

# Paralelni računarski sistemi



Komunikacija i sinhronozacija procesa u MIMD  
sistemima

# Komunikacija i sinhronozacija procesa u MIMD sistemima

\* Osnovna jedinica konkurentnog izvršenja u MIMD sistemu je proces.

- To je deo programa (niz naredbi) koji se izvršava sekvensijalno na jednom procesoru.
- Veoma retko su procesi koji se izvšavaju paralelno potpuno nezavisni, tj. procesi zahtevaju nekakvu razmenu podataka ili medjusobnu sinhronizaciju (zbog toga je korektnije reći konkurentni umesto paralelni procesi).
- Zbog toga MIMD sistem mora sadržati sredstva koja podržavaju interprocesorsku komunikaciju i sinhronizaciju

# Komunikacija i sinhronozacija procesa u MIMD sistemima

\* Fundamentalni postulat koji se odnosi na interprocesnu komunikaciju MIMD sistema je da se ne mogu uvoditi nikakve pretpostavke o relativnoj brzini izvršenja procesa.

- Ovo znači da ako dva procesa P1 i P2 treba da komuniciraju oni moraju biti korektno sinhronizovani da bi komunikacija mogla da se obavi.
- ova sinhronizacija mora biti ugradjena u sam proces, tj. mora biti programirana.
  - Tradicionalno su mehanizmi interprocesne komunikacije bili zasnovani na korišćenju deljivih promenljivih.
  - Ovaj prilaz je i dalje dominantan kod multiprocesorskih sistema, tj. sistema sa deljivom memorijom.
  - Alternativni prilaz je korišćenje tehnike slanja poruka.
    - Ovaj prilaz je postao atraktivan sa pojavom jezika kao što su CSP, OCCAM, Ada.
  - Na arhitektturnom nivou, mehanizmi sinhronizacije i komunikacije putem razmene poruka preovladajuju kod sistema sa distribuiranom memorijom (tj. kod multiračunarskih sistema).

\* Posmatrajmo dva procesa P1 i P2 koji se izvršavaju na dva različita procesora paralelno.

- U nekom trenutku svog izvršenja oba procesa zahtevaju da modifikuju zajedničku promenljivu D koja je zapamćena u glavnoj memoriji

P1: .....

LOAD D, Reg1  
ADD Reg1, #1  
STORE Reg1, D

} K.S. 1

.....

P2: .....

Load D, Reg2  
ADD Reg2, #2  
STORE Reg2, D

} K.S.2

.....

- pošto su P1 i P2 konkurentni procesi nikakve prepostavke o brzini izvršenja ovih procesa se ne mogu učiniti.
  - Ako je početna vrednost promenljive D bila 0, nakon izvršenja procesa P1 i P2 može se desiti da D ima vrednost 1, 2 ili 3, u zavisnosti od relativnih vremena izvršenja instrukcija

# Sinhronizacija i komunikacija procesa kod multiprocesorskih sistema

\* da bi obezbedili da vrednost D bude uvek 3 nakon izvršenja procesa P1 i P2, potrebno je da svaki od ovih segmenata programa bude nedeljiva celina i da procesi P1 i P2 pristupaju promenljivoj D uzajamno isključivo (tj. da se delovi programa koji pristupaju zajedničkim promenljivim izvršavaju sekvencialno).

- Kodni segment (deo programa) koji se obraća deljivoj promenljivoj predstavlja tvz. *kritičnu sekciju*.
  - Uzajamno isključivo pravo pristupa kritičnoj sekciji može se koncepcijски implementirati mehanizmom zaključavanja.
  - deljivoj promenljivoj D može se pridružiti ključ *k* nad kojim se mogu obavljati dve operacije **LOCK(k)** i **UNLOCK(k)** (zaključaj(k) i otključaj(k)) koje postavljaju *k* na 1 (zaključavanje) odnosno na 0 (otključavanje)
  - Proces može izvršiti **LOCK(k)** samo ako je *k* otvoren (tj. postavljen an 0).
    - Rezultat izvršenja **LOCK(k)** je postavljanje *k* na 1, i na taj način se sprečava da drugi proces obavi **LOCK(k)** operaciju.
    - Proces koji naiđe na zaključanu bravu, mora da čeka.
    - Operacijom **UNLOCK(k)** vrednost *k* se postavlja na 0. .

# kritične sekcije

\* Usvajajući da je k inicijalno postavljeno na 0, kritične sekcije u P1 i P2 mogu se sada ovako implementirati

P1: LOCK(k)

kritična sekcija 1

UNLOCK(k)

P2: LOCK(k)

kritična sekcija 2

UNLOCK(k)

\* kritične seskcije 1 i 2 trebalo je izvršavati uzajamno isključivo ali u bilo kom redosledu.

# uslovne kritične sekcije

- \* Neka su P1 i P2 kooperativni, komunicirajući , iterativni procesi pri čemu P1 generiše podatke za P2 i pamti ih u D.
- \* P2 čita D.
- \* Potrebno je obezbediti da P1 čeka pre upisa dok P2 ne pročita prethodni podatak, i da pošto P2 pročita podatak čeka dok P1 ne upiše novi podatak.
  - tj. potrebno je da P2 ne pročita dva puta isti podatak, ili da P1 ne upiše novi podatak preko podatka koji još nije pročitan.
  - Ovakva sinhronizacija zove se uslovna sinhronizacija.
  - I ovaj vid sinhronizacije moguće je konceptualno ostvariti mehanizmom zaključavanja.
  - Sada je potrebno uvesti dva ključa k1 i k2.
    - P1 otključava k2 da signalizira P2 da je upisao novi podatak.
    - P2 otključava k1 da signalizira P1 da je pročitao prethodno zapamćeni podatak.

# uslovne kritične sekcije

\* Usvajajući da je inicijalno  $k1=0$  I  $k2=1$ , kodni segmenti u P1 i P2 imaju sledeći izgled

P1: LOCK(k1)

generisanje podatka

UNLOCK(k2)

P2: LOCK(k2)

konzumiranje podatka

UNLOCK(k1)

- LOCK operacija sadrži dve aktivnosti: *testiranje* k i *postavljanje* vrednosti k na 1.
- UNLOCK sadrži samo jednu opraciju - postavljanje k na 0.
  - Da bi LOCK mehanizam pravilno funkcionisao potrebno je da operacije testiranja i postavljanja budu nedeljive (tj. atomične).
  - Jedan od načina realizacije LOCK mehanizma je pomoću TEST\_AND\_SET (TAS) instrukcije,
    - prvi put implementirana kao deo skupa instrukcija sistema IBM 360 (1964), a zatim i kod IBM 370 (1981).

# uslovne kritične sekcije

- \* Naredbom **TEST\_AND\_SET** se izvršavaju sledeće operacije kao jedna instrukcija:

$x = TAS(k)$ :  $x := k$

*if  $x = 0$  then  $k := 1$ ;*

- Nedeljivost ove operacije se obično postiže tako što procesor zadržava magistralu dok se ciklus ne završi.
- Korišćenjem TEST\_AND\_SET naredba LOCK(k) se može realizovati na sledeći način

**LOCK(k)**: *repeat {  $x = TAS(k)$ } until  $x = 0$*

➤ LOCK instrukcija koja koristi TAS ima za posledicu da proces koji pokušava da udje u kritičnu sekciju bude stalno upošljen testiranjem zajedničke promenljive.

➤ Ovo se zove “busy-waiting” (upošljen čekanjem) ili “spin lock”.

➤ Proces koji naiđe na zatvorenu bravu ne napušta procesor, već stalno testira vrednost ključa.

➤ To ima za posledicu da se procesorski ciklusi beskorisno troše, povećava saobraćaj na magistrali i broj (beskorisnih) obraćanja memoriji

# Semafori

\* Pobrojan nedostaci mogu se rešiti korišćenjem posebne strukture podataka SEMAFORA.

- Semafor je celobrojna promenljiva koja može uzeti samo nenegativne vrednosti.
- Jedine operacije, izuzev inicijalizacije, koje su dozvoljene nad semaforima su P i V operacije (WAIT i SIGNAL).
- Ove operacije su nedeljive, tako da se ne može desiti da više procesa izvršava ove operacije nad istim semaforom u isto vreme.
- Semafore je kao sredstvo sinhronizacije predložio danski naučnik Dijkstra (1968).
  - Nazivi operacija P i V potiču od danskih rečI Passern (proći) i Vrygeven (napustiti, oslobođiti).

# Semafori (nast.)

\* Svakom semaforu pridružena je lista procesa koji čekaju na ulazak u kritičnu oblast.

- Ove liste se najčešće implementiraju kao redovi čekanja.
- P i V operacije nad semaforom S imaju sledeće dejstvo
  - P(S): if  $S > 0$  then
    - $S := S - 1$
    - else { blokirati proces koji je izvršio P(S)  
i smestiti ga u red čekanja pridružen semaforu S}
    - 
    -
  - V(S):  $S := S + 1$ 
    - if { red čekanja pridružen semaforu S nije prazan }
    - then { selektovati jedan proces iz reda čekanja i smestiti ga u red spremnih}
- Izvršenje P(S) će blokirati proces ako je semafor zatvoren.
  - Blokirani proces neće okupirati procesor kao u slučaju TEST\_AND\_SET.
  - Procesor može izvršavati neki drugi proces koji je spreman za izvršenje.
- Izvršenje V(S) će proces koji je bio blokiran prevesti u stanje spreman.
  - eliminisan je "busy waiting".

# semafori (nast.)

## \* Semafor može biti

- binarni - uzimati samo vrednosti 0 ili 1, ili
- brojni (generalni) - uzimati vrednosti 0 do N.
- Osnovna razlika izmedju binarnog i brojnog semafora je što kod opšteg semafora više procesa može obaviti P operaciju i nastaviti sa izvršenjem.
- Prethodni primeri zahtevane sinhronizacije mogu se korišćenjem semafora napisati na sledeći način:

I)  $S:=1$  (inicijalno otvoren semafor)

P1:  $P(S)$

krit. sekc.1

$V(S)$

P2:  $P(S)$

krit.sekc.2

$V(S)$

II) Potrebna su dva semafora  $S1$  i  $S2$ . Inicijalno je  $S1:=1$ ,  $S2:=0$ ;

P1:  $P(S1)$

generisanje podatka

$V(S2)$

P2:  $P(S2)$

konzumiranje podatka

$V(S1)$

# Primer – opšti senafori

- \* Postoji nekoliko procesa (proizvođači) koji generišu podatke i predaju ih drugim procesima (potrošačima)
- \* Svi procesi se izvršavaju paralelno i nepoznatim brzinama.
- \* Proizvođači smeštaju podatke u bafer konačne veličine
- \* Potrošači čitaju podatke iz bafera
  - Potrebno je sprečiti
    - upis u pun bafer
    - Čitanje praznog bafera
- \* Bounded-buffer ili producer-consumer problem

# Primer (nast.)

- \* Neophodna sinhronizacija se može postići korišćenjem opštih semafora koji predstavljaju uslove čekanja:
- \* Semafori:
  - FULL – predstavlja broj popunjениh elemenata u baferu
    - Potrošač se blokira kada je FULL=0
    - Kada obavi upis u bafer proizvođač izvršava V operaciju nad semaforom (inkrementira vrednost semafora)
    - Potrošač izvršava P nad semaforom (testira ga) pre nego što pokuša da pročita podatak (izvršenje P omogućava da se potrošač blokira ako je bafer prazan)
  - EMPTY – predstavlja broj praznih lokacija u baferu
    - Proizvođač se blokira kada je EMPTY=0
  - UZISK – obezbeđuje uzajamno isključivo pravo pristupa baferu
    - samo jedan proces u jednom trnutku može da pristupa baferu, bilo radi čitanja bilo radi upisa.

# Primer (nast.)

## \* Promenljive:

- COUNT - Pamti broj podataka u baferu u svakom trenutku
- POINTER – pamti gde se nalazi sledeći podatak u baferu koji treba pročitati
  - $(\text{POINTER}+1)\bmod N$  omogućava ciklično čitanje iz bafera
- Pozicija upisa se određuje na osnovu
  - $(\text{POINTER} + \text{COUNT})\bmod N$  – omogućava ciklično upisivanje u bafer
  - Nakon upisa u bafer proizvođač inkrementira COUNT
  - Nakon čitanja iz bafera potrošač dekrementira COUNT
- BUFFER – polje koje simulira rad bafera

# Program

```
Var      BUFFER array[0..N-1] of integer; (*deljiva promenljiva*)
        POINTER: 0..N-1 ;      (*ukazuje na poziciju elementa koji se cita*)
        COUNT: 0..N ;          (*broj podataka u baferu*)
        FULL, EMPTY, UZISK : SEMAPHORE;
```

```
Begin {inicijalizacija}
```

```
    COUNT:=0; POINTER:=0;
    FULL:=0; EMPTY:=N;; UZISK:=1;
```

```
End;
```

(\*proizvodjac\*)

```
Var podatak: integer;
-- generisanje podatka
```

**P(EMPTY);**

**P(UZISK);**

```
BUFFER[(POINTER+COUNT)modN]:= podatak;
```

```
COUNT:=COUNT+1;
```

**V(UZISK);**

**V(FULL);**

(\*Potrošač\*)

```
Var podatak: integer;
```

**P(FULL);**

**P(UZISK);**

```
podatak:= BUFFER[POINTER];
```

```
COUNT:= COUNT-1;
```

```
POINTER:=(POINTER+1)mod N;
```

**V(UZISK);**

**V(EMPTY);**

-- korišćenje podatka ..

# Semafori - osobine

## \* Prednost:

- lako se može realizovati.

## \* Nedostaci:

- Semafori obezbedjuju veoma primitivan oblik sinhronizacije procesa i čine paralelni program nejasnim i podložnim greškama.
- Izostavljanje P ili V operacije ili izvršenje P operacije nad jednim a V nad drugim semaforom, može imati katastrofalne posledice jer uzajamno isključivo pravo pristupa više nije obezbedjeno.
- Ako procesi u sistemu imaju različite prioritete, sinhronizacija pomoću semafora postaje veoma komplikovana i nerazumljiva.
  - Semafori programi su nepregledni izmedju ostalog i što svaki proces zadržava za sebe tekst svoje kritične oblasti.

# Monitori

\* Sve kritične sekcije, tj. naredbe koje se obraćaju deljivim promenljivim, sadržane su u monitoru.

- Deklaracija monitora sadrži brojne procedure koje definišu operacije nad deljivim resursima (promenljivim).
- Opšti oblik deklaracije monitorai:
  - monitor ime;
  - \*deklaracija lokalnih promenljivih za monitor\*
  - \*deklaracija procedura lokalnih za monitor\*
  - begin
  - \*inicijalizacija lokalnih promenljivih\*
  - end ime;
- Monitorske procedure su kritične sekcije.
  - Monitorske procedure mogu pristupati samo promenljivim koje su lokalne za dati monitor.
  - Ovim lokalnim promenljivim ne sme se pristupati van monitora
  - Ako se ovo poštuje, može se garantovati uzajamna isključivost kod pristupa deljivim promenljivim

# Monitori (nast.)

- Kada proces želi da pristupi deljivom resursu, mora pozvati odgovarajuću monitorsku proceduru.
- U jednom trenutku samo jedan proces može izvršavati neku monitorsku proceduru i ona se mora završiti pre nego što se dozvoli da neki drugi proces pozove neku monitorsku proceduru.
  - sam monitor ne prouzrokuje nikakvu akciju u sistemu.
  - On je jednostavno skup procedura koje se mogu izvršavati u pojedinim procesima.
- Poziv monitorske procedure:
  - ime\_monitora.ime\_procedure (stvarni parametri)
- Nakon deklaracije monitor inicijalizuje svoje lokalne podatke
  - svi dalji pozivi koriste vrednosti lokalnih promenljivih dobijenih završetkom prethodnog poziva.
- Da bi se obezbedila potrebna sinhronizacija procesa unutar monitora se deklariše i poseban tip podatka CONDITION
  - Ovi podaci se koriste kada proces želi da zakasni svoje izvršenje ili pobudi prethodno blokirani proces

# Monitori (nast.)

## \* *Primer:*

*Var usl:* CONDITION

## \* Na ovaj način je deklarisan red čekanja *usl.*

- Nad promenljivima tipa CONDITION moguće je izvršavati samo dve operacije:
  - WAIT(*usl*)
  - SIGNAL(*usl*).
- WAIT(*usl*) operacija blokira proces i dodaje ga redu čekanja pridruženom promenljivoj *usl*.
  - Ova operacija automatski oslobadja pristup monitoru, tako da drugi procesi mogu da udju u monitor.
- SIGNAL(*usl*) omogućava da se aktivira proces koji je bio blokiran zbog *usl*.
  - Ako nema ni jednog procesa koji čeka, ova naredba je bez dejstva.

## \* Jezici koji sadrže monitore:

- Concurrent Pascal, Modula-2, MESA, CSP/K (serija jezika izvedena iz PL/1)

# Primer – proizvođač - potrošač

Program *proizvođačpotrošač*;

*const N= ... (\*veličina bafera\*)*

Monitor boundedbuffer;

var BUFFER: array[0..N-1] of integer; POINTER =0..N-1; COUNT=0..N;

nije\_pun, nije\_prazan : CONDITION;

**procedure dodaj** (v: integer );

begin

if COUNT=N then **WAIT**(nije\_pun);

BUFFER[(POINTER+COUNT)modN]:=v;

COUNT:=COUNT+1;

**SIGNAL**(nije\_prazan);

end;

**procedure uzmi** (var v: integer);

begin

if COUNT =0 then **WAIT**(nije\_prazan);

v:=BUFFER[POINTER];

POINTER:=(POINTER+1)modN;

COUNT:=COUNT-1;

**SIGNAL**(nije\_pun);

end;

begin {telo monitora}

POINTER:=0; COUNT:=0;

end;

# Primer (nast.)

Procedure proizvodjac;

```
var v: integer;
begin
    repeat
        -- generisanje podatka v
        boundedbuffer.dodaj(v); {poziv monitorske procedure}
    forever;
end;
```

Procedure potrosac;

```
var v: integer
begin
    repeat
        boundedbuffer.uzmi(v);
        -- korišćenje podatka;
    forever;
```

Begin {glavni program}

```
cobegin
    proizvodjac; potrosac
coend;
```

End.

# Komunikacija razmenom poruka

\* Primitivne operacije koje omogućuju razmenu poruka su SEND/RECEIVE tipa.

- Proces šalje poruku izvršavajući operaciju
  - SEND izraz TO odredišni\_proces
- Proces koji prima poruku izvršava operaciju
  - RECEIVE promenljiva FROM izvorni\_proces
- Tehniku slanja poruka prvi je u jezik CSP (Communicating Sequential Processes) ugradio Hoare, a zatim Brinch-Hansen u jezik DP (Distributed Processes).
- Kod CSP proces P šalje poruku Procesu Q izvršavajući naredbu:
  - Q!(izraz). (Npr. Q!(x+y) ).
- Proces Q prima poruku od P izvršavajući naredbu
  - P?(promenljiva). (Npr. P?(z) ).

# Komunikacija razmenom poruka

\* Da bi se ostvarila komunikacija izmedju dva konkurentna procesa procesi moraju proći kroz dva koraka.

- Prvi korak je sinhronizacija.

- Ona se odnosi na činjenicu da oba procesa moraju da stignu u tačku gde se zahteva razmena podataka.
- Ako neki proces stigne pre, mora da čeka.
- Kada oba procesa stignu u odgovarajuću tačku, sinhronizacija je postignuta.

- Sledeći korak je komunikacija.

# Randevu u Adi

\* Osnovna konkurentna jedinica izvršenja u programskom jeziku Ada je task.

- Task se sastoji iz specifikacijskog dela i tela.
  - Specifikacijski deo, ako postoji, sadrži entry deklaracije koje predstavljaju sredstvo pomoću kojeg jedan zadatak može komunicirati sa drugim.
  - Svakom entry je pridružen red čekanja.
  - Entry je sličan specifikaciji procedure.
  - Međutim, dok se telo procedure izvršava u pozivnom zadatku, Entry se izvršava u sopstvenom zadatku
  - Naredba accept specificira akcije koje treba izvršiti kada se pozove entry.
  - Opis u accept naredbi mora se slagati sa opisom u entry deklaraciji.

# Randevu –primer deklaracije

```
task M is      {specifikacijski deo}
    entry put (S:in INTEGER) {in ukazuje da je promenljiva S ulazna
        promenljiva}
    end M;

task body M is {telo zadatka}
    -deklaracija lokalnih promenljivih za zadatak
    begin
        .
        .
        .
    accept put (S:in INTEGER) do
        PRINT(S)      {kod randevua}
    end put;
    .
    .
    .
end.
```

- Entry deklaracija može sadržati više promenljivih od kojih neke mogu biti ulazne (in), neke izlazne (out), a neke i ulazne i izlazne (in out).

➤ entry A (*I: in INTEGER; x: out REAL; b: in out BOOLEAN*)

# Randevu (nast.)

- Poziv entry-a na randevu u nekom zadatku

*ime\_zadatka.ime-entry (stvarni parametri)*

➤ Npr. M.put(Q).

- Randevu - sinhronizacija i komunikacija izmedju procesa ostvaruje se kada se upare accept naredba i poziv entry-a u nekom zadatku.

➤ Ako neki proces prvi stigne u tačku koja zahteva komunikaciju, mora da čeka.

➤ Efekat randevua je da prouzrokuje izvršenje naredbi izmedju **do** i **end** u accept naredbi pozvanog zadatka, za parametre koji su specificirani u pozivu entry-a.

➤ Kada se randevu odvija parametri se razmenjuju po istim pravilima kao parametri procedura.

➤ Zadatak koji poziva entry drugog zadatka se zaustavlja dok pozvani zadatak izvršava kod u accept naredbi.

# Randevu (nast.)

- Ako nekoliko zadataka poziva isti entry oni formiraju FIFO red.
- Pozvani zadatak (zadatak u kome je entry deklarisan) može koristiti **COUNT** atribut entry-a da vidi koliko zadataka trenutno čeka na randevu.
  - Ovaj atribut se dodaje imenu entry-a i kao rezultat vraća broj zadataka koji čeka na randevu sa odgovarajućim entry-jem.
- Jeden zadatak može sadržati više entry deklaracija i može komunicirati sa više drugih zadataka.
- Pošto su zadaci koji se izvršavaju asinhroni ne može se predvideti redosled kojim zadaci komuniciraju.
  - Da bi se uvela nedeterminisanost u program u Adi je na raspolaganju naredba **select**.
  - Ova naredba se sastoji od niza alternativa koje se odnose na prihvatanje (accept) nekog poziva od kojih će samo jedan u datom trenutku biti prihvaćen.

# primer select naredbe

*select*

*accept M1 (formalni parametri) do*

-----

*end M1*

*or*

*accept M2 (formalni parametri) do*

-----

*end M2*

*or*

.

.

.

*end select;*

- Uslovno izvršenje select alternative može se postići uvodjenjem *when* naredbe

# uslovno izvršenje select naredbe

*select*

*when uslov =>*

*accept M1(formalni parametri) do*

-----

*end M1*

*or when M2 'COUNT >0 =>*

*accept M2(formalni parametri) do*

-----

*end M2*

*end select;*

- Select naredba može imati i *else* deo koji se izvršava ako ni jedan od uslova u selectnaredbi nije ispunjen:

*select*

-----

*or*

-----

*or*

-----

*else*

-----

*end select;*

# Randevu u Adi (nast)

- Zadatak može biti deklarisan u okviru tela glavnog programa ili kao nezavisna celina koja se nezavisno prevodi.
- Izgled glavnog programa u Adi

***program ime;***

*-deklaracija konstanti*

*-deklaracija tipova*

*-deklaracija taskova i procedura*

*-telo taska 1;*

*-telo taska 2;*

-----

*-deklaracija promenljivih*

***begin***

-----

-----

***end.***

# Randevu u Adi (nast)

- Ako se zadaci pišu kao nezavisne celine koje se posebno prevode onda u glavnom programu postoji samo deklarativni deo zadatka koji sadrži entry specifikacije, a umesto tela zadatka piše se
- task body ime je SEPARATE;
- Npr.

- Program MAIN;
  - task T1 is
    - entry M1(formalni parametri)
    - entry M2(formalni parametri)
  - end T1;
  - task T2 is
    - entry MM1(formalni parametri)
  - end T2;
  - task body T1 is SEPARATE;
  - task body T2 is SEPARATE;
  - begin
    -
  - end.

# Randevu u Adi (nast)

- Zadaci koji se pišu kao nezavisne celine moraju da sadrže oznaku pripadnosti glavnom programu:

*SEPARATE (MAIN)*

*task body T1 is*

*-deklaracija lokalnih promenljivih*

*-deklaracija procedura*

-----

*begin*

-----

*end T1;*

# Primer proizvođač-potrošač

## \* Tri zadatka:

- **Bounded\_buffer**

- upisuje podatke u bafer i čita podatke iz bafera;
- Ima dve entry deklaracije: **stavi** i **uzmi**

- **Proizvođač**

- Generiše podatke
- Kada želi da upiše podatak u bafer poziva odgovrajući entry zadatka bounded\_buffer

- **Potrošač**

- Koristi podatke
- Kada želi da pročita podatak iz bafera poziva odgovrajući entry zadatka bounded\_buffer

# Primer (nast.)

```
SEPARATE (MAIN)
task BOUNDED_BUFFER is
    entry stavi (podatak: in INTEGER);
    entry uzmi (podatak: out INTEGER);
end BOUNDED_BUFFER;
task body BOUNDED_BUFFER is
    N: constant INTEGER:=20;
    BUFFER : array(0..N-1) of INTEGER;
    POINTER: INTEGER range 0..N-1:=0;
    COUNT : INTEGER range 0..N:=0;
begin
    loop
        select
            when COUNT < N =>
                accept stavi (podatak: in INTEGER) do
                    BUFFER((POINTER+COUNT)modN):=podatak;
                end stavi;
                COUNT:=COUNT+1;
            or
            when COUNT > 0 =>
                accept uzmi (podatak: out INTEGER) do
                    podatak:=BUFFER(POINTER);
                end uzmi;
                POINTER:=(POINTER+1)modN;
                COUNT:=COUNT-1;
        end select;
    end BOUNDED_BUFFER;
```

# Primer (nast.)

```
SEPARATE(MAIN)
task PROIZVODJAC is
end PROIZVODJAC;
task body PROIZVODJAC is
    pod: INTEGER;
    begin
        loop
        -- generisanje podatka
        BOUNDED_BUFFER.STAVI(pod);
        end loop
    end PROIZVODJAC;
```

```
SEPARATE(MAIN)
task POTROSAC is
end POTROSAC;
task body POTROSAC is
    pod: INTEGER;
    begin
        loop
        BOUNDED_BUFFER.UZMI(pod);
        -- koriscenje podatka
        end loop
    end POTROSAC;
```

```
procedure MAIN
task BOUNDED_BUFFER is
    entry stavi (podatak: in INTEGER);
    entry uzmi (podatak: out INTEGER);
end BOUNDED_BUFFER;
task PROIZVODJAC is
end PROIZVODJAC;
task POTROSAC is
end POTROSAC;
task body BOUNDED_BUFFER is SEPARATE;
task body PROIZVODJAC is SEPARATE;
task body POTROSAC is SEPARATE;
begin
end.
```

## Primer

Zadata su dva skupa celih brojeva  $S$  i  $T$  sa  $n$  i  $m$  elemenata , respektivno. Napisati konkurentni program u Ada kojim se skupovi preuredjuju tako da  $S$  sadrži  $n$  najmanjih elemenata iz  $S \cup T$ , a  $T$   $m$  najvećih elemenata iz  $S \cup T$

**task manji is**

**entry** sendmax (xx: out INTEGER)

**end** manji

**task veci is**

**entry** sendmin(yy: out INTEGER)

**end** veci

**task body manji is**

    n: constant INTEGER:=20

    x,mx: INTEGER

    S: array (1..n) of INTEGER

**begin**

        mx:= max vrednost u S

**loop**

**accept** sendmax(xx:out INTEGER)

            xx:=mx

**end** sendmax

    ukloniti mx iz S

    veci.sendmin(x)

    dodati x u S

**exit when** x=mx

**end loop**

**end** manji

**task body veci is**

    m: constant INTEGER:=30

    y,mn: INTEGER

    T: array(1..m) of INTEGER

**begin**

**loop**

        manji.sendmax(y)

        dodati y u T

        mn:= min vrednost u T

**accept** sendmin(yy:out INTEGER)

            yy:=mn

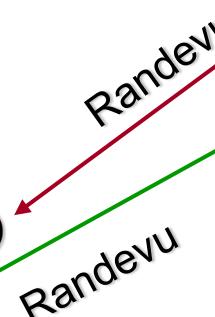
**end** sendmin

        ukloniti mn iz T

**exit when** mn=y

**end loop**

**end** veci



# Operativni sistemi multiprocesora

## \* Zadaci

- Dodjela resursa
- Upravljanje resursima
- Zaštita memorije i podataka
- Sprečavanje samrtnog zagrljaja (deadlock)
- Rekonfiguracija sistema

## \* Tri organizacije OS

- Distribuirani
- Gospodar-sluga
- Plivajući (simetrični)

# Distribuirani OS

## \* Svaki procesor ima odvojenu kopiju jezgra OS.

- Jezgro OS u svakom procesoru može kreirati, rasporediti ili uništiti proces.
- Procesi komuniciraju razmenom poruka
- Jezgro OS u svakom procesoru upravlja skupom privatnih tabela, redova čekanja i slanjem poruka iz svog procesa
- Pošto postoji interakcija izmedju procesora i pošto postoje zajednički resursi, pored privatnih tabela postoje i tabele koje su zajedničke za ceo sistem.
  - Umnožavanje jezgra OS u svakom procesoru zahteva mnogo memorije
  - Keš memorije se mogu iskoristiti za pamćenje često korišćenih delova OS, a ostali mogu biti centralizovani u deljivoj memoriji

## \* Korišćen u prvim multiprocesorskim sistemima

- Želaja je bila iskoristiti već opstojeće OS
  - Ovakav prilaz se danas retko koristi

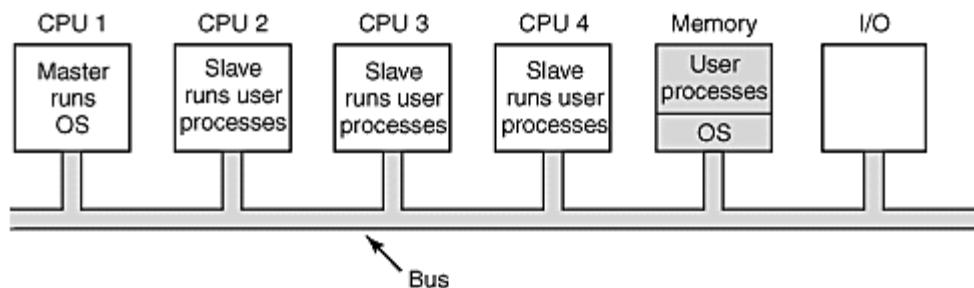
# Gospodar-sluga

\* Jedan procesor (gospodar) u sistemu uvek izvršava program OS

- Gospodar sadrži status svih procesora u sistemu i raspoređuje poslove procesorima slugama
- Ako je procesoru-slugi potrebna bilo koja usluga OS upućuje prekid glavnom procesoru i zahteva izvršenje odgovarajuće supervizorske rutine

## ● Problemi

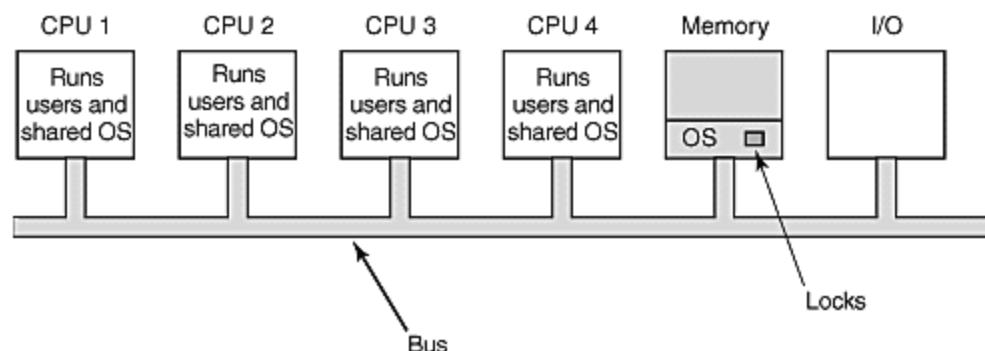
- Ako ima mnogo procesora gospodar će postati usko grlo sistema (mora da upravlja sistemskim pozivima svih procesora)
- Otkaz gospodra je katastrofalан за ceo sistem



# Plivajući (simetrični) OS

\* Postoji jedna kopija OS u memoriji, ali bilo koji procesor može da izvršava program OS

- Kada proces na nekom procesoru uputi sistemski poziv, isti procesor obrađuje sistemski poziv
  - Ako dva ili više procesora izvršavaju program OS u isto vreme može doći do katastrofalnih posledica
    - Npr. dva procesora simultano odaberu da izvršavaju isti proces ili zauzimaju istu slobodnu stranicu memorije
  - Rešenje: OS se pristupa kao jednoj velikoj kritičnoj sekciji ( u jednom trenutku samo jedan procesor može izvršavati neku rutinu OS)
    - Problem: dugi redovi čekanja za pristup OS



# Plivajući (simetrični) OS (nast.)

## ➤ Rešenje:

- Mnogi delovi OS su nezavisni jedan od drugog (npr. Nema nikakvi problema da jedan procesor izvršava planer, drugi obradjuje sistemski poziv za upravljanje fajlovima, ...)
- Ovo zahteva podelu OS na nezavisne kritične regije (delove) koji ne interaguju jedni sa drugima
- Svaki kritični region je zaštićen sopstvenim lock mehanizmom, tako da ga u jednom trenutku može izvršavati samo jedan procesor
- Ako neke tabele koriste različiti delovi OS, takvim tabelama se mora obezbediti uzajamno isključivo pravo pristupa (lock mehanizam)
- Problem je izvršiti podelu OS na nezavisne celine koje se mogu izvršavati konkurentno (jednovremeno)
- Posebno treba voditi računa da ne dođe do deadlocka (samrtni zagrljaj)
  - » Ako dva kritična regiona oba zahtevaju tabelu A i tabelu B iako prvi region zahteva prvo A a drugi region prvo B, doći će do deadlocka (situacije kada ni jedan proces ne može da nastavi sa izvršenjem)
  - » Problem se rešava tako što se svakoj tabeli dodeli celobrojna vrednost, a od kritičnih regiona zahtevaju tabele uvek u rastućem redosledu

- Moderni multiprocesorski sistemi koriste simetrične OS (Hydra na C.mmp)