Operacinės sistemos

N. Sarafinienė

2013m.

Kalbėsime

- Problemas, susijusias su kritine sekcija
- Kritines sekcijos sprendimo būdus:
  - Programinius sprendimus
  - Techninius sprendimus
  - Operacinės sistemos tiekiamus sprendimus

Konkurencija

- Konkurencija reiškia tai, kad procesai siekdami aptarnavimo turi konkuruoti dėl sistemoje esančių resursų:
  - procesoriaus laiko,
  - vietos pagrindinėje atmintinėje
  - Įrenginių
  - Soketų
  -
  - ir kitų.

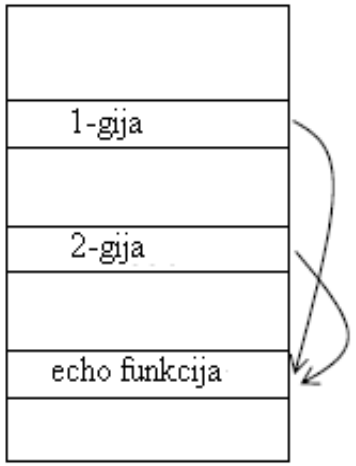
Kritinis resursas

- Lygiagrečiai vykdant procesus (arba gijas) dažnai susiduriame su situacija, kai procesai (ar gijos) nori vienu metu pasinaudoti kažkuriuo tai sistemos resursu.
- Bendrai naudojamu resursu gali būti:
  - failas
  - kintamasis
  - spausdintuvas
  - registras, ir t.t.
- Kritinis resursas: tai toks sistemos resursas, kurio panaudojimą turi kontroliuoti operacinė sistema
  - Jei viena iš proceso gijų nori naudotis bendrai prieinamu resursu tai kitoms gijoms tuo metu turi būti draudžiama naudotis šiuo resursu.
  - Jei nėra kontroliuojamas priėjimas prie bendrai naudojamų duomenų, tai procesai (gijos) gali gauti iškreiptas duomenų reikšmes.

- Kritinė sekcija: tai programos dalis, kurioje kreipiamasi ir vykdomi veiksmai su kritiniu resursu.

Konkurencija: Pavyzdys

- Pav.: “echo” funkcija
- **void echo (char in)**
  - **{**
  - **char out;**
  - **read (stdin, in);**
  - **out = in;**
  - **write (stdout, out);**
  - **}**



1 gija

```
read(stdin,in);
{ blokuota }
out=in;
write(stdout,out);
```

2-gija

```
read(stdin,in);
out = in;
write(stdout,out);
```

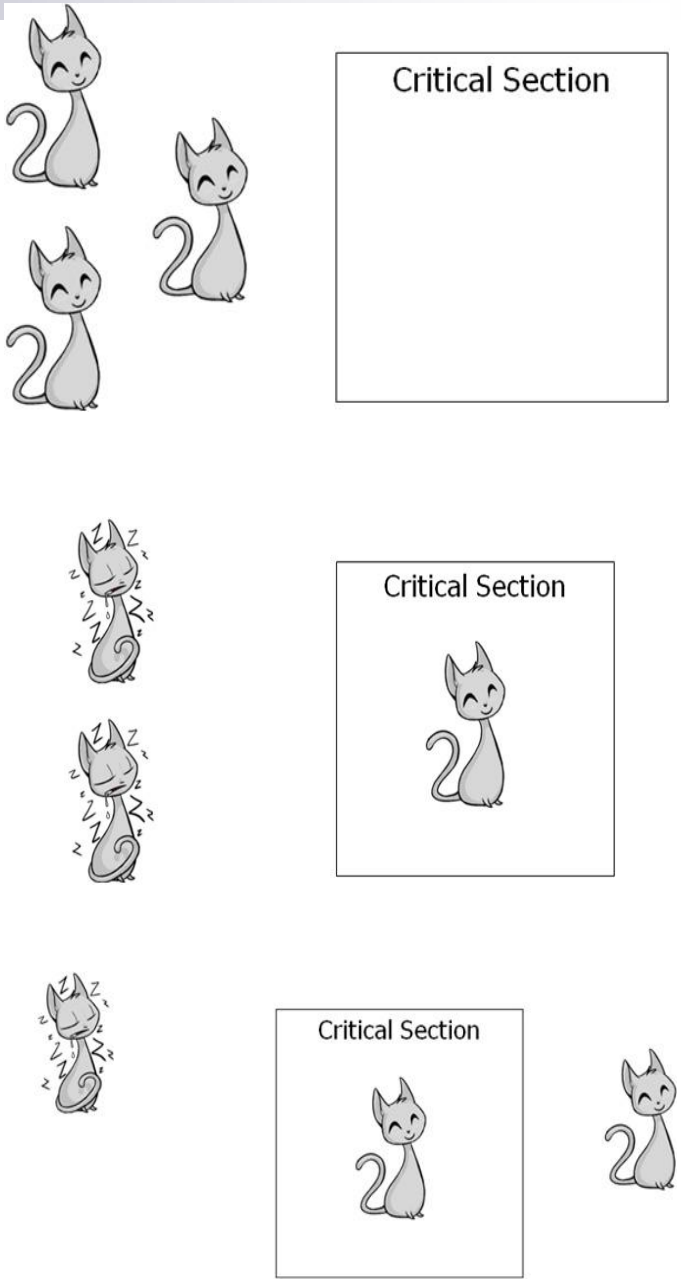
Kintamieji

in	out
x	
y	
y	y
y	y
y	y
y	y

- **Susidaro lenktynių situacija-** rezultatas priklauso nuo to , kokia seka bus vykdomos gijos

Kritinė sekcija

- Kai procesas vykdo kodą, kuris manipuliuoja bendrai naudojamais duomenimis (ar ištekliais), yra sakoma, kad procesas patenka į kritinę sekciją (CS) šių bendrai naudojamų duomenų atžvilgiu.
- Kritinės sekcijos veiksmų vykdymas turi vykti „**tarpusavio išskirtinumo**“ (mutual exclusion) režimu:
  - bet kuriuo laiko momentu tik vienam procesui yra leidžiama atlikti kritinės sekcijos veiksmus (netgi esant keliems CPU).
  - Tik po to, kai vienas procesas atlieka visus kritinės sekcijos veiksmus, į kritinę sekciją leidžiama įeiti kitam procesui.



Programa su kritine sekcija

Proceso vykdomo kodo struktūra:

```
repeat
   Įėjimo sekcija
   Kritinė sekcija (CS)
   Išėjimo sekcija
   Likusi sekcija (RS)
forever
```

Procesas turi užsiprašyti leidimo įeiti į kritinę sekciją (CS).

Toliau seka veiksmai kritinėje sekcijoje,

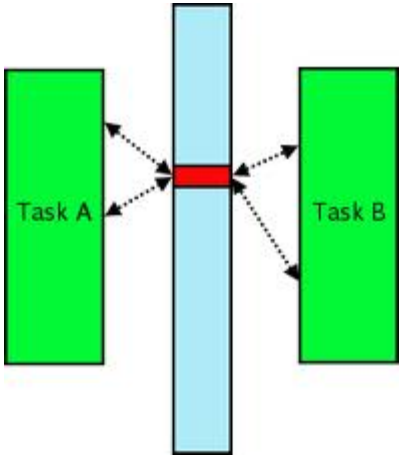
Išėjimo sekcijos dalyje yra pranešama apie tai, kad procesas atlaisvino kritinę sekciją.

Likusioje sekcijoje gali būti vykdomi veiksmai, nesurišti su bendrai naudojamais kintamaisiais.

Kritinės sekcijos veiksmų vykdymas

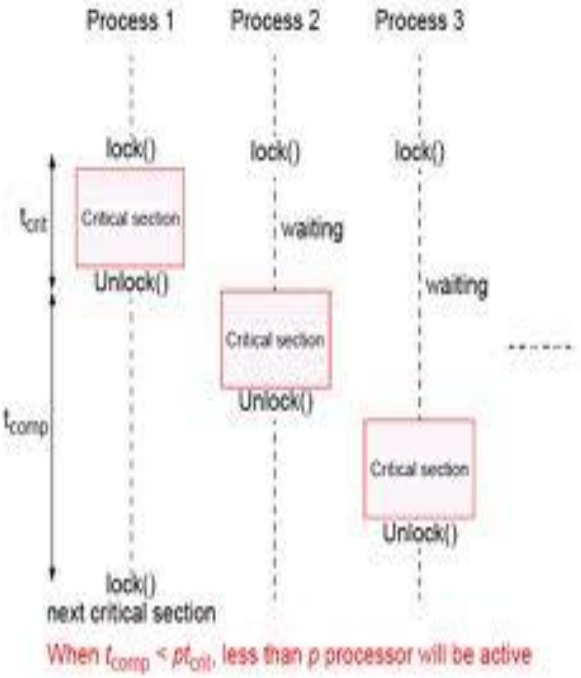
- Kritinės sekcijos atžvilgiu reikia numatyti tokias taisykles (protokolą), kurių prisilaikant procesams būtų nesvarbu:
  - kokia seka procesorius vykdys procesus
  - kokiame taške jie bus pertraukiami (net ir esant daugeliui procesorių).
- Šios taisyklės garantuotų, kad bet kuriuo momentu su kelių procesų bendrai naudojamais duomenimis operuoja tik vienas procesas.

- Iš anksto apie procesų vykdymą paprastai galima pasakyti tik tiek, kad kiekvienas procesas yra vykdomas nenuliniu greičiu, tačiau nėra jokių galimybių nusakyti :
  - santykinius procesų greičius vienas kito atžvilgiu,
  - kokia seka bus vykdomi skaičiavimai bei persijungiama nuo proceso prie proceso
  - kokia seka procesai naudosis bendrai naudojamais duomenimis.



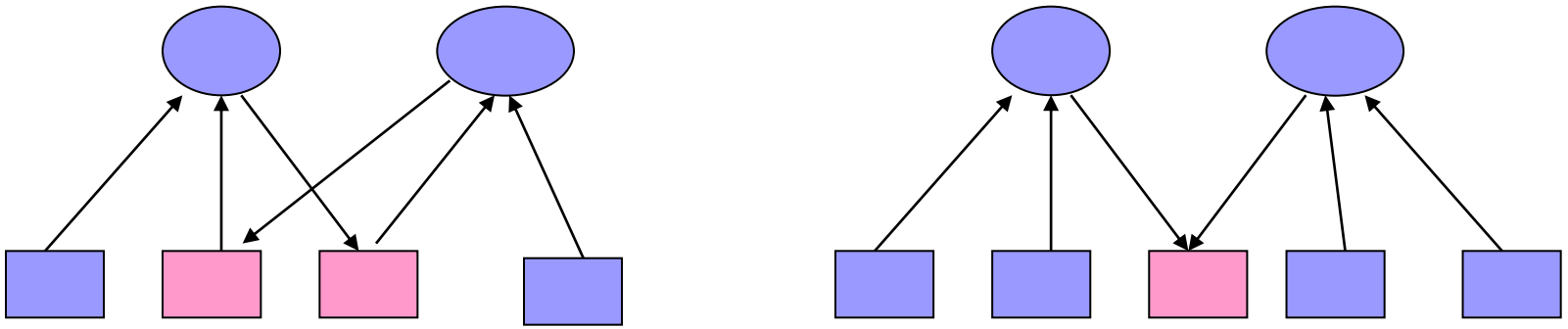
Reikalavimai efektyviam kritinės sekcijos problemos sprendimui

- **Tarpusavio išskirtinumo reikalavimas:**
  - bet kuriuo laiko momentu tik vienas procesas gali vykdyti kritinės sekcijos (CS) veiksmus.
- **Eigos-progreso reikalavimas**:
  - Tik tie procesai, kurie yra „įėjimo“ sekcijoje gali įeiti į savas CS. Šis išrinkimas negali būti atidėtas neapibrėžtam laikui.
- **Ribinio laukimo reikalavimas**:
  - Po to, kai procesas užsiprašo leidimo įeiti į kritinę sekciją (CS), egzistuoja tam tikra ribinė laukimo reikšmė, kuri nusako kiek kartų kitiems procesams bus leista įeiti į savas CS iki to momento, kol šis procesas įeis į CS



Tarpusavio išskirtinumo reikalavimas

- Tarpusavio išskirtinumo reikalavimo užtikrinimas gali atvesti prie :
  - mirties taško situacijos susidarymo
  - badavimo situacijos



Sprendimų tipai

- **Programiniai sprendimai**.
  - realizuojami programiniais principais, naudojami algoritmai, kurie užtikrina procesų tarpusavio išskirtinumą
- **Techninės įrangos sprendimai.**
  - Jie pagristi tam tikrų, specialių mašininių komandų panaudojimu.
- **Operacinių sistemų sprendimai**
  - pateikiamos tam tikros operacinės sistemos funkcijos bei duomenų struktūros, leidžiančios spręsti kritinės sekcijos problemas.

Programiniai sprendimai

- Programiniai sprendimai taikomi lygiagretiems procesams (gijoms), kurie gali būti vykdomi kompiuterinėse sistemose, turinčiose tiek vieną, tiek daug procesorių, kurie naudojasi bendra atmintine.
- Programinių sprendimų atveju nėra numatomas joks palaikymas nei iš operacinės sistemos pusės nei iš techninės įrangos pusės.
  - reikia programiškai garantuoti tai, kad kreipinį į kritinę sekciją gautų tik vienas procesas

Programiniai sprendimai

- Pirmas algoritmas:

```
var turn: 0..1;

PROCESAS P0                    PROCESAS P1:
    ...                            ...
     while(turn!=0)                 while(turn!=1)
   { /* laukti   */ };             { /* laukti   */};
        CS                             CS
     turn:=1;                       turn:=0;
        RS                             RS
    ...                            ...
```

Tenkinamas *tarpusavio išskirtinumo* reikalavimas
Eigos (progreso) reikalavimas nėra tenkinamas
Ribinio laukimo reikalavimas taip pat nėra tenkinamas

Pirmas algoritmas (antra realizacija)

- Kintamasis flag rodo, kuris kintamasis yra CS

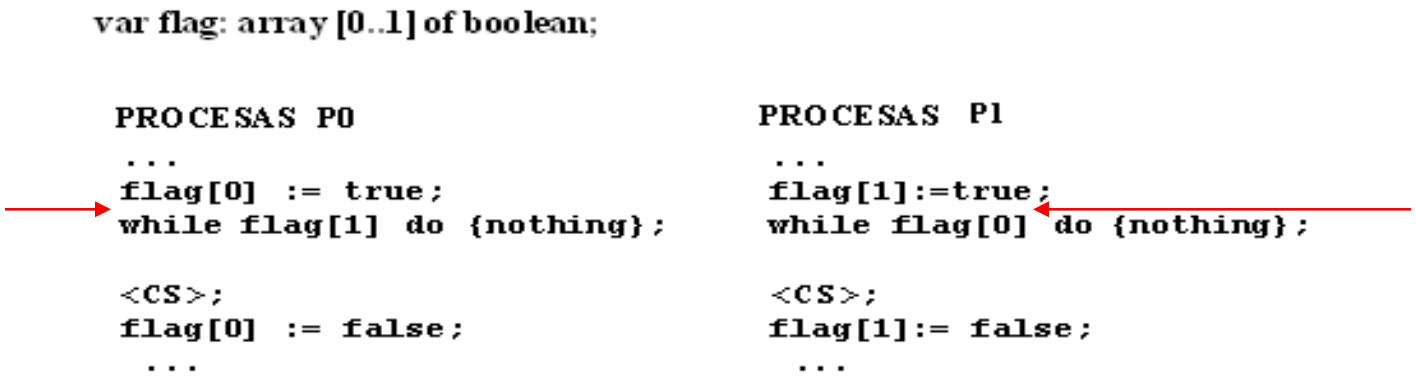
```
var flag: array[0..1] of boolean;

PROCESAS P0                          PROCESAS P1:

...                                  ...
while flag[1] do {nothing}           while flag[0]do{nothing};
flag[0]:=true;                       flag[1]:=true;
<CS>;                                <CS>;
flag[0] := false;                    flag[1]:=false;
...                                  ...
```

Kadangi procesai gali būti pertraukti bet kuriuo metu, galima situacija, kad abu procesai pradės kartu vykdyti kritinės sekcijos veiksmus ir galės vykdyti bendrai naudojamų kintamųjų keitimą.

Pirmas algoritmas: trečia realizacija



- Čia, siekiant išvengti antros realizacijos trūkumų, procesai iš anksto nustato savas vėliavėles į vienetinę padėtį, dar nepatikrinę kito proceso vėliavėlės būsenos. Tai turėtų rodyti atitinkamo proceso ketinimą įeiti į **CS**.
- Tarpusavio išskirtinumas yra patenkinamas, bet ne eigos reikalavimas

Dekkerio algoritmas

- Panaudojamos ne tik vėliavėlės, rodančios, kad atitinkamas procesas nori įeiti į kritinę sekciją, bet ir kintamasis **turn**, kurio reikšmė rodo, ar kažkuris iš procesų jau yra įėjęs į kritinę sekciją.

```
PROCESAS  P0
begin
  repeat
   flag[0] := true;
   while flag [1] do if turn =1 then
        begin
        flag[0] := false;
         while turn=1 do {nothing};
        flag[0] := true
        end;
   <critical section>;
   turn := 1;
   flag[0] := false;
   remainder
  forever
end;
```

```
PROCESAS  P1
begin
  repeat
   flag[1] := true;
   while flag [0] do if turn =0 then
        begin
        flag[1] := false;
         while turn=0 do {nothing};
        flag[1] := true
        end;
   <critical section>;
   turn := 0;
   flag[1] := false;
   remainder
  forever
end;
```

Pradinės sąlygos:

```
Begin
      flag[0] := false;
      flag[1] := false;
      turn :=1;
      parbegin
            P0; P1
      parend
end
```

Šiame algoritme išlieka reikalavimas, kad procesai eilės tvarka įeitų į kritines sekcijas.

Petersono algoritmas

- Jei abu procesai bando įeiti į kritines sekcijas vienu laiku, kintamasis **turn** leis tai atlikti tik vienam procesui. Išėjimo sekcija – ji nustatys tą faktą, kad **Pi** procesas nebeketina įeiti į CS

Pradinių reikšmių priskyrimas:

```
flag[0]:=flag[1]:=false
turn:=0 (arba 1)
```

Proceso *Pi* algoritmas:

```
Process Pi:
repeat
  flag[i]:=true;
  turn:=j;
  do {} while (flag[j] and  turn=j);
     CS
  flag[i]:=false;
     RS
forever
```

Algoritmas tenkina išskirtinumo, eigos bei riboto laukimo reikalavimus

Kepyklos (bakery) algoritmas sprendimas, esant n procesų

- Prieš įeinant procesams į savas CS, kiekvienas Pi procesas gauna savo numerį. Procesas, turintis mažiausią numerį įeina į CS
  - (a,b) < (c,d), jei a < c arba jei a = c ir b < d.
  - max(a0,...ak) tai toks skaičius b, kuris tenkina sąlygą: b > ai, visiems i=0, ...,k

```
Process Pi:
repeat
  choosing[i]:=true;                              /*  renkamas NR   */
  number[i]:=max(number[0]..number[n-1])+1;
  choosing[i]:=false;
  for j:=0 to n-1 do {
    while (choosing[j])  {}; ...../* laukiama kol nevyks Nr pasirinkimas
*/
    while (number[j]!=0
    and (number[j],j)<(number[i],i)) {}; /*palaukiama, kol praeis turintys
mazesnius Nr */
  }
  CS
        number[i]:=0;
      RS
      forever
```

Bakery algoritmas tenkina tiek tarpusavio išskirtinumo, tiek eigos, tiek ribinio laukimo sąlygas n procesų atveju

Programinio sprendimo trūkumai

- Procesai, kurie užsiprašo įėjimo į kritines sekcijas randasi „*užimto laukimo*“ (busy waiting) būklėje
  - jie yra aktyvūs
  - procesorius turi juos apklausti kiekvieną kartą pasirinkdamas procesą vykdymui
    - tai nenaudingai eikvoja CPU laiką.
    - Jei kritinėje sekcijoje vykdomų veiksmų seka yra pakankamai ilga, tai didesnė prasmė yra blokuoti procesus, laukiančius įėjimo į kritines sekcijas nei laikyti juos šiame „užimto laukimo“ būvyje.

Techniniai sprendimai

- Grindžiami tuo, kad yra *uždraudžiami pertraukimai* vykdant procesui kritinės sekcijos veiksmus :
  - garantuoja tarpusavio išskirtinumo užtikrinimą,
  - bet kenčia efektyvumas.
- Jei kompiuteris turi keletą CPU, tai šios priemonės netinka, nes negalima uždrausti kitiems CPU vykdyti kitus procesus.

```
repeat
   disable interrupts
    critical section
   enable interrupts
    remainder section
forever
```

Specialios mašininės komandos

- Operacinių sistemų projektuotojai yra pasiūlę mašinines komandas, kurios vykdo du veiksmus „atomiškai“ (neperskiriamai, nedalomai), t.y. kai po vieno veiksmo atlikimo nėra galimas pertraukimas kol nėra įvykdytas ir antras veiksmas su ta pačia atmintinės sritimi (pavyzdžiui, skaitymas ir testavimas):
  - “**test-and-set**” komanda

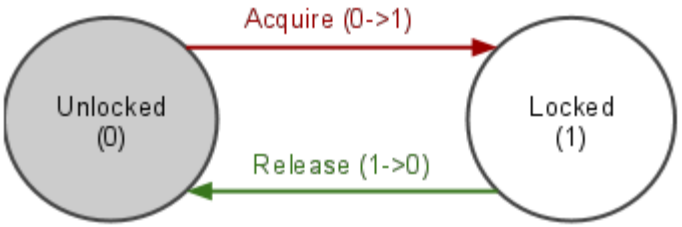
```
bool testset(int &i)
{
  if (i==0) {
    i=1;
    return true;
  } else {
    return false;
  }
}
```

```
Process Pi:
repeat
  repeat{}
  until testset(b);
     CS
  b:=0;
     RS
forever
```

Šis algoritmas pilnai užtikrina procesų tarpusavio išskirtinumą: procesui Pi įėjus į CS, visi kiti procesai Pj yra užimto laukimo būsenoje. Procesui Pi išėjus iš CS, išrinkimas sekančio proceso Pj, kuris sekančiu įeis į CS, yra atsitiktinis. To rezultate kažkuriam procesui gali gautis neribotas laukimo laikas, o taip pat galima ir „badavimo“ situacija

“Mutex” tipo užraktas

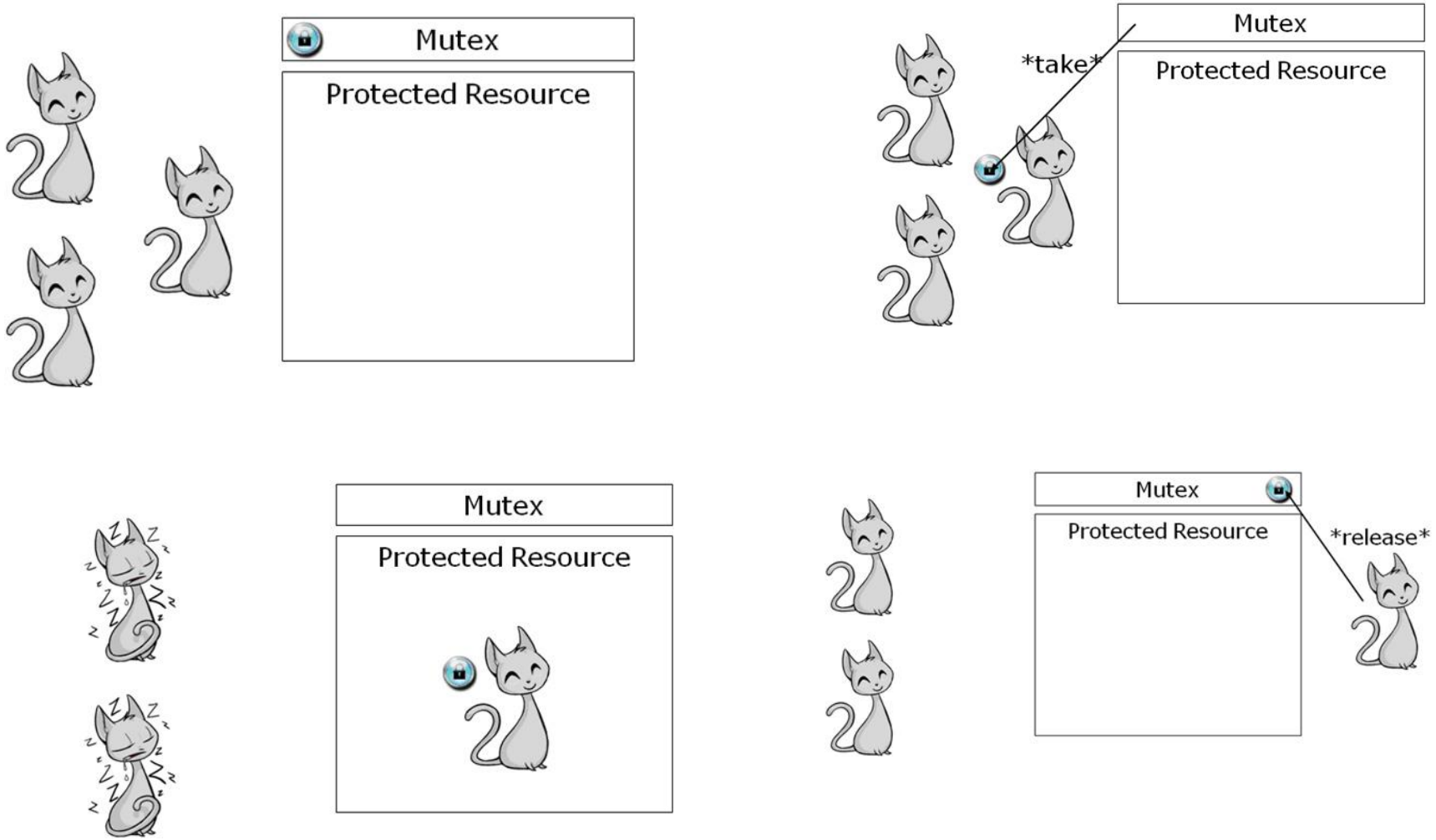
- “Mutex” užraktas naudojamas tik procesų tarpusavio išskirtumo užtikrinimui, valdant prieigą prie bendrai naudojamų duomenų ar bendros kodo dalies
- „Mutex“ tipo objektas gali būti viename iš dviejų būvių:
  - užrakintas arba atrakintas (1-0).
    - Šias dvi būsenas gali atvaizduoti ir dvi bet kurio bito reikšmės, tačiau dažniau yra naudojamas sveiko tipo kintamasis.
  - Mutex tipo užraktą gali atitikti dvejetainis semaforas, kuris gali įgyti tik dvi reikšmes nulį arba vienetą.



Perėjimai:

acquire : unlocked > locked
release : locked > unlocked

Mutex tipo užrakinimai

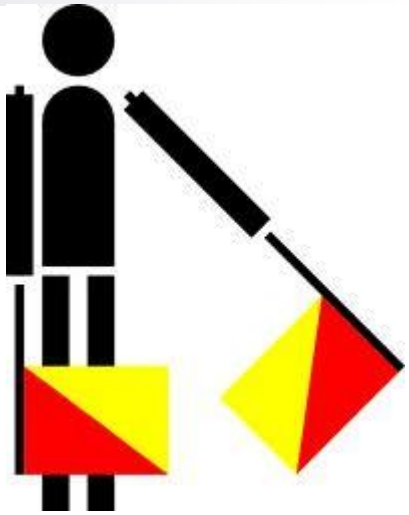


Semaforai

- Semaforai yra operacinės sistemos teikiamas įrankis, leidžiantis užtikrinti sinchronizaciją.
- Semaforas S gali būti traktuojamas kaip sveiko tipo kintamasis, su kuriuo gali būti atliekamos dvi operacijos: įjungimo ir išjungimo:
  - **signal(S)**
  - **wait(S)**

```
type semaphore = record
                   count: integer;
                   queue: list of process
                 end;
var S: semaphore;
```

Semaforai



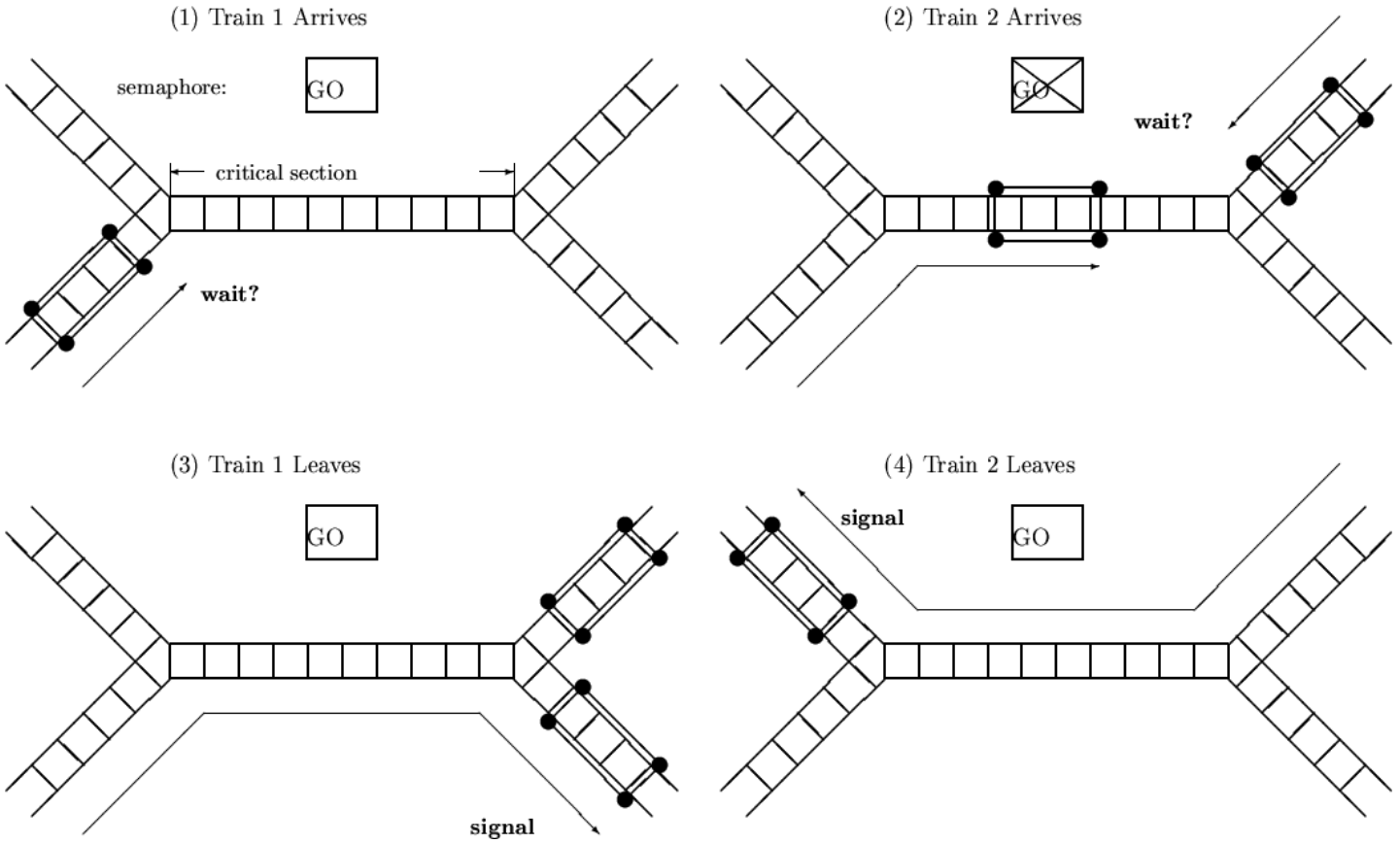
- Semaforai: Pirmą kartą panaudoti Dijkstra 1960 metais

- Jie turi du tikslus
  - Užtikrinti tarpusavio ***išskirtinumą*** (Mutex): T.y. užtikrinti, kad kelios gijos vienu metu neįeitų į kritinę sekciją.
  - Užtikrinti ***sinchronizaciją***: T.y. užtikrinti tai, kad gijos atliktų veiksmus tam tikra tvarka
- Pagal tai, kokias reikšmes gali įgyti semaforai galimi du semaforų tipai:
  - ***Skaitmeninis*** semaforas (Counting semaphore) – sveikojo tipo kintamasis gali įgyti neapibrėžtas reikšmes.
  - ***Dvejetainis*** semaforas (Binary semaphore) – sveikojo tipo kintamasis gali įgauti reikšmes 0 arba 1 (arba false-true).

Semaforai

- Norėdamas įeiti į kritinę sekciją procesas vykdo išjungimo operaciją **`wait(S):`**
  - Jei S>0, semaforo reikšmė yra mažinama.
  - Jei S=**0**, tai procesas yra užblokuojamas.
    - Taigi procesas vietoje buvimo užimto laukimo (busy waiting) būklėje, yra blokuojamas ir perkeliamas į blokuotų procesų eilę.
- **signal(S)** yra vykdomas išėjus iš kritinės sekcijos.
  - Jo metu patikrinama, ar yra blokuotų procesų.
  - Jei taip, tai vienas iš procesų yra atblokuojamas ir tęsia savo veiksmus.
  - Jei blokuotų procesų nėra, yra padidinama semaforo reikšmė.
- Operacijos atomiškumas žymi tai, kad jei yra prasidėjusi operacija **signal()** arba **wait()**, tai joks kitas procesas negali prieiti prie to semaforo šios operacijos metu

Veiksmai su semaforu



Veiksmai su semaforu

- Jei proceso laukimo būklė yra surišta su semaforu S, tai šis procesas yra blokuotas ir randasi semaforo S eilėje .
- Pradžioje **S.count** =1

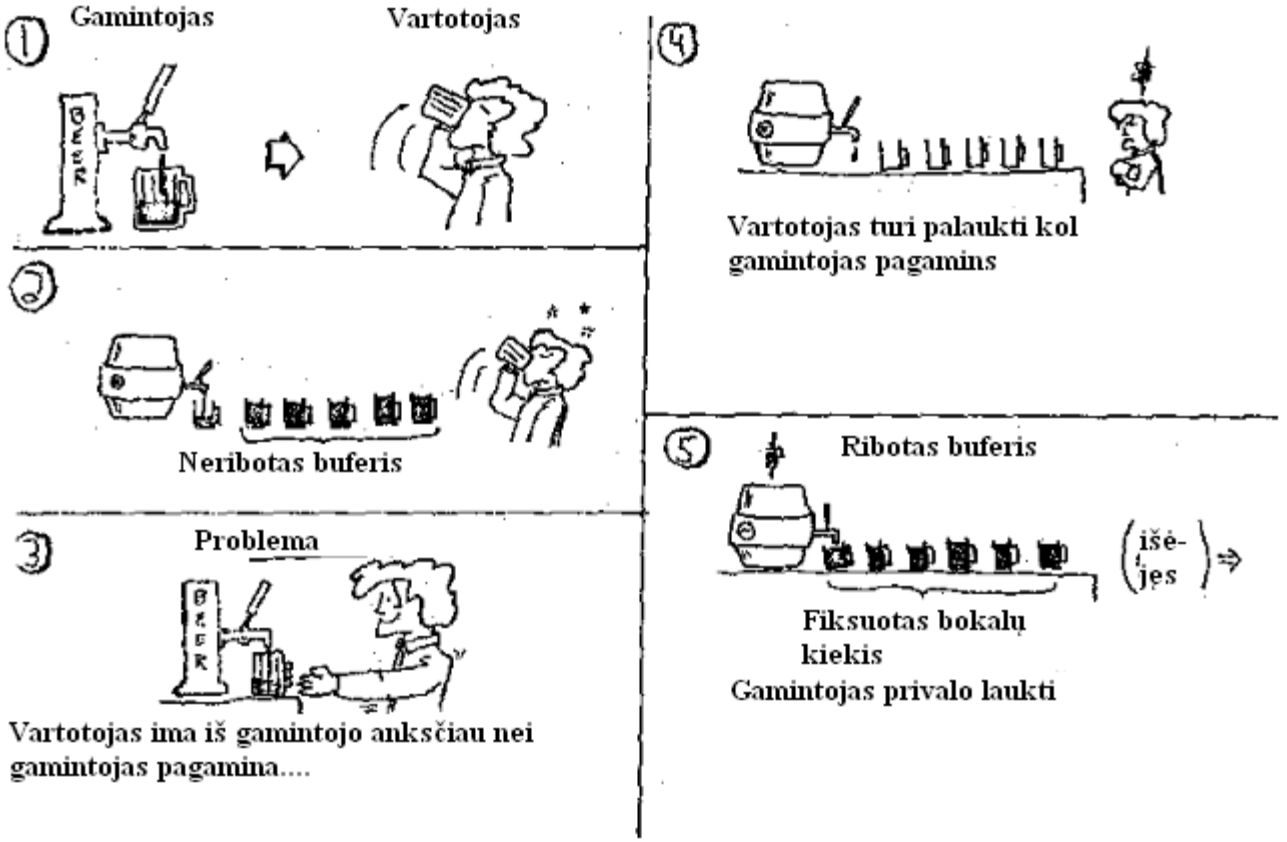
```
wait(S):
  S.count--;
  if (S.count<0)
{
    blokuoti šį procesą
    patalpinti šį procesą
    į S semaforo eilę
 }
```

```
signal(S):
  S.count++;
  if (S.count<=0)
{
    išstumti procesą P
     iš semaforo S eilės
    patalpinti šį procesą P
    į pasiruošusių procesų sąrašą
}
```

Realizacija wait(S) bei signal(S):
Vieno procesoriaus atveju: uždraudžiami pertraukimai vykdant šias operacijas (gana trumpam laiko intervale).

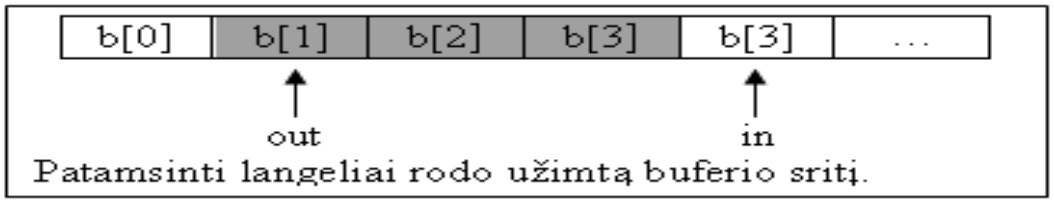
Daugelio procesorių atveju: yra galima pasinaudoti aukščiau paminėtomis programinėmis ar techninėmis realizacijomis.

Vartotojo/gamintojo problema



Vartotojo/gamintojo problema: neribotas buferis

- Neribotą buferį gali realizuoti paprasčiausias elementų masyvas, su kuriuo surišamos dvi nuorodos :
  - **in** – rodanti į sekantį gaminamą elementą;
  - **out** – rodanti į sekantį vartojamą elementą.
  - **b[i]** – informacijos vienetas
- Reikalingi semaforai:
  - Mutex tipo semaforas **S** kuris garantuotų procesų tarpusavio išskirtumą, jiems kreipiantis į buferį.
  - Skaitmeninis semaforas **N**, surištas su elementų kiekiu buferyje (**N=in–out**)



Vartotojo/gamintojo problema: neribotas buferis

```
Pradinės reikšmės:
  S.count:=1;
  N.count:=0;
  in:=out:=0;
```

```
Gamintojas (Producer):
repeat
  gaminti v;
  wait(S);
  append(v); /* kritinė sekcija */
  signal(S);
  signal(N);
forever
```

```
Vartotojas (Consumer):
repeat
  wait(N);
  wait(S);
  w:=take();/* kritinė sekcija */
  signal(S);
  vartoti(w);
forever
```

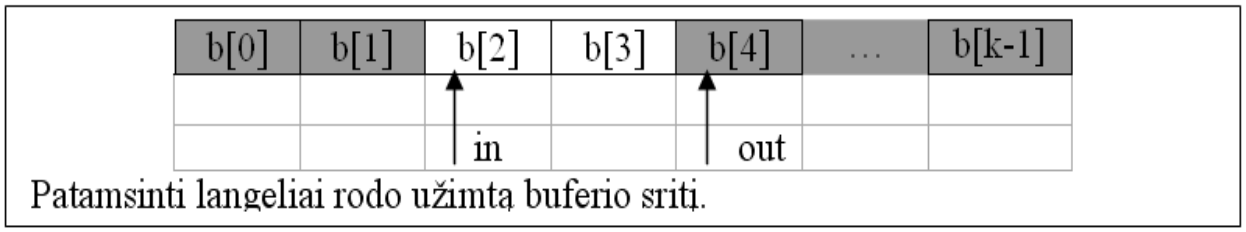
```
Funkcijos:

append(v):
      b[in]:=v;
      in++;

take():
      w:=b[out];
      out++;
      return w;
```

Vartotojo – gamintojo santykiai: baigtinio dydžio buferis

- gamintojas susidurs su tam tikra problema – jis galės gaminti tik tuo atveju, jei neužimta buferio erdvė **E**
  - esanti laisva – neužimta buferio erdvė **E** bus **>=1,** tai atspindės *semaforas E*.
  - *Semaforas S* skirtas “mutex” užtikrinimui.
  - *Semaforas N* rodys, kad buferis netuščias



Patamsinti langeliai rodo užimtą buferio sritį.

Vartotojo – gamintojo santykiai: baigtinio dydžio buferis

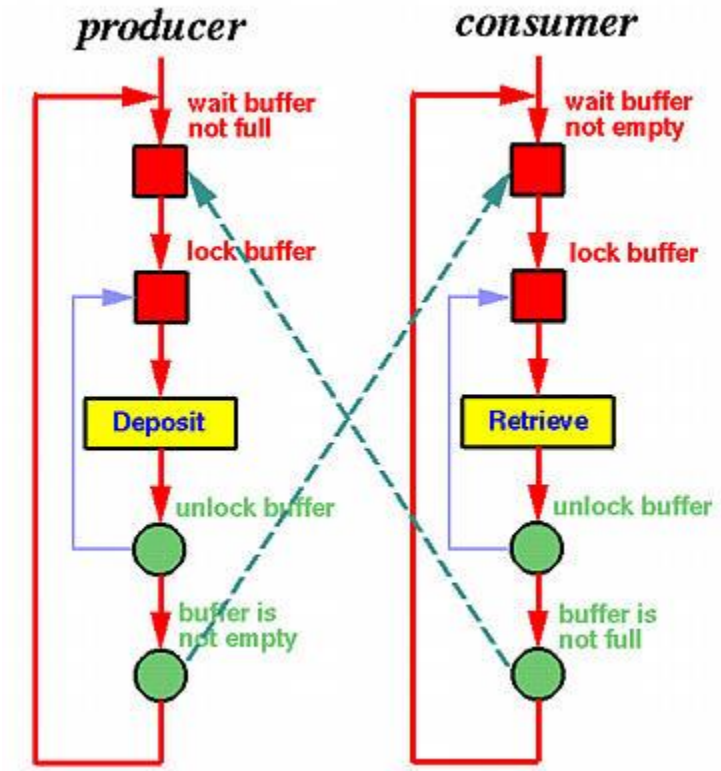
```
S.count:=1;
N.count:=0;
E.count:=k;
```

```
Gamintojas:
repeat
  produce v;
  wait(E);
  wait(S);
  append(v);
  signal(S);
  signal(N);
forever
```

```
Vartotojas:
repeat
  wait(N);
  wait(S);
  w:=take();
  signal(S);
  signal(E);
  consume(w);
forever
```

```
append(v):
b[in]:=v;
in:=(in+1) mod k;
```

```
take():
w:=b[out];
out:=(out+1) mod k;
return w;
```

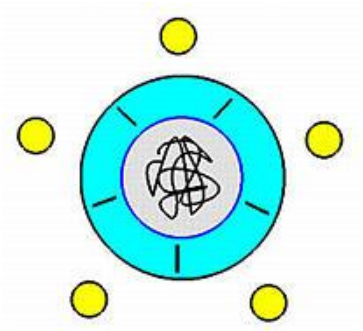


Pietaujančių filosofų problema

- Tai klasikinis pavyzdys, nagrinėtas E.W.Dijkstra, kuris demonstruoja sinchronizacijos bei resursų išskyrimo problemas.
- Įsivaizduokime 5 filosofus, kurie leidžia gyvenimą dviejose būsenose: arba jie galvoja, arba valgo.
  - Turime penkias šakutes.
  - Filosofai mąsto. Kai kuris nors iš jų išalksta, jis sėdasi prie stalo, ima dvi šakutes, kurios yra arčiausiai jo (valgymui reikalingos abi šakutės – tiek šakutė iš kairės, tiek šakutė iš dešinės pusės).
  - Jei filosofas gali pasiimti abi šakutes – jis kurį laiką valgo. Pavalgęs jis grąžina šakutes ir vėl mąsto.
- Šakutė-resursas, kuriuo turi dalintis du gretimi filosofai,
- filosofas negali atimti šakutės, kurią jau turi paėmęs greta sėdintis filosofas.
- Šakutės paėmimas turi būti vykdomas tarpusavio išskirtinumo (mutex) sąlygomis.

Pietaujančių filosofų problema

- galima mirties taško situacija
- galima ir badavimo situacija



```
Process Pi:
repeat
 think;
 wait(fork[i]);
 wait(fork[i+1 mod 5]);
 eat;
 signal(fork[i+1 mod 5]);
 signal(fork[i]);
forever
```

Pietaujančių filosofų problema

- Bazinė idėja yra ta, kad nors bet kuri konkuruojanti gija visada daro tai, kas joms atrodo geriausia jų atžvilgiu ir teisėta jų atžvilgiu, tačiau esant bendrai naudojamiems resursams tai gali vesti į chaosą (visi filosofai badauja...)
- Šis pavyzdys siekia parodyti, kad esant konkuruojantiems procesams, kurie varžosi dėl bendrai naudojamų resursų šių procesų aprašymui netinka tradicinio programavimo priemonės.
- Lygiagretaus programavimo technika teikia įvairias priemones, kurių pagalba gali būti sprendžiama pietaujančių filosofų problema.

Skaitytojų – rašytojų problema

Kintamuoju X (failu, įrašu) dalosi keli procesai, kurie su juo atlieka skaitymo ir rašymo operacijas.

Duomenų objektas (toks kaip failas ar įrašas) yra prieinamas keliems lygiagretiems procesams. Keletas šių procesų gali norėti tik skaityti duomenų objekto turinį, tuo tarpu kiti procesai gali norėti atnaujinti (t.y. skaityti ir rašyti) prieinamą objektą.

Reikia užtikrinti sekančias sąlygas :

- *Vienu metu leisti rašyti tik vienam procesui*.
- *Leisti skaitymo veiksmą daugeliui procesų.*
- *Negalima leist badavimo skaitymo procesams.*
- *Negalima leist badavimo rašantiems procesams.*
- *Apseiti be "busy waiting"*.

Sprendimas

X – skaitomas (rašomas) objektas

rc – skaitytojų kiekis

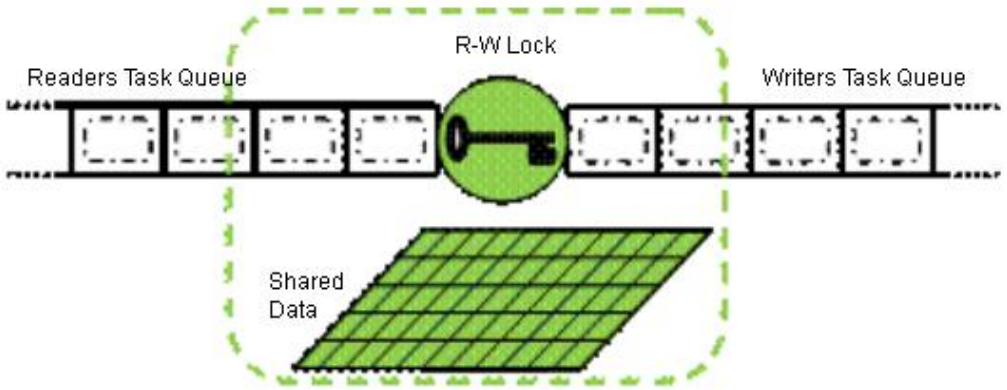
Procesai, norėdami gauti prieigą turės kviesti: StartRead(), EndRead(), StartWrite(), EndWrite().

- Semaforas xLock pradžioje 1; // kontroliuoja prieigą prie X
-
- integer rc = 0; // skaičius skaitančių arba norinčių skaityti
- Semaforas rcMutex = 1; // apsaugo rc

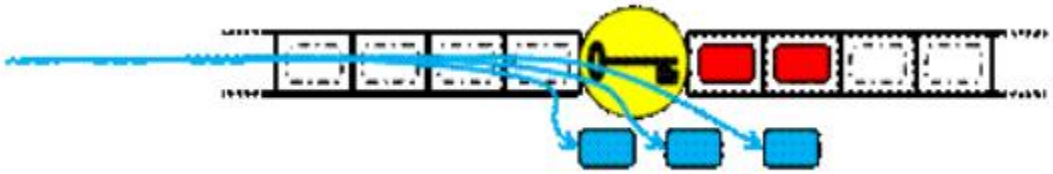
- Negalima X įdėti į kritinę sekciją, nes reikia prieigos daugeliui skaitančių procesų

- StartRead( )
  - { **wait** (**rcMutex**);
  - rc := rc + 1;
  - if rc = 1 then {**wait**(**xLock**) };
  - **signal**(**rcMutex**); }
  - **...**
  - **Vyksta skaitymas**
  - **...**
- EndRead( )
  - { **wait** (**rcMutex**);
  - rc := rc - 1;
  - if rc = 0 then { **signal** (**xLock**) };
  - **signal**(**rcMutex**); }
- StartWrite( )
  - { **wait**(**xLock**); }
  - **...**
  - **Vyksta rašymas**
  - **...**
- EndWrite( )
  - { **signal**(**xLock**); } }

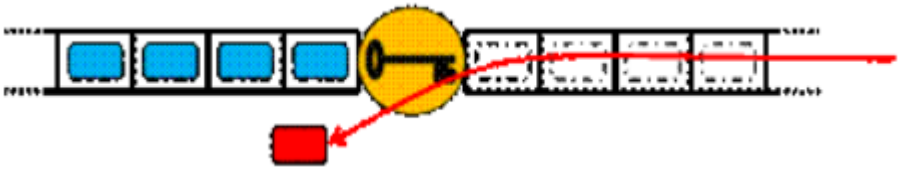
Skaitytojų – rašytojų problema



- Pirmumas suteikiamas skaitantiems procesams

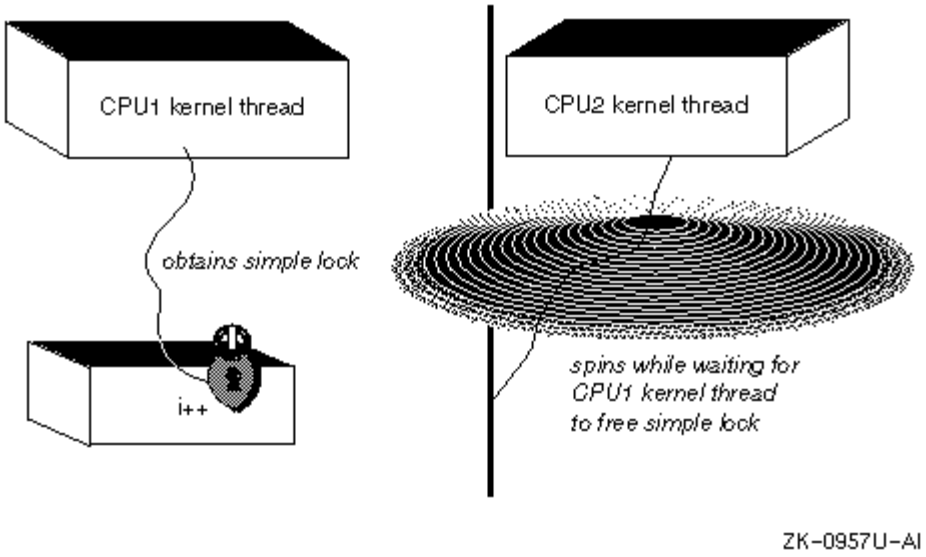


Pirmumas suteikiamas rašantiems procesams



Spinlocks

- Tai procesų sinchronizavimo priemonė, kuri vietoje proceso blokavimo naudoja procesų „užimto laukimo“ būvį (busy waiting).
- Jie naudotini tada, kai kompiuteris turi keletą procesorių (SMP), o veiksmai kritinėje sekcijoje yra neilgi.
- Kažkuriai gijai vykdant kreipinį į “spinlock” būdu užrakintą resursą, jis gali būt užimamas (įjungiama vėliavėlė) arba gija sukasi cikle, laukdama įėjimo (tai trunka kelis komandų vykdymo taktus).
- Tokiu atveju yra šiek tiek prarandama CPU laiko, bet nereikalingas procesų persijungimas.

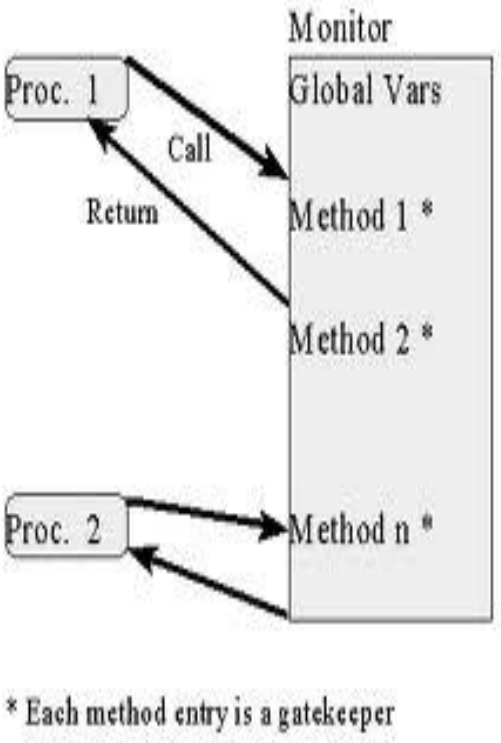


Monitoriai

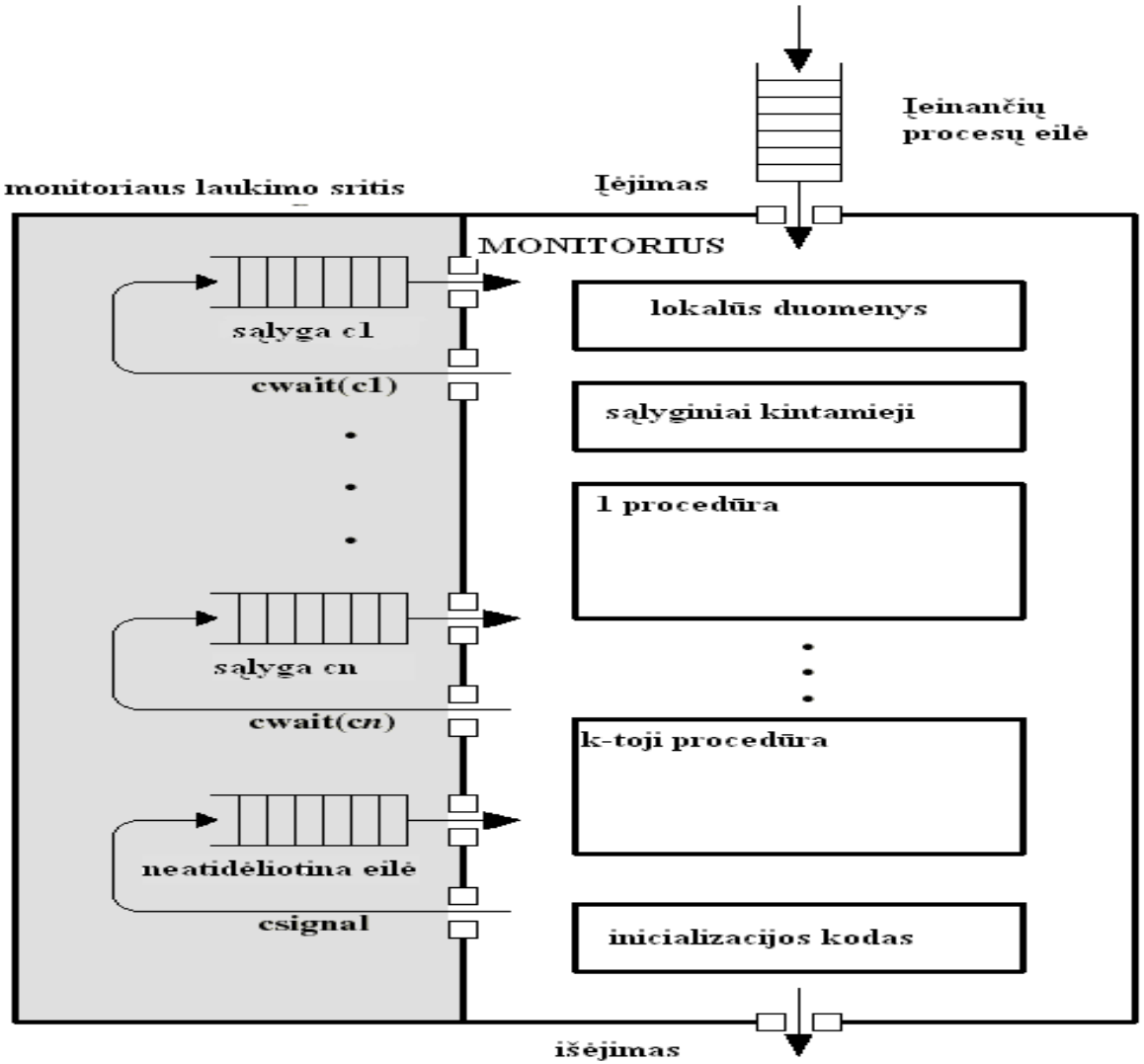
- Monitoriai – tai aukšto lygio programavimo kalboje realizuotos konstrukcijos, kurios teikia ekvivalentišką funkcionalumą kaip ir semaforai, bet juos yra paprasčiau kontroliuoti.
- Juos galima realizuoti naudojant C++, Java bei kitas lygiagrečiam programavimui pritaikytas kalbas.
- Monitoriai paprastai yra sudaryti:
  - iš vienos ar daugiau procedūrų,
  - turi pradinių duomenų nustatymo – inicializacijos seką,
  - naudoja lokalius kintamuosius.
  - Lokalūs kintamieji yra pasiekiami tik per monitoriaus procedūras.
  - Procesas gali vykdyti į monitoriaus sudėtį įeinančius veiksmus tik iškviesdamas kažkurią iš monitoriaus procedūrų.
  - Tik vienam procesui vienu metu leidžiama „būti“ monitoriuje.

Monitoriai

- Procesų sinchronizacija realizuojama naudojant sąlyginius kintamuosius.
  - Sąlyginiais kintamaisiais yra nusakomos sąlygos, kurių procesas turi laukti prieš pradėdamas vykdyti monitoriaus veiksmus.
- Sąlyginiai kintamieji yra lokalūs monitoriaus kintamieji – jie yra pasiekiami tik esant monitoriaus aplinkoje.
- Jie yra pasiekiami ir jų reikšmės gali būti pakeičiamos naudojant dvi funkcijas **cwait(a)** ir **csignal(a)**:
  - **cwait(a)** funkcija blokuoja kviečiančio proceso vykdymą ties sąlyginiu kintamuoju **a**.
  - **csignal(a)** funkcija atnaujina kažkurio proceso vykdymą, kurio blokavimas buvo susijęs su sąlyginiu kintamuoju **a**.



Monitoriaus schema



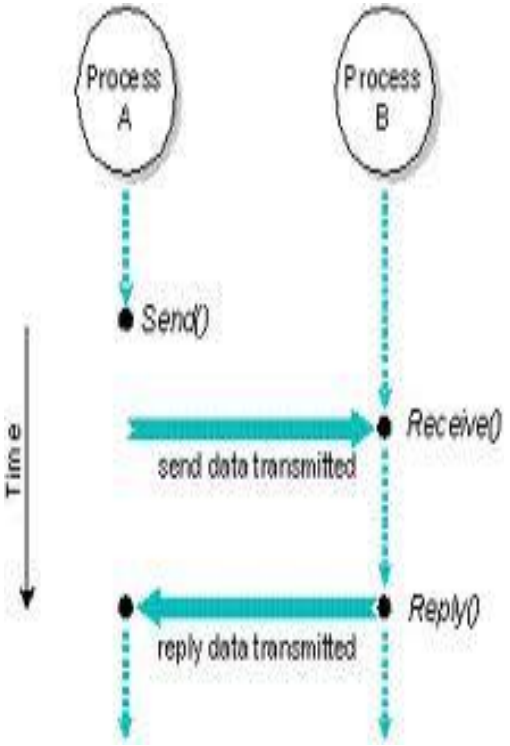
Monitoriaus pavyzdys

- Tarkime, kad turime baigtinio dydžio buferį. Monitoriuje jis gali būti aprašomas taip:
  - **buffer: array[0..k-1] of items;**
- Reikalingi du sąlyginiai kintamieji:
  - ***notfull*: true** kai **buffer** nėra užpildytas.
  - ***notemty*: true** kai **buffer** nėra tuščias.
  - **append(v)**, **take(v)** yra monitoriaus procedūros.

```
Monitor boundedbuffer:
  buffer: array[0..k-1] of items;
  nextin, nextout, count: integer;
  notfull, notempty: condition;
  Append(v):
    while (count=k) cwait(notfull);
    buffer[nextin]:= v;
    nextin:= nextin+1 mod k;
    count++;
    csignal(notempty);
  take(v):
    while (count=0) cwait(notempty);
    v:= buffer[nextout];
    nextout:= nextout+1 mod k;
    count--;
    csignal(notfull);
```

Pranešimų perdavimas

- Pranešimų perdavimas – tai vienas iš bendriausių būdų, naudojamų procesų tarpusavio komunikacijose (interprocess communicanion – IPC).
- Esant patikimam pranešimų perdavimui, tai gali būti panaudojama užtikrinant procesų tarpusavio sinchronizaciją bei tarpusavio išskirtinumą.
- Veiksmuose su pranešimais yra nurodoma:
  - **send()-** paskirties taškas bei siunčiamas pranešimas.
  - **receive()-** šaltinis bei pranešimas.



Pranešimų perdavimas

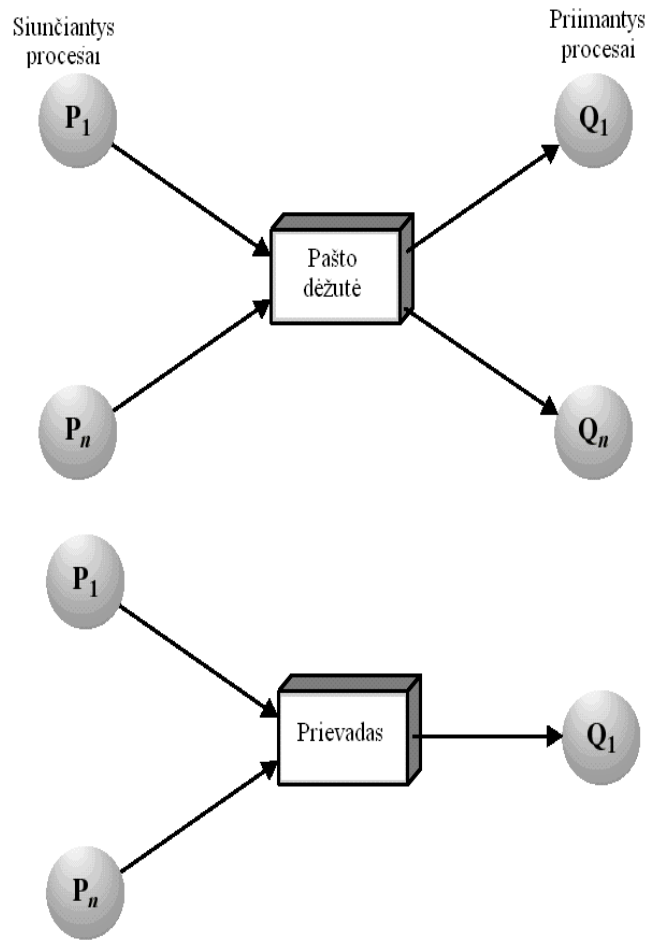
- Dauguma operacinių sistemų palaiko pranešimų perdavimus. Procesai gali siųsti pranešimus vienas kitam.
- “send()”, “receive()” funkcijos gali būti :
  - Blokuojančios arba ne.
  - Dažniausiai naudojama neblokuojanti “send” ir blokuojanti “receive”
- Kartais gali būti naudojamos abi blokuojančios funkcijos **send()** bei **receive()**, kai abu procesai yra blokuojami tol, kol nėra gautas pranešimas.
  -
  - Toks sprendimo variantas yra naudojamas tuo atveju, kai informacijos pasikeitimui tarp procesų nėra naudojamas buferis (pranešimų eilė).
  -
  - Taip yra užtikrinama kieta procesų veiksmų sinchronizacija

Adresacija

- Adresacija pranešimų siuntimo atveju gali būti naudojama dvejopa:
  - tiesioginė
  - netiesioginė.
- Tiesioginės adresacijos atveju nurodant pranešimo šaltinį arba pranešimo gavėją gali būti naudojamas specifinis proceso identifikatorius.
  - iš anksto sunku žinoti šį identifikatorių.
- Netiesioginė adresacija yra labiau priimtina.
  - pranešimai yra siunčiami į bendrai naudojamas pašto dėžutes, kurias sudaro pranešimų eilė.

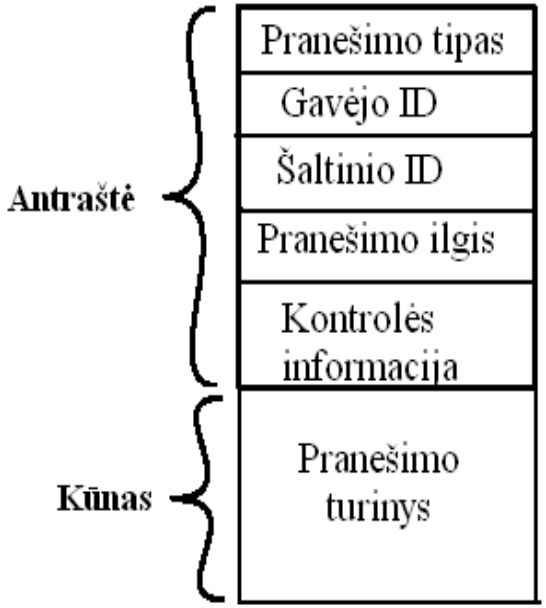
Pašto dėžutės ir prievadai (portai)

- Pašto dėžučių atveju, ji gali būti privati kiekvienos siuntėjo - gavėjo poros atžvilgiu.
- Galimas taip pat atvejis, kai ta pačia pašto dėžute dalinasi kelios siuntėjų - gavėjų poros.
  - Operacinė sistema gali leisti naudoti *pranešimų tipus* tam, kad identifikuoti ryšio porą.
- Prievadas yra pašto dėžutė, kuri yra surišta su vienu gavėju ir daugeliu siuntėjų.
  - Prievado sąvoka plačiai naudojama kliento-serverio modelio taikomosiose programose.



Pranešimų formatai

- Siunčiant pranešimus yra reglamentuojamas jų formatas
- pranešimas susideda iš :
  - pranešimo antraštės
  - pranešimo kūno.
- Kontrolės informacija :
  - apima nurodymus, kas turi būti daroma, jei neužtenka vietos buferyje,
  - nurodomas pranešimų sekos numeris,
  - prioritetas bei kita informacija.
- Pranešimus kaupiant į eilę, eilei dažniausiai yra taikoma FIFO disciplina, bet gali būti vertinami ir prioritetai.



Tarpusavio išskirtinumo užtikrinimas naudojant pranešimus

- Sukuriama pašto dėžutė, pavyzdžiui, vardu ***mutex***, kuria dalinsis **n** procesų.
- Tarkim, kad yra naudojama neblokuojanti **send()** funkcija,
  - **send(mutex,„go“)**.
- Funkcija **receive()** blokuoja procesą, kol **mutex** dėžutė yra tuščia.
- Kiekvieno iš **n** pranešimo gavėjų procesų programos kode naudojama parodyta algoritmo dalis.

```
Procesas Pi
        var msg:  message;
        repeat
                receive(mutex, msg);
                CS
                send(mutex, msg);
        RS
        forever
```

Riboto buferio gamintojo/vartotojo problema ir jos sprendimas siunčiant pranešimus

- Turime gamintojo bei vartotojo procesus.
- Naudojamos dvi pašto dėžutės:
  - mayconsume
  - mayproduce

- Gamintojas siunčia pagamintus dydžius į pašto dėžutę **mayconsume**.
  - Ši pašto dėžutė veiks kaip buferis (jo talpa k).
  - Iš jos vartotojo procesas ims pagamintus dydžius
- Pašto dėžutė **mayproduce** :
  - pradžioje yra užpildoma **k** nulinių pranešimų
  - **Mayproduce** dydis mažėja su kiekvienu pagamintu dydžiu ir didėja sulig kiekvienu suvartojimu.
- Naudojant šį algoritmą galima palaikyti daugelio gamintojų – vartotojų funkcionavimą.

Vartotojo-gamintojo santykiai realizuojant pranešimais

```
const  int
        capasity = /* buferio talpa */;
        null = /* tuščias pranešimas */
int i;

void producer ()
{       message pmsg;
        while (true) {
        receive (mayproduce, pmsg);
        pmsg = produce();
        send (mayconsume, pmsg);
        }
}

void consumer()
{       message cmsg;
        while (true) {
        receive (mayconsume, cmsg);
        consume (cmsg);
        send(mayproduce, null);
        }
}

void main()
{
        create_mailbox(mayproduce);
        create_mailbox(mayconsume);
        for (int i = 1; i <=capasity; i++) send (mayproduce,null);
        parbegin (producer,consumer);
}
```

Pašto dėžutės

- Operacinė sistema leidžia procesams:
  - Sukurti dėžutę
  - Siųsti ir gauti pranešimus per dėžutę
  - Išardyti dėžutę
- Dėžutės savininku gali būti :
  - operacinė sistema:
    - Dėžutė nepriklausoma, nepriskirta konkrečiam procesui
  - procesas
    - Savininkas – kuris sukūrė dėžutę (gali gauti pranešimus– tą gali daryti ir proceso vaikai)
    - Kiti – naudotojai –siunčia pranešimus (jie žino apie jos egzistavimą).
  - Dėžutės gyvavimas priklauso nuo dėžutės tipo.