Operativni Sistemi - Konkurentno programiranje

Veljko Petrović

Februar, 2024

Primeri štetnog preplitanja

Svojstva konkurentnih programa

- Mešanje izvršavanja raznih niti ili niti i obrađivača prekida naziva se *preplitanje* (eng. 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

- 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

```
1  class Position {
2      int x, y;
3  public:
4      Position();
5      void set(int new_x,int new_y);
6      void get(int* current_x,int* current_y);
7  };
8  Position::Position(){
9      x = 0;
```

```
10      y = 0;
11  }
```

Rukovanje pozicijom kursora

```
12  void Position::set(int new_x, int new_y){
13      x = new_x;
14      y = new_y;
15  }
16  void Position::get(int* current_x, int* current_y){
17      *current_x = x;
18      *current_y = y;
19  }
20  Position position;
```

Rukovanje pozicijom kursora

- Neka operacija `position.set()` bude ono što koristi obrađivač prekida da podesi poziciju kursora na osnovu podataka koje je dobio i dekodirao od samog hardvera, npr. miša.
- Neka operacija `position.get()` bude na raspolaganju procesima iz korisničkog sloja i služi da se preuzme tekuća pozicija kursora.
- Tada je moguće da u toku izvršavanja operacije `position.get()` proces bude prekinut ili prestignut radi obrade prekida, koja poziva operaciju `position.set()`.
- Ako se to desi dok je izvršavanje get u liniji 17, a izvršavanje set stigne do linije 14, onda će, kada se get završi preuzeti pola stare, a pola nove koordinate: rezultat koji je sigurno netačan.

Rukovanje slobodnim baferima

```
1   struct List_member {
2       List_member* next;
3       char buffer[512];
4   };
5
6   class List {
7       List_member* first;
8   public:
9       List() : first(0) {};
10      void link(List_member* member);
11      List_member* unlink();
12  };
13
```

```
14 void List::link(List_member* member){
15     member->next=first;
16     first=member;
17 }
```

Rukovanje slobodnim baferima

```
18 List_member* List::unlink(){
19     List_member* unlinked;
20     unlinked=first;
21     if(first != 0)
22         first=first->next;
23     return unlinked;
24 }
25
26 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.

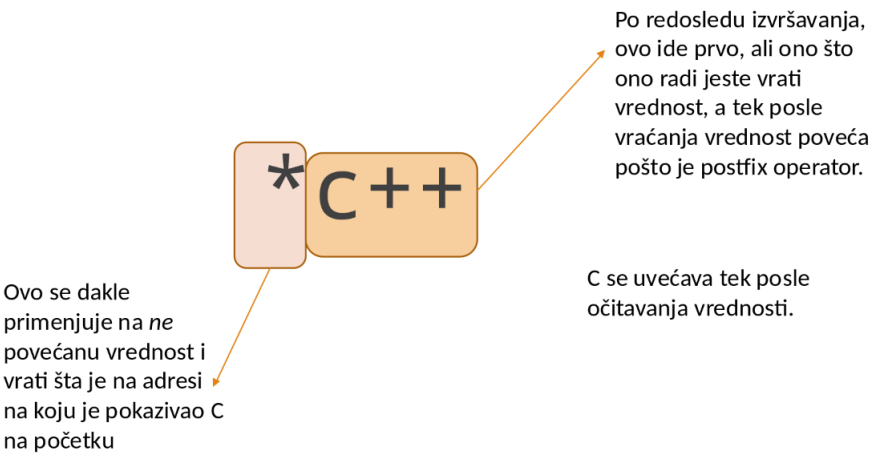
Rukovanje komunikacionim baferom

```
1  const unsigned int BUFFER_SIZE = 512;
2
3  class Buffer {
4      char content[BUFFER_SIZE];
5  public:
6    Buffer() {};
7      void put(char* c);
8      void get(char* c);
9  };
10
11 void Buffer::put(char* c){
12     unsigned int i;
13     for(i = 0; i < BUFFER_SIZE; i++)
14         content[i] = *c++;
15 }
```

Rukovanje komunikacionim baferom

```
16 void Buffer::get(char* c){
17     unsigned int i;
18     for(i = 0; i < BUFFER_SIZE; i++)
19         *c++ = content[i];
20 }
21 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

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 pretostavkom 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 promenljvih.

Kritične sekcije i sinhronizacija

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

Fundamentalna Rešenja problema međusobne isključivosti

- Postoje dva fundamentalna pristupa problemu međusobne isključivosti:
  - Pristup baziran na softveru
  - Pristup baziran na hardveru

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.
- To se može učiniti kroz algoritam ili kroz hardversku podršku

Algoritmi koji štite propusnicu

- Ovo je plodno polje istraživanja koje je *naročito* važno ako se bavite distribuiranim sistemima (nešto više o ovome kasnije)
- Ovde ćemo baciti pogled na Lamportovo rešenje poznato još i kao "Lamportova Pekara"
- Lamportova Pekara je mehanizam gde zamišljamo da imamo pekaru (radila bi i neka druga prodavnica) gde mušterije uzmu broj da bi ušle u prodavnicu.
- Na ulazu je znak koji prikazuje koji se broj uslužuje

Lamportova Pekara

- Algoritam je onda da uzmemo broj i čekamo (bukvalno čekanje: koristimo petlju za čekanje - tehnika koja je još poznata i kao spinlock ili busy-wait) dok naš broj nije 'na redu'
- Pošto se svi možemo složiti oko toga koji je broj veći ili manji, dok god možemo da delimo memoriju, možemo da se sinhronizujemo

Lamportova Pekara

- Algoritam se mora adaptirati za upotrebu na računaru (a ne u pekari) ali centralna ideja je ista
- Uvešćemo prvo dva niza, `Choosing` i `Number`
- Prvi je bool niz koji je 1 ako je neka nit u procesu biranja broja
- Drugi je niz brojeva koje niti uzimaju

Lamportova Pekara - pseudokod

```
1 bool choosing[NUM_THREADS] = {false};
2 int number[NUM_THREADS] = {0};
3 void lock(int i){
4   choosing[i] = true;
5   number[i] = 1 + max(number);
6   choosing[i] = false;
7   for(int j = 0; j < NUM_THREADS; j++){
8     while(choosing[j]);//busy-wait
```

```
9       while
10      (
11        (number[j] != 0) &&
12        (
13          (number[j] < number[i]) ||
14          (number[j] == number[i] && j < i)
15        )
16      );//busy-wait
17    }
18  }
```

Lamportova Pekara

- Primetite da je ograničenje Lamportove pekare da može da se desi da dve niti dobiju isti broj
- To nije problem: u tom slučaju broj **niti** postaje prioritet
- Jedino što je onda potrebno je imati brojeve niti koje se ne ponavaljaju

Lamportova Pekara - pseudokod

```
19  void unlock(int i){
20    number[i] = 0;
21  }
22
23  void nit(int pid){
24    while(true){
25      lock(pid);
26      //Kritična sekcija
27      unlock(pid);
28    }
29  }
```

Ograničenje Lamportove pekare i sličnih algoritama

- Glavni problem sa svim algoritmima ove vrste jeste što se zasnivaju na tome da se kod izvršava *tačno kako piše*
- Ovo apsolutno nije nešto u šta možemo da se uzdamo na modernim sistemima gde je često izvršavanje koda van redosleda
- Ako imamo izvršavanje koda van redosleda, onda algoritmi ovog tipa *neće raditi.*

Hardverska podrška

- Hardverska podrška je ono što se koristi na jedinstvenim računarima, tj. van distribuiranih sistema.
- Jednostavno je, brzo i efektno
- Na procesorima jednostrukog tipa se koriste *atomski regioni* koji su konstruisani tako što se zabrani obrada prekida. Bez prekida na takvim sistemima nema preplitanja.
- Procesori sa više sistemskih niti zahtevaju nešto malo komplikovanije.

Zabrana prekida

- Ako imamo jednostruk procesor jedini način na koji on može prestati da izvršava sekvencu instrukcija koju je uzeo da izvršava jeste da se desi prekid
- Taj prekid može da nešto promeni i da vrati izvršavanje (kao što je slučaj sa obrađivačem za kursor ili vreme), ali jednako lako može i da potpuno promeni šta se izvršava budući da obrađivač prekida može da promeni pointer instrukcija i sadržaj registara i steka i time potpuno i permanentno izmeni šta sistem radi.

Zabrana prekida

- Ako *zabranimo* prekid što se relativno lako može uraditi kroz instrukciju procesora koja tome služi (`sti` i `cli` na x86_64 arhitekturi), onda garantujemo da nas niko drugi neće prekinuti
- Ako nema potrebe da se bavimo nečim drugim, mi predpostavljamo da koristimo baš ovaj pristup, mada se on u praksi ne koristi.

Šta na procesorima sa više sistemskih niti?

- Jedan od metoda jeste da se zabrane prekidi *i* zaključa memorijska magistrala
- Ovim se garantuje da niti može da se desi prekid nit druga nit može da dirne memoriju
- U praksi ovo je previše sporo
- Umesto toga, kada je samo potrebna međusobna isključivost između dve korisničke niti onda se koriste posebne instrukcije koje implementiraju sve što je potrebno da bi se implementirala propusnica.

Poredi i zameni

- Poredi i zameni (eng. Compare and Swap) je instrukcija koja postoji na nekim arhitekturama i koristi se kao gradivni element da implementira različite primitive sinhronizacije.
- Na Intel arhitekturi ovo se zove `CMPXCHNG` i (uz prefiks `LOCK`) omogućava da se vrlo jednostavno napravi struktura koja bezbedno upravlja memorijom propusnice
- Ako pogledamo uputstvo možemo da vidimo kako je to Intel implementirao.

Citat iz Intel uputstva za x86-64 arhitekturu 7.3.1.2

“The CMPXCHG (compare and exchange) and CMPXCHG8B (compare and exchange 8 bytes) instructions are used to synchronize operations in systems that use multiple processors. The CMPXCHG instruction requires three operands: a source operand in a register, another source operand in the EAX register, and a destination operand.”

Citat iz Intel uputstva za x86-64 arhitekturu 7.3.1.2

“If the values contained in the destination operand and the EAX register are equal, the destination operand is replaced with the value of the other source operand (the value not in the EAX register). Otherwise, the original value of the destination operand is loaded in the EAX register.”

Citat iz Intel uputstva za x86-64 arhitekturu 7.3.1.2

“The status flags in the EFLAGS register reflect the result that would have been obtained by subtracting the destination operand from the value in the EAX register. The CMPXCHG instruction is commonly used for testing and modifying semaphores. It checks to see if a semaphore is free. If the semaphore is free, it is marked allocated; otherwise it gets the ID of the current owner.”

Citat iz Intel uputstva za x86-64 arhitekturu 7.3.1.2

“This is all done in one uninterruptible operation. In a single-processor system, the CMPXCHG instruction eliminates the need to switch to protection level 0 (to disable interrupts) before executing multiple instructions to test and modify a semaphore. For SMP systems, CMPXCHG can be combined with the LOCK prefix to perform the compare and exchange operation atomically.”

Kako dalje

- Nije bitno (mnogo) koja tehnika se koristi da se implementira propusnica
- Bitno je da radi
- Za naše potrebe, ako nas baš ne zanima neka varijacija na temu, plan će da se pratvaramo da imamo jednostruk procesor gde koristimo zabranu prekida zato što je to najprostije za razumevanje.

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 C++

- Međunarodni standard C++11 predviđa rukovanje nitima i deljenim promenljivama.
- Implementacije C++11 standarda i kasnijih revizija su dužne da pruže nekakvu implementaciju ovih standardnih osobina jezika
- Na Linux platformi GCC počevši od verzije ~5 podržava sve što je potrebno
- Potrebna je pomoć operativnog sistema da se niti implementiraju kako valja
- Ta pomoć se pruža preko POSIX Threads (pthreads) mehanizma.

Klasa `thread`

```
1 void thread_example(){
2     double x;
3     cout << "ZADAJ VREDNOST BROJA X" << endl;
4     cin >> x;
5     cout << endl << "x = " << x << endl;
6 }
7
8 int main(){
```

```
9       thread example(thread_example);
10      example.join();
11     //example.detach();
12  }
```

Klasa `thread`

- Sve što nam je potrebno jeste da napravimo instancu klase `thread` kojoj prosledimo funkciju koja će biti telo niti
- U tom trenutku nit počinje da radi
- Ono što nam je ostalo i što moramo da uradimo jeste da definišemo odnos između niti koja je pokrenula tu nit (sve niti pokreće neka druga nit osim niti koja izvršava 'main' koju dobijamo za džabe od operativnog sistema)
- Imamo dva izbora: `join` i `detach`

`join` i `detach`

- Prva opcija kaže da će nit koja je pokrenula drugu nit (nit-roditelj i nit-dete su tipični termini) da pauzira izvršavanje dok nit-dete ne terminira.
- Većinu vremena ovo je tačno ono što želimo zato što ne želimo da se nit `main` terminira (zajedno sa programom) dok nam se sve niti nisu završile.
- Druga opcija služi kada nam ne treba da čekamo da se nit-dete završi (možda se izvršava u beskonačnoj petlji) i jednostavno terminiramo program kada stignemo do kraja `main` niti.

Primer štetnog preplitanja

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

Primer štetnog preplitanja

- Šta je ovde deljeni resurs?
- Šta izaziva štetno preplitanje?
- Kako bi se manifestovalo?

Sprečavanje štetnog preplitanja 1

```
mutex terminal;

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

Sprečavanje štetnog preplitanja 2

```
mutex terminal;

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

Uslovna sinhronizacija

- Sinhronizacija kroz `mutex` je mehanizam koji omogućava da kontrolišemo koliko niti mogu biti u nekoj ključnoj sekciji koda
- Ponekad nam ne treba samo to, nego nam je neophodno da neku nit pauziramo dok neki proizvoljan uslov nije zadovoljen.
- Nije bitno koji je to uslov, samo da se nit mora pauzirati dok uslov nije ispunjen.

Uslovna sinhronizacija

- Klasa koja se koristi da ovo implementira je `condition_variable`
- Ako hoćemo da čekamo mi proverimo uslov (i to u while petlji) i u telu petlje pozivemo `.wait` nad `condition_variable` instancom.
- `.wait` prima lock pod kojim smo trenutno i oslobađa ga dok čekamo da ne bi blokirali ceo program tako što čekamo pod bravom
- Ostaćemo u čekanju dok neka druga nit, negde drugde, ne pozove komandu da nas otkoči, to može biti `.notify` ili `.notify_all`

gde prva otkoči nit koja najduže čeka na uslov, a druga otkoči sve.

Slobodni baferi - 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;
}
```

Slobodni baferi - neblokirajuća verzija

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

Slobodni baferi - 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();
}
```

Slobodni baferi - blokirajuća verzija

```
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;
}
```

Komunikacioni bafer

```
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);
};
```

Komunikacioni bafer

```
14 void Buffer::put(char* c){
15     unsigned int i;
16     unique_lock<mutex> lock(mx);
17     while (state == FULL)
18         empty.wait(lock);
19     for(i = 0; i < BUFFER_SIZE; i++)
20         content[i] = *c++;
21     state = FULL;
22     full.notify_one();
23 }
24
25 void Buffer::get(char* c){
26     unsigned int i;
27     unique_lock<mutex> lock(mx);
28     while (state == EMPTY)
29         full.wait(lock);
30     for(i = 0; i < BUFFER_SIZE; i++)
31         *c++ = content[i];
32     state = EMPTY;
33     empty.notify_one();
34 }
```

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 templejt klasa `Message_box`

Komunikacioni kanal kapaciteta jedne poruke

```
1  template<class MESSAGE>
2  class Message_box {
3      mutex mx;
4      enum Message_box_states { EMPTY, FULL };
5      MESSAGE content;
6      Message_box_states state;
7      condition_variable full;
8      condition_variable empty;
9  public:
10     Message_box() : state(EMPTY) {};
11     void send(const MESSAGE* message);
```

```
12      MESSAGE receive();
13  };
```

Komunikacioni kanal kapaciteta jedne poruke

```
14  template<class MESSAGE> void Message_box<MESSAGE>::send(const MESSAGE* message){
15      unique_lock<mutex> lock(mx);
16      while(state == FULL)
17          empty.wait(lock);
18      content = *message;
19      state = FULL;
20      full.notify_one();
21  }
22
23  template<class MESSAGE> MESSAGE Message_box<MESSAGE>::receive(){
24      unique_lock<mutex> lock(mx);
25      while(state == EMPTY)
26          full.wait(lock);
27      state = EMPTY;
28      empty.notify_one();
29      return content;
30  }
```

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 sinhrone 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 asinhrone razmene poruke.
- Iako na ovaj način primalac pristupa lokalnoj promenljivoj pošiljaoca, u kojoj se nalazi poruka, to ne predstavlja problem dok god mehanizam sinhrone 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).

Pitanja

Pitanja

- Šta je preplitanje?
- Da li preplitanje ima slučajan karakter?
- Šta izaziva pojavu preplitanja?
- Da li preplitanje može uticati na rezultat izvršavanja programa?
- Šta su deljene promenljive?
- Šta je preduslov očuvanja konzistentnosti deljenih promenljivih?

Pitanja

- Šta su kritične sekcije?
- Šta je sinhronizacija?
- Koje vrste sinhronizacije postoje?
- Šta je atomski region?
- Šta sužava primenu atomskih regiona?
- Čemu služi propusnica?

Pitanja

- Šta se dešava sa niti koja zatraži, a ne dobije propusnicu?
- Šta se dešava kada nit vrati propusnicu?
- Kako se štiti konzistentnost propusnica?
- Šta je isključivi region?

- Šta uvode poželjne osobine konkurentnih programa?
- Po čemu se konkurentno programiranje razlikuje od sekvencijalnog?

Pitanja

- Koje prednosti ima konkurentna biblioteka u odnosu na konkurentni programski jezik?
- Kako se opisuju niti?
- Kako se kreiraju niti?
- Kada se zauzima propusnica deljene promenljive?
- Kada se oslobađa propusnica deljene promenljive?
- Kakvu ulogu ima klasa mutex?

Pitanja

- Kakvu ulogu ima klasa unique_lock?
- Kakvu ulogu ima klasa condition_variable?
- Koje vrste razmene poruka postoje?
- U čemu se razlikuju sinhrona i asinhrona razmena poruka?