBER = 10;
const int INITIAL_AMOUNT = 100;
class Bank {
    mutex mx;
    int accounts[ACCOUNTS_NUMBER];
    short readers_number;
    short writers_number;
    short readers_delayed_number;
    short writers_delayed_number;
    condition_variable readers_q;
    condition_variable writers_q;
```

Kod

```
    void show();
    void reader_begin();
    void reader_end();
    void writer_begin();
    void writer_end();
public:
    Bank();
    void audit();
    void transaction(unsigned source, unsigned destination);
};

Bank::Bank(){
    for(unsigned i = 0; i < ACCOUNTS_NUMBER; i++)
        accounts[i] = INITIAL_AMOUNT;
    readers_number = 0;
    writers_number = 0;
    readers_delayed_number = 0;
    writers_delayed_number = 0;
}
```

Kod

```
void Bank::show(){
    cout << "RN: " << readers_number << " RDN: "
         << readers_delayed_number << " WN: "
         << writers_number << " WDN: "
         << writers_delayed_number << endl;
}

void Bank::reader_begin(){
    unique_lock<mutex> lock(mx);
    if((writers_number > 0) || (writers_delayed_number > 0){
        readers_delayed_number++;
        show();
        do { readers_q.wait(lock); }
        while((writers_number > 0) ||
              (writers_delayed_number > 0));
    }
```

Kod

```
    readers_number++;
    show();
    if(readers_delayed_number > 0){
        readers_delayed_number--;
        show();
        readers_q.notify_one();
    }
}

void Bank::reader_end(){
    unique_lock<mutex> lock(mx);
    readers_number--;
    show();
    if((readers_number == 0) && (writers_delayed_number > 0)){
        writers_delayed_number--;
        show();
        writers_q.notify_one();
    }
}
```

Kod

```
void Bank::writer_begin(){
    unique_lock<mutex> lock(mx);
    if((readers_number > 0) || (writers_number > 0)){
```

```
        writers_delayed_number++;
        show();
        do { writers_q.wait(lock); }
        while((readers_number > 0) || (writers_number > 0));
    }
    writers_number++;
    show();
}
```

Kod

```
void Bank::writer_end(){
    unique_lock<mutex> lock(mx);
    writers_number--;
    show();
    if(writers_delayed_number > 0){
        writers_delayed_number--;
        show();
        writers_q.notify_one();
    } else if(readers_delayed_number > 0) {
        readers_delayed_number--;
        show();
        readers_q.notify_one();
    }
}
```

Kod

```
const milliseconds WRITING_PERIOD(1);
void Bank::transaction(unsigned source,unsigned destination){
    int amount;
    writer_begin();
    sleep_for(WRITING_PERIOD);
    amount = accounts[source];
    accounts[source] -= amount;
    accounts[destination] += amount;
    writer_end();
}
Bank bank;
void thread_reader(){
    bank.audit();
}
```

Kod

```
const milliseconds READING_PERIOD(1);
void Bank::audit(){
    int sum = 0;
    reader_begin();
    sleep_for(READING_PERIOD);
    for(unsigned i = 0; i < ACCOUNTS_NUMBER; i++)
        sum += accounts[i];
    reader_end();
    if(sum != ACCOUNTS_NUMBER*INITIAL_AMOUNT) {
        unique_lock<mutex> lock(mx);
        cout << " audit error " << endl;
    }
}
```

Kod

```
const milliseconds WRITING_PERIOD(1);
void Bank::transaction(unsigned source,unsigned destination){
    int amount;
    writer_begin();
    sleep_for(WRITING_PERIOD);
    amount = accounts[source];
    accounts[source] -= amount;
    accounts[destination] += amount;
    writer_end();
}
Bank bank;
void thread_reader(){
    bank.audit();
}
```

Kod

```
void thread_writer0to1(){
    bank.transaction(0, 1);
}
void thread_writer1to0(){
    bank.transaction(1, 0);
}
int main(){
    cout << endl << "READERS AND WRITERS" << endl;
    thread reader0(thread_reader);
    thread reader1(thread_reader);
    thread writer0(thread_writer0to1);
```

```
    thread reader2(thread_reader);
    thread writer1(thread_writer1to0);
    reader0.join();
    reader1.join();
    writer0.join();
    reader2.join();
    writer1.join();
}
```

Komentar o rešenju

- Ovo rešenje, eksplicitno, je napravljeno tako da pisanje ima prednost u odnosu na čitanje
- Ako je plan da bude obrnuto onda su potrebne promene

Rizici konkurentnog programiranja

Rizici

- Opisivanje obrada podataka je jedini cilj sekvencijalnog, a osnovni cilj konkurentnog programiranja.
- Bolje iskorišćenje računara i njegovo čvršće sprezanje sa okolinom su dodatni ciljevi konkurentnog programiranja, po kojima se ono i razlikuje od sekvencijalnog programiranja.
- Od suštinske važnosti je da ostvarenje dodatnih ciljeva ne ugrozi ostvarenje osnovnog cilja, jer je on neprikosnoven, pošto je konkurentni program upotrebljiv jedino ako iza svakog od njegovih izvršavanja ostaju samo ispravno obrađeni podaci.

Rizici

- Tipične poželjne osobine sekvencijalnog, a to znači i konkurentnog programa obuhvataju tvrdnje uključivanja poželjnog, kao što je tvrdnja da nakon izvršavanja programa ostaju ispravno obrađeni podaci, i tvrdnje isključivanja nepoželjnog, kao što je tvrdnja da program ne sadrži beskonačne petlje.
- Tipične dodatne poželjne osobine konkurentnog programa obuhvataju tvrdnje uključivanja poželjnog, kao što je tvrdnja da su, u toku izvršavanja programa, deljene promenljive stalno konzistentne, i tvrdnje isključivanja nepoželjnog, kao što je tvrdnja da u toku izvršavanja programa ne dolazi do trajnog zaustavljanja aktivnosti niti.

Rizici

- Ispravnu obradu podataka ugrožava narušavanje konzistentnosti deljenih promenljivih u toku izvršavanja konkurentnog programa.
- Do narušavanja konzistentnosti deljenih promenljivih dolazi, ako na kraju isključivog regiona deljena promenljiva nije u konzistentnom stanju ili ako se operacija wait() pozove pre nego je deljena promenljiva dovedena u konzistentno stanje

Pozivanje `wait` pre dovođenja u konzistentno stanje

```
{
    unique_lock<mutex> lock(mx);
    //< exclusive region 1 >
    some_condition.wait(lock);
    //< exclusive region 2 >
}
```

Mrtva petlja među nitima

- Upotrebljivost konkurentnih programa ugrožava i pojava međuzavisnosti niti, poznata pod nazivom mrtva petlja.
- Ona dovodi do trajnog zaustavljanja aktivnosti niti, a to ima za posledicu da izvršavanje konkurentnog programa nema kraja.
- Konkurentni program, u toku čijeg izvršavanja je moguća pojava mrtve petlje, nije upotrebljiv, jer pojedina od njegovih izvršavanja, koja nemaju kraja, ne dovode do uspešne obrade podataka.
- Do mrtve petlje može da dođe, na primer, ako se iz jedne deljene klase pozivaju operacije druge deljene klase, pod uslovom da je bar jedna od pozivanih operacija blokirajuća.

Minimalni primer mrtve petlje

```
class Activity {
    mutex mx_activity;
    condition_variable activity_permission;
public:
    void stop();
    void start();
};
void Activity::stop(){
    unique_lock<mutex> lock(mx_activity);
    activity_permission.wait(lock);
```

```
}
void Activity::start(){
    unique_lock<mutex> lock(mx_activity);
    activity_permission.notify_one();
}
```

Minimalni primer mrtve petlje

```
class Manager {
    mutex mx_manager;
    Activity activity;
public:
    void disable_activity();
    void enable_activity();
};
void Manager::disable_activity(){
    unique_lock<mutex> lock(mx_manager);
    activity.stop();
}
void Manager::enable_activity(){
    unique_lock<mutex> lock(mx_manager);
    activity.start();
}
Manager manager;
```

Minimalna mrtva petlja

- Mrtve petlje, koje ilustruje prethodni primer, se mogu sprečiti, ako se blokirajuća operacija ne poziva iz isključivog regiona.
- Nenamerno izazivanje konačnog, ali nepredvidivo dugog zaustavljanja aktivnosti niti u toku isključivog regiona može da ima negativne posledice na izvršavanje programa.
- To se, na primer, desi, kada se iz isključivog regiona pozivaju potencijalno blokirajuće operacije, poput funkcije `sleep_for()`.
- Globalne const promenljive, koje služe za smeštanje podataka, raspoloživih svim nitima, ne spadaju u deljene promenljive.

Pitanja

Pitanja

- Opisati problem pet filozofa.
- Kako bi izgledala verzija problema pet filozofa koja bi se realistično mogla sresti tokom razvoja softvera?

- Šta je mrtva petlja?
- Po čemu se živa petlja razlikuje od mrtve petlje?207. Koji uslovi su potrebni za pojavu mrtve petlje?
- Kako se u praksi tretira problem mrtve petlje?
- Na čemu se temelji sprečavanje mrtve petlje?

Pitanja

- Šta karakteriše izbegavanje mrtve petlje?
- Šta karakteriše otkrivanje i oporavak od mrtve petlje?
- Šta karakteriše ignorisanje mrtve petlje?
- Opisati problem čitanja i pisanja.