

Konkurentno programiranje

UDŽBENIK, STRANICE 11-50

Konkurentno programiranje

•Svojstva konkurentnih programa:

Mešanje izvršavanja raznih niti i obrađivača prekida naziva se **preplitanje (interleaving)**.

Preplitanje niti i obrada prekida imaju **slučajan karakter** jer unapred nije poznato kada će se desiti **prekid** i **preključivanje**.

Stohastično izvršavanje konkurentnih programa može da menja rezultate izvršavanja od slučaja do slučaja, što može dovesti do pojave **štetnog preplitanja**.

Primeri štetnog preplitanja

UDŽBENIK, STRANICE 11-16

Primeri štetnog preplitanja

- Primeri štetnog preplitanja su mogući i u OS (sistemski pozivi su niti i mogu se preplitati sa obrađivačima prekida).
- Rukovanje pozicijom kursora je dobar primer štetnog preplitanja.

Rukovanje pozicijom kursora

```
class Position {
    int x, y;
public:
    Position();
    void set(int new_x,
            int new_y);
    void get(int* current_x,
            int* current_y);
};

Position::Position()
{
    x = 0;
    y = 0;
}

void
Position::set(int new_x, int new_y)
{
    x = new_x;
    y = new_y;
}
```

```
void
Position::get(int* current_x,
int* current_y)
{
    *current_x = x;
    *current_y = y;
}

Position position;
```

Rukovanje pozicijom kursora

- Neka operacija `position.set()` bude na raspolaganju samo **obrađivaču prekida** koji u okviru operativnog sistema registruje izmene pozicije kursora.
- Neka operacija `position.get()` bude na raspolaganju procesima iz **korisničkog** sloja.
- Tada je moguće da u toku izvršavanja operacije `position.get()` proces bude prekinut radi obrade prekida, koja poziva operaciju `position.set()`.
- Ako se to desi **nakon** izvršavanja **prvog** iskaza dodele iz tela operacije `position.get()` a **pre** izvršavanja **drugog** iskaza dodele iz njenog tela, tada će rezultat izvršavanja ove operacije biti **pogrešan**.

Rukovanje slobodnim baferima

```
struct List_member {
    List_member* next;
    char buffer[512];
};

class List {
    List_member* first;
public:
    List() : first(0) {};
    void link(List_member* member);
    List_member* unlink();
};

void
List::link(List_member* member)
{
    member->next=first;
    first=member;
}

List_member*
List::unlink()
{
    List_member* unlinked;
    unlinked=first;
    if(first != 0)
        first=first->next;
    return unlinked;
}

List list;
```

Rukovanje slobodnim baferima

- Neka su operacije `list.link()` i `list.unlink()` na raspolaganju samo modulu za rukovanje datotekama i neka se pozivaju iz operacija ovog modula.
- Tada je moguće da izvršavanje operacije `list.unlink()` bude pokrenuto u toku aktivnosti niti nekog procesa u modulu za rukovanje datotekama.
- Kao rezultat izvršavanja ove operacije na **steku** pomenute niti nastane primerak njene **lokalne** promenljive **unlinked**.
- Izvršavanje iskaza:
`.unlinked = first`
- smešta u ovaj primerak lokalne promenljive **adresu prvog slobodnog bafera** iz liste bafera, jasno, kada takav bafer postoji.

Rukovanje slobodnim baferima

- Neka, nakon izvršavanja prethodnog iskaza, pod uticajem obrade prekida dođe do preključivanja procesora na nit drugog procesa. Tada, u toku aktivnosti niti ovog procesa, može doći do pokretanja **još jednog** izvršavanja operacije `list.unlink()`.
- Tako na **steku i ove druge niti** nastaje njen primerak lokalne promenljive **unlinked**.
- Posledica ovakvog sleda događaja je da posmatrana **dva procesa** koriste **isti bafer**. To neminovno dovodi do **fatalnog** ishoda.
- Prema tome, **preplitanje** dva izvršavanja operacije `list.unlink()`, je **štetno**. Isto važi i za preplitanje izvršavanja operacija `list.link()` i `list.unlink()` kao i `list.link()` i `list.link()`

Rukovanje komunikacionim baferom

- Štetna međusobna preplitanja niti su moguća i za niti koje pripadaju **istom procesu**.
- Ovakva saradnja se može ostvariti tako što jedna od niti **šalje** podatke drugoj niti.
- Takva razmena podataka između niti se obično obavlja posredstvom **komunikacionog bafera**.
- Nit koja puni ovaj bafer podacima ima ulogu **proizvođača** (podataka), a nit koja prazni ovaj bafer ima ulogu **potrošača** (podataka).
- U pojednostavljenom slučaju, rukovanje komunikacionim baferom obuhvata punjenje: **put()** celog bafera, kao i pražnjenje: **get()** celog bafera.

Rukovanje komunikacionim baferom

```
const unsigned int
BUFFER_SIZE = 512;

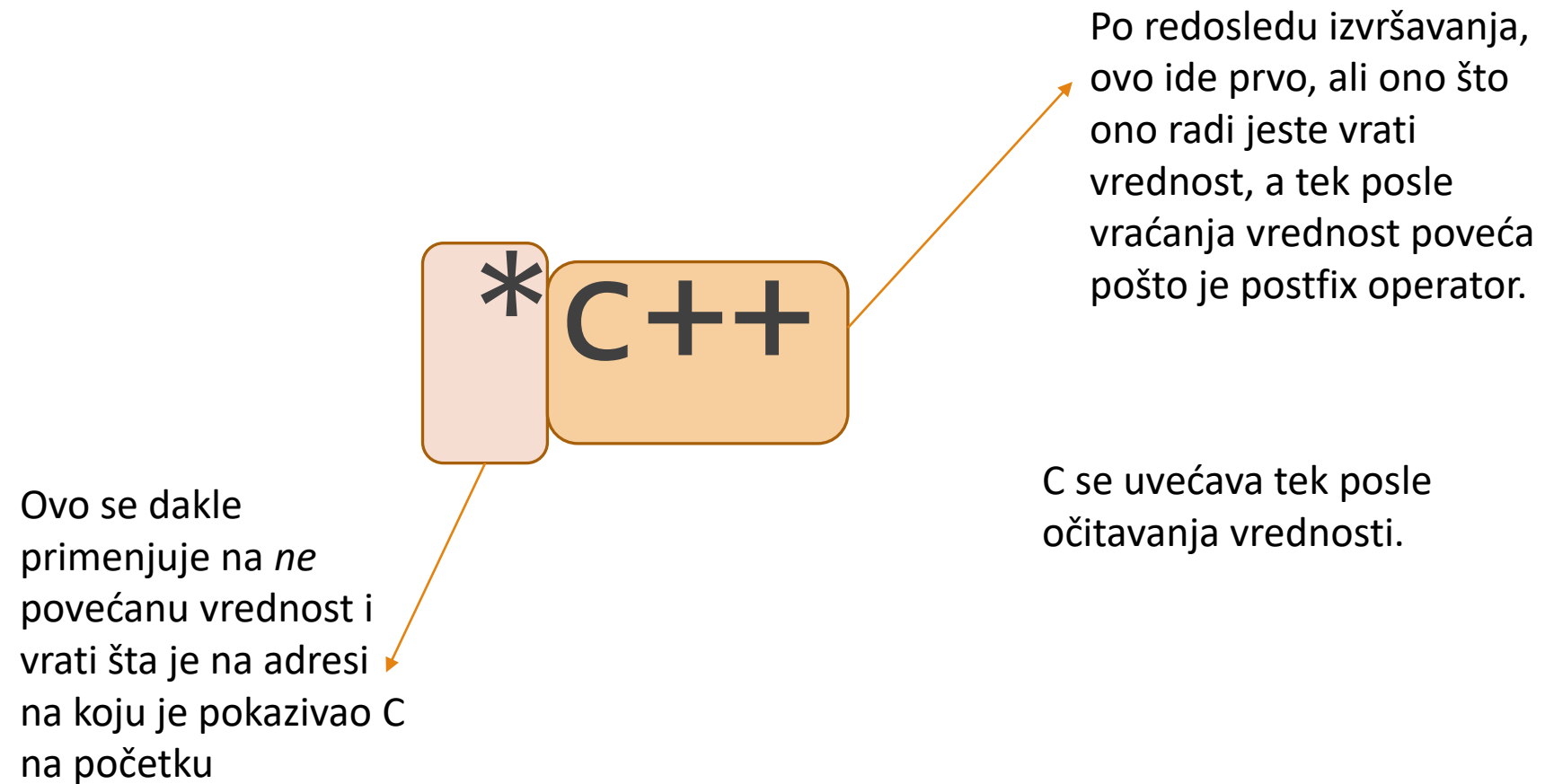
class Buffer {
    char content[BUFFER_SIZE];
public:
    Buffer() {};
    void put(char* c);
    void get(char* c);
};

void
Buffer::put(char* c)
{
    unsigned int i;
    for(i = 0; i < BUFFER_SIZE;
        i++)
        content[i] = *c++;
}

void
Buffer::get(char* c)
{
    unsigned int i;
    for(i = 0; i < BUFFER_SIZE; i++)
        *c++ = content[i];
}

Buffer buffer;
```

C haiku



Rukovanje komunikacionim baferom

- Operaciju **buffer.put()** poziva nit proizvođač.
- Operaciju **buffer.get()** poziva nit potrošač.
- U ovoj situaciji moguće je, da se desi **prekid sata** za vreme aktivnosti proizvođača u operaciji **buffer.put()**.
- Ako operacija **buffer.get()** bude pozvana u toku aktivnosti potrošača, tada postoji mogućnost da potrošač preuzme sadržaj delimično popunjenog bafera.

Sprečavanje štetnih preplitanja

UDŽBENIK STRANICE 16-18

Međusobna isključivost

- Promenljive kojima pristupaju više niti ili niti i obrađivači prekida istovremeno (**position**, **list** i **buffer**) se nazivaju **deljene promenljive**, a klase koje opisuju te promenljive **deljene klase**.
- Deljene klase su pravljene pod pretpostavkom da se rukovanja deljenim promenljivim obavljaju **sekvencijalno** odnosno strogo **jedna za drugom**.
- **Štetna preplitanja** negiraju pomenutu pretpostavku jer dopuštaju da novo izvršavanje neke operacije deljene promenljive započne **pre nego što se završilo** već započeto izvršavanje neke od operacija te promenljive.
- Problem štetnih preplitanja ne postoji ako se obezbedi **međusobna isključivost (mutual exclusion)** izvršavanja operacija deljenih promenljivih.

Kritične sekcije i sinhronizacija

- **Tela operacija** deljenih klasa ili delovi ovih tela, čije izvršavanje je **kritično** za konzistentnost deljenih promenljivih, se nazivaju **kritične sekcije**.
- Međusobna isključivost kritičnih sekcija se ostvaruje **sinhronizacijom** pristupa.
- Pored obične sinhronizacije postoji i **uslovna sinhronizacija**.

Atomski regioni

- U primeru rukovanja pozicijom kursora, štetna preplitanja nastupaju kao posledica **obrade prekida**.
- **Atomski regioni** omogućavaju **neprekidnost** izvršavanja unutar kritičnih sekcija deljene promenljive.
- Onemogućenje prekida odlaže obradu novih prekida i usporava reakciju procesora, pa atomski regioni treba da budu **što kraći**.

Propusnice i isključivi regioni

- **Propusnice** su drugi način ostvarenja međusobne isključivosti kritičnih sekcija.
- Propusnica može biti **slobodna** ili **zauzeta**.
- Samo jedna nit od svih niti koje se takmiče za propusnicu dobija istu i ulazi u kritičnu sekciju, dok sve ostale niti zaustavljaju svoju aktivnost i prelaze u stanje “**čeka**”.
- Kada nit koja napušta kritičnu sekciju oslobodi propusnicu (vrati), neka sledeća nit dobija propusnicu i prelazi iz stanja “**čeka**” u stanje “**spremna**”.
- Nit, koja tada **dobije propusnicu**, odmah prelazi iz stanja “**čeka**” u stanje “**spremna**”, ali u kritičnu sekciju ulazi tek kada postane **aktivna** (odnosno, kada se **procesor preključi na nju**).

Propusnice i isključivi regioni

- Rukovanje **propusnicom** deljene promenljive je, takođe, **ugroženo štetnim preplitanjima**.
- Zbog toga se **konzistentnost** propusnica mora zaštititi.
- Za to se obično koriste **atomske regioni**, jer su rukovanja propusnicama **kratkotrajna**.
- Po načinu ostvarenja međusobne isključivosti, **atomske regioni** se **razlikuju** od onih kritičnih sekcija koje međusobnu isključivost ostvaruju korišćenjem **propusnica**.
- Zato ove druge sekcije treba drugačije nazvati, na primer, **isključivi regioni**.

Propusnice i isključivi regioni

- Interesantno je uočiti da **preključivanja** u isključivim regionima **omogućuju pojavu štetnih preplitanja**, ali i **omogućuju njihovo sprečavanje**.
- Tako, ako nit, čiju aktivnost je omogućilo **preključivanje**, pokuša da uđe u isključivi region deljene promenljive sa **zauzetom propusnicom**, tada opet **preključivanje** dovodi do **zaustavljanja aktivnosti** ove niti.
- Prema tome, **prvo preključivanje** je stvorilo uslove za **pojavu štetnog preplitanja**, a **drugo preključivanje** je sprečilo tu pojavu.

Poželjne osobine konkurentnih programa

- Poželjne osobine konkurentnih programa uvode:

- Tvrdnju **isključivanja nepoželjnog (safety property)**.

- Tvrdnju **uključivanja poželjnog (liveness property)**.

- Primer tvrdnje **isključivanja nepoželjnog** je tvrdnja da se u izvršavanjima konkurentnog programa **ne javlja nekonzistentnost date deljene promenljive**.

- Primer tvrdnje **uključivanja poželjnog** je tvrdnja da se u toku izvršavanja konkurentnog programa **dese svi zatraženi ulasci u dati isključivi region**.

Poželjne osobine konkurentnih programa

- Za konkurentni program se može reći da ima neku od poželjnih osobina samo ako se dokaže (na neformalan ili formalan način) da važi tvrdnja kojoj pomenuta osobina odgovara.
- Ovakvo rezonovanje o ispravnosti konkurentnog programa je neophodno, jer slučajna priroda preplitanja može da bude uzrok nedeterminističkog (nepredvidivog) izvršavanja konkurentnog programa.

Poželjne osobine konkurentnih programa

- U takvoj situaciji, **ponovljena izvršavanja** konkurentnog programa, u toku kojih se obrađuju isti podaci, **ne moraju da imaju isti ishod**.
- Zbog toga, provera ispravnosti konkurentnog programa ne može da se zasniva samo na pokazivanju da **pojedina izvršavanja konkurentnog programa** imaju ispravan rezultat, jer tačan rezultat, dobijen u jednom ili više izvršavanja, **ne isključuje mogućnost postojanja izvršavanja** koja za iste ulazne podatke daju netačan rezultat.

Programski jezici za konkurentno programiranje

- Konkurentno programiranje se razlikuje od sekvencijalnog po rukovanju **nitima** i **deljenim promenljivama**.
- Konkurentni programski jezik može nastati kao rezultat pravljenja potpuno novog programskog jezika ili kao rezultat proširenja postojećeg sekvencijalnog programskog jezika **konkurentnim iskazima**.
- **Konkurentna biblioteka** omogućuje da se za konkurentno programiranje koristi već postojeći, poznat programski jezik.

Uvod u konkurentnu biblioteku koja implementira deo međunarodnog standarda C++11

- Međunarodni standard **C++11** predviđa rukovanje **nitima** i **deljenim promenljivama**.
- **CppTss** (C plus plus Thread subset) izlaže **podskup** načina za rukovanje nitima i deljenim promenljivama koje predviđa C++11.
- Za označavanje pomenutog **podskupa** u ovoj knjizi se koristi skraćenica **CppTss** (**C plus plus Thread subset**)

Klasa thread

```
void thread_example()  
{  
    double pi;  
    cout << "ZADAJ VREDNOST BROJA PI" << endl;  
    cin >> pi;  
    cout << endl << "PI = " << pi << endl;  
}  
  
int main()  
{  
    thread example(thread_example);  
    example.join();  
    //example.detach();  
}
```

Pojava štetnog preplitanja kod kreiranja više niti

```
int main()  
{  
    thread example1(thread_example);  
    thread example2(thread_example);  
    example1.join();  
    example2.join();  
}
```

Sprečavanje štetnog preplitanja kod kreiranja više niti (mutex)

```
mutex terminal;
```

```
void thread_example()  
{  
    double pi;  
    terminal.lock();  
    cout << "ZADAJ VREDNOST BROJA PI" << endl;  
    cin >> pi;  
    cout << endl << "PI = " << pi << endl;  
    terminal.unlock();  
}
```

```
void thread_example()  
{  
    double pi;  
    unique_lock<mutex> lock(terminal);  
    cout << "ZADAJ VREDNOST BROJA PI" << endl;  
    cin >> pi;  
    cout << endl << "PI = " << pi << endl;  
}
```

Sprečavanje štetnog preplitanja kod kreiranja više niti (unique_lock)

```
mutex terminal;

void thread_example()
{
    double pi;
    unique_lock<mutex> lock(terminal);
    cout << "ZADAJ VREDNOST BROJA PI" << endl;
    cin >> pi;
    cout << endl << "PI = " << pi << endl;
}
```

Uslovna sinhronizacija

- Ulazak u isključivi region nije moguć ako je **propusnica zauzeta** ili ako se ustanovi da traženi **uslov nije ispunjen**.
- Kada druga nit **ispuni uslov** ona **objavljuje ispunjenje traženog uslova** i omogućuje nastavak aktivnosti prve niti.
- Za ovakvu sinhronizaciju koristi se klasa **condition_variable**, koja nudi operacije **wait()**, **notify_one()** i **notify_all()**.
- Nakon aktiviranja niti koja je očekivala ispunjenje nekog uslova potrebno je da nit **ponovo proveri** da li taj uslov važi, jer je moguće **lažno buđenje** niti .

Sprečavanje štetnih preplitanja prilikom rukovanja slobodnim baferima (neblokirajuća verzija)

```
struct List_member {  
    List_member* next;  
    char buffer[512];  
};
```

```
class List {  
    mutex mx;  
    List_member* first;  
public:  
    List() : first(0) {};  
    void link(List_member* member);  
    List_member* unlink();  
};
```

```
void  
List::link(List_member* member)  
{  
    unique_lock<mutex> lock(mx);  
    member->next=first;  
    first=member;  
}
```

```
List_member*  
List::unlink()  
{  
    List_member* unlinked;  
    {  
        unique_lock<mutex> lock(mx);  
        unlinked=first;  
        if(first != 0)  
            first=first->next;  
    }  
    return unlinked;  
}
```

Sprečavanje štetnih preplitanja prilikom rukovanja slobodnim baferima (blokirajuća verzija)

```
struct List_member {
    List_member* next;
    char buffer[512];
};

class List {
    mutex mx;
    List_member* first;
    condition_variable nonempty;
public:
    List() : first(0) {};
    void link(List_member* member);
    List_member* unlink();
};

void
List::link(List_member* member)
{
    unique_lock<mutex> lock(mx);
    member->next=first;
    first=member;
    nonempty.notify_one();
}
```

```
List_member*
List::unlink()
{
    List_member* unlinked;
    {
        unique_lock<mutex> lock(mx);
        while (first == 0)
            nonempty.wait(lock);
        unlinked=first;
        first=first->next;
    }
    return unlinked;
}
```


Sprečavanje štetnih preplitanja prilikom rukovanja komunikacionim baferom

```
const unsigned int
BUFFER_SIZE = 512;

enum Buffer_states
{EMPTY, FULL};

class Buffer {
    mutex mx;
    char content[BUFFER_SIZE];
    Buffer_states state;
    condition_variable full;
    condition_variable empty;
public:
    Buffer() {state = EMPTY;};
    void put(char* c);
    void get(char* c);
};
```

```
void Buffer::put(char* c)
{
    unsigned int i;
    unique_lock<mutex> lock(mx);
    while (state == FULL)
        empty.wait(lock);
    for(i = 0; i < BUFFER_SIZE; i++)
        content[i] = *c++;
    state = FULL;
    full.notify_one();
}

void Buffer::get(char* c)
{
    unsigned int i;
    unique_lock<mutex> lock(mx);
    while (state == EMPTY)
        full.wait(lock);
    for(i = 0; i < BUFFER_SIZE; i++)
        *c++ = content[i];
    state = EMPTY;
    empty.notify_one();
}
```

Komunikacioni kanal kapaciteta jedne poruke

- Saradnja niti **proizvođača** i niti **potrošača**, u toku koje prva od njih prosleđuje rezultate svoje aktivnosti drugoj niti, može da se prikaže kao **razmena poruka**.
- U toku ove razmene **proizvođač** odlaže poruku u poseban **pregradak** iz koga tu poruku preuzima **potrošač**.
- Takvu razmenu poruka podržava templatejt klasa **Message_box**

Komunikacioni kanal kapaciteta jedne poruke

```
template<class MESSAGE>
class Message_box {
    mutex mx;
    enum Message_box_states
    { EMPTY, FULL };
    MESSAGE content;
    Message_box_states state;
    condition_variable full;
    condition_variable empty;
public:
    Message_box() : state(EMPTY) {};
    void send(const MESSAGE* message);
    MESSAGE receive();
};
```

```
template<class MESSAGE>
void
Message_box<MESSAGE>::
send(const MESSAGE* message)
{
    unique_lock<mutex> lock(mx);
    while(state == FULL)
        empty.wait(lock);
    content = *message;
    state = FULL;
    full.notify_one();
}

template<class MESSAGE>
MESSAGE
Message_box<MESSAGE>::receive()
{
    unique_lock<mutex> lock(mx);
    while(state == EMPTY)
        full.wait(lock);
    state = EMPTY;
    empty.notify_one();
    return content;
}
```

Komunikacioni kanal kapaciteta jedne poruke

- Templejt klasa **Message_box** omogućuje uspostavljanje komunikacionog kanala između niti pošiljaoca i niti primaoca.
- Njene operacije **send()** i **receive()** omogućuju **asinhronu** razmenu poruka jer se pošiljalac i primalac **ne sreću** prilikom razmene poruka (aktivnost pošiljaoca se **zaustavlja pri slanju** poruka samo kada je komunikacioni kanal **pun**, dok se aktivnost primaoca **zaustavlja pri prijemu** poruka samo kada je ovaj kanal **prazan**).
- Ako se kapacitet komunikacionog kanala poveća na **dve** ili **više poruka**, tada svakom prijemu mogu da prethode **dva** ili **više slanja**.

Komunikacioni kanal kapaciteta jedne poruke

- S druge strane, uz zadržavanje kapaciteta komunikacionog kanala na **jednoj poruci**, razmena poruka postaje **sinhrona**, ako se uvek **zaustavlja** aktivnost niti koja prva započne razmenu poruka.
- Aktivnost ove niti ostaje zaustavljena dok i druga nit ne započne razmenu poruka (dok se pošiljalac i primalac ne **sretnu**).
- Pri tome se podrazumeva da pošiljalac **nastavlja** svoju aktivnost tek kada primalac **preuzme poruku**.
- Prethodno dozvoljava da se u komunikacionom kanalu ne čuva poruka, nego njena **adresa**.
- To doprinosi **brzini** sinhronne razmene poruka, jer primalac može **direktno** preuzeti poruku od pošiljaoca.

Komunikacioni kanal kapaciteta jedne poruke

- Time se **izbegava potreba** da se poruka **prepisuje** u **komunikacioni kanal**, što je neizbežno kod **asinhronne** razmene poruke.
- Iako na ovaj način primalac pristupa **lokalnoj promenljivoj pošiljaoca**, u kojoj se nalazi poruka, to ne predstavlja problem dok god mehanizam sinhronne razmene poruka osigurava **međusobnu isključivost pristupanja pomenutoj lokalnoj promenljivoj**.
- Pošto **sinhrona razmena** poruka zahteva da se pošiljalac i primalac poruke **sretnu**, ona se naziva i **randevu (rendezvous)**.

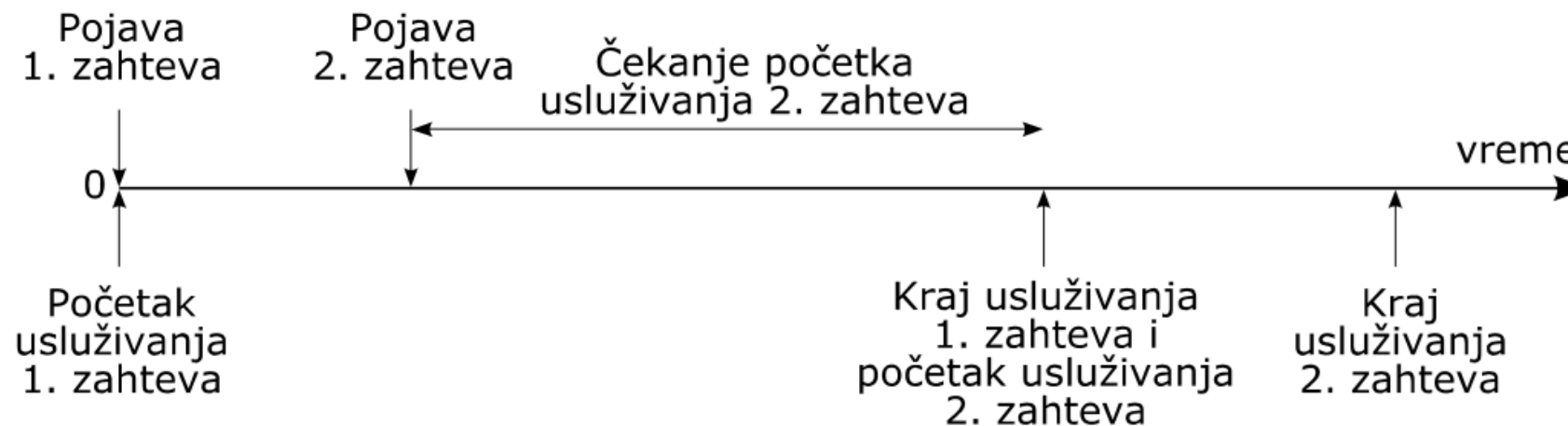
Primer simulacije

- Primer sistema, za koje su razvijeni uspešni simulacioni modeli, predstavljaju sistemi čiji elementi se nalaze u odnosu **korisnika** i **uslužioca**.
- Kod ovakvih sistema, korisnik upućuje uslužiocu zahteve za **uslugom**, pri čemu su **razmaci** između pojava susednih zahteva za uslugom, kao i **dužine** njihovog usluživanja, **slučajne** veličine.

Primer simulacije

.Ovakvi simulacioni modeli omogućuju, određivanje **srednjeg vremena čekanja početka usluživanja** pojedinih zahteva, ako su poznate:

- raspodela **verovatnoća razmaka** između pojava susednih zahteva
- raspodela **verovatnoća dužina** njihovog usluživanja



Primer simulacije

- **Srednje vreme čekanja** početka usluživanja pojedinih zahteva se dobije kada se **ukupnim brojem zahteva podeli suma vremena koja su protekla između pojava pojedinih zahteva i početaka njihovog usluživanja**.
- **Komunikacioni kanal** između proizvođača i potrošača uspostavlja deljena promenljiva **box**.
- Kroz ovaj kanal se mogu slati poruke koje se sastoje od **jednog celog broja (int)**.
- U implementiranom primeru simulacionog modela se **ne koriste slučajni brojevi**, radi jednostavnosti.

Primer simulacije

```
#include<thread>
#include<iostream>
#include"box.hh"

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

Message_box<int> Box;

const int TERMINATION = -1;
const int USER_INTERVAL = 1;
const int SERVER_INTERVAL = 2;
```

```
void thread_user()
{
    int request_time = 0;
    int time_limit = 3;
    cout << endl << "USER-SERVER
                     SIMULATION" << endl;

    while(request_time < time_limit) {
        box.send(&request_time);
        request_time += USER_INTERVAL;
    }
    box.send(&TERMINATION);
}
```

Primer simulacije

```
void
thread_server()
{
    int service_end_time = 0;
    int new_request_time;
    int request_count = 0;
    int mean_waiting_time = 0;

    while((new_request_time = box.receive()) != TERMINATION) {
        request_count++;
        if(new_request_time < service_end_time)
            mean_waiting_time += service_end_time - new_request_time;
        else
            service_end_time = new_request_time;
        service_end_time += SERVER_INTERVAL;
    }
    mean_waiting_time /= request_count;
    cout << endl << "mean waiting time = " << mean_waiting_time << '\n';
}
```

Primer simulacije

```
int  
main()  
{  
    thread user(thread_user);  
    thread server(thread_server);  
    user.join();  
    server.join();  
}
```

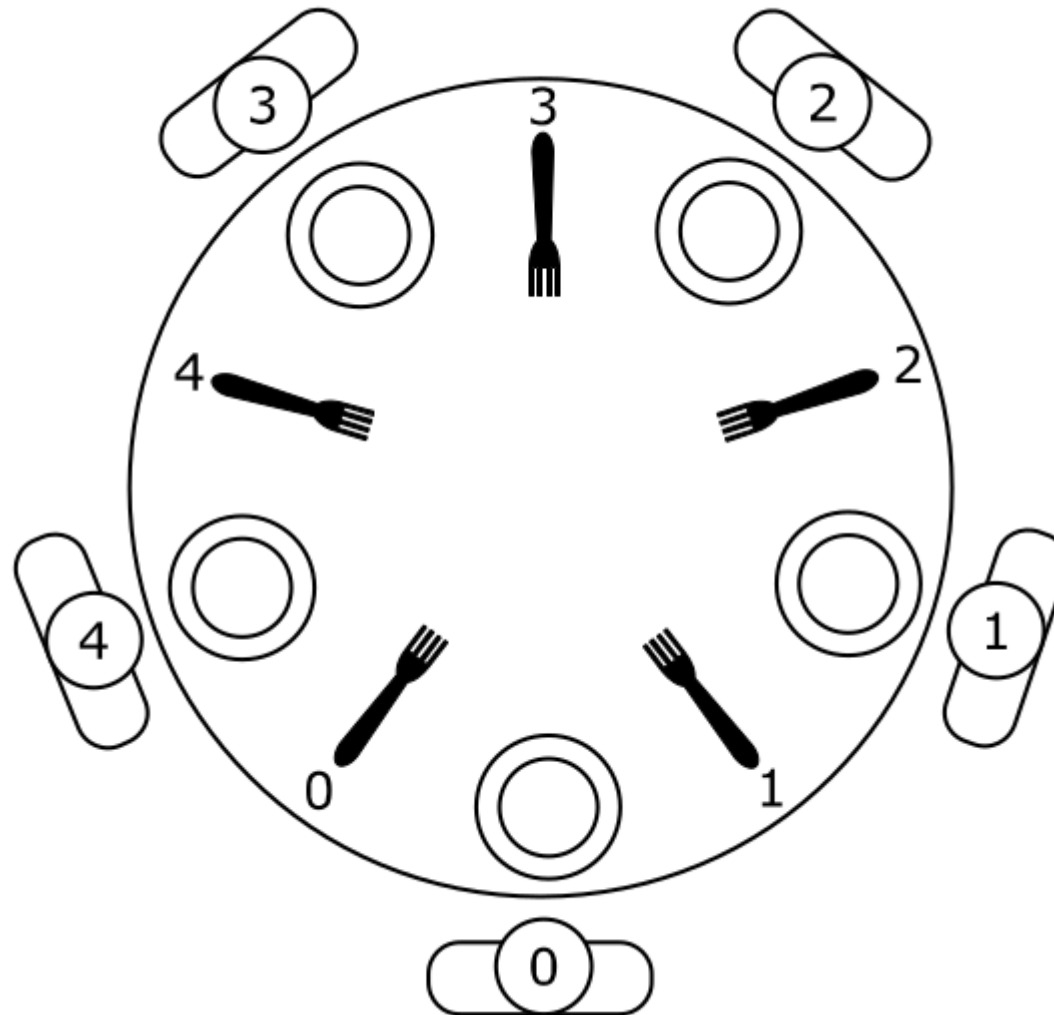
Uspavljivanje niti

- Uspavljivanje niti omogućuje funkcija **sleep_for()**.
- Njen parametar dopušta zadavanje **broja milisekundi** koji određuje najkraći period odlaganja aktivnosti niti pozivaoca funkcije **sleep_for()**.
- Poziv funkcije **sleep_for()** sa argumentom **većim od nula** dovodi do **uspavljivanja niti pozivaoca** i do **aktiviranja najprioritetnije spremne niti**.

Problem pet filozofa

- 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**.

Problem pet filozofa



Problem pet filozofa

- 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.

Problem pet filozofa

- 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**.

Problem pet filozofa

- 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 omogućiti nastavak aktivnosti svojih suseda.

Problem pet filozofa

- 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**.

Problem pet filozofa

- 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**).

Problem pet filozofa

- 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.

Problem pet filozofa

```
#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};
```

Problem pet filozofa

```
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);
};
```

Problem pet filozofa

```
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++;
}
```


Problem pet filozofa

```
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();
}
```

Problem pet filozofa

```
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);
    }
}
```

Problem pet filozofa

```
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();
}
```

Problem pet filozofa

• Prikaz izmenjenih stanja filozofa:

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

Problem čitanja i pisanja

- 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.

Problem čitanja i pisanja

•Prenos sredstava obuhvata četiri koraka:

- 1) Čitanje stanja računa s koga se prenose sredstva
- 2) Pisanje novog stanja na ovaj račun. Novo stanje se dobije **umanjivanjem** pročitano stanja za prenošeni iznos
- 3) Čitanje stanja računa na koji se prenose sredstva.
- 4)Pisanje novog stanja na račun na koji se prenose sredstva. Novo stanje se dobije **uvećavanjem** pročitano stanja za prenošeni iznos

Problem čitanja i pisanja

- 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.

Problem čitanja i pisanja

• Iz prethodne analize sledi:

- 1) Da je za ispravnost prenosa bitno da **prenosi** budu **međusobno isključivi**.
- 2) Da je za ispravnost proveru bitno da **provere** i **prenosi** budu **međusobno isključivi**.

- Pošto prenos sadrži **pisanje**, a provera samo **čitanje**, 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 čitanje** međusobna isključivost **nije potrebna**.

Problem čitanja i pisanja

```
#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;
```

Problem čitanja i pisanja

```
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;
}
```

Problem čitanja i pisanja

```
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));
    }
}
```

Problem čitanja i pisanja

```
        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();
    }
}
```

Problem čitanja i pisanja

```
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();
}
```

Problem čitanja i pisanja

```
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();
    }
}
```

Problem čitanja i pisanja

```
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();
}
```

Problem čitanja i pisanja

```
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;
    }
}
```


Problem čitanja i pisanja

```
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();
}
```

Problem čitanja i pisanja

```
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();
}
```

Problem čitanja i pisanja

READERS AND WRITERS

RN: 1	RDN: 0	WN: 0	WDN: 0
RN: 2	RDN: 0	WN: 0	WDN: 0
RN: 2	RDN: 0	WN: 0	WDN: 1
RN: 2	RDN: 1	WN: 0	WDN: 1
RN: 2	RDN: 1	WN: 0	WDN: 2
RN: 1	RDN: 1	WN: 0	WDN: 2
RN: 0	RDN: 1	WN: 0	WDN: 2
RN: 0	RDN: 1	WN: 0	WDN: 1
RN: 0	RDN: 1	WN: 1	WDN: 1
RN: 0	RDN: 1	WN: 0	WDN: 1
RN: 0	RDN: 1	WN: 0	WDN: 0
RN: 0	RDN: 1	WN: 1	WDN: 0
RN: 0	RDN: 1	WN: 0	WDN: 0
RN: 0	RDN: 0	WN: 0	WDN: 0
RN: 1	RDN: 0	WN: 0	WDN: 0
RN: 0	RDN: 0	WN: 0	WDN: 0

.Prethodno rešenje problema čitanja i pisanja nije dobro, ako je važno da **čitanja imaju prednost u odnosu na pisanja**, odnosno, da provere računa imaju prednost u odnosu na prenose sredstava.

Rizici konkurentnog programiranja

- **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 konkurentnog programiranja

- 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 konkurentnog programiranja

- 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:

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

Rizici konkurentnog programiranja

- 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**.

Rizici konkurentnog programiranja

```
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();
}
```


Rizici konkurentnog programiranja

```
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;
```

Rizici konkurentnog programiranja

- 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**.