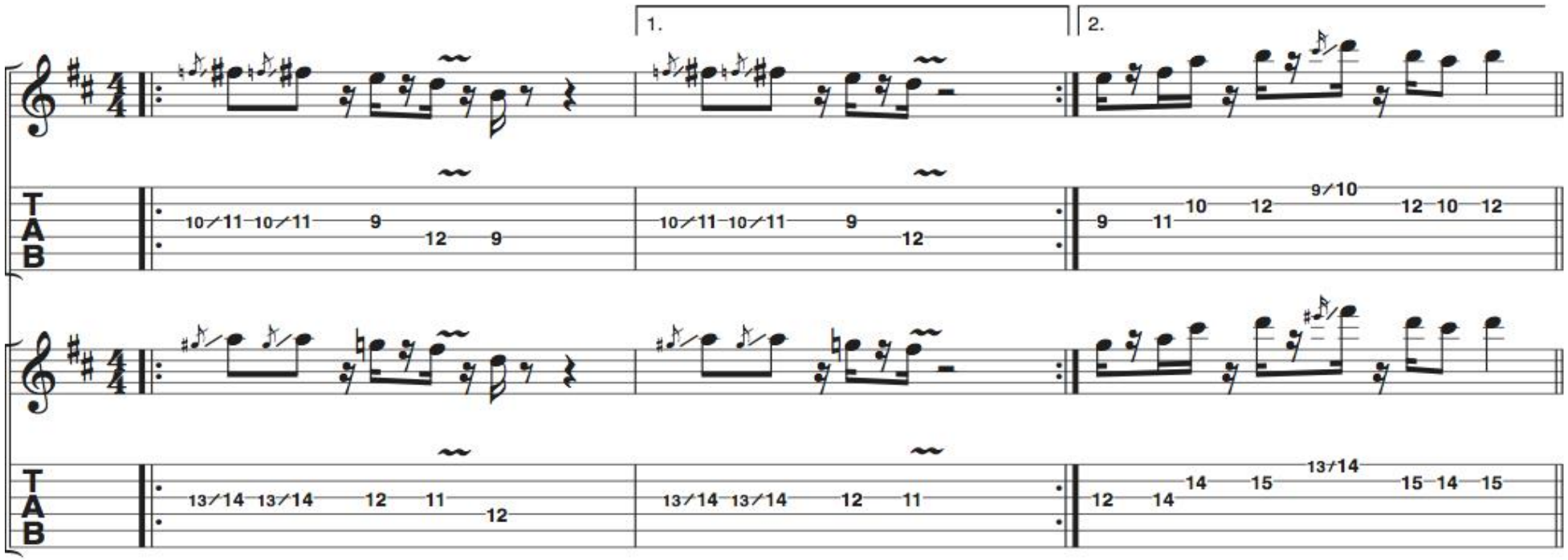
Sinhronizacija procesa

Nemanja Maček

Sadržaj

Master the art of playing harmonies …



* Slika (note i tablature) preuzeta sa Web sajta: http://www.musicradar.com/tuition/guitars

Problem sinhronizacije

- **Kooperatvni proces** može da utiče na druge procese ili da trpi uticaj drugih procesa.
  - Do toga dolazi usled deljenja podataka koji su smešteni u memoriji ili u datotekama.
- Ukoliko nema sinhronizacije, moguća je **nekonzistentnost** tih podataka!
- Primer:
  - Dva procesa P1 i P2 žele da privremeno sačuvaju neku vrednost na memorijskoj lokaciji A.
  - Proces P1 proverava da li je memorijska lokacija A slobodna.
    - Lokacija A je slobodna $\rightarrow$ proces P1 je obavešten da je lokacija A slobodna.
  - Proces P2 proverava da li je memorijska lokacija A slobodna.
    - Lokacija A je slobodna $\rightarrow$ proces P2 je obavešten da je lokacija A slobodna.
  - Proces P1 upisuje podatak na memorijsku lokaciju A.
  - Proces P2 upisuje drugi podatak na memorijsku lokaciju A.
  - Proces P1 čita **pogrešan** podatak sa memorijske lokacije A!
- Da bi se izbegle slične situacije, OS mora da obezbedi mehanizme za očuvanje konzistentnosti podataka, odnosno mehanizme **sinhronizacije procesa**.

Problem sinhronizacije

- Problem **proizvođača i potrošača**.
- Rešenje sa baferom kapaciteta *N* elemenata i promenljivim `in` i `out` radi za *N*-1 elemenata u baferu.
  - Hoćemo da izmenimo kod tako da bafer radi sa punim kapacitetom.
  - Uvodimo novu deljivu promenljivu `counter` koja opisuje broj zauzetih elemenata u baferu.
    - Početna vrednost promenljive `counter` je 0.
    - Promenljiva se uvećava za 1 svaki put kada proizvođač doda novi element u bafer.
    - Promenljiva se umanjuje za 1 svaki put kada potrošač uzme jedan element iz bafera.

```
typedef struct {
   /* Jedan element bafera */
} item;
item buffer [N]; // Bafer veličine N
int in = 0; // Prvo slobodno mesto
int out = 0; // Prvo zauzeto mesto
int counter = 0; // Broj zauzetih elemenata
```

Potrošač →

Problem sinhronizacije

- Proizvođač v1.0:

```
item next_produced;
while (1) {
   /* Proizvođač proizvodi informaciju */
   while (((in+1) % N) == out);
   /* Sinhronizacija (bafer je pun).
      Proizvođač čeka da potrošač nešto
      uzme iz bafera */
   buffer[in] = next_produced;
   in = (in+1) % N;
}
```

- Potrošač v1.0:

```
item next_consumed;
while (1) {
   while (in == out);
   /* Sinhronizacija (bafer je prazan).
      Potrošač čeka se da proizvođač
      nešto stavi u bafer /*
   next_consumed = buffer[out];
   out = (out+1) % N;
   /* Potrošač konzumira informaciju */
}
```

- Rešenje je OK u slučaju da se u baferu mogu naći najviše *N*-1 informacija.
- Bafer je pun kada `(in+1) mod N = out`.
- Bafer je prazan kada je `in=out`.

Problem sinhronizacije

- Proizvođač v2.0:

```
item next_produced;
while (1) {
   /* Proizvođač proizvodi informaciju */
   while (counter == N)
   /* Sinhronizacija (bafer je pun).
      Proizvođač čeka da potrošač nešto
      uzme iz bafera */
   buffer[in] = next_produced;
   in = (in+1) % N;
   counter++;
}
```

- Potrošač v2.0:

```
item next_consumed;
while (1) {
   while (counter==0);
   /* Sinhronizacija (bafer je prazan).
      Potrošač čeka se da proizvođač
      nešto stavi u bafer /*
   next_consumed = buffer[out];
   out = (out+1) % N;
   counter--;
   /* Potrošač konzumira informaciju */
}
```

- Rešenje za bafer sa punim kapacitetom: proizvođač i potrošač koriste promenljivu `counter`.
- Bafer je pun kada je `counter=N`. Bafer je prazan kada je `counter=0`.
- Pitanje: čemu „crveno slovo“ nad promenljivom `counter`?

Problem sinhronizacije

- Proces proizvođač i potrošač se mogu izvršavati konkurentno samo ako se operacije `counter++` i `counter--` izvršavaju **atomarno** ili nedeljivo, odnosno **bez prekidanja**.
- Problemi nastaju kao posledica implementacije operacija uvećanja i umanjenja promenljive `counter` koje **nisu atomarne** već se izvršavaju se na sledeći način:
  - Vrednost promenljive `counter` se upiše u lokalni CPU regitar.
  - Vrednost registra se uveća ili umanji za 1.
  - Vrednost registra se vraća u promenljivu `counter`.
- Kod za operacije `counter++` i `counter--` na simboličkom mašinskom jeziku:

- Operacija `counter++`:

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- Operacija `counter--`:

```
register2 = counter
register2 = register2 - 1
counter = register2
```

Problem sinhronizacije

- **Primer preklapanja mašinskih instrukcija** ako globalne operacije nisu atomarne.
  - Vrednost promenljive counter u početnom trenutku je 5.
  - Proizvođač i potrošač obavljaju svoje operacije `counter++` i `counter--` konkurentno.
    - Izvršava se šest mašinskih instrukcija.
  - Nakon izvršenja ovih operacija vrednost promenljive `counter` može biti 4, 5 ili 6.
  - Ispravna vrednost je 5.

Naredba	Izvršava	Instrukcija	register1	register2	counter
T0	proizvođač	`register1 = counter`	**5**	/	5
T1	proizvođač	`register1 = register1 + 1`	**6**	/	5
T2	potrošač	`register2 = counter`	6	**5**	5
T3	potrošač	`register2 = register2 - 1`	6	**4**	5
T4	proizvođač	`counter = register1`	6	4	**6**
T5	potrošač	`counter = register2`	6	4	**4**

Problem sinhronizacije

- **Primer preklapanja mašinskih instrukcija** ako globalne operacije nisu nedeljive.
  - Ova sekvenca za posledicu ima pogrešnu vrednost promenljive `counter` (4).
  - Promena redosleda naredbi T4 i T5 takođe dovodi do nekorektne situacija, odnosno pogrešne vrednosti promenljive `counter` (6).
  - U oba slučaja greška nastupa zato što oba procesa konkurentno pristupaju zajedničkoj promenljivoj.

Naredba	Izvršava	Instrukcija	register1	register2	counter
T0	proizvođač	`register1 = counter`	**5**	/	5
T1	proizvođač	`register1 = register1 + 1`	**6**	/	5
T2	potrošač	`register2 = counter`	6	**5**	5
T3	potrošač	`register2 = register2 - 1`	6	**4**	5
T4	potrošač	`counter = register2`	6	4	**4**
T5	proizvođač	`counter = register1`	6	4	**6**

Problem sinhronizacije

- Ako više procesa pristupa i modifikuje iste podatke konkurentno, krajnja vrednost zajedničkih podataka zavisi od sekvence instrukcija koje tim podacima pristupaju.
- Ova sekvenca se naziva **stanje trke** (engl. *race condition*) i zavisi od:
  - prekidnih signala,
  - načina raspređivanja procesa.
- U trci se pobednik obično ne zna, tako da je krajnji rezultat sekvence neizvestan.
  - Ovo se ne sme dozvoliti!
- Stanje trke se može sprečiti ukoliko se obezbedi mehanizam sinhronizacije koji će dozvoliti pristup zajedničkom resursu samo jednom od konkurentnih procesa u jednom vremenskom trenutku.

Kritična sekcija

- **Kritična sekcija** (engl. *critical* section) je deo koda u kome proces pristupa ili menja zajedničke podatke (memorijske promenljive, tabele ili datoteke).
  - OS može dozvoliti samo jednom procesu da bude u svojoj kritičnoj sekciji, tj. izvršavanje kritičnih sekcija procesa je međusobno isključivo u vremenu (engl. *mutual exclusion*).
- Proces traži od OS dozvolu da uđe u kritičnu sekciju pomoću specijalnog koda (**ulazne sekcije**).
- Kada prođe ulaznu sekciju, proces može ući u svoju kritičnu sekciju.
- Po završetku proces mora objaviti svim ostalim procesima da više nije u kritičnoj sekciji, što se postiže delom koda koji se zove **izlazna sekcija**.
- Određivanje dela koda koji predstavlja kritičnu sekciju je **zadatak programera**!

```
do {
   entry section
      critical section
   exit section
   /* ostatak koda koji ne pripada kritičnoj sekciji */
} while (1);
```

Softverska realizacija kritične sekcije

- Pretpostavke softverskog rešenja kritične sekcije:
  - **Međusobno isključenje**.
    - Unutar kritične oblasti u jednom trenutku može naći najviše jedan proces.
  - **Mogućnost ulaska**.
    - Svaki proces ima mogućnost ulaska u svoju kritičnu sekciju.
  - **Sprečavanje ulaska**.
    - Proces van kritične sekcije ne sme sprečiti druge procese da uđu u svoje kritične sekcije.
    - Proces u kritičnoj sekciji može sprečiti ostale procese da uđu u kritičnu sekciju.
  - **Čekanje**.
    - Proces ne sme neograničeno dugo da čeka da uđe u svoju kritičnu sekciju.
    - Proces se sme neograničeno dugo da ostane u svojoj kritičnoj sekciji.
      - Proces mora da izađe i omogući drugim procesima da uđu u svoje kritične sekcije.

Algoritam stroge alternacije

- Zajednička promenljiva `turn` određuje koji proces može ući u svoju kritičnu sekciju.
- Samo proces čija je vrednost indeksa jednaka vrednosti promenljive `turn` može ući u svoju kritičnu sekciju.
  - Ulazana sekcija: while petlja (čeka da promenljiva `turn` postane jednaka indeksu procesa).
  - Izlazna sekcija: promenljivoj `turn` se dodeljuje vrednost indeksa drugog procesa.
- Primer za dva procesa P0 i P1:

- Kod procesa P0:

```
do {
   while (turn != 0) ; // ulazna sekcija
      /* kritična sekcija */
   turn = 1; // izlazna sekcija
      /* ostatak koda */
} while (1);
```

- Kod procesa P1:

```
do {
   while (turn != 1); // ulazna sekcija
      /* kritična sekcija */
   turn = 0; // izlazna sekcija
      /* ostatak koda */
} while (1);
```

Algoritam stroge alternacije

- Poštuje se **stroga alternacija** procesa po pitanju ulaska u svoje kritične oblasti.
  - Može se dešavati samo sekvenca P0 → P1 → P0 → P1 …
- Rešenje zadovoljava uslov međusobnog isključenja (engl. *mutual exclusion*).
  - Samo jedan proces može da uđe u svoju kritičnu sekciju.
- Nedostaci rešenja:
  - P1 ne može ući u svoju kritičnu sekciju ako P0 ne prođe svoju kritičnu sekciju i postavi vrednost promenljive `turn` na 1.
    - Može doći do blokiranja sistema.
  - *Polling* tehnika ulazne sekcije.
    - Stalno se proverava stanje promenljive `turn` što troši procesorsko vreme.
  - Slučaj dva procesa od kojih je jedan znatno sporiji u odnosu na drugi.
    - Brži proces će dugo čekati da spori proces dodeli odgovarajuću vrednost promenljivoj `turn` i time mu omogući ulazak u kritičnu sekciju.

Realizacija kritične sekcije bez stroge alternacije

- Osnovni nedostatak prethodnog algoritma je sam **princip stroge alternacije**.
- Algoritam ne sadrži dovoljno informacija o procesima.
  - Jedina informacija u algoritmu tiče se procesa koji ima pravo da uđe u kritičnu sekciju, bez obzira da li proces to želi ili ne.
- Stroga alternacija se može eliminisati uvođenjem binarne promenljive `flag` za svaki proces posebno.
  - Ako je `flag[i]=1`, proces Pi je spreman i želi da uđe u svoju kritičnu sekciju.
  - Ako je `flag[i]=0`, proces Pi ne želi da uđe u svoju kritičnu sekciju jer njegovo izvršenje trenutno ne zavisi od zajedničkih podataka.
- Inicijalno, za dva procesa P0 i P1 uvode se dve kontrolne promenljive `flag[0]` i `flag[1]` sa početnom vrednošću 0.

```
int flag[2];
flag[0] = flag[1] = 0; // boolean
```

Realizacija kritične sekcije bez stroge alternacije

- Kod procesa Pi je:

```
do {
   flag[i] = 1;
   while (flag[j]); // ulazna sekcija
      /* kritična sekcija */
   flag[i] = 0; // izlazna sekcija
      /* ostatak koda */
} while (1);
```

- Rešenje deluje korektno.
  - Zadovoljava međusobno isključenje.
  - Van kritičnih sekcija procesi ne utiču jedan na drugog.
- Proces postavlja fleg na 1 (želi da uđe u svoju kritičnu sekciju).
- Proces proverava da li je drugi proces postavio svoj fleg na 1.
- Ako jeste, čeka u petlji da drugi proces izađe iz kritične sekcije i postavi svoj fleg na 0.

- Sledeća sekvenca dokazuje da je ovaj algoritam **pogrešan** (otpada kao rešenje):
  - P0 postavlja `flag[0]=1`. Posle toga se kontrola dodeljuje procesu P1.
  - P1 postavlja `flag[1]=1`. Oba procesa upadaju u beskonačnu petlju jer su oba flega postavljena na 1 i ni jedan ne može da uđe u svoju kritičnu sekciju.

Dekker-Petersonov algoritam

- Kod procesa Pi je:

```
do {
   flag[i] = 1;
   turn = j;
   while (flag[j] && turn = j);
      /* kritična sekcija */
   flag[i] = 0;
      /* ostatak koda */
} while (1);
```

- Promenljiva `flag` ukazuju da proces želi da u uđe u svoju kritičnu sekciju.
- Promenljiva `turn` određuje koji proces ima prednost ulaska u kritičnu sekciju.
- Način rada:
- Proces Pi najpre postavlja `turn=j` odnosno daje šansu drugom procesu (Pj) da uđe u svoju kritičnu sekciju.
- Proces Pi čeka na ulazak u kritičnu sekciji samo ako su ispunjeni sledeći uslovi:
  - Proces Pj želi da uđe u svoju kritičnu sekciju: `flag[j]=1`
  - Procesu Pj je data šansa za ulazak u kritičnu sekciju: `turn=j`.
- U svim drugim situacijama proces Pi ulazi u svoju kritičnu sekciju.

Dekker-Petersonov algoritam

- Svi uslovi su zadovoljeni.
  - Poštuje se međusobno isključenje.
    - Promenljiva `turn` obezbeđuje da u kritičnu sekciju može ući samo jedan proces i to onaj koji ima prednost.
    - Proces koji nema prednost ne odustaje, ali čeka na red.
  - Nema stroge alternacije.
    - Neka se P1 zaglavi u ostatku koda koji ne pripada kritičnoj sekciji dovoljno dugo.
    - Proces P0 kreće da izvršava do-while petlju iz početka.
    - Tada je `flag[1]=0`, `flag[0]=1`, `turn=1`.
    - Vrednost logičkog izraza while petlje procesa P0 biće 0.
    - Proces P0 može ponovo ući u kritičnu sekciju a da ne čeka na proces P1.
  - Nema zaglavljivanja u beskonačnoj petlji.
    - Ne može biti istovremeno `turn=0` i `turn=1`.
    - Ne postoji mogućnost da oba procesa beskonačno dugo ostanu unutar while petlje.

Pekarski algoritam

- Pekarski (engl. *bakery*) algoritam je generalizacija prethodnog rešenja za realan slučaj sa N procesa u sitemu.
- **Ideja**: poštovanje reda, čime se sprečava haos.
  - Red se (navodno) poštuje u bankama, poštama, pa i pekarama – otud naziv algoritma.
- **Pravilo**:
  - Svaki kupac u pekari dobija **broj** (engl. *ticket*).
  - Radnik **uslužuje kupce redom**, počev od onog sa najmanjim brojem.
- Slično pravilo primenjuje se i na procese:
  - Svakom procesu se dodeljuje broj.
  - Pekarski algoritam ne garantuje da će dva procesa uvek dobiti različite brojeve.
  - Ako dva procesa dobiju isti broj, odlučiće ime, odnosno indeks procesa (*process ID*).

Pekarski algoritam

- **Pravila pekarskog algoritma**.
  - Sekvenca brojeva se generiše u rastućem redu i dodeljuje procesima (npr. 1,2,3,3,3,3,4,5...)
  - Procesi se opslužuju na osnovu vrednosti dodeljenih brojeva počev od procesa kome je dodeljen najmanji broj *i*.
  - Ako dva procesa Pi i Pj dobiju iste brojeve prvo se opslužuje proces sa manjim indeksom.
    - Primer: ako je $i<j$ tada se prvo opslužuje Pi.
- **Leksikografski poredak** dva elementa (*ticket*, *process ID*) dat je na sledeći način:
  - $(a, b) < (c, d)$ ako je $a < c$ ili ako je $a = c$ i $b < d$.
  - max $(a_0, \ldots, a_{n-1})$ je broj a_k, takav da je $a_k \geq a_i$ za svako $i \in [0, n-1]$
- **Zajednički podaci** za pekarski algoritam: `int choosing[n]` (*boolean*) i `int number[n]`.
- Za svaki proces se definišu promenljive:
  - Binarna `choosing[i]` koja služi za sinhronizaciju prilikom dobijanja broja.
  - Celobrojna `number[i]` koja sadrži broj koji je proces Pi dobio i koji mu definiše prioritet.
- Ni jedan proces ne može ući u kritičnu sekciju dok neki proces dobija broj.
- Svi podaci su inicijalno postavljeni na 0.

Pekarski algoritam

```
do {
   choosing[i] = 1;
   /* Pi dobija broj. Kandidati za ulazak u kritičnu sekciju moraju da čekaju */
   number[i] = max (number[0], … , number[n-1]) + 1;
   /* Pi dobija najveći broj u tom trenutku, uvećan za 1, što mu garantuje najniži prioritet */
   choosing[i] = 0;
   /* Pi oslobađa fleg choosing[i]=0 i pristupa ulaznoj sekciji */
   for (j=0; j<n; j++) {
      while (choosing[j]);
      while ( (number[j] != 0) && ((number[j], j) < (number[i], i)) );
      /* U ulaznoj sekciji, proces ostaje sve dok ne postane proces sa najmanjim brojem.
         To se proverava u petlji (number[j], j) < (number[i], i)) */
   }
   /* kritična sekcija */
   number[i] = 0;
   /* Pi anulira svoj broj i više ne učestvuje u takmičenju za ulazak u kritičnu sekciju */
   /* ostatak koda */
} while (1);
```

Hardverska realizacija kritične sekcije

- Kritična sekcija je rešiva na nivou hardvera ako postoje **nedeljive procesorske instrukcije**:
  - Za čitanje i izmenu memorijske reči ili
  - Razmenu sadržaja dve memorijske reči.
- VAŽNO: instrukcije se ne smeju prekidati u toku izvršavanja!
- Mašinska naredba procesora `TestAndSet (&target)`:
  - (1) Čita vrednost korisničke promenljive `target`
  - (2) Kopira tu vrednost u korisničku promenljivu `rv`
  - (3) Postavlja vrednost promenljive `target` na true (1).
  - U praksi `rv` predstavlja neki registar procesora, a `target` neku memorijsku lokaciju.
- Naredba swap:
  - Razmenjuje vrednosti dve memorijske promenljive preko privremenog registra procesora (neka se radi ilustracije zove `temp`).
  - (1) `temp = a`
  - (2) `a = b`
  - (3) `b = temp`

Hardverska realizacija kritične sekcije

- Algoritam koji koristi hardversku podršku za realizaciju kritične oblasti (naredba `TestAndSet`).
- Koristi se promenljiva `lock` čija je inicijalna vrednost 0: `int lock = 0;`
- Kod za sve procese:

```
do {
   while (TestAndSet(lock));
      /* kritična sekcija */
   lock = 0;
      /* ostatak koda */
} while (1);
```

- Hardver garantuje da: (a) samo jedan proces može da pročita vrednost promenljive `lock` i da je neposredno posle toga postavi na 1, (b) pri tome ne može biti prekinut.
- Proces koji želi da uđe u svoju kritičnu sekciju postavlja `lock` na 1 i ulazi u kritičnu sekciju.
- Svi ostali procesi čekaće u while petlji dok je `lock=1`.
- Kada proces napusti K.S. postaviće `lock` na 0 i omogućiti drugim procesima da uđu u svoju K.S.

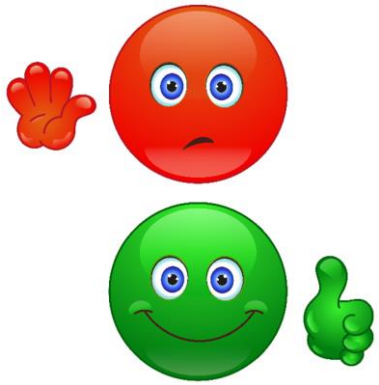
Hardverska realizacija kritične sekcije

- Algoritam koji koristi hardversku podršku za realizaciju kritične oblasti (naredba `swap`).
- Kod za sve procese je:

```
do {
   key = 1;
   while (key == 1) swap(lock, key);
      /* kritična sekcija */
   lock = 0;
      /* ostatak koda */
} while (1);
```

- Promenljivoj `lock` se inicijalno dodeljuje vrednost 0.
- Proces koji želi da uđe u svoju kritičnu sekciju zamenjuje vrednosti `lock` i `key` (`lock=1`, `key=0`).
- Proces ulazi u svoju kritičnu sekciju.
- Ostali procesi će detektovati `lock=1` i čekaće da proces napusti svoju KS i postavi `lock=0`.
- Nakon toga neki drugi proces može ući u svoju kritičnu sekciju.

Semafori



- **Semafor** je celobrojna promenljiva nenegativne vrednosti koja:
  - Štiti neki resurs i
  - Omogućava komunikaciju između procesa:
    - Mehanizam međusobnog isključenja ili
    - Sinhronizaciju aktivnosti kooperativnih procesa.
- Vrednost semafora određuje da li je resurs kog taj semafor štiti slobodan.
  - Ukoliko je vrednost semafora `val(s)=0` resurs je zauzet.
  - Resurs je slobodan ukoliko je vrednost semafora `val(s)>0`.
- Svaki semafor ima svoju **početnu vrednost**.
- Nad semaforom se mogu izvršiti dve **nedeljive operacije**:
  - Operacija `signal(s): increment s;`
  - Operacija `wait(s): when (s > 0) do decrement s;`
- Nedeljivost ovih operacija se ogleda u sledećim činjenicama:
  - Operacije `signal` i `wait` se ne mogu podeliti na više ciklusa.
  - Dva procesa ne mogu istovremeno izvršavati ove operacije nad istim semaforom.

Semafori

- Problem kritične sekcije može se rešiti upotrebom binarnog semafora `mutex` (*mutual exclusion*).

```
semaphore mutex; /* inicijalno, mutex = 1 */
```

- Kod koji važi za sve procese na sistemu obezbediće zaštitu kritičnih sekcija:

```
do {
   wait(mutex); // ulazna sekcija
      /* kritična sekcija */
   signal(mutex);
     /* ostatak koda */
} while (1);
```

- Samo jedan proces proći će semafor, oboriti mu vrednost na 0 i ući u svoju kritičnu sekciju.
- Svi drugi procesi čekaju na semaforu.
- Kada proces napusti kritičnu sekciju oslobodiće semafor komandom `signal(mutex)`.

Semafori

- Saradnju između procesa čije su aktivnosti sinhronizovane obezbeđuje semafor `proceed` .
- Inicijalna vrednost ovog semafora je 0.
- Primer:
  - Procesu P1 je u tački S1 neophodna informacija koju proces P2 obezbeđuje u tački S2.
  - P1 ne može da nastavi izvršenje u tački S1 dok P2 ne prođe tačku S2 i obezbedi neophodnu informaciju.

Proces P1	Proces P2
…	…
…	…
wait (proceed)	S2
S1	signal (proceed)
…	…
…	…

- Program koji se izvršava u procesu P1:

```
/* P1 čeka na infomaciju koju obezbeđuje P2 */
wait (proceed);
/* Proces P1 nastavlja izvršenje */
```

- Program koji se izvršava u procesu P2:

```
/* P2 obezbeđuje infomaciju neophodnu za
   dalje izvršenje procesa P1 */
signal (proceed);
```

Semafori

- **Nedostaci i mogući problemi**.
- **Ignorisanje prioriteta** procesa.
  - Može se desiti da najprioritetniji proces uđe u kritičnu sekciju posle velikog broja neprioritetnih procesa.
- Mogućnost da proces bude **zaposlen čekanjem** (*busy waiting*).
  - Proces se neprestano vrti u while petlji i ne radi ništa korisno.
  - U petlji proverava vrednost semafora kako bi saznao da li može ući u svoju kritičnu sekciju.
  - Iako ne radi ništa korisno proces troši procesorsko vreme.
  - Primer:
    - 1.000 procesa od kojih se jedan nalazi u kritičnoj sekciji.
    - Ostali čekaju na ulazak u svoje kritične sekcije.
    - Jedan proces radi nešto korisno i 999 procesa čekajući u petlji troše procesorsko vreme.

Semafori

- **Rešenje problema**.
- Kritična sekcija se realizuje pomoću sistemskih poziva kojima se proces može blokirati.
- Proces koji čeka da uđe u kritičnu sekciju postaje blokiran dok ga drugi proces ne oslobodi.
  - U kontekstu semaforskih tehnika: proces koji čeka na semaforu čija je vrednost 0 blokira se i kao takav ne troši procesorsko vreme.
- **Redefinicija pojma semafora**.
- Za svaki semafor se uvodi poseban red čekanja (**semaforski red**).
- Svi procesi koji izvršavaju operaciju `wait` nad semaforom čija je vrednost negativna ili jednaka nuli se pomoću sistemskog poziva `sleep` blokiraju i prevode u semaforski red.
- Procesor se oslobađa i predaje nekom drugom procesu koji nije blokiran.
- Proces nastavlja svoje izvršenje nakon sistemskog poziva `wakeup` koji ukida blokadu procesu.
  - Blokada se može ukinuti samo ako je proces:
    - Prvi u semaforskom redu ili ima najviši prioritet na semaforu, i
    - Neki drugi proces obavi `signal` operaciju nad tim semaforom.

Proširena definicija semafora

- Semafor u proširenom kontekstu je struktura čiji su elementi:
  - Celobrojna vrednost semafora (koja može biti i negativna) i
  - Lista pokazivača na procese koji čekaju na semafor (semaforski red).

```
typedef struct {
   int value;
   struct process *L;
} semaphore;
```

- Vrednost semafora `s` može biti:
  - `val(s)>0`. Resurs je slobodan.
  - `val(s)=0`. Resurs je zauzet, ali je semaforski red prazan.
  - `val(s)<0`. Resurs je zauzet, u semaforskom redu postoje procesi koji čekaju na taj semafor.

Proširena definicija semafora

- Operacija `wait` :
  - Umanjuje vrednost semafora.
  - Ako je vrednost nakon toga negativna, stavlja proces u semaforski red i blokira proces.
- Operacija `signal` :
  - Uvećava vrednost semafora.
  - Ako je vrednost posle toga negativna ili jednaka nuli uklanja jedan od procesa iz liste (izvršava `wakeup` sistemski poziv i prebacuje proces u red čekanja za spremne procese).

```
wait (S) {
   S.value--;
   if (S.value < 0) {
      /* Dodaj proces u semaforski red */
      sleep();
   }
}
```

```
signal (S) {
   S.value++;
   if (S.value <= 0) {
      /* Ukloni proces P iz semaforskog reda */
      wakeup(P);
   }
}
```

Problem zastoja i zakucavanja

- Loša implementacija semafora može dovesti do situacije u kojoj jedan ili više procesa beskonačno dugo čekaju na događaje koji se nikada neće dogoditi.
- Takvi procesi se nalaze u situaciji koja se zove **zastoj**.
- Primer:
  - Procesi P0 i P1 izvršavaju operacije nad semaforima S i Q (inicijalne vrednosti 1).
  - P0: `wait(S)` → S = 0, P1: `wait(Q)` → Q = 0
  - Procesi P0 i P1 će se nakon toga naći u stanju zastoja:
    - P0 čeka na Q, P1 može uvećati Q naredbom `signal(Q)`.
    - P1 čeka na S, P0 može uvećati S naredbom `signal(S)`.
  - Nijedna `signal` operacija se neće izvršiti, jer oba procesa čekaju na semaforima.
- Do **zakucavanja** (engl. *starvation*) dolazi ako procesi beskonačno dugo čekaju u semaforskom redu.
- Ova situacija je moguća ako se procesi iz semaforskog reda uklanjaju po LIFO principu.

Proces P0	Proces P1
…	…
wait (S)	wait (Q)
wait (Q)	wait (S)
…	
signal(S)	signal(Q)
signal(Q)	signal(S)

Binarni i brojački semafori

- Realizacija brojačkog semafora S (engl. *counting semaphore*) pomoću dva binarna semafora S1 i S2 (promenljive tipa `binary-semaphore`) i celobrojne promenljive C.

```
binary-semaphore S1, S2;
S1 = 1; S2 = 0;
int C; // početna vrednost semafora S
```

```
wait(S) {
   wait(S1);
   C--;
   if (C < 0) {
      signal(S1);
      wait(S2);
   }
   signal(S1);
}
```

```
signal(S) {
  wait(S1);
  C++;
  if (C <= 0) signal(S2);
  else signal(S1);
}
```

Klasični problemi sinhronizacije i semafori

- Primena semaforskih tehnika za rešavanje sledećih problema:
  - Proizvođač – potrošač (problem ograničenog bafera)
  - Problem čitalaca i pisaca
  - Problem večere filozofa.

Proizvođač – potrošač

- Bafer kapaciteta *N*.
- Sinhronizacija: brojački semafori čije vrednosti pripadaju intervalu [0,*N*-1].
  - `item_available` (potrošač ne može uzeti ništa iz bafera, ukoliko proizvočač to prethodno nije stavio u bafer)
  - `space_available` (ukoliko je bafer pun, proizvođač ne može ubaciti informaciju).
- Mutex semafor `buffer_ready` štiti bafer kao nedeljivi resurs.
  - Sprečava mogućnost zabune sa pokazivačima koja nastaje ukoliko jedan proces čita informacije iz bafera, dok drugi proces nešto upisuje u bafer.

- Promenjljive:

```
typedef struct {} item; /* Jedan element bafera */
item buffer [N]; // Bafer veličine N
int in = 0; // Prvo slobodno mesto
int out = 0; // Prvo zauzeto mesto
```

Proizvođač – potrošač

- Inicijalizacija semafora:

```
item_available = 0;
space_available = N;
buffer_ready = 1;
```

- Proizvođač:

```
do {
   /* proizvođač proizvodi informaciju */
   item next_produced;
   wait (space_available);
   wait (buffer_ready);
      buffer[in] = next_produced;
      in = (in+1)%N;
   signal (buffer_ready);
   signal (item_available);
} while (1);
```

- Potrošač:

```
do {
   item next_consumed;
   wait (item_available);
   wait (buffer_ready);
      next_consumed = buffer[out];
      out = (out+1)%N;
   signal (buffer_ready);
   signal (space_available);
   /* potrošač konzumira informaciju */
} while (1);
```

Problem čitalaca i pisaca

- Konkurentni procesi mogu da dele objekte podataka.
- Zavisnosti od toga šta procesi žele da rade sa podacima izdvajamo dve kategorije procesa:
  - procese **čitaoce** (engl. *readers*) – žele samo da čitaju deljive objekte,
  - procese **pisce** (engl. *writers*) – žele da menjaju sadržaj objekata.
- Pravila:
  - Dva procesa čitaoca mogu istovremeno da čitaju sadržaj deljenog objekta.
  - Proces koji piše ne sme dozvoliti konkurentni pristup deljenom objektu.
    - Dva pisca ne mogu istovremeno da menjaju sadržaj zajedničkog objekta.
    - Nema čitanja dok pisac ne obavi svoje aktivnosti.
- Napomena: za razliku od bafera koji ima *N* elemenata, ovde je reč o jednom objektu, čiji se sadržaj čita ili modifikuje.

Problem čitalaca i pisaca

- Problem čitalaca i pisaca može se rešiti pomoću jedne promenljive i dva mutex semafora:
  - Semafor `write` služi za međusobno isključenje procesa pisaca i čitaoca.
  - Promenljiva `read_count` čuva informaciju o broju aktivnih čitalaca.
  - Semafor `counter_ready` obezbeđuje međusobno isključenje pristupa kontrolnoj promenljivoj `read_count`.
- Pisac će nešto upisati pod uslovom da niko od čitaoca ili drugih pisaca nije aktivan.

- Inicijalizacija:

```
write=1;
counter_ready=1;
readcount=0;
```

- Pisac:

```
wait(write);
   /* proces modifikuje sadržaj objekta */
signal(write);
```

Problem čitalaca i pisaca

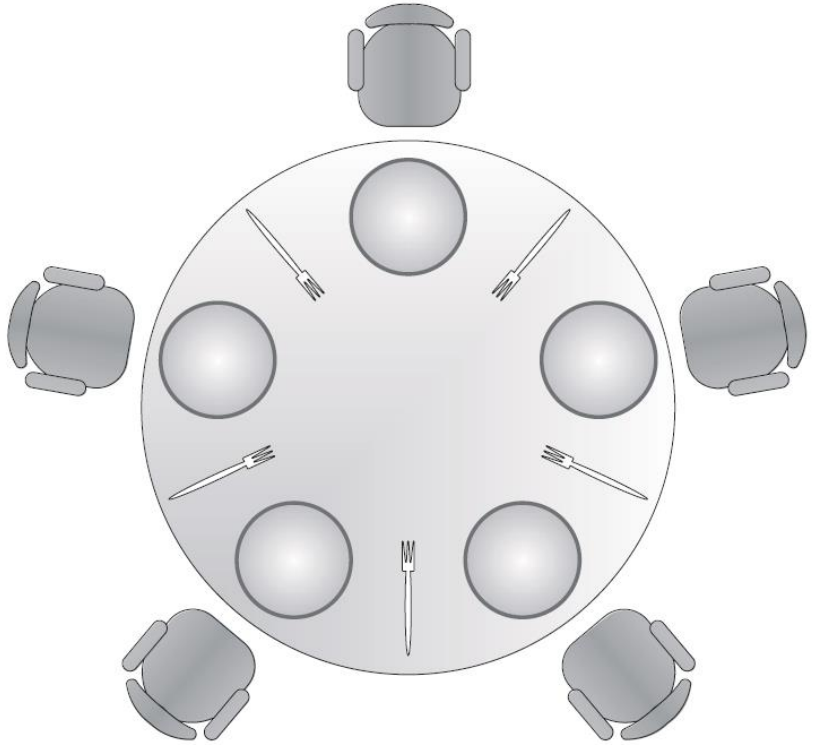
- Čitalac:

```
wait(counter_ready);
   readcount++;
   if (readcount == 1) wait(write);
signal(counter_ready);
/* proces čita sadržaj objekta */
wait(counter_ready);
   readcount--;
   if (readcount == 0) signal(write);
signal(counter_ready);
```

- Čitalac proverava koliko ima aktivnih čitalaca u sistemu.
- Čitalac pomoću semafora `counter_ready` kontroliše pristup promenljivoj `readcount`.
- Ako je vrednost promenljive `readcount` nakon inkrementiranja jednaka 1:
  - On je jedini čitaoc i mora eventualno sačekati na pisca - `wait(write)`.
- U protivnom, on nije prvi i može slobodno da čita.
- Proces završava čitanje i dekrementira vrednost promenljive `readcount`.
- Ako je čitaoc poslednji, odnosno jedini:
  - Izvršiće operaciju `signal(write)` i time osloboditi objekat za pisce.

Problem večere filozofa

- Pet filozofa sede za okruglim stolom.
- Ispred svakog filozofa nalazi se po jedan tanjir sa špagetama.
- Između svaka dva tanjira postavljena je po jedna viljuška.
- Filozofi su svoj život posvetili razmišljanju.
- Ako postanu gladni, oni jedu pa ponovo razmišljaju.
- Da bi filozof mogao jesti, potrebne su mu dve viljuške.
- Filozof koji postane gladan:
  - Najpre pogleda da li može uzeti obe viljuške.
  - Ako može, onda jede.
    - Za to vreme njegovi susedi ne mogu jesti.
  - Ako ne može, filozof misli i čeka dok obe viljuške ne postanu slobodne.
- Kada završi sa jelom, filozof spušta obe viljuške na sto i nastavlja da misli.



Problem večere filozofa

- Rešenje: svaki filozof predstavlja proces, a svaka viljuška semafor.
- Potrebno je pet semafora kojima se štite viljuške (`viljuska [i]`) čiju ćemo početnu vrednost postaviti na 1.
- Kod za svakog filozofa koji rešava problem večere filozofa je sledeći:

```
philosopher (i) {
  do {
     wait(fork[i]); /* jedna viljuška*/
     wait(fork[(i+1)%5]); /* druga viljuška*/
        /* filozof jede */
     signal(fork[i]);
     signal(fork[(i+1)%5]);
       /* filozof misli */
  } while (1);
}
```

- Problem: svi filozofi ogladne odjedanput i uzmu viljušku sa leve strane.
- Tada niko neće razrešiti drugu `wait` operaciju (niko neće doći do druge viljuške).
- Može se sprečiti ako se dozvoli da filozof uzme viljuške samo ako su obe slobodne.
- Asimetrična rešenja: neparni filozofi (prvi, treći i peti) uzimaju prvo svoje leve viljuške a zatim desne, a parni prvo desne pa leve.

Kritični regioni

- Kritični regioni štite od grubih, nehotičnih programerskih grešaka u sinhronizaciji.
- **Kritični region** zahteva promenljivu `v` tipa `T`, koja je zajednička za veći broj procesa.
- Simbolički se obeležava na sledeći način: `v: shared T`;
- Promenljivoj `v` se može pristupati samo unutar regiona naredbi `S`, uslovno, sa sledećom sintaksom: `region v when (B) S`;
- Ova konstrukcija ima sledeće značenje:
  - Dok se naredba `S` izvršava, nijedan drugi proces neće pristupati promenljivoj `v`.
  - Izraz `B` je logički izraz, koji upravlja radom u kritičnom regionu.
  - Kada proces uđe u kritični region, izraz `B` se ispituje i ukoliko je tačan, naredba `S` se izvršava.
  - Ako izraz nije tačan, međusobno isključenje se napušta i odlaže, dok `B` ne postane tačan.

Kritični regioni

- Primer: rešenje problema ograničenog bafera pomoću kritičnih regiona.
- Zajednički podaci su:

```
struct buffer {
   int pool[n];
   int count, in, out;
}
```

- Proizvođač:

```
region buffer when (count < n) {
   pool[in] = next_produced;
   in = (in+1)%n;
   count++;
}
```

- Potrošač:

```
region buffer when (count > 0) {
   next_consumned = pool[out];
   out = (out+1)%+n;
   count--;
}
```

Monitori

- Monitori omogućavaju programeru da resurse posmatra kao objekte.
- Struktura monitora podseća na instance klasa kod objektno orijentisanog programiranja.
- Svaki **monitor** se sastoji iz:
  - Promenljivih koje opisuju deljeni resurs (objekat) čije vrednosti definišu stanje monitora.
    - Lokalne promenljive vidljive su samo unutar monitora,
  - Skupa procedura i funkcija kojima se pristupa objektu, odnosno promenljivima monitora.
  - Dela programa koji inicijalizuje objekat (izvršava se samo jednom, prilikom kreiranja objekta).

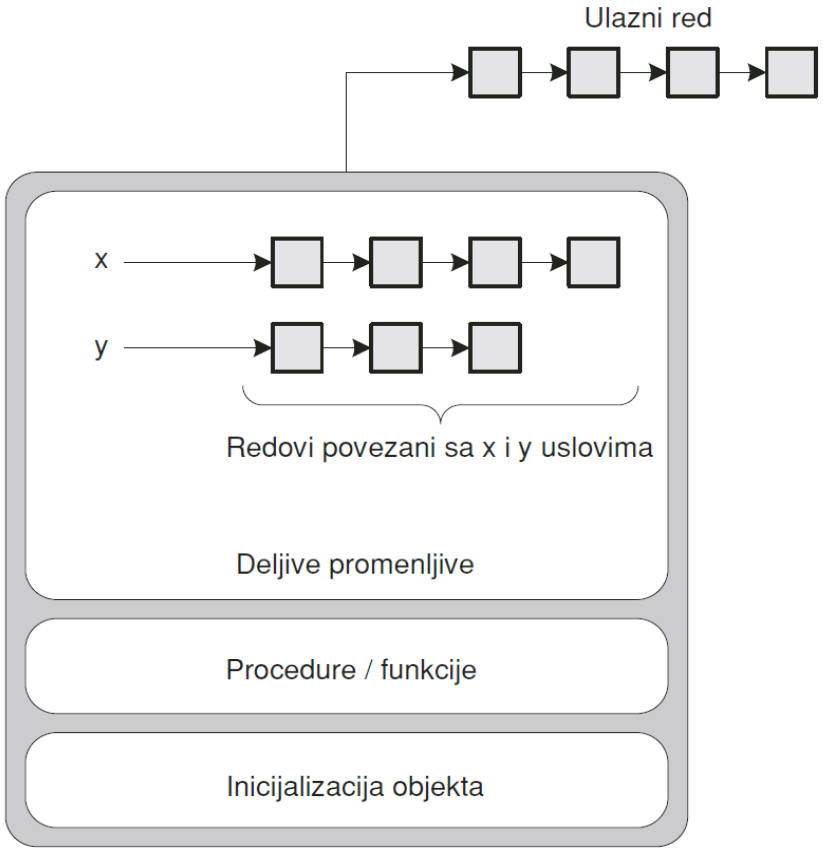
```
monitor ime_monitora {
   /* deklaracije deljenih promenljivih */
   P1 (...) { /* definicija procedure/funkcije P1 */ }
   P2 (...) { /* definicija procedure/funkcije P2 */ }
   Pn (...) { /* definicija procedure/funkcije Pn */ }
            { /* inicijalizacija monitora */        }
}
```

Monitori

- Primer - bafer koji se koristi prilikom rešavanju problema proizvođač-potrošač može se predstaviti monitorom koji čine:
  - Baferski prostor i pokazivači na prvo slobodno i prvo zauzeto mesto.
  - Procedure za dodavanje elemenata u bafer i uzimanje elemenata iz bafera.
  - Procedure za inicijalizaciju pokazivača na početak bafera.
- Konstrukcija monitora dozvoljava da samo jedan proces bude aktivan u monitoru.
- Ako su potrebni dodatni sinhronizacioni mehanizmi, programer mora sam da definiše promenljive kojima će dodatni problem razrešiti.
- Ako je dodatni sinhronizacioni mehanizam neka uslovna konstrukcija, definišu se tzv. uslovne promenljive: `condition x, y`;
  - Operacije koje mogu da se izvode nad ovim promenljivim su `signal` i `wait`.
  - Operacija `x.wait()` će blokirati proces koji je tu operaciju pozvao.
  - Operacija `x.signal()` će ukinuti blokadu sa tačno jednog blokiranog procesa.
  - Ako nema blokiranih procesa, operacija signal ne radi ništa.

Monitori

- Struktura monitora sa uslovnim promenljivama.



- Rešenje problema večere filozofa pomoću monitora.
- Struktura podataka `state` predstavlja trenutno stanje filozofa: misli, gladan je ili jede.
- Kada misli ne treba mu ništa, kade je gladan pokušava da jede, a ne misli.
- Filozof može da jede samo ako njegovi susedi ne jedu.
- Uvode se i uslovne promenljive `state[i]` koje će blokirati filozofa ukoliko je gladan, a ne može da dođe do obe viljuške.

Monitori

- Definisaćemo monitor dp:

```
monitor dp {
   enum {thinking, hungry, eating} state[5];
   condition self[5];

void pickup(int i) {
   state[i] = hungry;
   test[i];
   if (state[i] != eating) self[i].wait();
}

void release(int i) {
   state[i] = thinking;
   test((i+4) % 5);
   test((i+1) % 5);
}
```

```
void test(int i) {
   if ((state[i] == hungry) &&
   (state[(i+4)%5] != eating) &&
   (state[(i+1)%5] != eating)) {
      state[i] = eating;
      self[i].signal();
   }
}

void init() {
   for (int i = 0; i < 5; i++)
   state[i] = thinking;
}

}
```

Monitori

- Najpre se u monitoru svi filozofi dovedu u stanje `think` (svi misle, niko nije gladan, niko ne jede).
- Svaki filozof mora obaviti sledeće operacije:

```
dp.pickup(i);
   /* filozof jede */
dp.release(i);
```

- Svaki filozof mora da pozove proceduru `pickup` iz monitora `dp` pre nego što počne da jede.
  - U okviru procedure filozof će ili doći do obe viljuške ili će biti blokiran operacijom `self[i].wait`.
- Ako dođe do obe viljuške, filozof jede, a zatim ih spusti na sto.
- Posle toga, poziva se procedura `release` koja oslobađa obe viljuške a po potrebi i blokirane filozofe.

Literatura

1. B. Đorđević, D. Pleskonjić, N. Maček (2005): Operativni sistemi: teorija, praksa i rešeni zadaci. Mikro knjiga, Beograd.
2. R. Popović, I. Branović, M. Šarac (2011): Operativni sistemi. Univerzitet Singidunum, Beograd.

Hvala na pažnji

Pitanja su dobrodošla.