# Operációs rendszerek I.

## Folyamatok együttműködése, kommunikációja

Várkonyiné Kóczy Annamária
Professzor
Informatika Tanszék
(koczya@ujs.sk)
varkonyi-koczy@uni-obuda.hu

**Felhasznált irodalom:**
- **Kóczy-Kondorosi (szerk.): Operációs rendszerek mérnöki megközelítésben**
- **Tanenbaum: Modern Operating Systems 2nd. Ed.**
- **Silberschatz, Galvin, Gagne: Operating System Concepts**

# Tartalom

- Bevezetés
- Szinkronizáció
- A kritikus szakasz megvalósítási módozatai
- Információcsere folyamatok között

# 3.1 Folyamatokból álló rendszerek

Modell: rendszer sok folyamattal, amelyek aszinkron módon dolgoznak, kommunikálnak egymással, osztoznak az erőforrásokon, adatokon, stb.

Motiváció (általános):

- Információ megosztás: több felhasználót is érdekelhet ugyanaz
- Erőforrás jobb kihasználása: átlapolt használat („egyidőben”)
- Gyorsabb feladat megoldás: részekre bontás, párhuzamos futtatás (de csak akkor hatékony, ha több CPU, I/O csatorna van)
- Modularitás: könnyebb értelmezhetőség, a feladat szerkezetét követő program (folyamat) struktúra
- Kényelem: egyszerre több mindent lehet csinálni

# 3.1 Folyamatokból álló rendszerek

Egy rendszer sok folyamatból állhat. Ezek között csatolás lehet:

- független folyamatok
  - egymás működését semmilyen formában nem befolyásolják,
  - nincs információjuk egymásról (sebesség, befejeződési idő, erőforrás használat, stb.)
  - aszinkron működés
    - egymással párhuzamosan is végrehajtódhatnak,
    - nincs időbeli függőség
- függő (csatolt)folyamatok,
  - Versengő folyamatok: logikailag független folyamatok, de megosztott erőforrás használat (pl. több user, azonos gépen dolgozik)
  - Együttműködő folyamatok: logikailag függő folyamatok
    - közösen oldanak meg valamely feladatot
    - együttműködnek, kommunikálnak, közös változók, stb.

# 3.1 Folyamatokból álló rendszerek

- Szorosan csatolt rendszerek (1 processzor, multiprogramozás)
  - A kommunikáció ált. közös memórián keresztül valósul meg → szinkronizáció
- Lazán csatolt rendszerek
  - A kommunikáció kommunikációs csatornán keresztül valósul meg → a szinkronizáció ebből a szempontból nem lényeges
  - A kommunikáció a fontos
- A szinkronizáció tárgyalható a kommunikáció egy aleseteként is, de mi külön vesszük

# Együttműködő folyamatok használatának indokai

- Erőforrások megosztása
  - átlapolt működés, jobb kihasználtság
- Számítások felgyorsulása (több processzor)
  - Számítások párhuzamosítása, végrehajtási sebesség nő.
- Felhasználók kényelme
  - Egy időben több feladat megoldása.
- Modularitás
  - Egy adott folyamat kisebb részekre való bontása
  - Jobb áttekinthetőség
  - Bizonyos feladatoknál (párhuzamos részek) kézenfekvő modell.
  - Pl. vezérlés, folyamatirányítás
  - A függőséget valahogy biztosítani kell

# 3.2 Szinkronizáció

- A folyamat végrehajtásának olyan időbeli korlátozása, ahol ez egy másik folyamat futásától illetve egy külső esemény bekövetkezésétől függ.

- Gyakori feladatok:
  - Precedencia (előidejűség)
  - Egyidejűség
  - Kölcsönös kizárás (versenyhelyzet, kritikus szakasz)

- Egyéb kapcsolódó fogalmak:
  - Holtpont (deadlock)

# Precedencia

- Meghatározott sorrend biztosítása.
- Egy $P_k$ folyamat $S_k$ utasítása csak akkor mehet végbe ha a $P_i$ folyamat $S_i$ utasítása már befejeződött.
- pl:

  szakács-kávét főz ← kukta-cukrot vesz

  (különben kihűl a kávé)

# Egyidejűség

- Két vagy több folyamat bizonyos utasításait ($S_k$; $S_j$) egyszerre kell elkezdeni.
- Két folyamat találkozása (randevú).
- Két folyamat bevárja egymást mielőtt további működését elkezdené.
- pl.

  szakács-kávét főz || kukta-habot ver

  (különben kihűl a kávé vagy összeesik a hab)

# Kölcsönös kizárás (mutual exclusion)

- A résztvevő folyamatok utasításainak sorrendjére nincs korlátozás, de egy időben csak egyik futhat.
- pl.

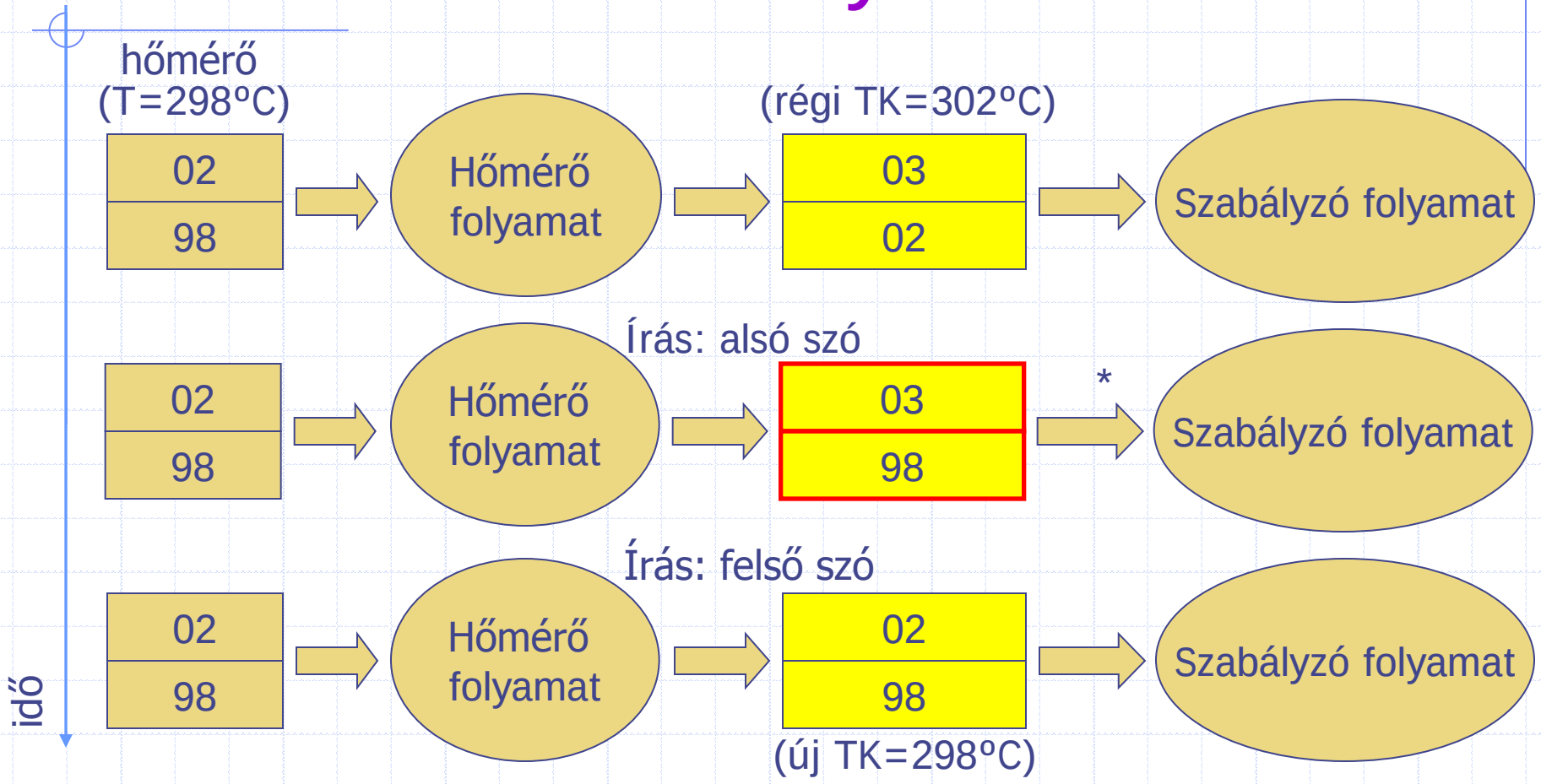
  szakács-habot ver X kukta-habverőt mosogat

# Versenyhelyzet

- Több párhuzamosan futó folyamat közös erőforrást használ. A futás eredménye függ attól, hogy az egyes folyamatok mikor és hogyan futnak, ezáltal hogyan (milyen sorrendben) férnek az erőforráshoz.
- Elkerülendő, nagyon nehéz debuggolni!
- Példa:

  *A mért hőmérséklet (T) értékét egy két szó hosszúságú változóban (TK) tároljuk. A hőmérő folyamat a hőmérsékletet szavanként beírja a változóba, a szabályzó folyamat pedig kiolvassa a változót és annak értékét használja.*

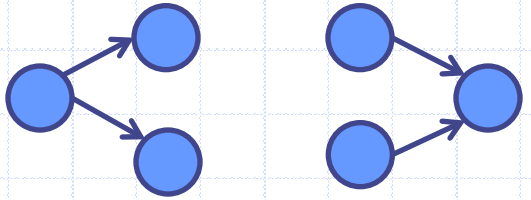
# Példa versenyhelyzetre: hőmérséklet-szabályzó




*Ha ebben a pillanatban olvas, hibás, inkonzisztens értéket kap.

# Párhuzamosság leírása

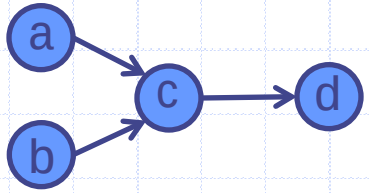
- Precedencia gráf
  - Fork, join struktúrák



- Fork-join szerkezet
  - Ua., csak nyelvi szerkezet írja le
- Parbegin-parend
  - Köztük az utasítások, amik párhuzamosan futhatnak
- Deklarált folyamatok
  - Maguktól indulnak

# Párhuzamosság leírása

- a:= x+y
- b:= z+1
- c:= a-b
- d:= c+1




- R: read set (input)
- W: write set (output)

- Bernstein, 1966

$R(S1) \cap W(S2) = 0$

$W(S1) \cap R(S2) = 0$

$W(S1) \cap W(S2) = 0$

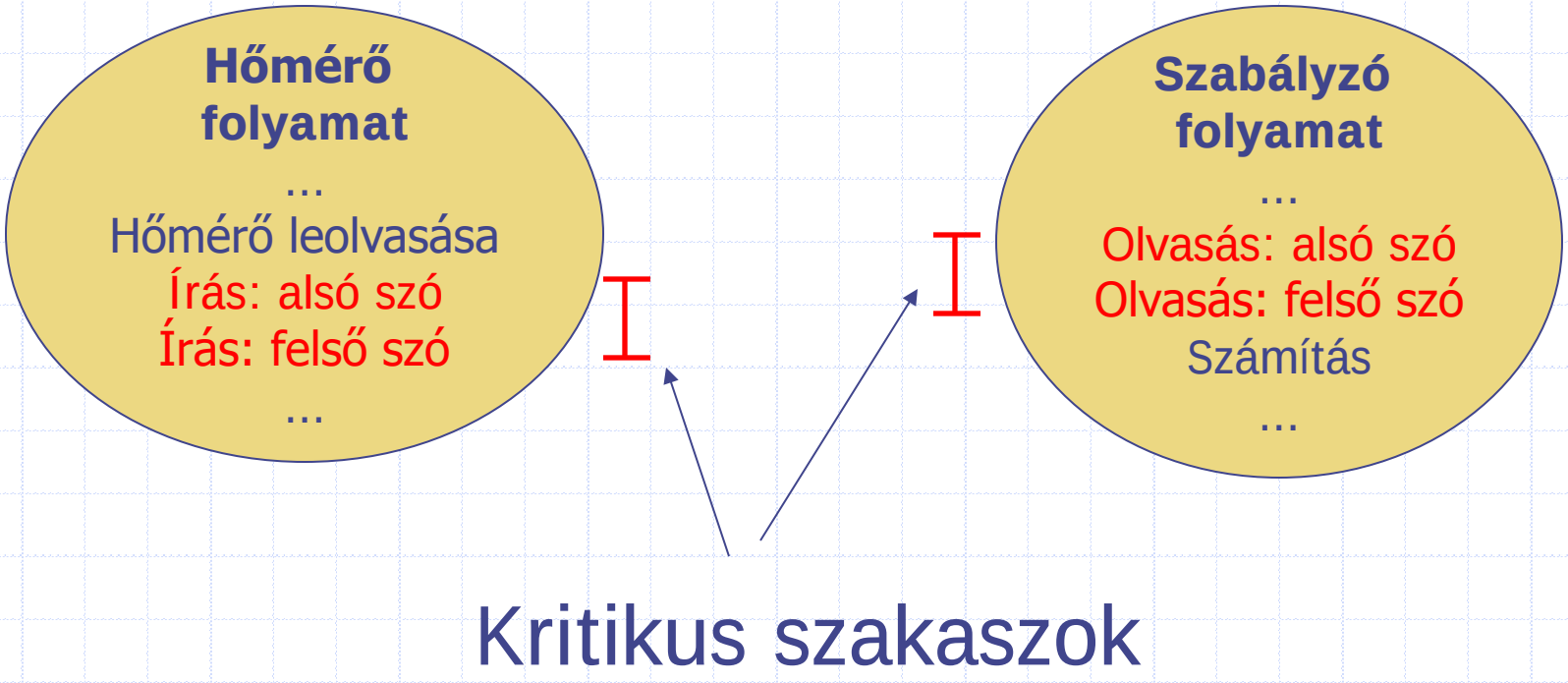
# Kritikus szakasz

- Kritikus szakaszoknak nevezzük a program olyan (általában osztott változókat használó) utasításszekvenciáit, amelyeknek egyidejű (párhuzamos) végrehajtása nem megengedett.
- Versenyhelyzet elkerülésére a kritikus szakaszok kölcsönös kizárását biztosítani kell.

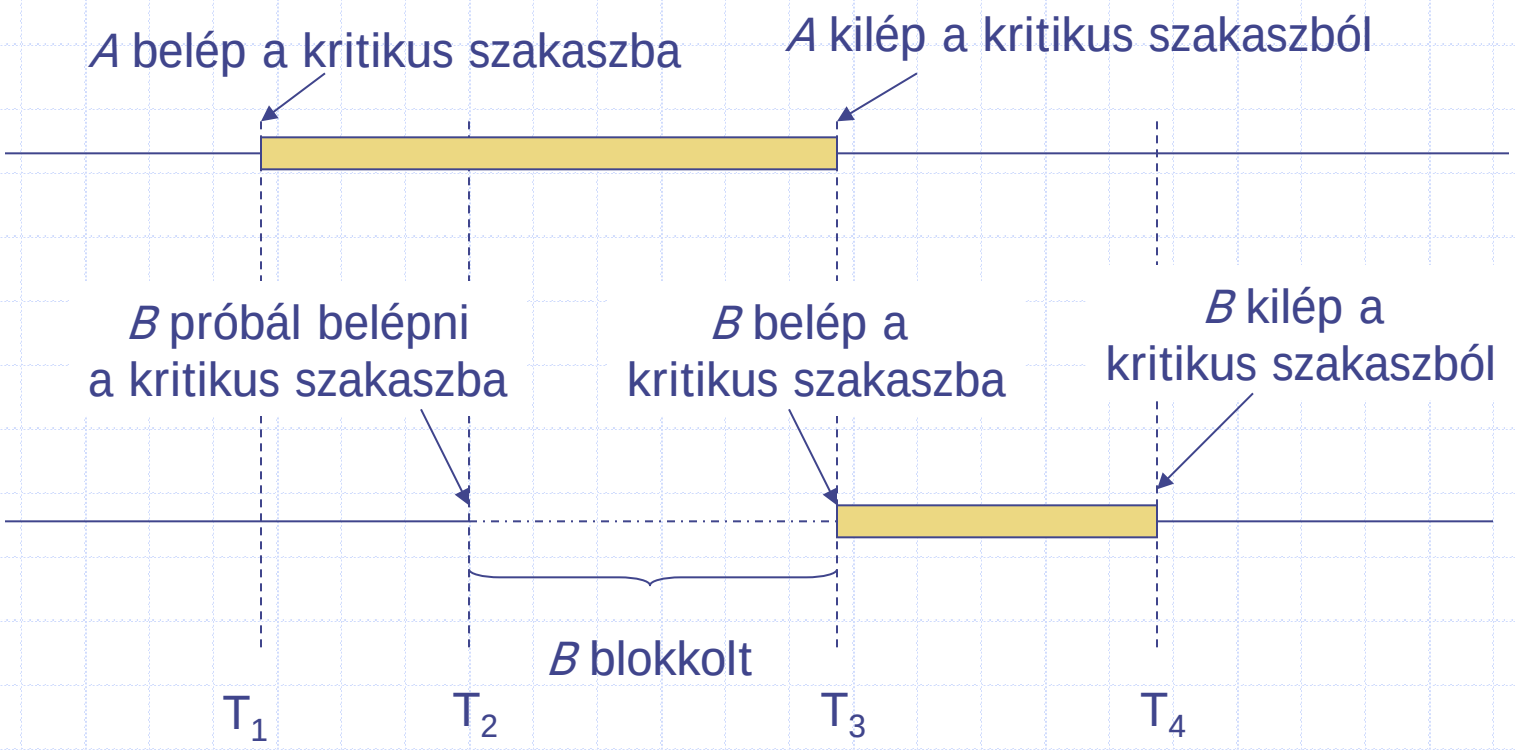
  (Ha az egyik folyamat már a kritikus szakaszában van, akkor más folyamat nem léphet be a (természetesen saját) kritikus szakaszába.)
- Pl.

  Tipikus kritikus szakasz a közös memória használata.

# Példa kritikus szakaszra: hőmérséklet-szabályzó



# Kritikus szakaszok és a kölcsönös kizárás




A kritikus szakaszban lévő *A* folyamat blokkolja *B* végrehajtását, amikor az is a kritikus szakaszba kíván lépni ($T_2$). Amikor *A* kilép a kritikus szakaszból ($T_3$), a blokkolás véget ér és *B* most már beléphet a kritikus szakaszba.

# 3.3 A kritikus szakasz megvalósítási kritériumai

A megvalósítás követelményei

1. Biztosítsa a kölcsönös kizárást
   - Egy időben csak egyetlen folyamat hajthatja végre a kritikus szakaszban lévő utasításokat.
2. Haladjon
   - Ha nincs folyamat a kritikus szakaszban, de van több belépő, akkor az algoritmus ezek közül véges idő alatt kiválaszt egyet és beengedi a kritikus szakaszba.
3. Folyamat várakozása korlátozott legyen (ne éheztessen)
   - Csak véges számú esetben előzhetik meg.

- A 3. kritériumot nem veszik annyira szigorúan (nem minden algoritmus teljesíti).

# Kritikus szakasz megvalósítása

A folyamat általános struktúrája:

```
//Process i
while (TRUE){
    non_critical_region1();
    entry_section();
    critical_region();
    exit_section();
    non_critical_region2();
}
```

# Kritikus szakasz megvalósítási módozatai

- Interrupt tiltása
- *Busy waiting* megoldások
  - Nincs HW támogatás, tiszta SW megoldás.
    - Naiv megközelítés (ROSSZ)
    - Strict alternation (JOBB)
    - Peterson algoritmusa (HELYES)
  - HW támogatással:
    - TestAndSet
- Szemafor
- Magas szintű módszerek

# Interrupt tiltása

- Megoldás:

```
disable_interrupt();
critical_region();
enable_interrupt();
```

- Egyszerű megoldás:
  - Nincs IT, tehát nem lehet a végrehajtást megszakítani. (Ütemező sem tud futni!)
- Probléma:
  - A folyamatnak joga van letiltani az IT-t. Az egész rendszert lefagyaszthatja.
- Hasznos az OS szintjén
- Nem célszerű a felhasználói szinten.

# Naiv programozott megközelítés

- Közösen használt változókon (flag) alapul.
- Egy foglaltsági bitet használ, amit a belépni kívánó folyamat tesztel.
- Gond: a változót többen is olvashatják, mielőtt az első foglaltra állítja!

```
//Process 0
while (TRUE){
    while (flag!=0) /*loop*/;
    flag = 1;
    critical_region();
    flag = 0;
    non_critical_region();
}
```

```
//Process 1
while (TRUE){
    while (flag!=0) /*loop*/;
    flag = 1;
    critical_region();
    flag = 0;
    non_critical_region();
}
```

# Szigorú váltás (*strict alternation*) algoritmus

- Közös változó: *turn*. Jelentése: ki van soron.
- Csak felváltva lehet belépni.
- Gond:
  - Ha jelentős a sebességkülönbség, akkor az egyik feleslegesen sokat vár.
  - Nem halad!


```
//Process 0
while (TRUE){
    while (turn!=0) /*loop*/;
    critical_region();
    turn = 1;
    non_critical_region();
}
```

```
//Process 1
while (TRUE){
    while (turn!=1) /*loop*/;
    critical_region();
    turn = 0;
    non_critical_region();
}
```

# Peterson algoritmusa 1.

```
define FALSE 0
define TRUE 1
define N 2                              /* a folyamatok száma */
int turn;                               /* ki van soron? */
int interested[N];                      /* kezdetben csupa 0 */
void enter_region(int process);         /* a folyamat belép a kritikus szakaszba*/
{
    int other;
    other = 1 – process;                /* a másik folyamat */
    interested[process] = TRUE;         /* az érdeklődés jelzése */
    turn = process;                     /* flag beállítása */
    while (turn == process && interested[other] == TRUE) /* nop */;
}
void leave_region(int process);         /* a folyamat elhagyja a kritikus szakaszt */
{
    interested[process] = FALSE;
}
```

# Peterson algoritmusa 2.

- Ha csak egy folyamat akar belépni, az `enter_region()` visszatér és a folyamat a kritikus szakaszba léphet.
- A másik folyamat addig nem léphet be, amíg a `leave_region()` az `interested` változót nem törli.
- Ha két folyamat egyszerre akar belépni, a turn változót utoljára állító folyamat várakozik, míg a másik végre nem hajtja a saját `leave_region()` eljárását.
- Megjegyzés: A `turn` változó írásának megszakíthatatlanak kell lenni (ez könnyen teljesíthető).

# Hardver támogatás kritikus szakasz megvalósításához

- Speciális **megszakíthatatlan** gépi utasítások (2 memória ciklus) + 1 változó, HW úton megvalósítva

1. TestAndSet
   - Kiolvassa és visszaadja a bit értéket, majd azonnal 1-be állítja.

```
int TestAndSet(int *flag){
   tmp = *flag;
   *flag = 1;
   return(tmp)
}
```

oszthatatlan

2. Swap
   - Két változó értékét cseréli fel

```
void swap(int *a, int *b){
   tmp = *a;
   *a = *b;
   *b = tmp;
}
```

oszthatatlan

# Kritikus szakasz TestAndSet utasítással

```
//init
flag = 0;        valamelyik foly. krit. szakaszban van, ha 1
...


//Process i
while (TRUE){
    while (TestAndSet(&flag)!=0) /*empty loop*/;
    critical_region();
    flag = 0;  /*exit critical section*/
    non_critical_region();
}
```

- Nem garantálja az éhezés kiküszöbölését!

# Kritikus szakasz swap utasítással

```
//init
flag = 0;     valamelyik foly. krit. szakaszban van, ha 1
...

//Process i
...
while (TRUE){
    myflag = 1;       a foly. krit. szak.-t akar végrehajtani
    while (myflag!=0) swap(&myflag, &flag);
    critical_region();
    flag = 0;  /*exit critical section*/
    non_critical_region();
}
```

- Nem garantálja az éhezés kiküszöbölését!

# Szemafor

E. W. Dijkstra, 1965

- Speciális adattípus (nemnegatív egészek),
- Az *s* szemafor-változó két oszthatatlan utasítással érhető el:

- **P(s)** - vizsgál/belép művelet
  - Működési elv:

```
while s < 1 do üres utasítás;
s := s - 1
```

- **V(s)** – kilép művelet
  - Működési elv:

```
s := s + 1
```

# Mutex

- A mutex olyan speciális szemafor, amelynek csak két értéke lehet:
  - nyitott (unlocked)
  - zárt (locked)
- Műveletei:
  - mutex_lock
  - mutex_unlock

# Szinkronizáció szemaforral

- Előidejűség ($U_1$ előbb mint $U_2$)

```
s := 0
P1 : ... U1; V(s); ...
P2 : ... P(s); U2; ...
```

- Randevú ($U_1$ és $U_2$ egyszerre)

```
s1 := 0; s2 := 0
P1 : ... V(s1); P(s2); U1; ...
P2 : ... V(s2); P(s1); U2; ...
```

- Kölcsönös kizárás

```
s := 1
P1 : ... P(s); U1; V(s); ...
P2 : ... P(s); U2; V(s); ...
```

  - Az *s* kezdő értékétől függ, hogy hány folyamat lehet egyszerre a kritikus szakaszban. Legtöbbször bináris szemafort (mutex) használunk.

# Szemaforok megvalósítása

- Probléma a szemafor elvi megvalósításával: busy waiting

```
P(s):
while s < 1 do üres utasítás;
s := s – 1
```

  - Ez blokkolt állapotban is állandóan futna, állandóan használná a CPU-t. Így nem használható.
- Multiprogramozott megoldás:
  - A *P*(*s*) az üres utasítás helyén a folyamattól elveszi a CPU-t és várakozó állapotba teszi (feljegyzi a szemaforhoz tartozó valamilyen adatszerkezetekbe).
  - A *V* (*s*)-nél nem csak a számláló nő, hanem egy folyamatot (ha van) futásra kész állapotba tesz (többféle stratégia lehet).

# Szemaforok megvalósítása

```
type semaphore  = record
                    value: integer
                    list : list of process
                  end;
P(s):
   s:value := s:value - 1
   if s:value < 0 then
   begin
        a folyamat felfűzése a s.list-re
        a folyamat felfüggesztése, újra ütemezés
   end

V(s):
   s:value := s:value + 1
   if s:value <= 0 then
   begin
        leveszünk egy folyamatot s:list -ről
        felébresztjük ezt a folyamatot
   end
```

# (Feltételes) kritikus régió

- A szemafor hasznos szinkronizációs eszköz, de alacsony szintű.
  - Sok hibalehetőség, ráadásul nehéz a hibakeresés.
- Kritikus régió: Az erőforrás és a rajta műveletet végző, kritikus  szakaszhoz tartozó utasítások magas szintű nyelvi szerkezetben
- feltétel nélküli
- feltételes

# Feltétel nélküli kritikus régió

```
type V  = shared region

   region V do S
```

V meg van osztva több folyamat között

V csak a régión belül érhető el!
ha szabad bejut, végrehajtja S-t

# Feltételes kritikus régió

```
type V  = shared region
```

V meg van osztva több folyamat között

```
region V when B do S
```

V csak a régión belül érhető el!
B teljesülése feltétel a bejutáshoz
ha szabad és bejut, végrehajtja S-t

Problémák:

- Az azonos erőforráshoz tartozó kritikus szakaszok a program szövegben szétszórtan fordulnak elő
- Az erőforrásra várakozó összes folyamatot fel kell ébreszteni B kiértékeléséhez
- Egy kritikus szakasz közben megváltozhat B értéke, így mindig újra ki kell értékelni
- Sok felesleges idő

# Monitor

- További nyelvi elemekkel való támogatás: *monitor*, egy magas szintű szinkronizációs primitív.
- A monitor eljárások, változók, adatszerkezetek speciális gyűjteménye.
- A monitor eljárásai szabadon hívhatóak, de a változókhoz nem lehet kívülről közvetlenül hozzáférni.
- Egyszerre csak egy eljárás lehet aktív a monitoron belül!

# Monitor példa

- Egyszerű monitor példa:
- Van *n* eljárás (pl. *producer* és *consumer*) és több belső változó (pl. *i* és *c*)
- Az *i* és *c* belső változókat csak a belső eljárások kezelhetik
- Legfeljebb egy eljárás lehet aktív a monitoron belül

```
monitor example
        integer i;
        condition c;

        procedure producer();


        end;

        procedure consumer();

        end
end monitor;
```

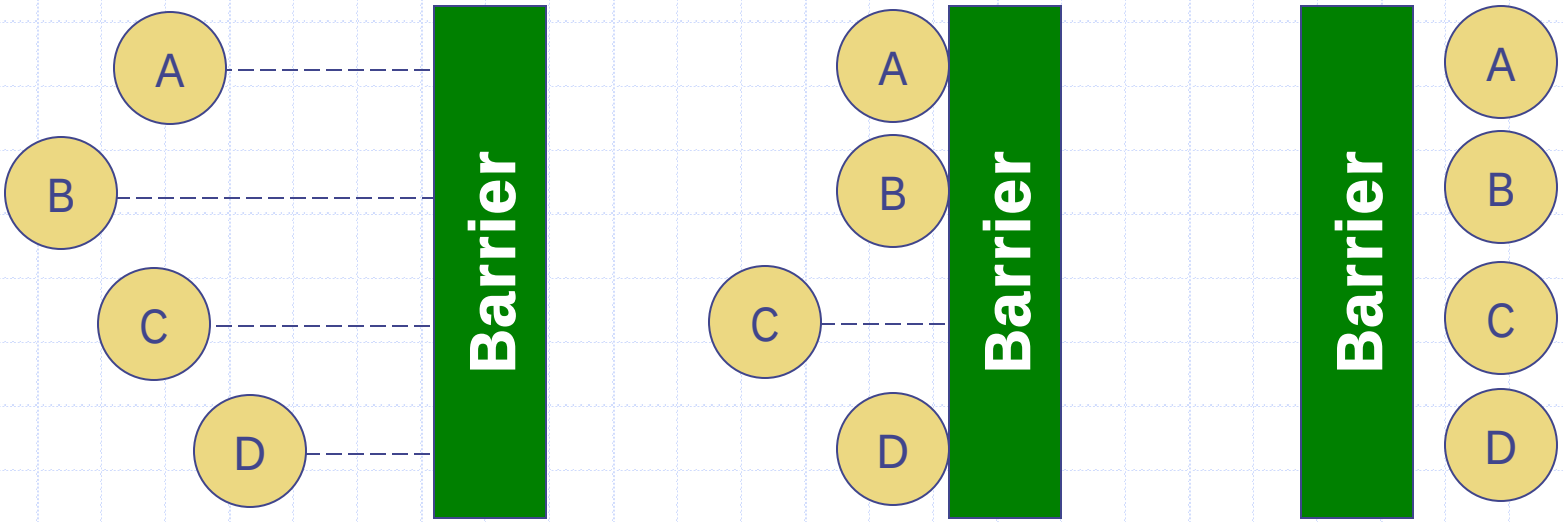
# Monitor

- Önmagában nem elég hatékony eszköz bizonyos problémák helyes modellezésére
- + szinkronizációs mechanizmus kell
- Condition típusú változó (pl. *x*)
- 2 utasítás: *wait*, *signal*
- *x.wait* $\rightarrow$ fel van függesztve, amíg nem jön egy signal
- *x.signal* $\rightarrow$ egy felfüggesztett folyamatot felébreszt, ha nincs ilyen, akkor nincs hatása (ellentétben a szemaforral)

# Monitor + condition

- Milyen sorrendben legyen a végrehajtás, ha jön egy signal?
- Q-t (aki várt a *signal*ra) engedje be és P-t (aki kiadta) tegye várakozóvá, mert ha Q továbbra is várna, elromolhatna a feltétel
- P fusson tovább és Q várjon, mert P már bent van
- Kompromisszumos megoldás: A *signal* legyen P utolsó utasítása a kritikus eljáráson belül, így Q futhat

# Az akadály (barrier)




a) Több folyamat (A, B, C, D) szinkronizálására szolgál
b) Az akadályt elérő folyamatok blokkolódnak
c) Az összes folyamat megérkezésekor az akadály ledől, az összes folyamat egyszerre folytatja futását.

# Szinkronizációs feladat osztályok

- Osztályon belül bármely feladat megoldására visszavezethető az összes többi csoportbeli feladat megoldása (~ algoritmuselmélet)
- Termelő-fogyasztó probléma (véges puffer)
- Ebédelő filozófusok problémája
- Irók-olvasók problémája

# Termelő-fogyasztó probléma (korlátos puffer)

- Termelő → puffer → fogyasztó
- Puffer mérete: n rekesz
- Ha üres a puffer: a fogyasztó vár
- Ha tele a puffer: a termelő vár
- + kölcsönös kizárás (versengő helyzet)
- Globális változók: in: az első üres helyre mutat, out: az első tele helyre mutat, counter: hány adat van a pufferben (redundáns, de kényelmes)
- Lokális változók: nextp: ide teszi be az új megtermelt adatot, nextc: innen veszi a következő feldolgozandó adatot

# Termelő-fogyasztó probléma (korlátos puffer)

Producer (termelő)

```
Repeat
    Produce an item in nextp
    while counter=n do noop
    buffer[in]:=nextp
    in:=in+1 mod n
    counter:=counter+1
Until false
```

Consumer (fogyasztó)

```
Repeat
    while counter=0 do noop
    nextc:=buffer[out]
    out:=out+1 mod n
    counter:=counter-1
    consume the item in nextc
Until false
```

Kölcsönös kizárás?!

# Termelő-fogyasztó probléma (korlátos puffer) - szemafor

Közösen használt változók: Mutex: bináris; empty, full: nem bináris szem.

Init: mutex:=1; empty:=n, full:=0

Producer (termelő)

Repeat

Produce an item

P(mutex)
P(empty)

Enter item

V(full)
V(mutex)

Until false

Consumer (fogyasztó)

Repeat

P(mutex)
P(full)

Take item

V(empty)
V(mutex)

Consume item

Until false

# Termelő-fogyasztó probléma (korlátos puffer) – kritikus régió

A puffert most hívjuk pool-nak

Közösen használt változók: count, in, out

Producer (termelő)

```
Region buffer when count < n
Do begin
   pool[in]:=nextp
   in:=in+1 mod n
   count:=count+1
End
```

Consumer (fogyasztó)

```
Region buffer when count >0
Do begin
   nextc:=pool[out]
   out:=out+1 mod n
   count:=count-1
End
```

# Ebédelő filozófusok problémája

- Dijskra, 1965
- 5 filozófus ül egy asztal körül
- Mindegyik előtt egy tál spagetti
- A szomszédos filozófusok között egy-egy villa
- 2 villával lehet enni
- Stratégia, hogy ne haljanak éhen (innen jön a starvation = kiéheztetés)

# Ebédelő filozófusok problémája

- 2 életfázis: eszik, gondolkozik
- Pl. felvesz egy villát és felveszi a másikat is, vagy vár, míg szabad lesz
- Pl. először felveszi a bal villát, utána megnézi szabad-e a jobb. Ha igen eszik, ha nem leteszi a balt és vár kicsit, majd újra próbálja

# Ebédelő filozófusok problémája

- 3 életfázis: eszik, gondolkozik, éhes
- Pl. mutató megy körbe, akire rámutat, aki megnézheti
- Ha éhes, és egyik szomszédja sem eszik, akkor ő eszik (= felveszi a két villát), egyébként blokkolódik
- Mi kell kölcsönös kizárásban történjék?
  - Villa felvétel
  - Villa letétel
  - Az evés nem!

# Ebédelő filozófusok problémája

Take forks

P(mutex)
State[i]=hungry
Test[i]
V(mutex)

Put forks

P(mutex)
State[i]=thinking
Test[left]
Test[right]
V(mutex)

Test

*If*
state[i]=hungry *and* state[left]≠eating *and* state[right]≠eating
*Then*
state[i]=eating
*Else*
BLOCK i

```
type dining-philosophers = monitor
    var state : array [0..4] of (thinking, hungry, eating);
    var self : array [0..4] of condition;

    procedure entry pickup (i: 0..4);
        begin
            state[i] := hungry;
            test (i);
            if state[i] ≠ eating then self[i].wait;
        end;

    procedure entry putdown (i: 0..4);
        begin
            state[i] := thinking;
            test (i+4 mod 5);
            test (i+1 mod 5);
        end;

    procedure test (k: 0..4);
        begin
            if state[k+4 mod 5] ≠ eating
              and state[k] = hungry
              and state[k+1 mod 5] ≠ eating
              then begin
                       state[k] := eating;
                       self[k].signal;
                   end;
        end;

    begin
        for i := 0 to 4
            do state[i] := thinking;
    end.
```

# Irók-olvasók problémája

- Courtois, 1971
- Adatbázis hozzáférés, pl. helyfoglalás
- Vannak olvasók és írók, párhuzamosan dolgoznak
- Több olvashat egyszerre, de csak egy írhat
- Ha író van benn, ki kell zárni mindenki mást

# Irók-olvasók problémája

- Mutex (init =1): KK, rc (init=0): hány olvasó van bent
- Pl. első olvasó: P(mutex),
- Utolsó olvasó: V(mutex)
- Minden olvasó: belépésnél rc++, kilépésnél rc—
- Író: belépésnél P(mutex), kilépésnél V(mutex)
- Kinek van elsőbbsége? → az íróknál kiéheztetés léphet fel
- És ha az íróknak adunk elsőbbséget?

# 3.4 Információcsere folyamatok között

- Folyamatok együttműködéshez információ cserére (interprocess communication) van szükség.
- Alapvetően két mód:
  - közös tárterületen keresztül (szorosan csatolt) (kölcsönös kizárás témaköre)
  - kommunikációs csatornán keresztül (lazán csatolt)
- Kommunikációs csatorna
  - Két folyamat között valamilyen - fizikai, virtuális (logikai) - csatorna van.
  - Egyik folyamat küld, a másik vesz.

# Információcsere közös tárterületen keresztül

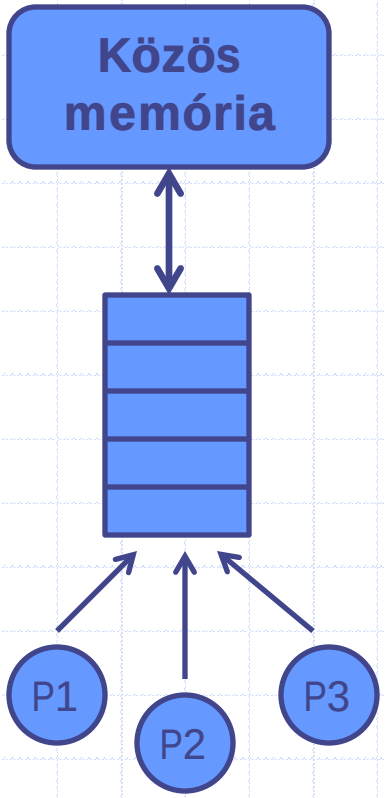
- Minden együttműködő folyamat a saját címtartományában lát egy közös memóriatartományt
- Az elérést (írás, olvasás) valamilyen adatátviteli rendszer teszi lehetővé
- A futás párhuzamos, modell szinten az olvasás, írás oszthatatlan
- Modellezés: a RAM modell kiterjesztése a PRAM (pipelined Random Access Memory)

# Információcsere közös tárterületen keresztül: RAM modell

- A memória tároló rekeszekből áll
- A rekeszek 1 dimenzióban címezhetőek
- Műveletek: írás (felülír), olvasás (nem)
- Írás, olvasás oszthatatlan

# Információcsere közös tárterületen keresztül: PRAM modell

- Kiegészítések:
- Olvasás-olvasás ütközés:
  - Az eredmény ua. (a rekesz tartalma)
- Olvasás-írás ütközés:
  - A rekesz felülíródik, az olvasás eredménye vagy a régi vagy az új tartalom
- Írás-írás ütközés:
  - A rekesz tartalma az egyik írás értékét veszi fel
- Kölcsönös kizárás =>Nincs interferencia, csak a sorrend véletlen
- A helyes működés + szinkronizációt igényel!
  - Kölcsönös kizárás; akkor olvassa, ha már ott van



# Információcsere kommunikációs csatornán keresztül: tulajdonságok

- milyen közeg biztosítja az átvitelt:
  - Pl: párhuzamos vagy soros, lokális hálózat, virtuális esetén közös tár
- kapcsolat egy vagy kétirányú
- kapcsolatot két vagy több folyamat használhatja
  - két folyamat között pont-pont kapcsolat
  - ugyanazt a csatornát  több folyamat is használhatja
    - egy időben, vagy időmultiplexelt rendszerben
- van-e közbülső tárolás (puffer)
- megbízhatóság

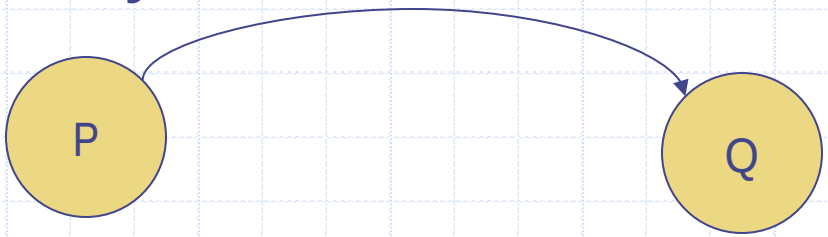
# Folyamatok megnevezése (naming)

- Az a módszer, amellyel a kommunikációban résztvevő folyamatok egymásra hivatkoznak.

- Közvetlen kommunikáció
- Közvetett kommunikáció
- Aszimmetrikus megnevezés
- Üzenetszórás

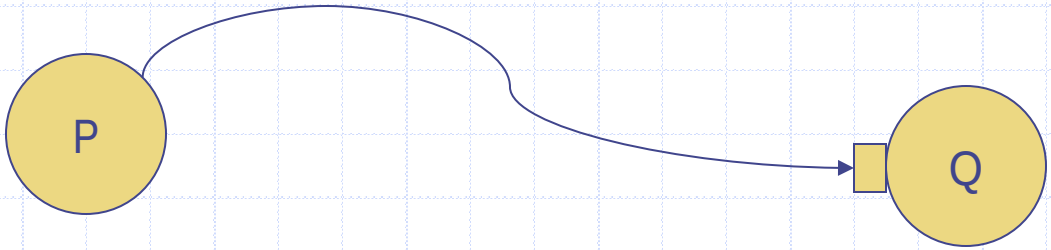
# Közvetlen (direkt) kommunikáció

- Folyamatok: *P*, *Q*
- üzenet küldés: *send*(*Q; message*),
- üzenet fogadás: message=*receive*(*P*)*;*
- csak egy csatorna létezik 2 folyamat között,
- automatikusan létrejön az első *send* vagy *receive* utasítás megjelenésekor,
- más folyamatok nem használhatják,
- lehet egyirányú, de általában kétirányú



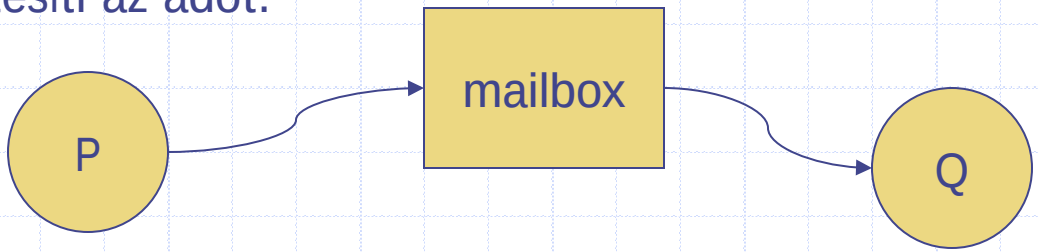
# Aszimmetrikus megnevezés

- Adó vagy vevő megnevezi, hogy melyik folyamattal akar kommunikálni
- A másik fél egy kaput (port) használ, ezen keresztül több folyamathoz, is kapcsolódhat.
- Tipikus eset: a vevőhöz tartozik a kapu, az adóknak kell a vevő folyamatot és annak a kapuját megnevezni.
  - (Pl. szerver, szolgáltató folyamat)
- Küldés: *send*(*Q:port*,*message*);
- Fogadás: message=*receive*();



# Közvetett (indirekt) kommunikáció

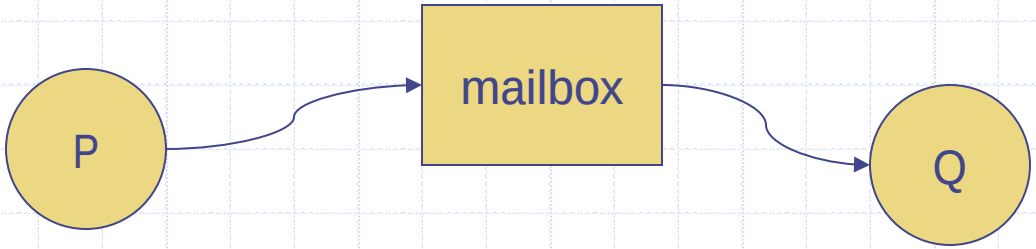
- Közbülső adatszerkezeten (pl. postaládán (mailbox)) keresztül valósul meg.
- Küldés: *send*(*mailbox, message*);
- Fogadás: message=*receive*(*mailbox*);
- Csatorna létrejötte 2 foly. között, ha van közös postaládájuk
- Két folyamat között lehet több postaláda.
- Ugyanazt a postaládát több folyamat is használhatja egyidejűleg.
- A postaláda megosztása a vevők között:
  - csak két folyamat használhatja
  - egy időben csak egy vevő lehet
  - a rendszer választ, hogy melyik vevőnek küldi az üzenetet és erről értesíti az adót.



# Közvetett kommunikáció - mailbox

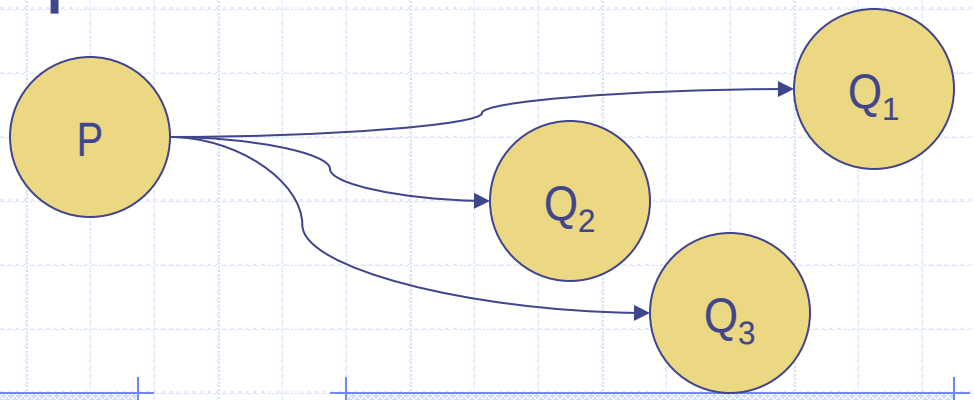
Kié a mailbox?

- **Processzé** – ő hozta létre
- Csak ő (és a gyerekei) kaphatnak benne üzenetet benne
- Owner, user (ő küldhet
- Ha a folyamat meghal a mailbox is eltűnik, a rendszer értesíti a küldőket erről
- **A rendszeré**
- Önállóan él
- OS-nek meg kell szerveznie a működést



# Üzenetszórás (broadcasting)

- A közeg több folyamatot köt össze.
- Az üzenet amelyet több folyamat, esetleg mindegyik veheti.
- Küldés: *send*(*message, to_all*);
- Fogadás: message=*receive*();
- Csoport kommunikáció



# Járulékos (implicit) szinkronizáció

- Folyamatok között információcsere implicit szinkronizációval jár, ennek típusa a puffertől függ.
- Tárolás nélküli átvitel
  - Nincs tárolás, így az adás és vétel egy időben zajlik (randevú).
- Véges kapacitású tároló
  - Egy időben csak az egyik folyamat használhatja (kölcsönös kizárás)
  - Vevő várakozik, amíg legalább egy elem nincs elküldve;
  - adó várakozik, ha a puffer megtelt.
  - A két folyamat utasításai között adott sorrendiségnek kell teljesülni (precedencia).
- "Végtelen" puffer
  - Az adó folyamat nem várakozhat (pl. külső folyamat, amelyre az OS-nek nincs hatása).
  - A valóságban természetesen a puffer véges, így megtelhet.
  - Ilyenkor információvesztés: új adat figyelmen kívül hagyása vagy régi adat felülírása a pufferben.
- Ha van puffer, a küldő nem tudja, hogy a vevő megkapta-e

# Átviteli hibák kezelése

- Elosztott rendszer hibája
  - Folyamat befejeződése: adó vagy vevő folyamat váratlanul befejeződik
- Csatorna hibája
  - Elveszett, torzult üzenetek

# Elosztott rendszer hibája

Adó folyamat áll le
- A vevő hiába várakozik a következő üzenetre.
- Nem holtpont (csak egy folyamatot érint).
- Felismerése: timeout.
- Feloldása: az OS terminálhatja a vevőt vagy jelzi neki az adó megszűnését.

- Vevő folyamat áll le
  - Az adó nulla vagy véges kapacitású puffer esetén várakozni kényszerül, megoldás az előzőekhez hasonló
  - Végtelen puffer esetén az adó az esetleges nyugták megszűnéséből veheti észre a vevő megszűnését.
- Timeout:
  - egy időkorlát túllépése (az az időtartam, amelyen belül valaminek biztosan történnie kell)
  - nehéz megállapítani.

# Csatorna hibája

- Elveszett, torzult üzenetek
- Az OS felismeri és újraküldi
- Az OS felismeri, az adót értesíti, így az újraküldi
- Vevő ismeri fel, az adót értesíti, így az újraküldi
- Torzult üzenetek felismerése egyszerű (HW vagy SW)
  - CRC (Cyclic Redundancy Codes)
  - checksum (pl: összeg mod 256)
  - ECC (Error Correcting Codes).
- Elveszett üzenetek detektálása bonyolultabb eljárás, pl. időkorlátok figyelése.