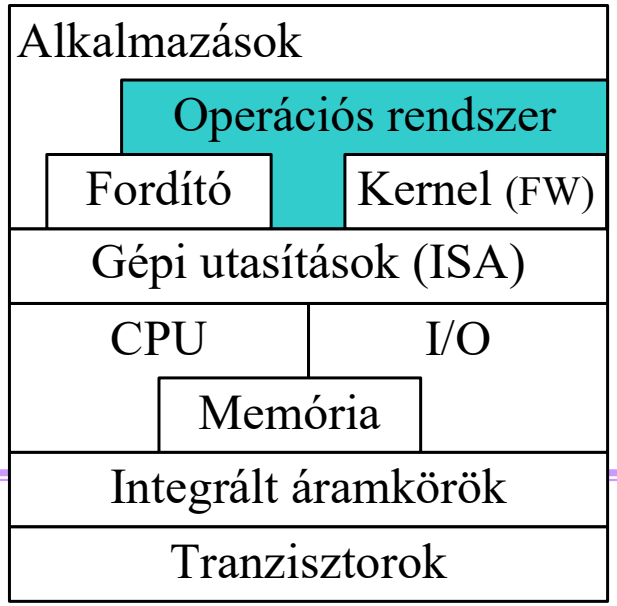
INFORMATIKA I.

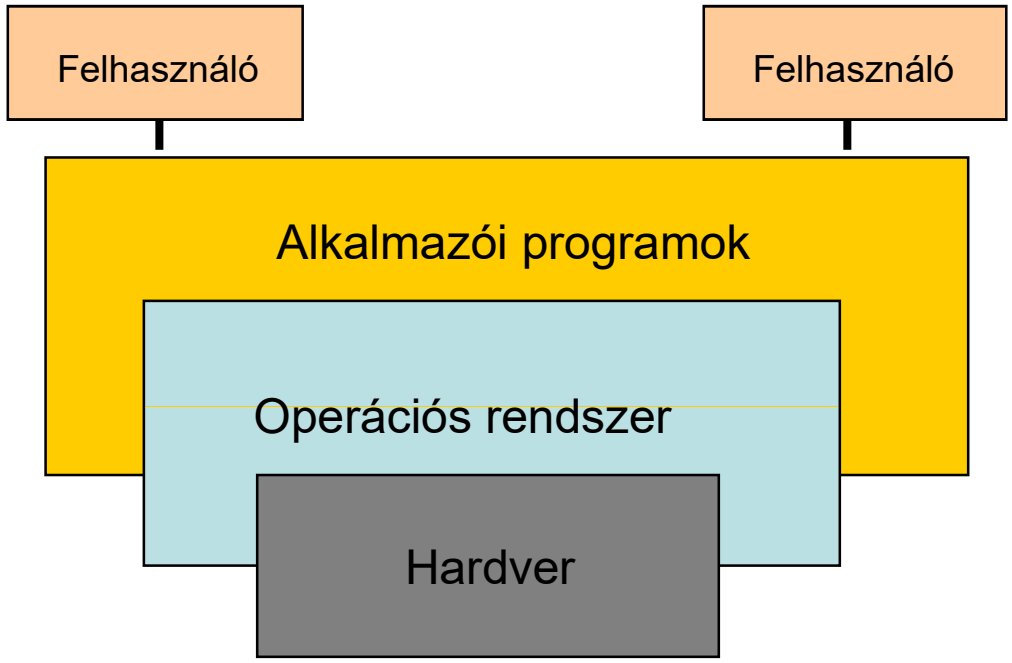
BMEVIIIAB04

Operációs rendszerek



Operációs rendszerek

Számítógép rendszer



Operációs rendszer
kapcsolat a hardver és a felhasználó között

Cél
Hatékony hardver kihasználás
A felhasználó kényelme

Folyamatok

- Egymással párhuzamosan futnak
- Van kezdetük és végük
- Betöltésük az operációs rendszer feladata
- (memória terület hozzárendelése, vezérlés átadása)
- Önmagukban szekvenciális végrehajtásúak
- Véges, **nem nulla** sebességgel hajtódnak végre
- Végrehajtásuk a többi folyamathoz képest aszinkron

  Nem felételezhetünk semmit a többi folyamathoz képest

Operációs rendszerek

Szükség van olyan eszközökre, amelyek a folyamatok egyidejű működését korlátozzák

→ **Szinkronizáció**

Szükség van olyan eszközökre, amelyek a folyamatok között információcserét tesznek lehetővé

→ **Kommunikáció**

Szinkronizáció alapesetei

- Egyidejűség (randevú)
- Precedencia
- Kölcsönös kizárás

Kölcsönös kizárás - két folyamatra

1. **Foglaltság jelző flag a közös memóriában**
2. **Változó, amely kijelöli ki léphet be**
3. **Mindenkinek flag a bent tartózkodásra**
4. **Folyamatonként flag + alternáló kapcsoló (Peterson 1981)**
   2 folyamatra működik, n folyamatra jóval bonyolultabb

Több folyamat - Bakery algoritmus (Lampart 1974)

- Minden folyamat sorszámot kap
  `int counter[N] = { 0, …}`
- A sorszám kérését flaggel védjük
  `int counter_flag[N] = { False, …}`
- Egy folyamat akkor léphet a kritikus szakaszba ha az ő sorszáma a legkisebb

P_i

```
// Entry
  counter_flag[i] = True;                    // védjük a sorszámkérést
  counter[i] = max(counter)+1;
  counter_flag[i] = False;
  for (j = 0; j < n; j++) {
     while (counter_flag[j] == True) …;      // más éppen kap, megvárjuk
     while ((counter[j] != 0) &&
             (counter[j] < counter[i])) …; // van kisebb sorszám, várunk
  }
// Exit
  counter[i] = 0;
```

Nem véd az azonos sorszám ellen – sorszám összehasonlítás folyamat azonosítóval

A szoftver algoritmusok túl bonyolultak
→ hardver támogatás kell

1. test_and_set
2. swap

```
int test_and_set(int *flag)
{
   int tmp;
   tmp = *flag;
   *flag = True;
   return tmp;
}
```

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

Oszthatatlan műveletek

1 processzor : IT tiltás
Több processzor: Bus lock

Kölcsönös kizárás megvalósítása

```
  int flag = False;

// Entry
  while (test_and_set(&flag) == True ) …;

// Exit
  flag = False;
```

```
  int flag = False;

// Entry
  int Myflag = True;
  for (swap(&flag,&Myflag); Myflag == True; swap(&flag,&Myflag)) …;

// Exit
  flag = False;
```

A (3) feltételt nem teljesíti

A (3) feltételt is kielégítő megoldás (Burns – 1978)

```
int varok[N} = { False, … }
int flag = False;
```

```
int j;
int Myflag;

// Entry(i)
varok[i] = True; Myflag = True;
while (varok[i] && Myflag) Myflag = test_and_set(&flag);
varok[i] = False;

// Exit(i)
for(j=(i+1)%N; (j != i) && varok[j]== False; j = (j+1)%N);
if (j == i) flag = False;                    // senki nem vár belépésre
else varok[j] = False;                       // a következőt beengedjük
```

Nem szimpatikus

- A folyamatok hatnak egymásra
- Tudni kell hány folyamat van

Szemafor (Dijkstra – 1965)

Legyen **s** egész változó
Két elemi (oszthatatlan) művelet
- P(s) - Lefoglalás (*Proberen*)
- V(s) - Felszabadítás (*Verhogen*)

P(s)

```
while (s < 1) …;
s = s - 1;
```

V(s)

```
s = s + 1;
```

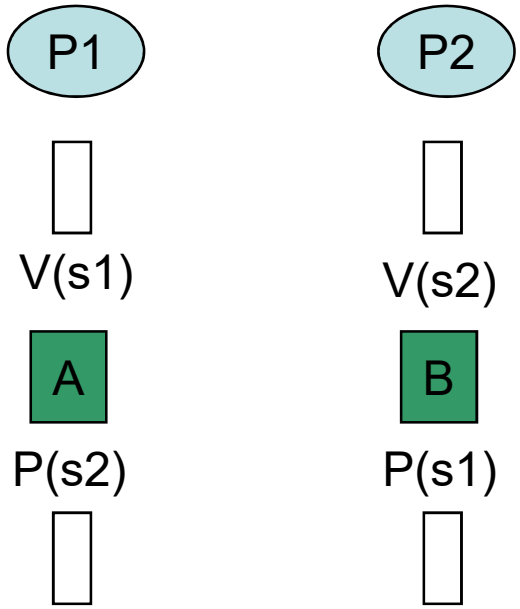
Kezdeti értékadás
Init(s,v)

```
s = v;                  // s=0 - foglalt
```

Szinkronizációs feladatok megoldása szemaforral

Egyidejűség

Init(s1,0); Init(s2,0); // két foglalt szemafor



Szakasz, amire az egyidejűséget biztosítani kell

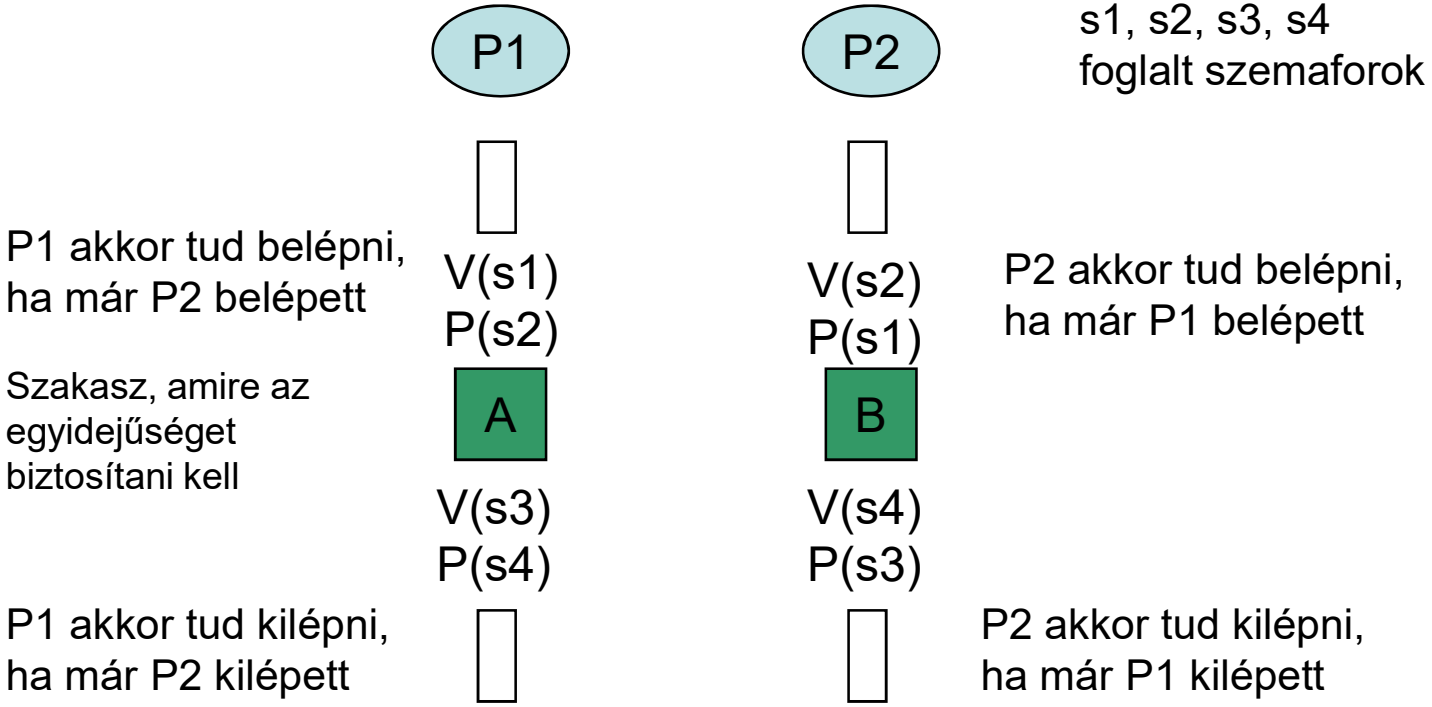
P1 akkor tud kilépni, ha már P2 belépett

P2 akkor tud kilépni, ha már P1 belépett

Szinkronizációs feladatok megoldása szemaforral

Egyidejűség – meghosszabbított randevú

P1 nem léphet be, míg P2 nem lép be és viszont
P1 nem léphet ki, míg P2 nem lép ki és viszont

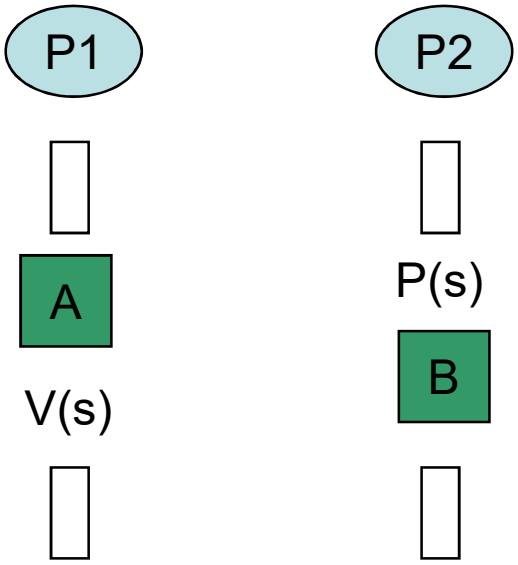


Szinkronizációs feladatok megoldása szemaforral

Precedencia

P1 folyamat **A** művelete meg kell előzze P2 folyamat **B** műveletét

Init(s,0); // foglalt szemafor



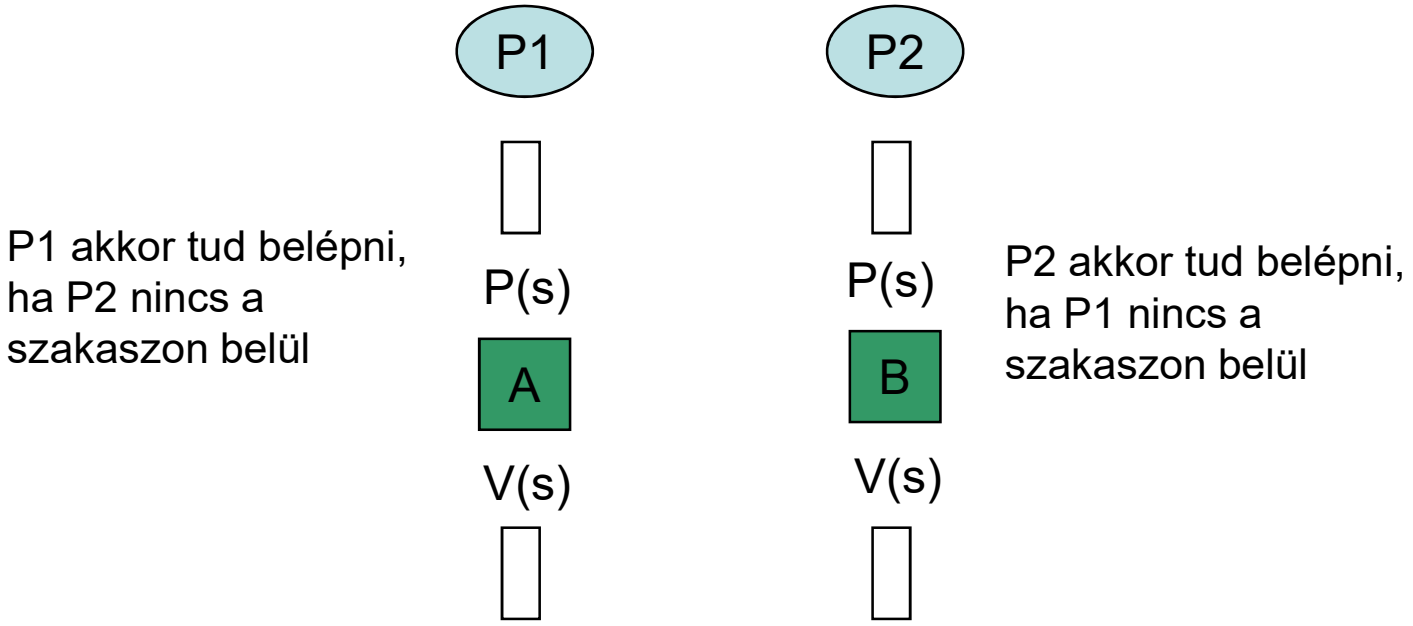
P2 akkor tud belépni, ha már P1 kilépett

Szinkronizációs feladatok megoldása szemaforral

Kölcsönös kizárás

P1 folyamat **A** művelete és P2 folyamat **B** művelete egyszerre nem hajtható végre

Init(s,1); // szabad szemafor

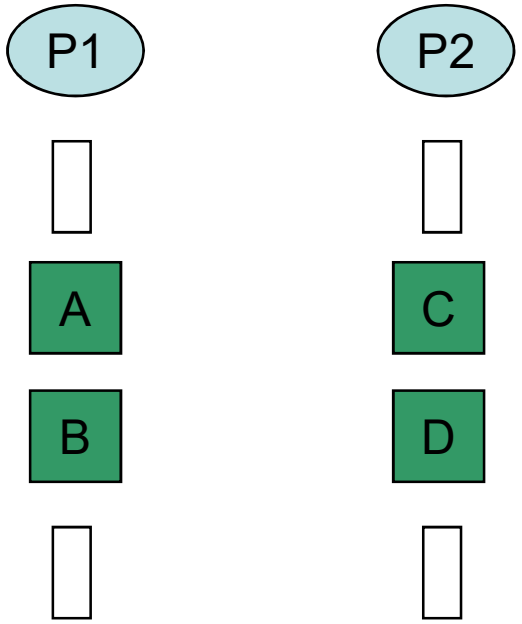


Szinkronizációs feladatok megoldása szemaforral

Szemafor alkalmazásával biztosítsuk hogy a műveletek végrehajtási sorrendje

A – B – C – D

legyen



Szinkronizációs feladatok megoldása szemaforral

Szemafor alkalmazásával biztosítsuk hogy a műveletek végrehajtási sorrendje

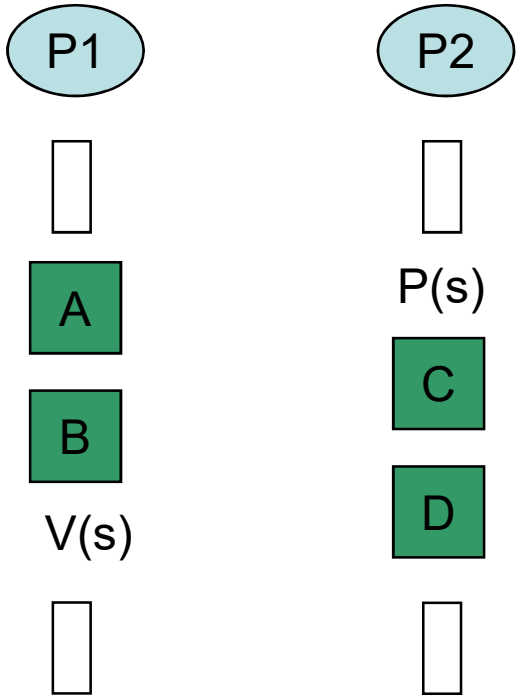
A – B – C – D

legyen

s: foglalt szemafor

P1: A, B, V(s)

P2: P(s), C, D

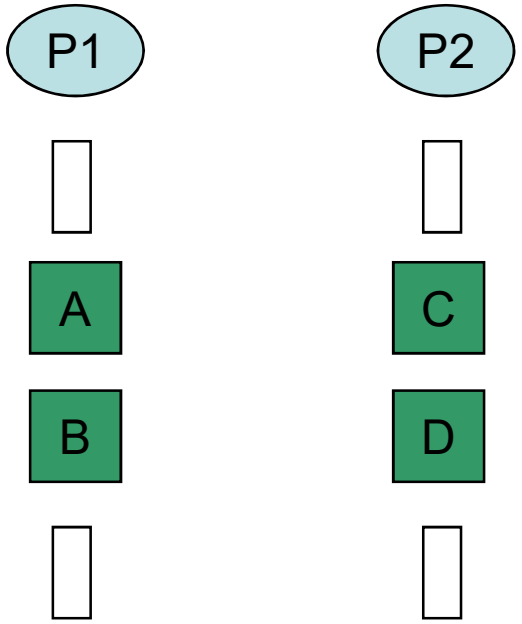


Szinkronizációs feladatok megoldása szemaforral

Szemafor alkalmazásával biztosítsuk hogy a műveletek végrehajtási sorrendje

A – C – D – B

legyen



Szinkronizációs feladatok megoldása szemaforral

Szemafor alkalmazásával biztosítsuk hogy a műveletek végrehajtási sorrendje

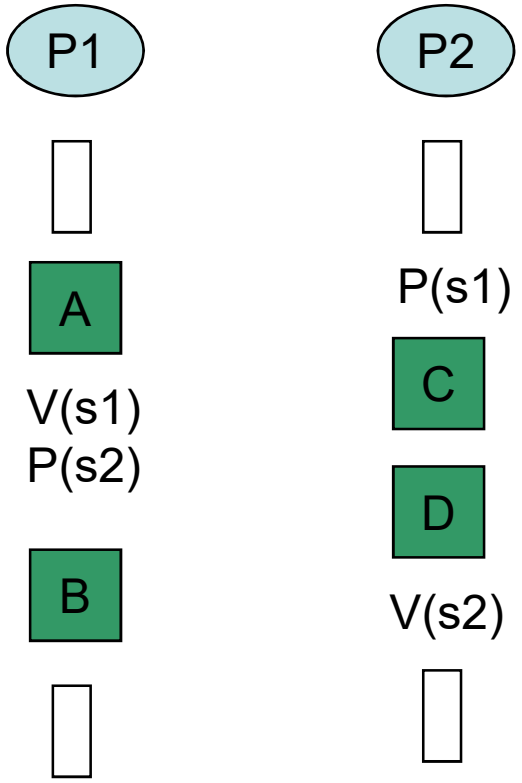
A – C – D – B

legyen

s1, s2: foglalt szemafor

P1: A, V(s1), P(s2), B

P2: P(s1), C, D, V(s2)

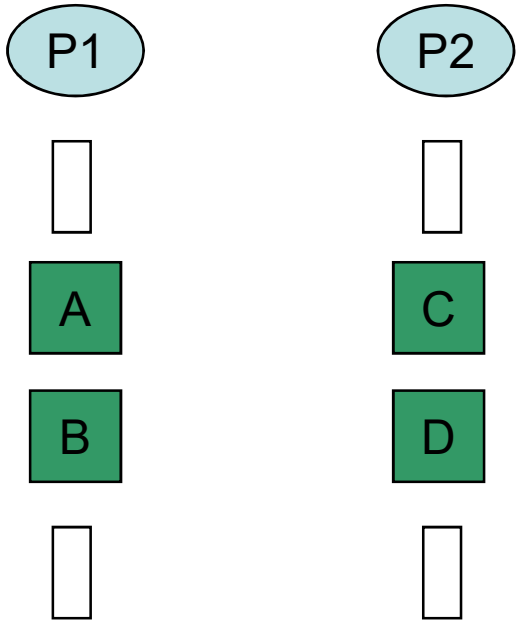


Szinkronizációs feladatok megoldása szemaforral

Szemafor alkalmazásával biztosítsuk hogy a műveletek végrehajtási sorrendje

A – C – B – D

legyen



Szinkronizációs feladatok megoldása szemaforral

Szemafor alkalmazásával biztosítsuk hogy a műveletek végrehajtási sorrendje

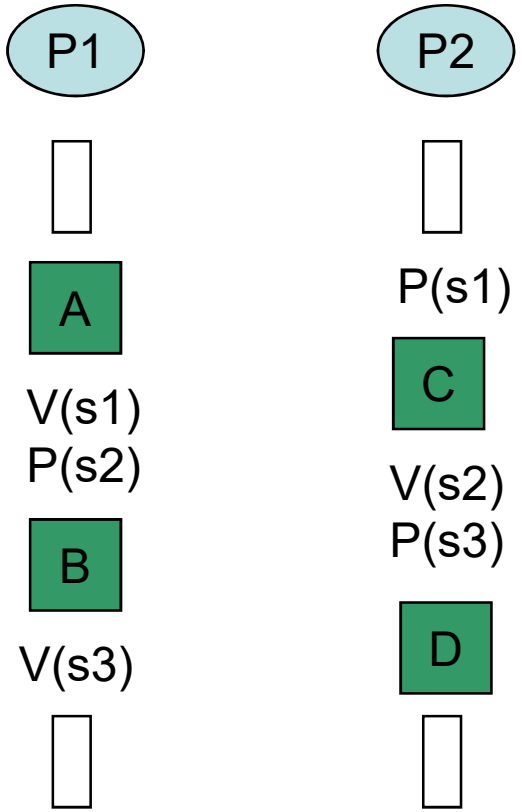
A – C – D – B

legyen

s1, s2, s3: foglalt szemafor

P1: A, V(s1), P(s2), B, V(s3)

P2: P(s1), C, V(s2), P(s3), D



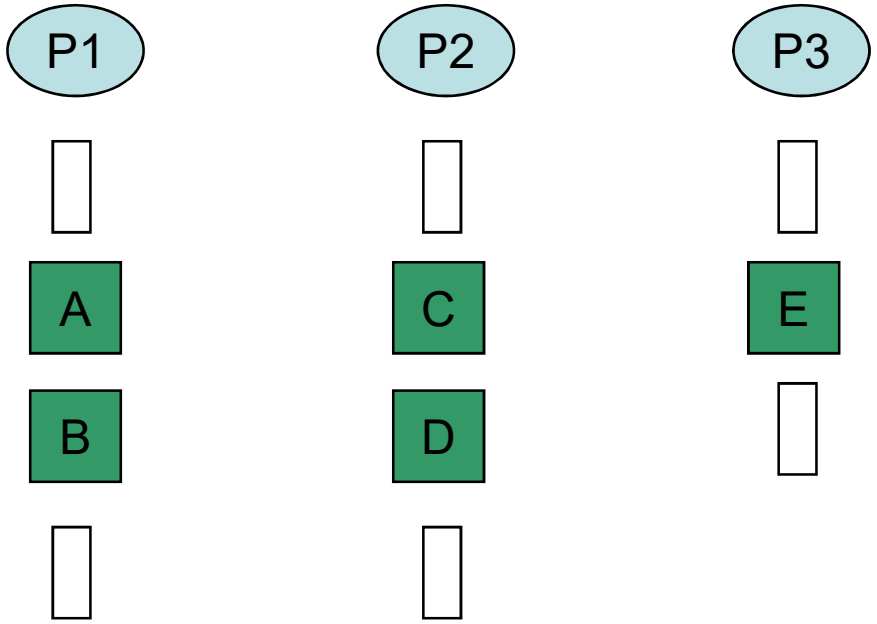
Szinkronizációs feladatok megoldása szemaforral

Szemafor alkalmazásával biztosítsuk

A és C egyidejűségét

valamint

B,D,E kölcsönös kizárását



Szinkronizációs feladatok megoldása szemaforral

Szemafor alkalmazásával biztosítsuk

A és C egyidejűségét

valamint

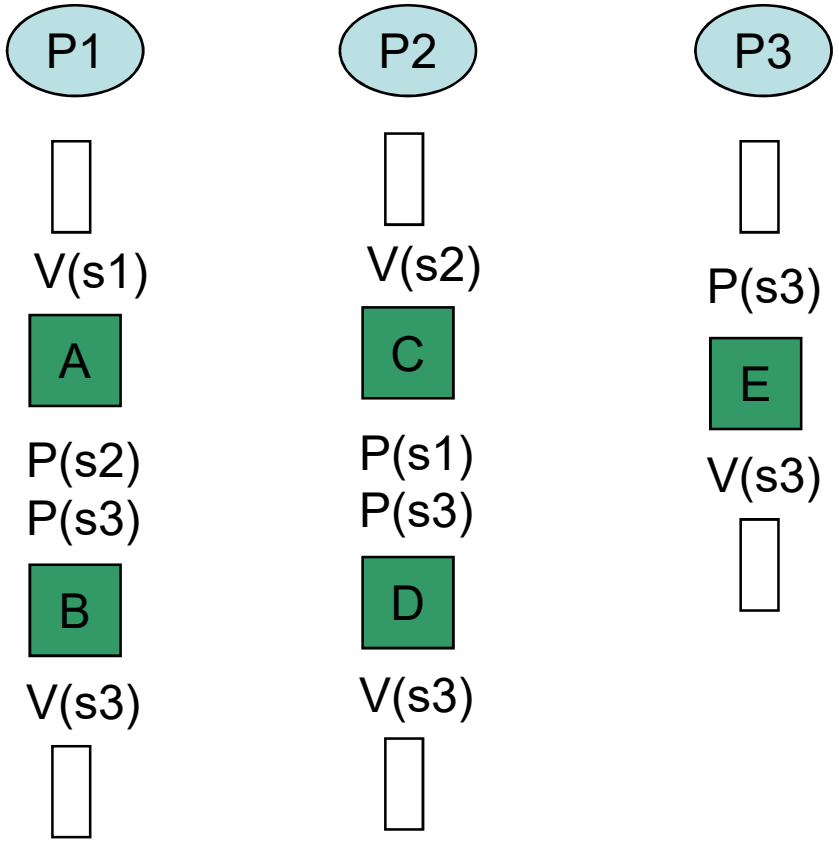
B,D,E kölcsönös kizárását

s1, s2: foglalt szemafor
s3: szabad szemafor

P1: V(s1), A, P(s2), P(s3), B, V(s3)

P2: V(s2), C, P(s1), P(s3), D, V(s3)

P3: P(s3), E, V(s3)



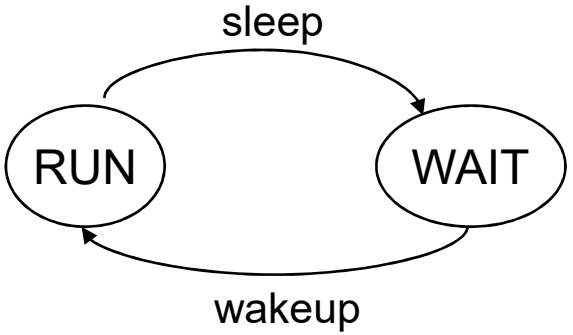
Szemafor multiprogramozott rendszerekben

P(s)

```
while (s < 1) …;
```

Az aktív várakozás nem hatékony

sleep() – a folyamat várakozó állapotba kerül
wakeup(PID) – a folyamat ismét fut



A várakozó processzekhez: listaszerkezet

LISTA implementálása:

- FIFO
- prioritásos
- más megvalósítás

Szemafor multiprogramozott rendszerekben

Szemafor → struktúra

```
struct {
   int value;
   struct LISTA *list;
} SEMAPHORE;
```

```
void felfűz(LISTA *list, PID p);
PID lefűz(LISTA *list);
```

```
void P(SEMAPHORE *s)
{
   s->value--;
   if (s->value < 0) {
      felfűz(s->list, myPID);
      sleep();
   }
}
```

Ha value kezdeti értéke > 1
Több folyamat is beléphet
A kritikus szakaszba

```
void V(SEMAPHORE *s)
{
   s->value++;
   if (s->value < 1) wakeup(lefűz(s->list));
}
```

Szemafor multiprogramozott rendszerekben

LISTA implementálása

FIFO → fair kezelés, előbb-utóbb minden folyamat belép

Prioritásos → bizonytalan ideig tartó várakozás, extrém esetben lehet olyan folyamat, amely soha nem jut szóhoz → **éhezés**

sleep – az operációs rendszer elmenti a folyamat kontextusát

wakeup – kontextus visszaállítás, a felébresztett folyamat szabad szemafort lát

Erőforrás

Korlátozottan hozzáférhető eszköz, használata kölcsönös kizárást igényel

RQST(E) – E erőforrás lefoglalása. Ha foglalt, várakozunk. (request)
RLSE(E) – E erőforrás felszabadítása. (release)

Esemény

Bekövetkezhet, várakozhatunk rá

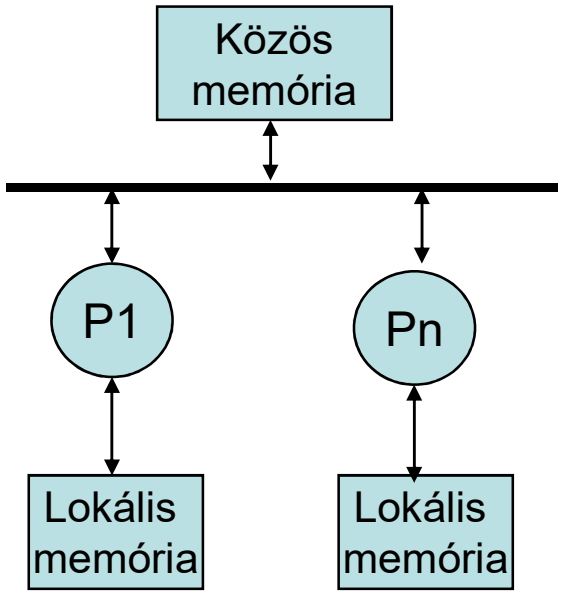
SIGNAL(ES) – ES eseményt generál. Aki várakozik rá „felébred”.
WAIT(ES) – Várakozás ES esemény bekövetkezésére

Egy eseményre többen is várakozhatnak. Bekövetkezésekor mindenki felébred.
Szemafor/Erőforrás: a felszabadítás megőrződik
Esemény: ha nem vár rá senki, elvész

+ szolgáltatások
Várakozás időzítéssel
- Várakozás adott időtartamra
- Várakozás adott időpontig

Operációs rendszerek

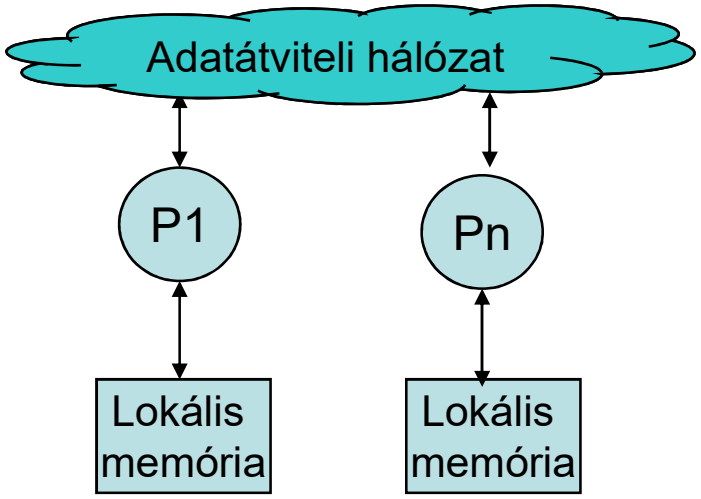
Együttműködés közös memórián



PRAM modell
2 művelet:
read – nem rombolhatja az adatot
write – felülírja az adatot

Szorosan csatolt rendszer

Üzenet alapú együttműködés (message-passing)

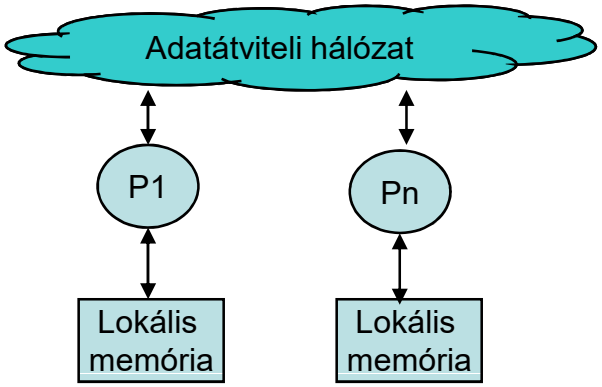


Nincs tiszta modell
2 művelet:
send
receive

Lazán csatolt rendszer

Üzenet alapú együttműködés (message-passing)

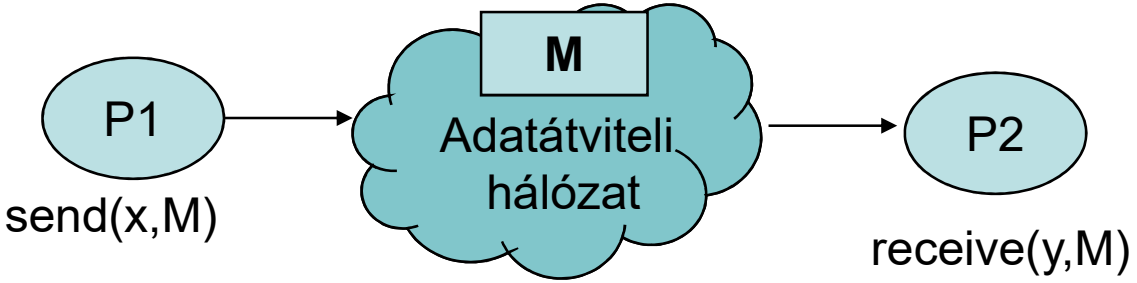
- Hogyan nevezhetjük meg a partnert?
- A partnert látjuk, vagy egy közbülső objektumot?
- Egy partner veheti az üzenetet vagy több is?
- Amikor a küldés befejeződött (visszakaptuk a vezérlést) az üzenet már elért a címzetthez?
- Honnan tudjuk meg, hogy az üzenetünk megérkezett?
- Hogyan működik a receive
  - Mi van, ha még nem küldtek üzenetet?
  - Kell-e visszaigazolás, vagy ez automatikus?



Üzenet alapú együttműködés (message-passing)
Hogyan nevezhetik meg egymást a partnerek?

Direkt megnevezés

Indirekt megnevezés



M
mail box (**postaláda**)
message queue (**üzenetsor**)
pipe (**csővezeték**)

Üzenet alapú együttműködés (message-passing)
Hogyan nevezhetik meg egymást a partnerek?

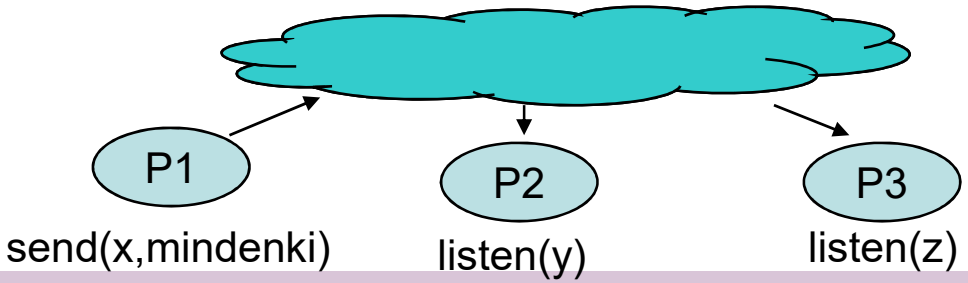
Csatorna

- Egyirányú (simplex)
- Osztottan kétirányú (half duplex) - átkapcsolás
- Kétirányú (duplex)

Átmeneti tároló mérete: végtelen véges, 0

Aszimmetrikus megnevezés (kommunikációs port)

Broadcast



Kommunikációs megoldások

Jelzés (signal)

A küldő jelzéssel(signal) értesíti a fogadót

Hálózati kommunikáció

Kommunikáció kliens-szerver modell szerint

Távoli eljáráshívás (Remote Procedure Call)

Kommunikáció kliens-szerver modell szerint
Egy másik folyamatban lévő eljárás meghívása
Küldő átadja az eljárás nevét, paramétereit és fogadja az eredményt
Fogadó végrehajtja az eljárást és visszaadja az eredményt
Pontosan meghatározható az átvitt adatok típusa

send() szemantikája

Visszatérése mit jelent, várkozhat-e?

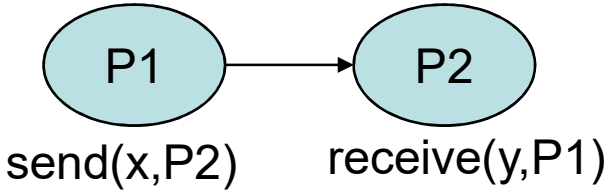
1. send() akkor fejeződik be, ha az üzenet a fogadó oldalra megérkezett
   → send() is várakozhat
2. send() visszatér ha az üzenet bekerült a kommunikációs rendszerbe, de további sorsáról nem tudunk
   → send() soha nem vár

Minimális konzisztencia feltétel: ha elküldtünk egy üzenetet és send() visszatért, a saját terület felülírható, az üzenet nem sérülhet.

receive() szemantikája

- Egy végrehajtás – egy üzenet (még akkor is, ha több üzenet van a pufferben)
- Az üzenetek érkezési sorrendben fogadhatók
- Az üzenet fogadása törli az üzenetet a kommunikációs rendszerből
- Ha a kapuból (port) vagy postaládából nem derül ki a feladó, célszerűen magának az üzenetnek kell ezt tartalmaznia (belső protokoll)

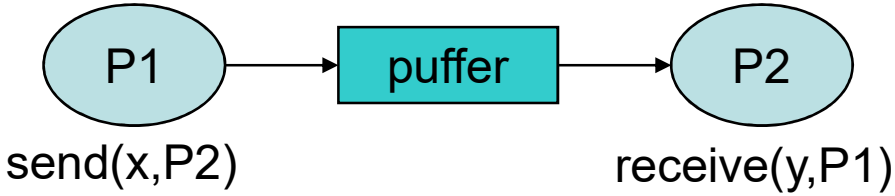
Implicit szinkronizáció



Nincs puffer

Ha P1 van előbb: megvárja míg P2 receive-hez ér
Ha P2 van előbb: megvárja míg P1 send-hez ér
→ **Egyidejűség (randevú)**

Implicit szinkronizáció



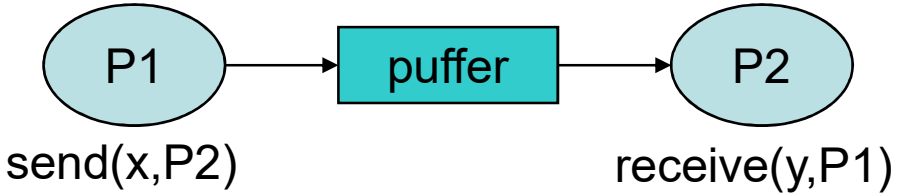
Véges méretű puffer

Végrehajtódhat-e receive() a send() előtt? (És ha igen mi történik)
Jöhet-e két receive() egymás után?
Jöhet-e két send() egymás után?
Mi történik, ha send() és receive() párhuzamosan hajtódik végre
A send() végrehajtása után mit lehet tudni
- P2 megkapta az üzenetet?
- Az üzenet beíródott a pufferbe?

Üres puffer esetén receive() vár
Tele puffer esetén send() vár (vagy felülír?)
send() és receive() kölcsönösen kizárják egymást

→ Hosszú távon **precedencia**
→ Rövid távon **kölcsönös kizárás**

Implicit szinkronizáció



Végtelen méretű puffer

Ha P1 van előbb: nem vár
Ha P2 van előbb: megvárja míg P1 send-hez ér
→ **Precedencia**
(pufferműveletre: kölcsönös kizárás)

Kérdések

- Van-e implicit visszajelzés (nyugta)
- Mi lehet az üzenet típusa
  - Csak egyféle, előre definiált méretű üzenet küldhető
  - Tetszőleges üzenet küldhető
    Honnan tudja a fogadó, hogy mekkora üzenetet kapott
- Mi van, ha pl. a küldő soha nem küld üzenetet (nem fut a folyamat)
    Time-out-os várakozás kell
- Hogyan kezeljük a kommunikációs hibákat
  - Távoli kommunikáció – megbízhatatlan
    Kezelendő hiba: a másik oldal megáll → time-out
  - Üzenet vesztés
    - Az operációs rendszer jelzi és kezeli
    - Az operációs rendszer jelzi, a felhasználó kezeli → hibakód
    - A felhasználó veszi észre és kezeli
  - Üzenet sérülése
    Üzenettovábbítás hibajavító kódolással

Közös memória

- read, write műveletek
- Műveletek pipe-lineban
- Közvetett címzés
- Gyors adatcsere
- Egyszerű használat
- Inicializálás után a hozzáférés nem igényel rendszerhívásokat
- A közös memória mérete korlátoz
- A használók között szinkronizáció szükséges (read – write, write – write)

Üzenetalapú kommunikáció

- Adatátviteli közeg
- send, receive műveletek
- Implicit szinkronizáció
- Szinkron/aszinkron adatátvitel
- Direkt, indirekt, többes címzés
- Távoli gépekkel is használható (hálózat)
- A kommunikációs hibákat kezelni kell