Konkurentno i distribuirano programiranje

Slajdovi sa predavanja

2015. godina

Zoran Jovanović

Šta je konkurentno programiranje?

- Sekvencijalni program – jedna nit kontrole
- Konkurentan program – više niti kontrole
- Zajednički program – komuniciraju preko deljenih promenljivih (shared variables) ili prosleđivanja poruka (message passing)
- Međusobno isključivanje (Mutual exclusion) – iskazi koji se ne izvršavaju u isto vreme
- Uslovna sinhronizacija – zakašnjavanje sve dok uslov ne postane true

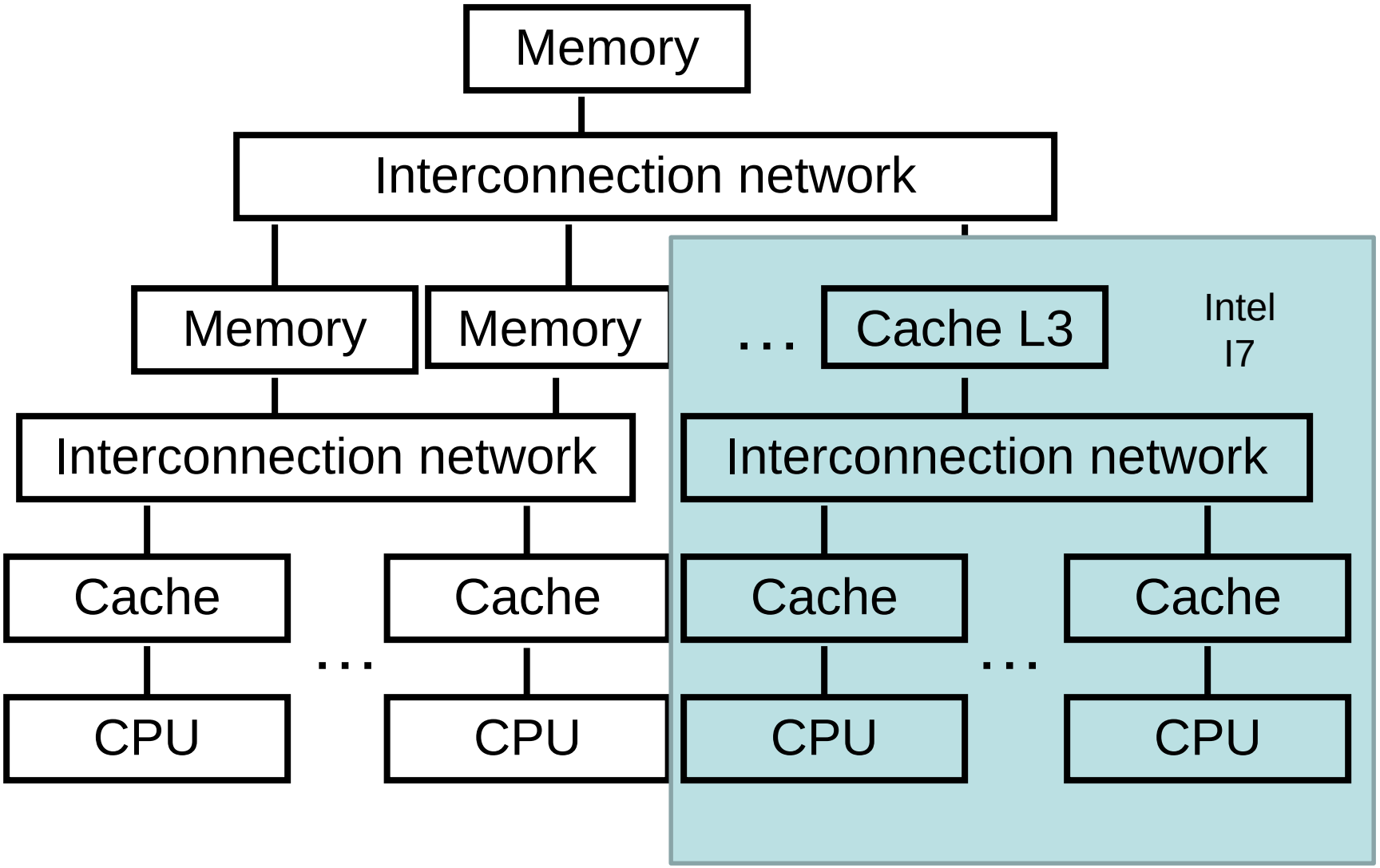
Tipovi mašina

- Shared memory - multiprocesori
- Distributed memory - multikompjuteri
- Mreže – WAN i LAN
- Mobilne wireless mreže – WLAN, Mobilna telefonija

Shared memory mašine

- Shared memory multiprocesori – svaki CPU poseduje sopstveni cache (neki put takođe i lokalnu memoriju) i memoriju koju dele sa drugim procesorima kroz interkonekcionu
- UMA – Uniform Memory Access (Symmetric multiprocessing)
- NUMA – No Uniform Memory Access
- Consistencija kopija u cache memorijama

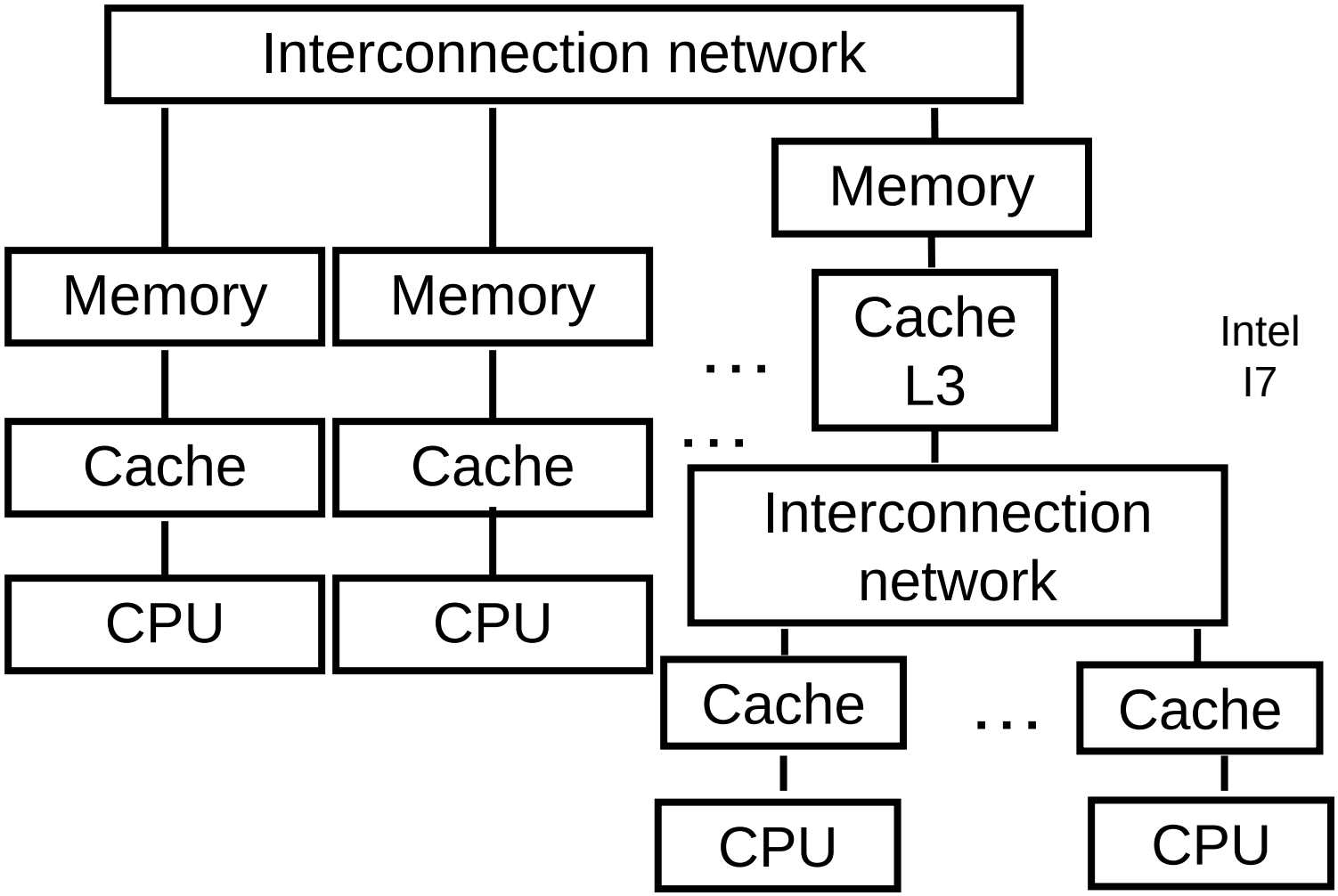
Shared memory mašine



Šta je distribuiran računarski sistem?

- Autonomni procesoru koji ne poseduju deljenu memoriju.
- Komunikaciona mreže za prosleđivanje poruka
- Distributed memory multiprocessors
- **Tightly coupled –** distribuiran sistem sa širokim propusnim opsegom i malim kašnjenjem komunikacione mreže, regularnom topologijom i lokalnom pouzdanom interkonekcionom mrežom. (Hypercube, Mesh, Star-Graph).
- **Loosely coupled –** distribuirani sistem sa LAN ili WAN mrežom. Mreža radnih stanica, klaster radnih stanica, Internet. Veća kašnjenja, manja pouzdanost veza i rutiranje nije bazirano na unapred poznatoj topologiji.

Distributed memory mašine



Tightly coupled

- Regularna topologija interkonekcione mreže
- Mali dijametar interkonekcionog grafa
- Mala fizička rastojanja
- Jednostavno rutiranje – hardverski bazirano sa ugrađenim izbegavanjem deadlock-a
- Distribuirano rutiranje – odluka o izlaznom kanalu mora se zasnivati na zaglavlju (adresa odredišta, ili čak deo adrese odredišta)
- Adaptivno (alternativno) rutiranje – primarno zbog umanjene osetljivosti na otkaze

Loosely coupled

LAN – Bitske brzine 10-100.000 Mbps

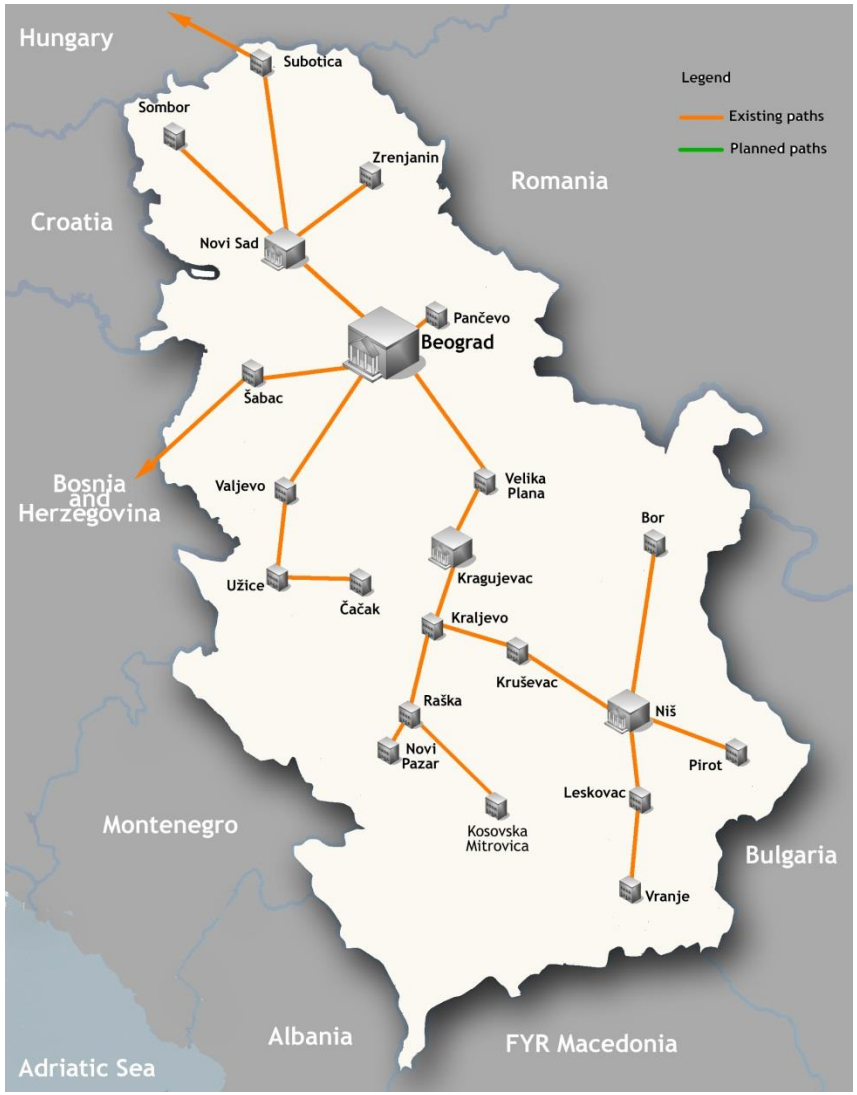
- Bez umanjene osetljivosti na otkaze (bus, star i ring)
- Sa umanjenom osetljivošću na otkaze - (token-ring, FDDI i star ring)
- Hibridne (HUB, switch)

WAN – globalne mreže – nepouzdani komunikacioni kanali. Modemi, digitalni telefonski kanali - 2B+D ISDN kanali 64Kbs ili 128 Kbps, 1,5Mbps(T1), 2Mbps (E1), 8Mbps (E2), 34Mbps (E3), STM1 (155Mbps), STM4 (622Mbps). Fiber optički kanali 1 Gbps Ethernet, STM16 (2.5 Gbps) čak 40 i 100 Gbps Ethernet

Internet

- Milioni računara u mreži sa procesima koji rade kao konkurentni (distribuirani) programi
- Standardni interfejsi, protokoli – standardi za konkurentno programiranje
- Protocol layering – izolovanje interfejsa da bi se enkapsulirao softver na svakom sloju (nema komunikacije ili sinhronizacije između softvera na različitim slojevima)

Kičma Akademske mreže



WLAN i Mobilna telefonija

- WLAN bežični TCP/IP – Internet okruženje
- Mobilna telefonija – SMS – Short messages
- WAP – Wireless application protocol
- GPRS i 3G – dovoljan kapacitet za mobilne Internet servise

Distributed shared memory

- Distribuirana implementacija shared memory apstrakcije – softver ili hardver
- Virtuelni memorijski adresni prostor sa stranicama distribuiranim u memorijama distributed memory machine
- Replicirane kopije i protokoli za konzistenciju stranica
- Razlog – kreiranje shared memory apstrakcije za shared memory konkurentne programe u distribuiranim sistemima

Klase konkurentnih i distribuiranih aplikacija

- **Redukovanje vremena izvršavanja –** preduslov: mala komunikaciona kašnjenja i paralelizam u aplikaciji. Različiti nivoi paralelizma: coarse grain, medium grain (loop level) i fine grain. Tightly coupled sistemi za medium i fine grain.
- **Fault tolerance**: otkazi se prevazilaze replikacijom funkcija ili podataka uz automatski oporavak od greške. Distribuirani sistemi mogu imati geografsku distribuciju i nema kritičnog resursa. Primer: fly by wire.
- **Specialne funkcionalnosti distribuiranih sistema**: file serveri (primer Picture Archiving and Communication Systems), numerički serveri, proizvodnja IC i komunikacioni serveri.
- **Aplikacija je distribuirana**: rezervacija avio karata i Email

Konkurentno programiranje
Kritična sekcija, AMOP, Await

Međusobno isključivanje

- Nedeljivo izvršavanje kombinacije susednih atomskih akcija (AA) kao krupnija nedeljiva akcija
- Taj skup iskaza se naziva kritična sekcija *(critical section)*. Izvršavanje kritične sekcije se mora obaviti sa međusobnim isključivanjem. To se mora uraditi samo za deljene resurse (promenljive).

At-Most-Once-Property

- Kritična referenca – referenca na promenljivu koja se menja od strane drugog procesa
- Kada dodeljivanje x:= v ima At-Most-Once-Property (AMOP)?

  Kada v sadrži najviše jednu kritičnu referencu i x se ne čita od strane drugog procesa ili v ne sadrži kritičnu referencu, i x se čita od strane drugog procesa => x:=v se javlja kao AA

Primer

int x:=0, y:=0;

co x:=y+1; // y:=y+1; oc

krajnja vrednost x je 1 ili 2 – ali **da** -AMOP
(y nije kritična referenca za proces 2)

Javljaju se kao atomske akcije dodeljivanja

int x:=0, y:=0;

co x:=y+1; // y:=x+1; oc – **nije** AMOP

Await iskaz

- Ako nije ispunjen At-Most-Once-Property – neophodna je coarse grained (krupnija) atomska akcija
- Primer operacije u vezi povezane liste ili bafera – umetanje ili izostavljanje elemenata – za jedan element
- <await (B) S;>
- B – boolean – specificira delay condition – uslov kašnjenja
- S sekvenca iskaza za koje je garantovano da terminiraju

Međusobno isključivanje i sinhronizacija sa await

- <S;> - međusobno isključivanje
- <await (B);> - uslovna sinhronizacija – B ==true tada nema kašnjenja.
- B – kada ima At-most-Once-Property – while not(B); – condition synchronization – prazno telo petlje => **spin loop** (isto kao while not(B) skip;)
- **bezuslovna atomska akcija <S;>**
- **uslovna atomska akcija <await (B) S;>**

Proizvođač/potrošač sinhronizacija

- Deljeni buffer za kopiranje niza
- Rešenje sa lokacijom za jedan element i dve deljene promenljive za brojanje
- Uvodi se sinhronizacioni predikat – logički iskaz koji mora biti true
- Sinhronizacioni predikat: c <= p <= c+1
- p==c empty lokacija; p==c+1 full lokacija
- Rešenje sa zaposlenim čekanjem

Program – kopiranje niza (1)

```
int buf, p = 0, c = 0;
process Producer {
int a[n];
while (p < n) {
<await (p == c);> /* Sinhronizacija na empty*/
buf = a[p];
p = p+1; }}
```

Program – kopiranje niza (2)

```
process Consumer {
int b[n];
while (c < n) {
<await (p > c);> /* Sinhronizacija na full*/
b[c] = buf;
c = c+1; }}
```

Problem kritične sekcije

- Više (n) procesa sa kodom

```
process CS[i = 1 to n] {
        while (true) {
            entry protocol;
            kritična sekcija;
            exit protocol;
     nije kritična sekcija; }}
```

Osobine i loša stanja

- Mutual exclusion (**međusobno isključivanje**) – najviše jedan proces izvršava svoju kritičnu sekciju
- Ne sme da postoji **Deadlock (Livelock**) – bar jedan proces će uspeti
- **Nema nepotrebnog kašnjenja** – ako drugi nisu u kritičnoj sekciji, i jedan proces čeka – neće biti zakašnjen
- **Eventual entry** – na kraju će svaki proces koji želi, uspeti da uđe u kritičnu sekciju

Rešenja – dva procesa

- < > za sve kritične sekcije sa await
- ¬(in1 and in2) predikat za dva procesa
- Dva procesa –

prvi: <await (!in2) in1 = true;> /* entry */

kritična sekcija;

in1 = false; (At-Most-Once-Property)

Nema nepotrebnog kašnjenja

Kritična sekcija – dva procesa (1)

```
bool in1 = false, in2 = false;
process CS1 {
   while (true) {
<await (!in2) in1 = true;>          /* entry */
kritična sekcija;
in1 = false;                        /* exit */
nije kritična sekcija; }}
```

Kritična sekcija – dva procesa (2)

```
process CS2 {
   while (true) {
<await (!in1) in2 = true;>          /* entry */
kritična sekcija;
in2 = false;                        /* exit */
nije kritična sekcija; }}
```

Kritična sekcija – upotrebom lock-a

- jedna promenljiva za dva stanja
- lock == (in1 or in2 or ....)
- Ako je lock = false – niko nije u kritičnoj sekciji
- Ako je lock = true – postoji proces u kritičnoj sekciji
- simetrično rešenje => za više procesa

Rešenje

```
bool lock = false;
process CSi {
   while (true) {
<await (!lock) lock = true;>        /* entry */
Kritična sekcija;
lock = false;                       /* exit */
nije kritična sekcija; }}
```

Test and Set

Specijalna instrukcija za procesore

Atomska akcija – uzme staru boolean vrednost i vraća je, ujedno nedeljivo postavlja vrednost na true

```
bool TS(bool lock) {
< bool initial = lock;    /* sačuvaj vrednost */
  lock = true;              /* postavi true */
  return initial; > }     /* vrati inicijalnu vrednost */
```

Test and Set – Kritična sekcija

```
bool lock = false;
process CSi {
   while (true) {/*entry spin lock ⇔ while lock
    skip;*/
while (TS(lock)) skip;  /* i lock = true; kao AA,
   kada se izlazi iz petlje (ulazi u kritičnu sekciju)*/
kritična sekcija;
lock = false;                                  /* exit */
nije kritična sekcija; }}
```

Problemi sa Test and Set

- spin locks – zagušenje memorije i cache-evi
- upis u lock izaziva invalidaciju cache-eva svih procesora u ShMem multiprocesorima
- izmena entry protokola koja povećava verovatnoću uspeha, a ne upisuje stalno

while (lock) skip; /* spin dok je lock true */

while (TS(lock)) **{**/* Probaj da prvi uzmeš lock */

while (lock) skip; **}** /* spin ponovo ako nisi uspeo */

Invalidacije samo kada neko ulazi u kritičnu sekciju

Tie-Breaker, Ticket i Bakery Algoritmi

Tie-Breaker (Petersonov) algoritam

- dodatna promenljiva koja indicira ko je zadnji u kritičnoj sekciji - ujedno fairness
- jednostavne promenljive i sekvencijalni iskazi
- Entry protokoli koji ne rade: in1 i in2 inicijalno false

```
P1:   while (in2) skip;
      in1 = true;
P2:   while (in1) skip;
      in2 = true;
```

Nije ispunjeno međusobno isključivanje zbog AA!

Tie-Breaker – coarse grain (1)

```
bool in1 = false, in2 = false;
int last =1;
process CS1 {
   while (true) {
      in1 = true; last = 1;          /* entry protokol */
      <await (!in2 or last ==2);>
      kritična sekcija;
      in1 = false;                   /* exit protokol */
      nije kritična sekcija; }}
```

```
process CS2 {
   while (true) {
      in2 = true; last =2; /* entry protokol */
      <await (!in1 or last ==1);>
      kritična sekcija;
      in2 = false;              /* exit protokol */
      nije kritična sekcija; }}
```

Tie-Breaker - fine-grain rešenje

```
bool in1 = false, in2 = false;
int last =1;
process CS1 {
   while (true) {
      in1 = true; last = 1; /* entry protokol */
      while (in2 and last == 1) skip;
      kritična sekcija;
      in1 = false;
      nije kritična sekcija }}
```

```
process CS2 {
   while (true) {
      in2 = true; last = 2; /* entry protokol */
      while (in1 and last ==2) skip;
      kritična sekcija;
      in2 = false;
      nije kritična sekcija }}
```

Objašnjenje (1)

- <await(B)> ⇔ while (not B) ako je B AMOP
- ! (!in2 or last == 2) ⇔ in2 and !(last == 2)
- last je ili 1 ili 2 => !(last == 2) ⇔ last == 1
- in2 and last == 1 da li poseduje AMOP ?
  Ne (i in2 i last su kritične reference)

Ali: Ako je in2 false za CS1 i pre ispitivanja last == 1, in2 postane true jer CS2 upravo izvrši in2 = true, da li mogu istovremeno u kritičnu sekciju?

Objašnjenje (2)

Ishod 1 – proces CS2 prvi izvrši last = 2; tada mora da čeka u petlji, jer je in1 and last == 2 sada true (CS2 je dao prioritet CS1 za istovremeno in1 i in2 true). Promena in2 nije uticala na ulazak, jer CS2 daje prioritet CS1.

Ishod 2 – proces CS1 ispituje last == 1, ali je in2 već ispitao i bilo je false => ulazi u kritičnu sekciju. =>

Proces koji je prvi promenio last ulazi u CS ako su i in1 i in2 postali true. (u prethodnom slučaju CS1). Upis u last je AA.

Sveukupno, nije AMOP, ali se javlja kao AMOP

Tie-Breaker za n procesa

- n stanja – koji proces prelazi u sledeće stanje određuje Tie-Breaker (Petersonov) algoritam za dva procesa
- Najviše jedan u jednom trenutku može da prođe svih n-1 stanja
- Uvode se dva integer niza in[1:n], last[1:n]
- in[i] – kroz koje stanje proces CS[i] prolazi
- last[j] – koji je proces poslednji započeo (ušao u) stanje j

n procesa

- spoljna petlja n-1 puta
- unutrašnja petlja – CS[i] čeka u dostignutom stanju ako postoji proces u višem ili istom stanju i CS[i] je bio poslednji koji je ušao u stanje j
- ako nema procesa u višem stanju koji čeka na ulazak u kritičnu sekciju ili drugi proces ulazi u stanje j - CSi može da pređe u sledeće stanje
- Najviše n-1 procesa CSi može da prođe prvo stanje, n-2 procesa CSi drugo, ..., jedan proces CSi poslednje stanje – Kritična sekcija

Algoritam za n procesa

```
int in[1:n]  = ([n] 0), last[1:n] = ([n] 0);
process CS[i = 1 to n] {
   while (true) {/* entry protokol */
      for [j = 1 to n]  {
           in[i] = j; last[j] = i;
           for [k = 1 to n st i != k] {
                    while (in[k] >= in[i] and last[j] = =i) skip; }}
      kritična sekcija;
      in[i] = 0;
      nije kritična sekcija}}
```

Objašnjenje (1)

- Spoljna petlja – prolazak kroz stanja – inicijalno u prvo stanje i markiranje da je CSi poslednji koji je ušao u prvo stanje
- Unutrašnja petlja proverava sa svim drugim procesima da li postoji proces u istom ili višem stanju. Ako se naiđe na proces u istom ili višem stanju, dodatno se ispituje da li postoji drugi proces koji je posle CSi ušao u tekuće stanje procesa CSi
- Ako se to desilo, while uslov je 0, za bilo koje k i CSi prelazi u više stanje

Objašnjenje (2)

- Poslednji koji je ušao u neko stanje izgura sve ostale procese u tom stanju u više stanje.
- Jedan proces stiže do najvišeg stanja i za taj proces CSi ne postoji in[k] >= in[i] => tipično (ali ne garantovano) prolazi sve iteracije unutrašnje i spoljašnje petlje
- Dakle, tipično se izvrše sve iteracije unutrašnje petlje za sve preostale više vrednosti j, pritom označavajući da je bio zadnji proces koji je ušao u stanje j.

Ticket algoritam

- Svi koji pokušavaju da uđu u kritičnu sekciju prvo dobiju ticket sa brojem u redosledu dolaska
- Pravična (fair) kritična sekcija
- Jedan po jedan u redosledu ticket-a (redosledu dolaska)
- Restoranski servisi – fair servisi

Ticket algorithm - coarse grain

```
int number = 1, next = 1, turn[1:n] = ([n] 0);
process CS[i = 1 to n] {
   while (true) {
        < turn[i] = number; number = number +1;>
        <await (turn[i] = = next);>
      critical section;
      <next = next +1;>
      noncritical section }}
```

Fetch and add

- Inkrementiranje promenljive sa konstantom kao atomska akcija uz vraćanje stare vrednosti
- FA(var, incr):

  <int tmp = var; var = var + incr; return(tmp);>
- upotreba FA za < turn[i] = number; number = number + 1;>
- ostalo ima AMOP jer svaki proces ima svoj turn[i], a inkrementiranje next je na kraju kritične sekcije – samo jedan proces ga modifikuje u jednom trenutku!

Ticket algorithm – fine grain

```
int number = 1, next = 1, turn[1:n] = ([n] 0);
process CS[i = 1 to n] {
   while (true) {
      turn[i] = FA(number,1); /* entry protocol */
     while (turn[i] != next) skip;
      kritična sekcija;
      next = next +1;                  /* exit protocol */
      nije kritična sekcija }}
```

Bakery algoritam

- liči na ticket algoritam bez next. Izvlači se broj veći od bilo kog drugog procesa interakcijom sa njima
- Uzajamna provera između procesa (ko je poslednji?)
- uzima se za jedan veća vrednost od bilo koje druge u turn[i]

Bakery – coarse grain

```
int turn[1:n] = ([n] 0);
process CS[i = 1 to n] {
   while (true) {
         < turn[i] = max (turn[1:n]) + 1;>
          for [j = 1 to n st j != i]
      <await (turn[j] == 0 or turn[i] < turn [j]);>
       kritična sekcija;
       turn [i] = 0;
       nije kritična sekcija }}
```

Bakery 2 – traženje max fine grain

- Dva procesa prvo
- **Nije rešenje 1**: inicijalno turn1 i turn2 su 0

```
turn1 = turn2 + 1;     /* entry 1 */
while (turn2 != 0 and turn1 > turn 2) skip;
turn2 = turn1 + 1;     /* entry 2 */
while (turn1 != 0 and turn2 > turn 1) skip;
```

turn1 i turn2 mogu oba da postanu 1 i oba da uđu u kritičnu sekciju jer su AA load, add, store

Bakery 2 – traženje max fine grain

- Dva procesa
- **Nije rešenje 2:** inicijalno turn1 i turn2 su 0

turn1 = turn 2 + 1; /* entry 1 CS1*/

while (turn2 != 0 and turn1 **>** turn 2) skip;

turn2 = turn 1 + 1; /* entry 2 CS2*/

while (turn1 != 0 and turn2 **>=** turn 1) skip;

CS1 čita turn2 (0) i onda CS2 prođe sve korake i uđe u CS jer je turn1 još jednako 0

turn 1 tada postaje 1 i CS1 ulazi u CS

Prioritet dat CS1 nema efekta za ovaj redosled AA

Bakery 2 – traženje max fine grain

- Dva procesa korektno

turn1 = 1; turn1 = turn 2 + 1; /* entry 1 CS1*/

while (turn2 != 0 and turn1 **>** turn 2) skip;

turn2 = 1; turn2 = turn 1 + 1; /* entry 2 CS2*/

while (turn1 != 0 and turn2 **>=** turn 1) skip;

- Za inicijalne vrednosti turn različite od 0, OK!
- Nije simetrično rešenje – kada bi se našlo simetrično rešenje – može se proširiti na n procesa za traženje max od n

Simetrično rešenje za 2 procesa

- Uvedimo uređen par: (a,b) > (c,d) == true akko a > c ili a == c i b > d i false u svim drugim slučajevima

turn1 > turn2 ⇔ (turn1,1) > (turn2,2)

turn2 **>=** turn1 ⇔ (turn2,2) **>** (turn1,1)

Ako turn2 == turn1 onda 2 > 1

Simetrično rešenje (turn[i],i) > (turn[j],j)

Proces CS[i] – Postavlja svoj turn na 1, traženje max od turn[j], i dodavanje 1.

Bakery n – nalaženje max fine grain

```
int turn[1:n] = ([n] 0);
process CS[i = 1 to n] {
   while (true) {
    turn[i] = 1; turn[i] = max (turn[1:n]) + 1;
    for [j = 1 to n st j != i]
        while (turn[j] != 0 and (turn[i],i) > (turn[j],j))
    skip;
     kritična sekcija;
     turn [i] = 0;
     nije kritična sekcija }}
```

Bakery - komentari

- Dva ili više procesa mogu da imaju istu vrednost turn – kao kod dva procesa !!!
- Unutrašnja while petlja uvodi redosled čak i kada dva ili više procesa imaju isto turn
- Atomske akcije su iste kao i u slučaju dva procesa, **max je aproksimirano!**

Aproksimacioni algoritmi

- Algoritmi sa sukcesivnim aproksimacijama za nizove, isto izračunavanje za sve elemente niza, zasnovano na elementima niza iz prethodne aproksimacije
- Iterativni algoritmi sa zavisnostima po podacima prenetim petljom
- Svaka aproksimacija – procesi koji su cobegin iskazi – paralelno izvršavanje

Neefikasan pristup

While (true) {

co [i=1 to n]

Kod za task i;

oc}

Co započinje kreiranje n procesa u svakoj iteraciji - neefikasno

Ideja: kreirati n procesa na početku i onda samo sinhronizovati na kraju iteracije

Sinhronizacija na barijeri
Barrier synchronization

Process Worker[i = 1 to n] {

While (true) {

kod kojim se implementira task i;

čekanje na svih n taskova da se završe; }}

Barijere su najčešće na krajevima iteracije, ali mogu čak i kod međustanja

Jedna ideja: deljeni brojač koji broji do n, inkrementiraju ga svi procesi, a kada se dostigne n, onda svi prolaze barijeru

Shared counter barrier

<count = count + 1;>

<await (count == n);>

Sa Fetch and add

FA(count,1);

While (count != n) skip;

Problemi – reset brojača posle svih i mora se proveriti pre početka inkrementiranja u narednoj iteraciji; up/down counter kao rešenje; problem update-a cache-a;

Flags and coordinators

<await (arrive[1] + … + arrive[n] == n);>

Ako se suma izračunava u svakom procesu, opet contention u memoriji - cache

Continue array – Koordinator čita arrive niz i postavlja continue niz da se izbegne contention

for [i = 1 to n] <await (arrive[i] == 1);>

for [i = 1 to n] continue[i] =1; /* ko radi reset ? */

Flag sinhronizacija

- Pravila za korektno izvršavanje:

Proces koji čeka na flag da bude postavljen na 1 treba da ga vrati na 0

Flag treba da bude set-ovan samo ako je vraćen na 0

- Radni proces briše continue, a Coordinator briše arrive
- Coordinator treba da obriše (vrati na 0) arrive pre set-ovanja continue

Sinhronizacija na barijeri sa procesom koordinatorom (1)

```
int arrive[1:n] = ([n] 0); continue[1:n] = ([n] 0);
process Worker[i = 1 to n] {
   while (true) {
      kod taska i;
      arrive[i] = 1;
      <await (continue[i] == 1);>
      continue[i] =0;
      }
    }
```

```
Process Coordinator {
    while (true)  {
      for [i = 1 to n] {
        <await (arrive[i] == 1);>
       arrive[i] = 0;
         }
    for [i = 1 to n] continue[i] = 1;
    }}
```

Semafori

Problemi sa zaposlenim čekanjem

- Kompleksni algoritmi
- Nema razlike između promenljivih namenjenih sinhronizaciji i računanju
- Neefikasno kod multithreading-a i kod multiprocesora, ako je broj procesa > od broja procesora
- Javlja se potreba za specijalnim alatima za sinhronizaciju - Semafori – analogija sa železničkim semaforima – kritična sekcije

Semaforske operacije

- Deklaracija sem s; sem lock =1;
- Ako nema deklaracije inicijalnih vrednosti, => inicijalizovano na 0
- Menja se samo operacijama P(s) i V(s)
- Wait P(s): <await (s>0) s = s - 1;>
- Signal V(s): < s = s + 1; >
- Binarni semafor: mnogi procesi pokušavaju P(s), ali samo jedan može da prođe do sledećeg V(s)
- Generalni semafor: bilo koja pozitivna vrednost ili 0
- Ako više procesa čeka = > tipično buđenje je u redosledu u kome su bili zakašnjeni

Kritična sekcija sa semaforima

```
sem mutex=1;
process CS[i = 1 to n] {
    while (true) {
        P(mutex);
        kritična sekcija;
        V(mutex);
        nije kritična sekcija;
    }
}
```

Barijere sa semaforima

- Busy waiting flags zamenjene sa semaformom za svaki flag
- Signaling semaphore – inicijalno 0, jedan proces izvršava V(s) da sinhronizuje drugi, a drugi čeka sa P(s).
- Barijera za dva procesa:

```
sem arrive1 = 0; sem arrive2 = 0;
process Worker1 {…
V(arrive1);
P(arrive2);  …}
process Worker2 {…
V(arrive2);
P(arrive1);  …}
```

Producers and consumers (n)

Split binary semaphore (raspodeljeni binarni semafor) - Više procesa, jedna lokacija

P na nekom semaforu mora da bude ulaz

Dva binarna semafora: empty i full – kao jedan semafor za kritičnu sekciju raspodeljen u dva semafora

Na svakom tragu izvršavanja, operacija P mora da bude praćena sa V na kraju dela koda kritične sekcije. Jedan semafor (empty) je inicijalno 1

Kada god postoji neki (jedan) proces između P i V => svi semafori raspodeljenog binarnog semafora su 0

Producers and consumers (n) – raspodeljeni semafori

```
typeT buf;
sem empty =1, full = 0;
process Producer[i = 1 to M] {
   while (true) { …
        /* proizvedi podatak i unesi */
        P(empty);
        buf = data;
        V(full); }
}
```

Producers and consumers (n) – raspodeljeni semafori

```
process Consumer[j = 1 to N] {
    while (true) { …
        /* uzmi rezultat */
        P(full);
        result = buf;
        V(empty);
        …}
}
```

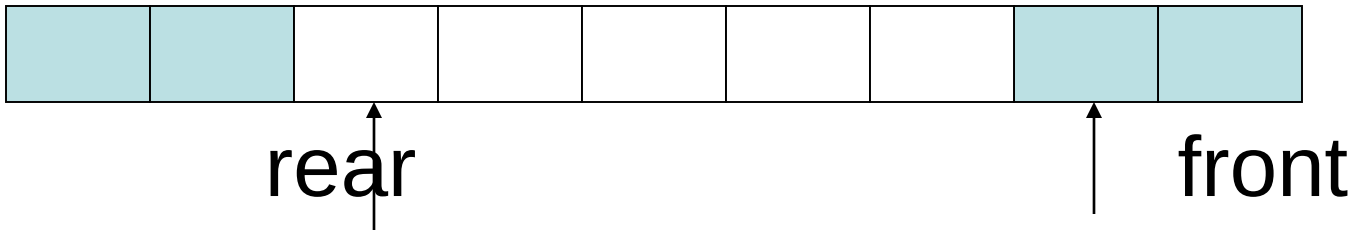
Bounded buffer

Deposit:

buf[rear] = data; rear = (rear + 1) % n;

Fetch:

Result = buf[front]; front = (front + 1) % n;



Bounded buffer – po jedan proizvođač i potrošač

```
typeT buf[n];
int front = 0, rear = 0;
sem empty = n, full = 0;
process Producer {
    while (true) { …
        /* proizvedi podatak */
        P(empty);
        buf[rear] = data; rear = (rear + 1) % n;
        V(full); }
}
```

```
process Consumer {
    while (true) { …
        /* dohvati rezultat */
        P(full);
        result = buf[front]; front = (front + 1) % n;
        V(empty);
        …}
}
```

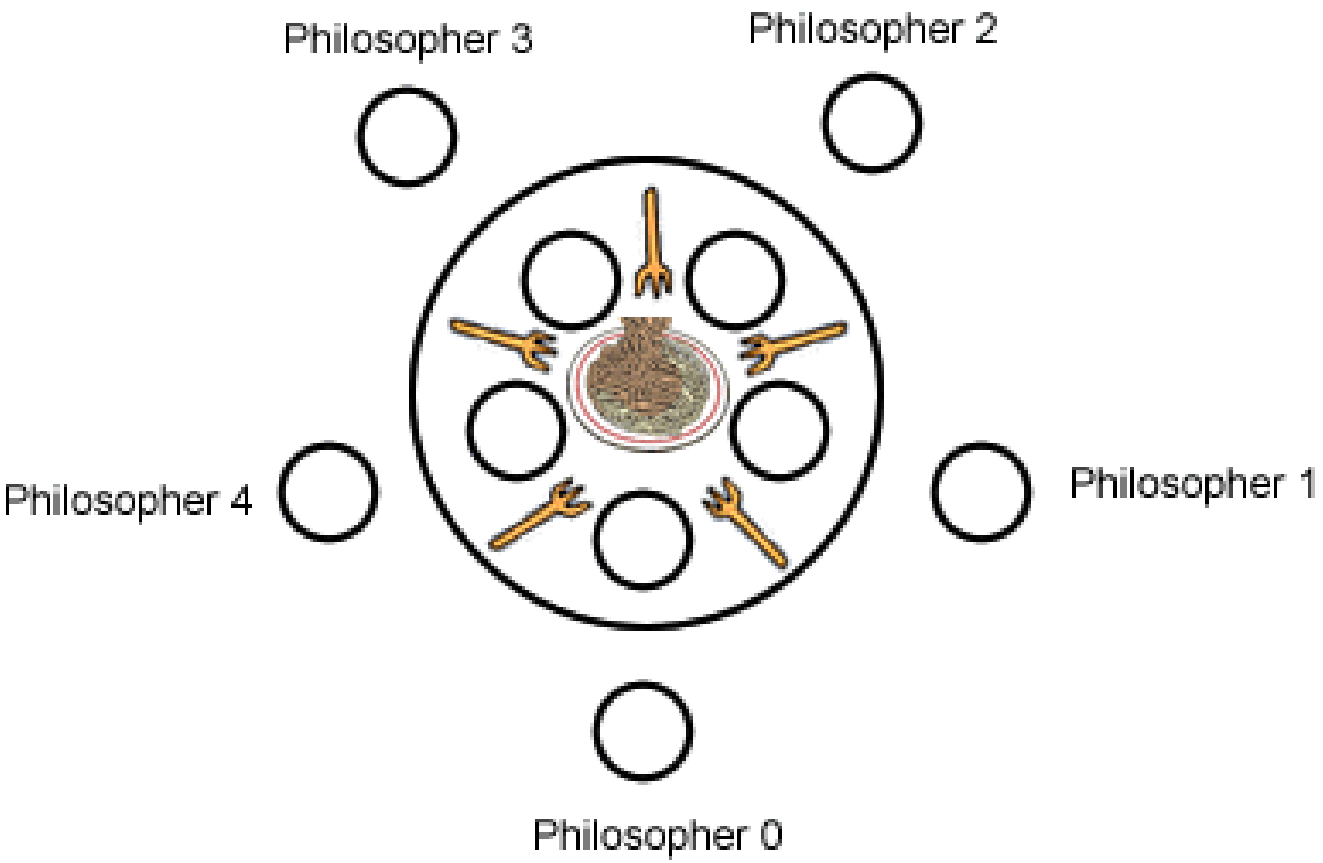
Bounded buffer – M proizvođača i N potrošača

```
typeT buf[n];
int front = 0, rear = 0;
sem empty = n, full = 0;
Sem mutexD = 1, mutexF = 1;
process Producer[i = 1 to M] {
    while (true) { …
        /* proizvedi podatke */
        P(empty); P(mutexD);
        buf[rear] = data; rear = (rear + 1) % n;
        V(mutexD); V(full); }
}
```

Bounded buffer (n)

```
process Consumer[j = 1 to N] {
    while (true) { …
        /* dohvati resultat i potroši */
        P(full); P(mutexF);
        result = buf[front]; front = (front + 1) % n;
        V(mutexF); V(empty)
        …}
}
```

Dinning philosophers (1)



Dinning philosophers (2)

```
process Philosopher[i = 0 to 4] {
    while (true) {
        razmišljaj;
        dohvati viljuške;
        jedi;
        oslobodi viljuške; }
}
```

Viljuške su deljene – kritične sekcije

Deadlock ako svi procesi imaju isti redosled dohvatanja (npr. leva pa desna)

Kružno čekanje (zatvoren put zahteva za resursima)

Dinning philosophers (3)

```
sem fork[5] = {1,1,1,1,1};
process Philosopher[i = 0 to 3] {
    while (true) { /* leva pa desna */
        P(fork[i]); P[fork[i+1];
        eat;
        V(fork[i]); V(fork[i+1]);
        think; }
}
```

Dinning philosophers (4)

```
process Philosopher[4] {
    while (true) {
    /* desna pa leva */
        P(fork[0]); P[fork[4];
        eat;
        V(fork[0]); V(fork[4]);
        think; }
}
```

Readers/Writers problem

- Procesi koji čitaju (Readers) i procesi koji pišu (Writers) dele bazu podataka, datoteku, listu, tabelu, ...
- Readers – samo čitaju, Writers – čitaju i pišu
- Transakcije iz konzistentnog stanja u konzistentno stanje
- Writer zahteva ekskluzivni pristup, ako nema Writer-a, bilo koji broj Reader-a
- Reader-i se takmiče sa Writer-ima, Writeri sa drugim Writer-ima

Jednostavan kritičan region – rešenje za R/W

```
sem rw = 1;
process Reader[i = 1 to m] {
   while (true) { …
        P(rw);
        read database;
        V(rw); … }}
process Writer[j = 1 to n] {
   while (true) { …
        P(rw);
        write database;
        V(rw); … }}
```

Izbegavanje preteranih ograničenja

- Reader-i nisu mogli istovremeno da čitaju po prethodnom rešenju
- Ideja – samo prvi Reader zahteva međusobno isključivanje
- Dokle god ima još Reader-a koji čitaju bazu podataka – nije potrebno međusobno isključivanje
- Potreban je brojač **aktivnih** Reader-a
- Inkrementiranje i dekrementiranje brojača aktivnih Reader-a

Rešenje 1 Readers/Writers – coarse grain

```
int nr = 0; sem rw = 1;
process Reader[i = 1 to m] {
   while (true) { …
        < nr = nr + 1;    if (nr == 1) P(rw); >
        read database;
         < nr = nr - 1;    if (nr == 0) V(rw); > … }}
process Writer[j = 1 to n] {
   while (true) { …
        P(rw);
        write database;
        V(rw); … }}
```

Rešenje 1 – fine grain (1)

```
int nr = 0; sem rw = 1; sem mutexR = 1;
process Reader[i = 1 to m] {
   while (true) { …
        P(mutexR);
           nr = nr + 1;
           if (nr == 1) P(rw);
        V(mutexR)
        read database;
```

Rešenje 1 – fine grain (2)

```
        P(mutexR);
           nr = nr - 1;
           if (nr == 0) V(rw);
        V(mutexR); … }}
process Writer[j = 1 to n] {
   while (true) { …
        P(rw);
        write database;
        V(rw); … }}
```

Rešenje 1 - problemi

- Preferiraju se Reader-i, nije ravnopravno
- Kada jednom počne niz čitanja, Reader-i imaju prioritet
- Drugo rešenje koje je bazirano na raspodeljenim binarnim semaforima u kojima se definiše tačan predikat koji uključuje i broj aktivnih Writer-a
- RW: (nr == 0 or nw == 0) and nw <= 1

Coarse grain – predicate based

```
int nr = 0; nw = 0;
process Reader[i = 1 to m] {
   while (true) { …
        < await (nw == 0) nr = nr + 1; >
        read database;
         < nr = nr - 1; > … }}
process Writer[j = 1 to n] {
   while (true) { …
         < await (nr == 0 and nw == 0) nw = nw + 1; >
                write database;
        < nw = nw - 1; >  … }}
```

Prosleđivanje štafete

- Jedan semafor e za ulazak u svaki atomski iskaz
- Svaki boolean u <await (B) S;> je zamenjen sa semaforom i brojačem
- Semafori r i w i broj zakasnelih reader-a i Writer-a dr i dw
- Raspodeljeni semafori – najviše jedan može da bude 1 u svakom trenutku, P je uvek praćen sa V
- SIGNAL – kôd koji definiše za koji semafor je V

Signal kôd

```
if (nw == 0 and dr > 0) {
      dr = dr – 1; V(r);
      }
elseif (nr == 0 and nw == 0 and dw > 0) {
      dw = dw – 1; V(w);
      }
else
      V(e);
```

Prosleđivanje štafete – kôd (1)

```
int nr = 0, nw = 0; /* broj aktivnih readera i writera */
sem e = 1, r = 0, w = 0; /* raspodeljeni semafori */
int dr = 0; dw = 0; /* broj zakasnelih readera i writera */
process Reader[i = 1 to m] {
    while (true) { …
        P(e);
        if (nw > 0) { dr = dr + 1; V(e); P(r); }
        nr = nr + 1;
        SIGNAL;
        read database;
        P(e);
        nr = nr - 1;
        SIGNAL; … }}
```

Prosleđivanje štafete – kôd (2)

```
process Writer[j = 1 to n] {
    while (true) { …
          P(e);
        if (nr > 0 or nw > 0) { dw = dw + 1; V(e); P(w); }
        nw = nw + 1;
        SIGNAL;
        write database;
        P(e);
        nw = nw - 1;
        SIGNAL; … }}
```

Pojednostavljenje Signal kôda

- U Readers procesu – prvi signal nw == 0

if (dr > 0) { dr = dr – 1; V(r); }

else V(e);

- U Readers procesu – drugi signal nw == 0 i dr == 0
- U Writers procesu – prvi signal nr == 0 i nw == 1
- U Writers procesu – drugi signal nw == 0 i nr == 0

Prosleđivanje štafete kompletno (1)

```
int nr = 0, nw = 0; /* Broj aktivnih readera i writera */
sem e = 1, r = 0, w = 0; /* Raspodeljeni semafori */
int dr = 0; dw = 0; /* Broj zakasnelih readera i writera */
process Reader[i = 1 to m] {
    while (true) { …
        P(e);
        if (nw > 0) { dr = dr + 1; V(e); P(r); }
        nr = nr + 1;
        if (dr > 0) {dr = dr - 1; V(r); }
        else V(e);
```

Prosleđivanje štafete kompletno (2)

```
        read database;
        P(e);
        nr = nr - 1;
        if (nr == 0 and dw > 0)  { dw = dw – 1; V(w);}
        else V(e);       … }}
process Writer[j = 1 to n] {
   while (true) { …
         P(e);
        if (nr > 0 or nw > 0) { dw = dw + 1; V(e); P(w); }
        nw = nw + 1;
        V(e);
```

Prosleđivanje štafete kompletno (3)

```
write database;
        P(e);
        nw = nw - 1;
        if (dr > 0)  { dr = dr – 1; V(r);  }
        elseif (dw > 0)  { dw = dw – 1; V(w);
        else V(e); … }}
```

Menjanje preferenci

- Reader-i se zakašnjavaju ako Writer čeka
- Zakasneli Reader se budi samo ako nema Writer koji trenutno čeka

  Zakašnjavanje Readera: if (nw > 0 or **dw > 0**)

  { dr = dr + 1; V(e); P(r); }

  Buđenje Writera:

  if (dw > 0) { dw = dw – 1; V(w); }
  elseif (dr > 0) { dr = dr – 1; V(r); }
  else V(e);

Alokacija resursa i Monitori

Problem alokacije resursa

- Kada se procesu može dozvoliti pristup resursu?
- Pristup kritičnoj sekciji, štampaču, bazi podataka, datoteci, lokacije u baferu ograničene veličine, ...
- U kritičnoj sekciji – da li proces ima dozvolu – ne kom procesu se daje dozvola
- Readers/Writers, Readers su imali prioritet, ali ne na nivou pojedinog procesa
- Generalno rešenje gde zahtev ima parametre – tip jedinica i broj jedinica

Generalni zahtev i oslobađanje

request(parameters) :

<await (request satisfied) take units;>

...

release(parameters) :

<return units;>

- atomske akcije su neophodne
- brojevi uzetih i vraćenih jedinica nisu nužno isti.

Prosleđivanje štafete

request(parameters) :
P(e);
if (request not satisfied) *DELAY*;
take units;
SIGNAL;
release(parameters) :
P(e);
return units;
SIGNAL;

DELAY i SIGNAL kôd

- **DELAY izgleda kao**

if (request not satisfied) { dp = dp + 1; V(e); P(rs);}

gde je rs request semafor

- **za svaki delay condition (uslov za zakašnjavanje), postoji poseban semafor**
- **SIGNAL kod zavisi od tipa problema za alokaciju resursa**

Shortest Job Next Allocation SJN

- jedan deljeni resurs
- request(time,id), time je integer – koliko dugo
- resurs free => odmah alokacija
- resurs not free => delay queue poređan u skladu sa parametrom time – vremenom koliko će proces da zadržava resurs
- release – resurs se daje procesu u delay queue sa minimalnom vrednošću parametra time
- minimizacija srednjeg vremena završetka posla

SJN predikat

- Unfair i starenje procesa sa velikim vremenima – potrebne modifikacije
- **free** – Boolean i **pairs** – skup record-a (time,id) procesa poređanih po time

SJN: **pairs** je uređen skup $\wedge$ **free** => (pairs == 0)

- Resurs ne sme da bude free ako postoje zahtevi koji čekaju

SJN coarse grain

- Ako ignorišemo SJN politiku

bool free = true;

request(time,id) : <await (free) free = false;>

release() : <free = true;>

- Potrebno je za proces sa najmanjim time
- Fine grain rešenje zasnovano na raspodeljenim semaforima

Fine grain rešenje - bazično

request(time,id) :
P(e);
if (!free) *DELAY*;
free = false;
SIGNAL;
release() :
P(e);
free = true;
SIGNAL;

SJN kroz DELAY i SIGNAL

- DELAY: umetni uređeni par parametara time i id u skup pairs zakašnjenih procesa i oslobodi kritičnu sekciju sa V(e), zakasni na semaforu za request
- Svaki proces ima različit delay uslov zavisno od položaja u pairs skupu
- b[n] niz semafora za n procesa – *id* u opsegu 0 do n-1
- DELAY uključuje uređivanje pairs skupa

Privatan semafor (1)

```
bool free = true;
sem e = 1, b[n] = ([n] 0);
typedef Pairs = set of (int, int);
Pairs pairs = ∅;
request(time,id) :
P(e);
if (!free) {
    umetni (time,id) u pairs;
    V(e); P(b[id]);}
free = false;
V(e);
```

Privatan semafor (2)

release() :

P(e);

free = true;

if (pairs != ∅) {

ukloni prvi uređeni par (time,id) iz skupa pairs;

V(b[id]); /* prosleđivanje štafete do procesa sa određenim proces id */}

else V(e);

Generalizacija alokacije korišćenjem prosleđivanja štafete

- Zamena boolean free sa integer-om **available**
- Request – ispitivanje **amount <= available** i alokacija nekog broja units
- Release – povećavanje **available** sa **amount, i ispitivanje amount <= available za** proces koji čeka sa najmanjom vrednošću time. Ako true => alokacija, u suprotnom V(e)

Problemi sa Semaforima

- P i V operacije treba da budu u parovima, raširenо po kodu
- Izvršavanje operacija na pogrešnom semaforu može da izazove deadlock, nije zaštićen kritičnom sekcijom, …
- Za međusobno isključivanje i uslovnu sinhronizaciju se koriste iste primitive.

Monitori

- Mehanizam apstrakcije podataka – enkapsulacija reprezentacije apstraktnih objekata – skup operacija je jedini put pristupa objektima
- Monitori – apstrakcija podataka + procesi koji pozivaju monitorske procedure
- Međusobno isključivanje – implicitno – kao <await>
- Sinhronizacija - condition promenljive

Modularizacija

- Aktivni procesi i pasivni monitor
- Sve što se radi u aktivnim procesima su pozivi procedura monitora – sva konkurentnost je sakrivena u monitoru za programere aktivnih procesa
- Programeri monitora mogu da menjaju implementaciju, dokle god se efekti ne menjaju
- Lako razgraničavanje odgovornosti među programerima

Struktura monitora

monitor mname {

deklaracija stalnih promenljivih

inicijalizacija

procedure }

Stalne promenljive – zadržavaju svoje vrednosti

Samo imena procedura su vidljiva preko poziva call mname.opname(argumenti)

Enkapsulacija

- Pristup samo kroz pozive procedura
- Pravila vidljivosti unutar monitora – nema referenciranja izvan monitora
- Stalne promenljive su inicijalizovane pre poziva procedura
- Programer monitora ne može da zna redosled pozivanja monitorskih procedura
- Monitorska invarianta – predikat smislenih stanja, uspostavljenih inicijalizacijom, održavanih procedurama

Međusobno isključivanje

- Monitorska procedura je aktivna ako proces izvršava iskaz u proceduri
- Najviše jedna instanca monitorske procedure je aktivna u jednom trenutku
- U praksi, najčešće skriveni locks ili semafori obezbeđuju međusobno isključivanje

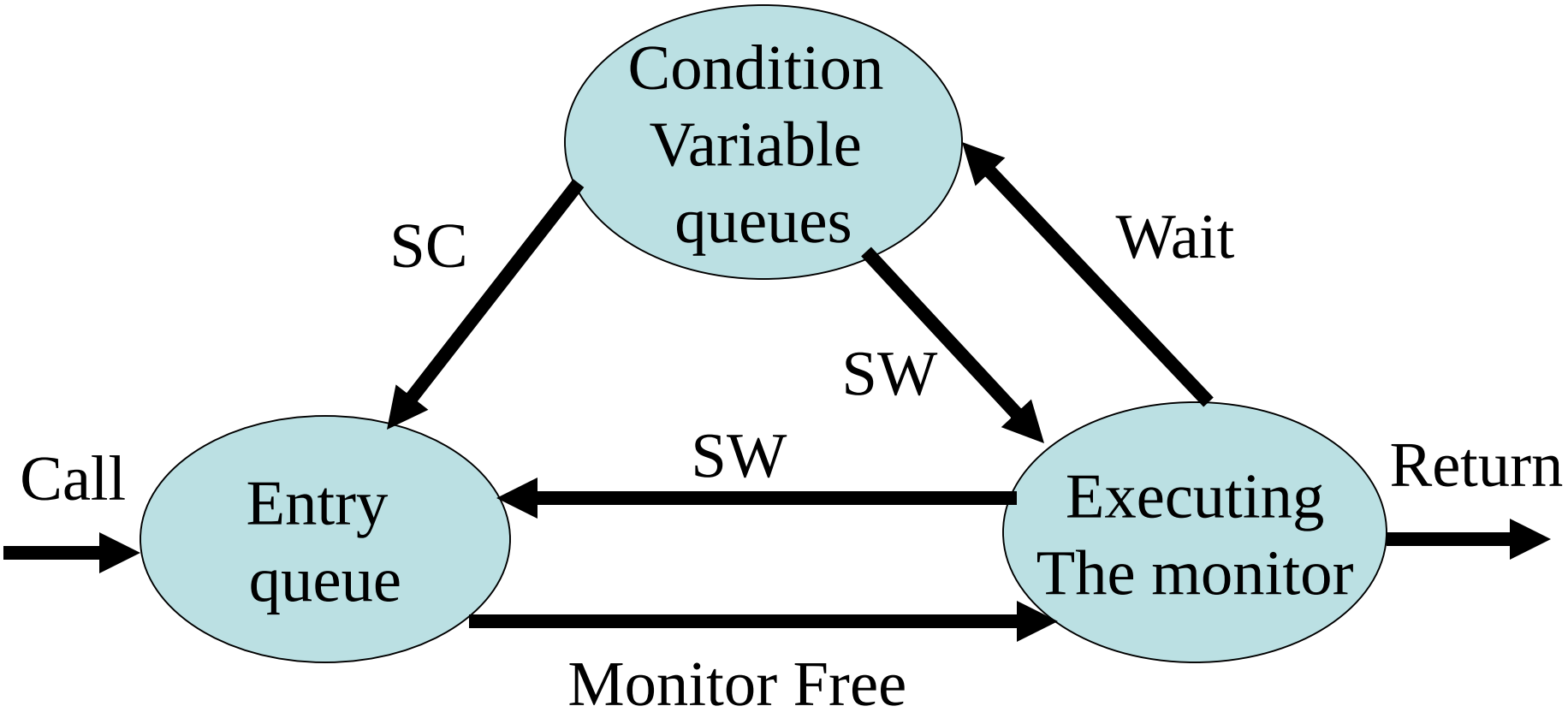
Uslovne promenljive

- Zakašnjavanje ako stanje monitora ne zadovoljava neki boolean uslov
- cond cv;
- cv je queue zakašnjenih procesa koji nisu direktno vidljivi programeru
- empty(cv) funkcija za ispitivanje stanja queue-a
- wait(cv) kašnjenje i odricanje ekskluzivnog pristupa – drugi procesi ulaze
- signal(cv) signal budi proces koji je na početku queue – u suprotnom **nema efekta**

Signal discipline

- Proces dolazi do signal i ima implicitni lock (međusobno isključivanje) – šta je sledeće ako signal budi drugi proces
- Signal and Continue – nonpreemptive, zadržava ekskluzivnu kontrolu
- Signal and Wait – preemptive, prosleđuje kontrolu probuđenom procesu, odlazi na red čekanja (Signal and Urgent Wait – Prioritet)

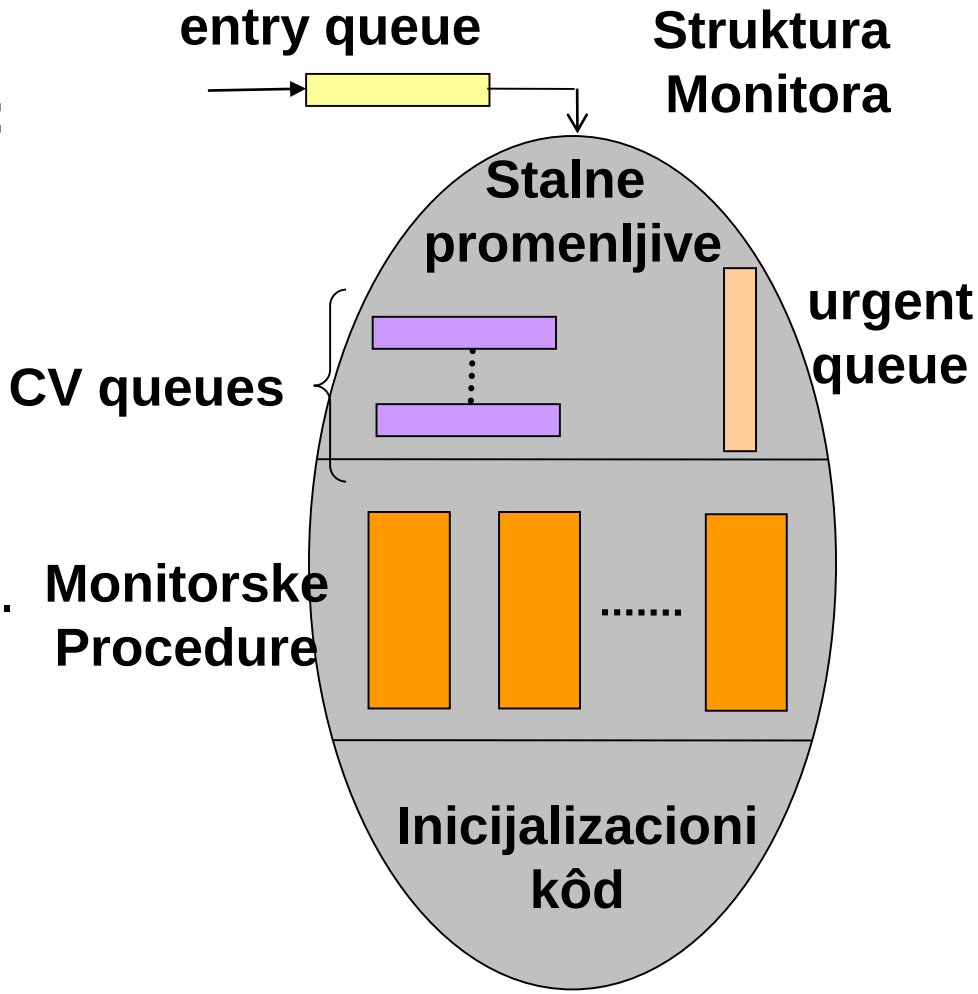
Dijagram stanja za monitore



Redovi u monitorima

Monitori održavaju puno različitih redova čekanja:

- Entry Queue → Postoji da se poređaju procesi i obezbedi međusobno isključivanje u pristupu monitoru.
- CV Queues → redovi za suspendovane procese koji čekaju na ispunjenje uslova.
- Urgent Queue → Da se implementira Signal and Urgent Wait – ređa procese koji su dali signal drugima i često treba da napuste monitor.



Implementacija Monitora pomoću Semafora (to uradi kompajler)

Monitor pomoću semafora:

```
P(mutex)
//najviše jedan proces aktivan u
     monitoru, međusobno
     isključivanje monitorskih
     procedura
…
  tela monitorskih procedura
…
if next.count > 0   V(next);
  else V(mutex);
// next je semafor koji služi za
     urgent queue !!, a next.count
     brojač zakasnelih na urgent
     queue
```

CV.wait: →

```
CV.count ++;
if next.count > 0 V(next);
else V(mutex);
P(CV_sem);
CV.count --;
```

CV.signal: →

```
if CV.count > 0 {
  next.count ++;
  V(CV_sem);
   P(next);
   next.count --;
}
```

Monitor za Semafore

```
monitor Semaphore {
    int s = 0; /* ili neka druga inicijalna vrednost */
    cond pos; /* signal kada je s inkrementirano */
    procedure Psem() {
        while (s == 0) wait(pos);
        s := s-1; }
    procedure Vsem() {
        s:= s + 1;
        signal(pos); }}
```

Signal and Continue za Semafore

- Predikat s >= 0 mora da bude očuvan
- Posle signal, proces koji izvršava Vsem nastavlja – vrednost s nije neminovno > 0 kada probuđeni proces dobije lock
- To je razlog zbog koga je while petlja neophodna – da se ponovo ispita s!
- Nije garantovano FIFO semafor – više procesa može da bude u entry queue – posle čekanja na uslovnoj promenljivoj queue

Signal and Wait za Semafore

- FIFO je zagarantovan – direktno na osnovu uslovne promenljive queue
- Nema potrebe za while petljom – ponovno ispitivanje vrednosti s nije neophodno =>

  if (s == 0) wait(pos);
- Proces iz pos queue, koji je dobio Signal, izvršava se odmah
- Takođe delovi procedura između signal-a su kritične sekcije (atomske akcije)

Prosleđivanje uslova - FIFO semafor

```
monitor FIFOsemaphore {
    int s = 0; /* ili druga inicijalna vrednost */
    cond pos; /* signal kada bi trebalo s > 0 */
    procedure Psem() {
        if (s == 0)    wait(pos);
        else   s := s-1; }
    procedure Vsem() {
        if (empty(pos))     s:= s + 1;
        else                signal(pos); }}
```

Osobine Prosleđivanja uslova

- Sada se ne menja stalna promenljiva s kada ima zakasnelih, već prosleđivanje uslova vidi **samo** proces koji je došao na red u **CV queue**
- Nema mogućnosti drugi proces da vidi promenu – samo monitor!
- Nema while petlje ni za SC
- Garantovan FIFO, jer se koristi FIFO CV

Signal and Continue

- Koristi se u Unix i Javi
- Kompatibilno sa raspoređivanjem na osnovu prioriteta
- Jednostavnija semantika – kompletne procedure su kritične sekcije
- wait(cv, rank) – najmanji rank - prioritet
- signal_all(cv) – probudi sve procese
- minrank(cv) – rank na početku queue

Bounded buffer(1)

```
monitor Bounded_Buffer {
    typeT buf[n];
    int front = 0, rear = 0, count = 0;
    cond not_full, not_empty;
    procedure deposit(typeT data) {
    while (count == n) wait(not_full);
    buf[rear] = data; rear = (rear + 1) % n; count++;
    signal(not_empty);    }
```

Bounded buffer(2)

```
procedure fetch(typeT &result) {
    while (count == 0) wait(not_empty);
    result = buf[front];  front = (front + 1) % n;
    count--;
     signal(not_full);
    }
}
```

Readers/Writers – broadcast signal

- Monitor samo daje dozvole za pristup
- Request i release za read i write
- Broj reader-a i writer-a nr i nw
- Predikat – monitorska invarianta

$(nr == 0 \vee nw == 0) \wedge nw <= 1$

Readers/writers monitor (1)

```
monitor RW_Controller {
   int nr = 0, nw = 0;
   cond oktoread; /* signaled when nw == 0
   */
   cond oktowrite; /* both nr and nw are 0 */
procedure request_read() {
   while (nw > 0) wait(oktoread);
   nr = nr + 1; }
```

Readers/writers monitor (2)

```
Procedure release_read() {
        nr = nr - 1;
        if (nr == 0) signal(oktowrite); }
Procedure request_write() {
   while (nr > 0 | | nw > 0) wait(oktowrite);
   nw = nw + 1; }
Procedure release_write() {
        nw = nw - 1;
        signal(oktowrite); signal_all (oktoread);  }}
```

Shortest next allocation monitor

```
monitor Shortest_Job_Next {
    bool free = true;
    cond turn;
    procedure request(int time)  {
        if (free) free = false;
        else wait(turn, time);  }
    procedure release()
        if (empty(turn)) free = true
        else signal(turn); }}
```

Interval Timer i Brica koji spava

Interval Timer Monitor

- Dve operacije (procedure)

delay (interval) i tick – proces koga budi hardverski tajmer sa visokim prioritetom

wake_time = tod + interval;

Svaki proces koji poziva – privatno vreme buđenja wake_time

Po jedna uslovna promenljiva za svaki proces?

Kada je tick, stalna promenljiva se proverava i radi se signal ako je uslov ispunjen - glomazno

Covering condition

- Jedna uslovna promenljiva
- Kada bilo koji od prekrivenih uslova (covered condition) može da postane istinit – svi procesi se bude
- Svaki proces ponovo ispituje svoj condition
- Buđenje i procesa koji mogu mogu da pronađu da je delay condition false

Covering condition monitor

```
monitor Timer {
    int tod = 0;        cond check;
    procedure delay(int interval) {
        int wake_time;
        wake_time = tod + interval;
        while (wake_time > tod) wait(check); }
    procedure tick() {
        tod = tod + 1;
        signal_all(check); }}
```

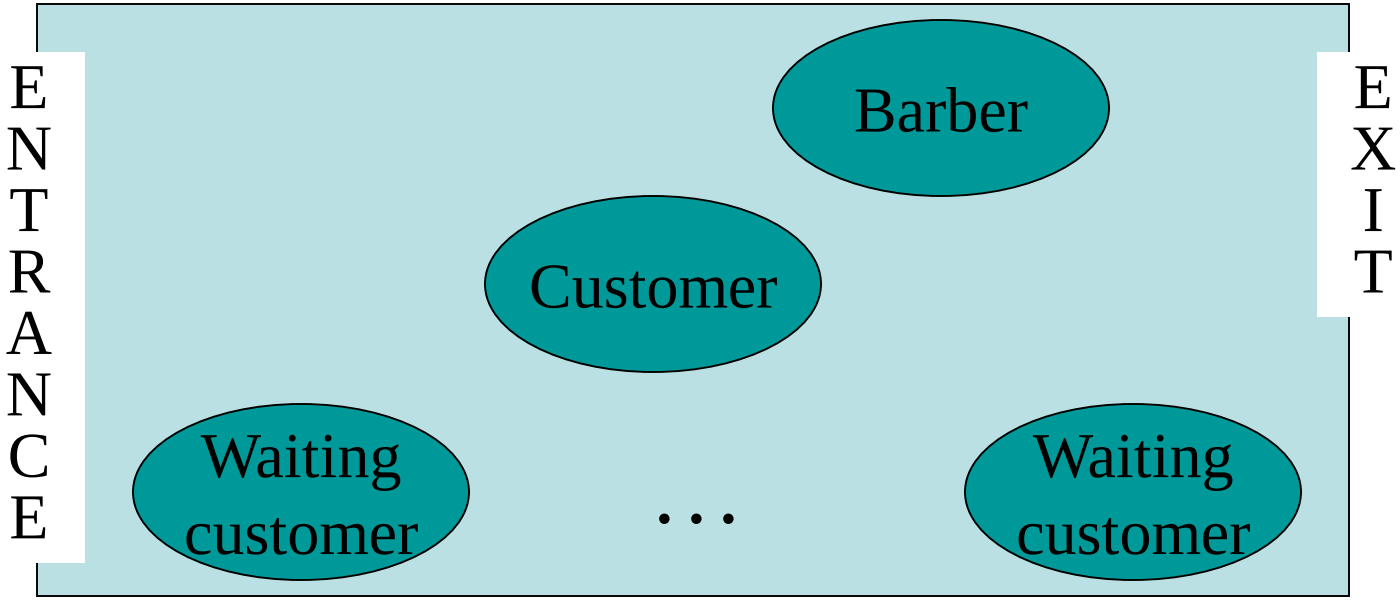
Priority Wait Timer

```
monitor Timer {
    int tod = 0;        cond check;
    procedure delay(int interval) {
     int wake_time;
     wake_time = tod + interval;
     if (wake_time > tod) wait(check, wake_time); }
    procedure tick() {
     tod = tod + 1;
     While (!empty(check) && minrank(check) <=
    tod) signal(check); }}
```

Prednosti priority wait timer-a

- Procesi su poređani u skladu sa vremenima buđenja
- minrank se koristi za odredjivanje da li da se probudi sledeći proces koji je zakašnjen
- While petlja za signal je neophodna za slučaj kada nekoliko procesa ima isto wake_time
- Kompaktno, efikasno – zasnovano na statičkom redosledu između delay uslova

Sleeping Barber Rendezvous



Client/Server relationship

- Frizerska radnja kao monitor, klijenti zahtevaju šišanje – procesi, brica – proces
- Procedure get_haircut, get_next_customer finished_cut
- Brica – get_next_customer i finished_cut za napuštanje radnje
- Potreba za sinhronizaciju brice i klijenta – rendezvous – barijera za dva procesa (ali brica sa bilo kojim klijentom)

Sinhronizaciona stanja

- Klijenti – sedanje u stolicu i napuštanje
- Brica – postaje raspoloživ, šišanje i napuštanje
- cinchair i cleave, bavail, bbusy i bdone brojači, inicijalno 0 i inkrement za beleženje koliko je procesa dostiglo to stanje

C1: cinchair >= cleave $\wedge$ bavail >= bbusy >= bdone

C2: cinchair <= bavail $\wedge$ bbusy <= cinchair

C3: cleave <= bdone **C1 $\wedge$ C2 $\wedge$ C3**

Redefinisanje brojača

- Problem beskonačnog inkrementiranja
- Promena promenljivih uvođenjem razlika

barber == bavail – cinchair

chair == cinchar – bbusy

open == bdone – cleave

Vrednosti su samo 0 i 1

Monitor Barber_Shop

int barber = 0, chair = 0, open = 0;

cond barber_available; /* signal kada barber > 0 */

cond chair_occupied; /* signal kada chair > 0 */

cond door_open; /* signal kada open > 0 */

cond customer_left; /* signal kada open == 0 */

/* četiri sinhronizaciona uslova: za customera - barber raspoloživ i barber da otvori vrata, za barbera – customer da stiže u stolicu i customer da napušta radnju */

Monitor Barber_Shop (1)

```
procedure get_haircut() {
 while (barber == 0)
wait(barber_available);
 barber = barber – 1;
 chair = chair + 1; signal(chair_occupied);
 while (open == 0) wait(door_open);
 open = open – 1; signal(customer_left);
}
```

Monitor Barber_Shop (2)

```
procedure get_next_customer() {
 barber = barber +1;
signal(barber_available);
 while (chair == 0) wait(chair_occupied);
 chair = chair – 1; }
procedure finished_cut() {
 open = open + 1; signal(door_open);
 while (open > 0) wait(customer_left); }}
```

Problem kružnog toka

Tekst zadatka

- Раскрсница кружног тока има 3 двосмерне улице са по једном саобраћајном траком у сваком смеру повезане на кружни ток. Предност у кружном току имају возила која се већ налазе у кружном току. Реализујте монитор у коме ће се регулисати саобраћај кружног тока. Сматрајте да се, ако у претходном сегменту кружног тока има возила, мора чекати на улазу у кружни ток. Такође, возила морају чекати возила испред себе на улазној улици у кружни ток (једна улазна саобраћајна трака).

У решењу, непотребно задржавање процеса за возила у монитору није дозвољено. Позивање више различитих мониторских процедура за процесе возила је дозвољено. Смер кретања у кружном току је прва_ул-друга_ул-трећа_ул-прва_ул. Претпоставите да сегмент кружног тока може да прими неограничен број возила.

Početni komentari

- U postavci su rečenice: У решењу, непотребно задржавање процеса за возила у монитору није дозвољено. Позивање више различитих мониторских процедура за процесе возила је дозвољено. – **VIŠAK**
- Kada bi se pravile procedure tipa: **od-ulaza-do-izlaza**, proces bi nepotrebno zadržavao monitor i **dozvoljavala bi da samo jedno vozilo bude u raskrsnici** – zadržavanje skrivene kritične sekcije jako dugo. Ni sam proces za vozilo ne bi mogao da radi.

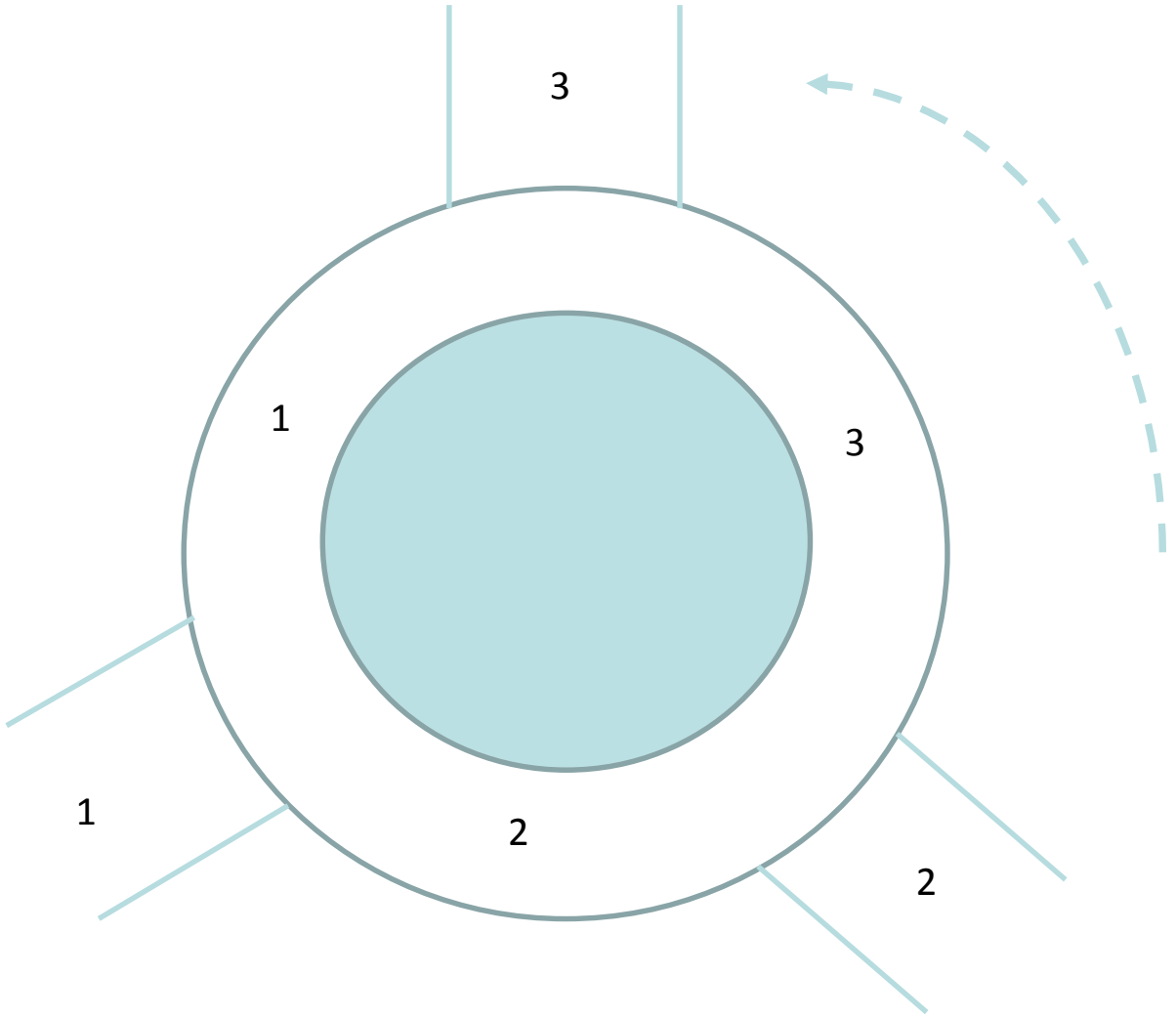
Komentar - konkurentnost

- Kada bi se pravile procedure od **ulaza segmenta do izlaza tog segmenta**, **zadržavao bi se monitor sve vreme dok vozilo putuje kroz segment** – mnogi procesi bi čekali da započnu monitorsku proceduru, pa bi čekanja kada ima više vozila bila primarno zbog zadržavanja u entry queue monitora
- Jedno vozilo u segmentu zadržava sve segmente zbog međusobnog isključivanja monitorskih procedura

Jedino rešenje za procedure

- Procedure za ulaz u segment i napuštanje segmenta sigurno ako se prolazi samo jedan segment
- Šta ako se prolazi kroz više segmenata?
- Prelaz iz segmenta u segment je trenutan – treba uvesti dodatnu proceduru za prelazak iz segmenta u segment **jer ne sme da bude zadržavanja monitora**!!!

Numerisanje segmenata i ulaza



Pozivajući procesi

- Šta sa pozivajućim procesima?
- Ako je jedan segment (ulazi na 1 a izlazi na 2): call kruzni_tok.start (1), **radi nešto malo lokalno**, call kruzni_tok.leave (2)
- Npr. Od 1 do 3
- Za dva segmenta (ulazi na 1 a izlazi na 3): call kruzni_tok.start (1), **radi nešto malo lokalno**, call kruzni_tok.moveto(3) **radi nešto malo lokalno** i call kruzni_tok.leave (3)

Kako održati FIFO queue na ulazu

- FIFO na ulazu u kružni tok je obezbeđen u Signal and Wait disciplini
- Rešenje koje obezbeđuje FIFO za Signal and Continue je kompleksnije
- Ako je moguć izbor discipline – Signal and Wait

Kod monitora kruzni_tok – Signal and Wait

```
Monitor  kruzni_tok;
const N = 3;
var i : integer; var count : array[1..N] of integer;
enter : array[1..N] of condition;

procedure start(segment : integer);
begin
  if ((enter[segment].queue()) OR (count[segment] <> 0)) then enter[segment].wait(ID);
  count[(segment mod N) + 1] := count[(segment mod N) + 1] + 1;
end;
```

```
procedure leave(segment : integer);
begin
 count[segment] := count[segment] - 1;
 while((enter[segment].queue()) and (count[segment] = 0)) do enter[segment].signal();
end;
```

```
procedure moveTo(segment : integer);
begin
  count[segment] := count[segment] + 1;
  count[(segment + N - 1) mod N) := count[(segment + N - 1) mod N) – 1;
while((enter[segment + N - 1) mod N].queue()) and (count[segment + N - 1) mod N] = 0)) do enter[segment + N - 1) mod N].signal();
end;

begin
  ID := 0;
  for i := 1 to N do count[i] := 0;
end.
```

Distribuirano programiranje

- Definisanje specijalnih mrežnih operacija koje uključuju sinhronizaciju – message passing primitives
- Procesi dele kanale – komunikacione puteve između procesa
- Tipično kanali su jedini objekat koji procesi dele
- Protok informacija u jednom ili dva smera
- Blokirajući ili neblokirajući

Asinhroni message passing

- Send i receive primitive
- Kanal chan ch($type_1$ id_1, …, $type_n$ id_n); tipovi su neophodni, id su opcioni
- send ch($expr_1$, …, $expr_n$); tipovi usklađeni sa onima za kanal
- receive var(var_1, …, var_n); tipovi usklađeni sa onima za kanal
- Receive kasni proces sve dok se ne pojavi bar jedna poruka – blokirajući
- Send je neblokirajući – neograničen queue
- Queue je FIFO – čuva se redosled

Filterski proces - mailbox

```
chan input(char), output(char [MAXLINE]);
process Char_to_Line {
    char line[MAXLINE]; int i = 0;
    while (true) { receive input(line[i]);
      while (line[i] != CR and i < MAXLINE) {
      i = i + 1; receive input(line[i]); }
      line[i] = EOL;
      send output(line);
      i = 0; }}
```

Filtri – mreža za sortiranje

- Filter – više ulaznih i više izlaznih kanala
- Izlaz – funkcija ulaza i inicijalnog stanja
- Predikat – povezuje izlaz sa ulazom, svaki put kada filter šalje na izlaz
- Primer sortiranje – ulazni kanal prima n vrednosti koje treba da budu sortirane, izlaz daje sortirane vrednosti

Predikat i terminacija

- $\forall$ i: 1 <= i < n: sent[i] <= sent[i+1] $\wedge$ permutacija ulaznih vrednosti
- Terminacija : n poznato unapred, n prva vrednost, i sentinel value
- Procesi u OS – jedan proces u 3 koraka: primi sve ulaze, sortiraj, pošalji sortirane brojeve
- Mreže za sortiranje – mreža malih procesa koji se izvršavaju u paraleli
- Mreže za stapanje – od dve sortirane liste se formira jedna sortirana lista

Predikati i procesi mreže za stapanje

- in1 and in2 empty $\wedge$ sent[n+1] == EOS $\wedge$ ($\forall$ i: 1 <= i < n: sent[i] <= sent[i+1]) $\wedge$ permutacija ulaznih vrednosti in1 and in2 – kada terminira
- Prima sve i pamti ili stvara tok kreiran stalnim poređenjem ulaznih vrednosti
- Nizovi koji kreiraju izlaze koji teku sa potencijalnim paralelizmom
- Mreža procesa i kanala
- pogodno kada je n stepen broja 2

Merge (1)

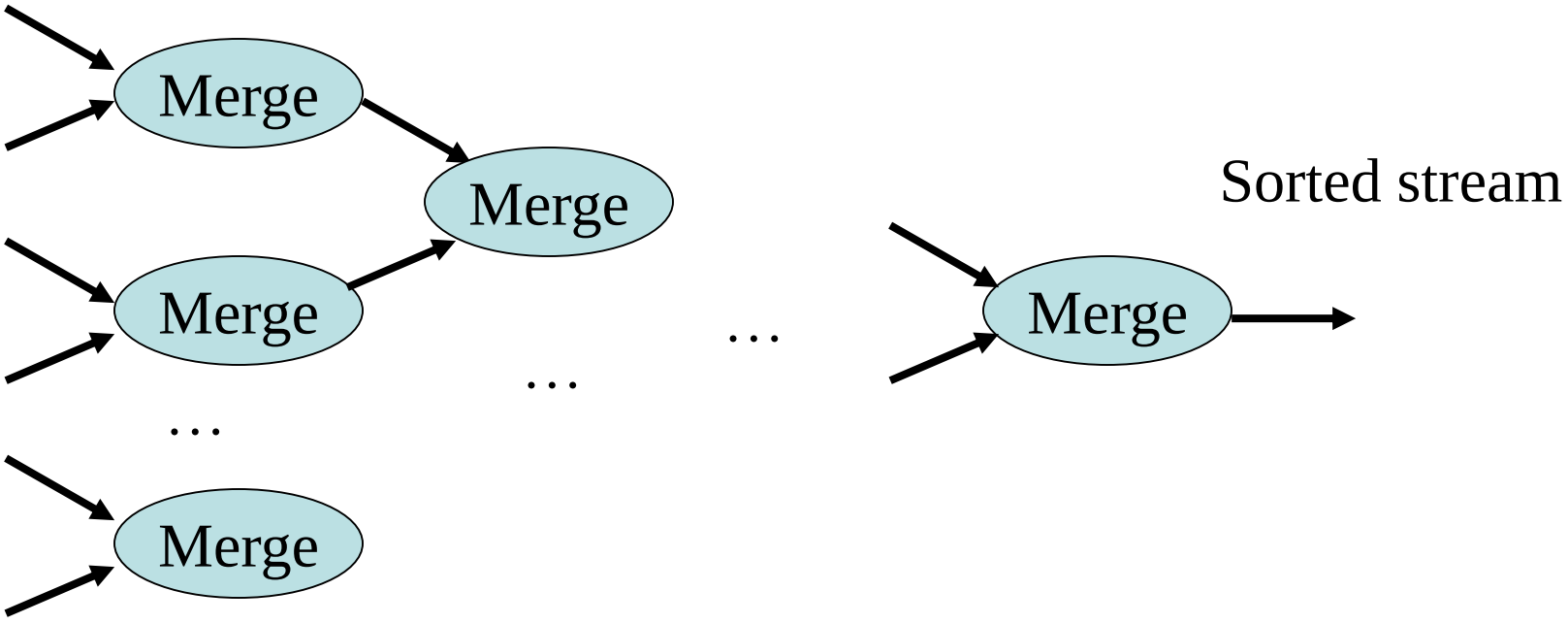
```
chan in1(int), in2(int), out(int);
process Merge {
    int v1,v2;
    receive in1(v1); receive in2(v2)
    while (v1 != EOS and v2 != EOS) {
        if (v1 <= v2)
                { send out (v1); receive in1(v1); }
        else
                { send out(v2); receive in2(v2); }}
```

Merge (2)

```
    if (v1 == EOS)
        while (v2 != EOS)
                { send out(v2); receive in2(v2); }
    else
        while (v1 != EOS)
                { send out(v1); receive in1(v1); }
    send out(EOS);
}
```

Interconnecting merge processes

- Binarno stablo sa n-1 procesa



Statičko i dinamičko imenovanje

- Statičko – svi kanali – globalan niz i pojave Merge (n-1) se preslikavaju u 2n-1 kanala – $Merge_i$ se preslikava u kanale – ugradnja stabla u niz
- dinamički – opet svi kanali su globalni, pridružuje svakom procesu tri kanala kada su kreirani – glavni proces dinamički kreira kanale i prosleđuje ih Merge procesima
- Fleksibilnost sa dinamičkim imenovanjem i kreiranjem

Klijenti i serveri – Aktivni Monitori

- Simulacija monitora korišćenjem serverskih procesa i message passing-a
- Stalne promenljive – Lokalne promenljive servera
- Server u petlji opslužuje pozive preko request i reply kanala
- Statičko imenovanje fiksan broj kanala i clientID
- Dinamičko – svaki klijent kreira klijenta sa privatnim reply kanalom

Aktivni monitori sa jednom operacijom

```
chan request(int clientID, types of input values);
chan reply[n] (types of results);
process Server { int clientID; declaration of
    permanent variables; initialization code;
    while (true) {
        receive request(clientID, input variables);
        results = f(input variables, permanent
    variables);
        send reply[clientID] (results); }
```

Aktivni monitori sa više operacija

process Client[i = 0 to n-1] {
send request(i, value arguments);
receive reply[i] (result arguments); }

Kompleksnije je jer ima više operacija

Različiti su argumenti, operacije i rezultati

Argumenti i rezultati se razlikuju po tipovima i broju parametara

Aktivni monitor – više operacija

type op_kind = enum(op_1, …, op_n);

type arg_type = union(arg_1, …, arg_n);

type result_type = union(res_1, …, res_n);

chan request(int clientID, op_kind,
arg_type);

chan reply[n] (result_type);

Aktivni monitor – više operacija (2)

process Server { int clientID; op_kind kind; arg_type args; result_type results; permanent variables; initialization code;

while (true) {

receive request(clientID, kind, args);

if (kind == op_1)

{body of op_1; }

…

else if (kind == op_n)

{body of op_n; }

send reply[clientID] (results); }}

Aktivni monitor – više operacija (3)

process Client[i = 0 to n-1] {

arg_type myargs; result_type myresults;

set value arguments in myargs;

send request(i, op_j, myargs);

receive reply[i] (myresults);

}

Razmatranje je bez uslovnih promenljivih.

Monitor based – Message based

Stalne promenljive – Lokalne serverske promenljive

Identifikatori procedura – kanal za zahtev i tipovi op

Pozivi procedura – send request(); receive reply()

Monitor entry – receive request()

Procedure return – send reply()

Wait iskaz – čuvanje zahteva koji čekaju

Signal iskaz – Budi se proces koji je čekao

Tela procedura – Delovi case iskaza zavisno od vrste operacije

Interacting Peers

- Komunikacioni pattern-i centralizovani, simetrični i prsten
- Svaki proces ima lokalnu vrednost – cilj: učenje najveće i najmanje vrednosti
- Rešenje 1: slanje vrednosti jednom procesu – nalazi max i min i šalje do svih 2(n-1)
- Rešenje 2: simetrično – šalje lokalne vrednosti svim drugim procesima i onda prima od svih drugih procesa i lokalno izračuna - n(n-1) poruka
- Rešenje 3: logički prsten – min i max primljenih i min i max od lokalnih vrednosti 2(n-1) poruka

Rešenje 1

```
chan values(int), result[n] (int smallest, int largest);
process P[0] {  # coordinator
    int v; # inicijalizovana vrednost
    int new, smallest = v, largest = v;
    for [i = 1 to n-1] {
        receive values(new);
        if (new < smallest)  smallest = new;
        if (new > largest) largest = new; }
```

Rešenje 1 (1)

```
 for [i = 1 to n-1]
   send results[i] (smallest, largest)
}
process P[i = 1 to n-1] {
   int v; # inicijalizovana vrednost
   int smallest, largest;
   send values(v);
   receive results[i] (smallest, largest); }
```

Rešenje 2 - Symmetric

```
chan values[n] (int);
process P[i = 0 to n-1] {  # symmetric no coordinator
    int v; # inicijalizovana vrednost
    int new, smallest = v, largest = v;
    for [j = 0 to n-1 st j != i]
        send values[j] (v);
    for [j = 0 to n-1st j != i ] {
        receive values[i] (new);
        if (new < smallest)  smallest = new;
        if (new > largest) largest = new; }}
```

Rešenje 3 - Ring

```
chan values[n] (int smallest, int largest);
process P[0] {  # initiator
    int v; # inicijalizovana vrednost
    int new, smallest = v, largest = v;
    send values[1] (smallest, largest);
    receive values[0] (smallest, largest);
    send values[1] (smallest, largest);
    }
```

Rešenje 3 – Ring (1)

```
Process P[i = 1 to n-1] {
    int v; # initialized
    int smallest, largest;
    receive values[i] (smallest, largest);
    if (v < smallest)  smallest = v;
    if (v > largest) largest = v;
    send values[(i+1) mod n] (smallest, largest);
    receive values[i] (smallest, largest);
    if (i < n-1) send values [i+1] (smallest, largest);
}
```

Sinhroni message passing

Sinhroni message passing

- Send je blokirajući synch_send
- Postoji granica u veličini bafera
- Direktno kopiranje iz adresnog prostora pošiljaoca u adresni prostor primaoca je moguće – adrese poruka koje čekaju da budu poslate
- Mane – smanjena konkurentnost, deadlock se lako napravi

Primer za deadlock

```
channel in1(int), in2(int);
process P1 {
    int value1 = 1, value2;
    synch_send in2(value1);
    receive in1(value2); }
process P2 {
    int value1, value2 = 2;
    synch_send in1(value2);
    receive in2(value1); }
```

CSP

process A { … B!e; … }

process B { … A?x; … }

Izlazni i ulazni iskazi. Direktno imenovanje. Distribuirano dodeljivanje

Destination!port(e_1, …, e_n);

Source?port (x_1, …, x_n);

Source[*] – bilo koji element niza procesa

Zaštićena (Guarded) komunikacija

- Kako sprečiti čekanje na portovima dok postoje drugi procesi koji mogu da komuniciraju na drugim portovima
- B; C -> S;
- B Boolean uslov, C komunikacioni iskaz zaštićen sa B, S lista iskaza
- zaštita uspeva ako je B true i izvršavanje C ne izaziva delay – neko čeka
- Zaštita ne uspeva ako je B false. Zaštita blokira ako je B true i C ne može da bude izvršeno – niko ne čeka

Izbegavanje deadlock-a

```
process P1 {
    int value1 = 1, value2;
    if P2!value1 -> P2?value2;
    [] P2?value2 -> P2!value1;
    fi }
process P2 {
    int value1, value2 = 2;
    if P1!value2 -> P1?value1;
    [] P1?value1 -> P1!value2;
    fi }
```

LINDA

- Generalizovanje deljenih promenljivih i asinhroni message passing
- Nije programski jezik – 6 primitiva za pristup prostoru torki - tuple space
- Tuple space – deljena asocijativna memorija etiketiranih zapisa - tagged data records
- Jedinstven deljeni komunikacioni kanal ka prostoru torki
- Pamti distribuirane strukture podataka kojima procesi konkurentno pristupaju – čak može trajno da pamti

Operacije u LINDA

- OUT – kao send
- IN – kao receive koji skuplja poruku sa kanala, blokirajući
- READ – kao receive koji čita poruku sa kanala i ostavlja je (ustvari ostavlja u prostoru torki), blokirajući
- EVAL – kreiranje procesa
- INP – nonblocking IN koji daje predikat
- RDP – nonblocking READ koji daje predikat

Tuple space (Prostor torki)

- Data tuple – tagged record-i
- Process tuples – procesi koji se izvršavaju asinhrono
- Data tuple (“tag”, $value_1$, … , $value_n$)
- Pamćenje data tuple OUT(“tag”, $expr_1$, …, $expr_n$);
- Evaluacija izraza expressions i pamćenje u prostoru torki

IN

- IN (“tag”, a:t, 55,…, true, k:t); izvlači tuple smešten (zapamćen) u tuple space, ako postoji **usklađena** torka
- IN parametri se zovu template – moraju se upariti: tagovi su identični, broj polja je isti i odgovarajuća polja imaju isti tip – potreban uslov
- “tag”, 55, true, su stvarni parametri i predstavljaju deo “naziva” torke. a:t, k:t su formalni parametri i označavaju lokacije u koje će se smestiti odgovarajuće vrednosti polja iz pročitane torke. ?a I ?k su alternativni na;ini obele\avanja lokacija. Torka je **usklađena**, ako su jednaki svi stvarni parametri.

Sinhronizacija na barijeri

Inicijalizacija vrednosti u prostoru torki (samo jedan proces inicijalizuje)

OUT(“barrier”, 0);

Svaki proces radi sledeće

IN(“barrier”, ?counter);

OUT(“barrier”, counter + 1);

Svih n procesa čekaju sve dok se ne dostigne vrednost n

RD(“barrier”,n); gde n ima konkretnu vrednost !!!

Procesne torke

- EVAL(“tag”, $expr_1$, …, $expr_n$);
- Jedan ili više izraza su pozivi procedura ili funkcija
- Teoretski paralelno izvršavanje za sva polja – u realnosti samo za polja koja su funkcije
- Postaje pasivan - data tuple kada sva polja dobijaju vrednost izraza u EVAL

Cobegin i send i receive sa torkama

Co [i = 1 to n]

    a[i] = f(i);

Ekvivalent u LINDI je:

for (i = 1; i <= n; i++)

    EVAL(‘a”, i, f(i));

Slanje na i prijem sa kanala

OUT(“ch”, expressions);

IN(“ch”, variables);

Remote Procedure Calls (RPC) i Rendezvouz (Randevu)

- Client/Server kod unidirekcionog message passinga je rađen sa dve eksplicitne razmene poruka
- Svaki klient je zahtevao zaseban reply kanal
- Uvodi se dvosmerni (bidirekcioni) komunikacioni kanal
- Client – send praćen sa receive
- Server – receive praćen sa send

Razlika između RPC i Randevua

- RPC – konceptualno svaki poziv operacije (procedure) – kreira novi proces
- Rendezvous – u okviru postojećeg procesa, postoji operacija koja može da se izvrši sa accept iskazom, gde serverski proces čeka da bude pozvan.
- Oba su sinhrona - blokirajuća

RPC

```
Module mname
    op opname1(formals) [return result];
    … # headers of exported operations
Body
Variable declarations;
Initialization code;
Proc opname1(formal identifiers) returns result identifier
    declarations of local variables;
    statements
End
… # other procedure bodies
```

Intermodulni i intramodulni pozivi

- Pozivanje je call mname.opname(arguments)
- mname se može izostaviti kod intramodulnih poziva
- Za operacije unutar modula se pretpostavlja da su u istom adresnom prostoru
- Novi procesi se kreiraju kad god je poziv intermodulni
- Serverski proces može da postane klijentski da bi ispunio svoj deo odgovornosti

Rendezvous - Randevu

- Pozivi su isti, ali se ne kreiraju novi procesi – opslužuju se od strane postojećeg procesa.
- Podrazumevaju i komunikaciju i sinhronizaciju
- Sekvencijalno opsluživanje operacija
- in opname(formal identifiers) -> S; ni
- S je zaštićeni iskaz u kome su formalni identifikatori vidljivi
- in (accept) je blokirajuće – kako da izbegnemo da se blokiraju programi?

Zaštićene alternativne operacije

in $op_1(formals_1)$ and B_1 by e_1 -> S_1;

[] …

[] in $op_n(formals_n)$ and B_n by e_n -> S_n;

ni

B_i – sinhronizacioni izrazi – Boolean – određuju kada je alternativa otvorena

e_i – izraz za raspoređivanje (prioritet) -

U Ada programskom jeziku – select iskaz i accept iskaz

Zaštićeni iskazi

- Uspevaju ako je operacija pozvana i sinhronizacioni iskaz je true
- Izvršavanje in je zakašnjeno sve dok neka zaštita ne uspe
- Ako više od jedne zaštite uspeva, o prioritetu odlučuje izraz za raspoređivanje
- Ako nema izraza za raspoređivanje – najstariji poziv operacije koja uspeva
- Obezbeđuju fleksibilnost da se ispune različiti zahtevi

Ada

- **SERVER**

```
task A is
   entry check_point
        (Input : INTEGER; Output : out INTEGER)
   end A;
task body A is
   Local : INTEGER;
begin
   ...
   accept check_point
       (Input : INTEGER; Output : out INTEGER) do
           Local := Input;
           Output := f(Input);
   end check_point;
   ...
end A;
```

ADA klient

- **CLIENT**

```
task B;
task body B is
  To_A, From_A : INTEGER;
begin
   To_A := ...;
   ...
   A.check_point(To_A, From_A);
   ...
end B;
```

Licitacija u ADI

```
task BID is
   entry raise_bid (From:BIDDER_ID;By:NATURAL);
   entry look_at (Value:out NATURAL);
end BID;
task body BID is
   Current_Bid : NATURAL := 0;
   Last_Bidder : BIDDER_ID;
begin
   accept raise_bid (From:BIDDER_ID;By:NATURAL) do
      Last_Bidder := From;
      Current_Bid := Current_Bid + By;
   end raise_bid;
```

Licitacija u ADI(1)

```
loop
      select
         accept raise_bid (From:BIDDER_ID;By:NATURAL) do
         Last_Bidder := From;
         Current_Bid := Current_Bid + By;
         end raise_bid;
     or accept look_at (Value:out NATURAL) do
           Value := Current_Bid;
        end look_at;
    or delay 10.0;
       exit;
end select;  ...
end BID;
```

Izmenjena licitacija u ADI

```
select
   accept raise_bid ...
or when raise_bid'COUNT = 0 =>
accept look_at ...
or when Current_Bid >= Reserve_Value =>
   delay 10.0;
   exit;
end select;
```