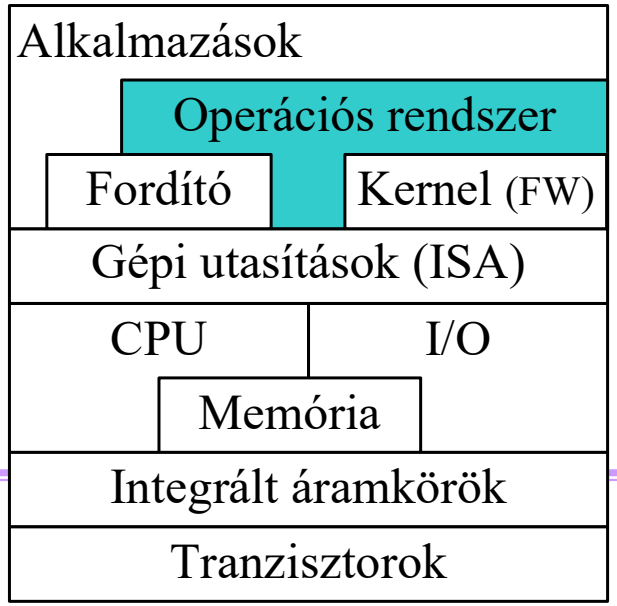
INFORMATIKA I.

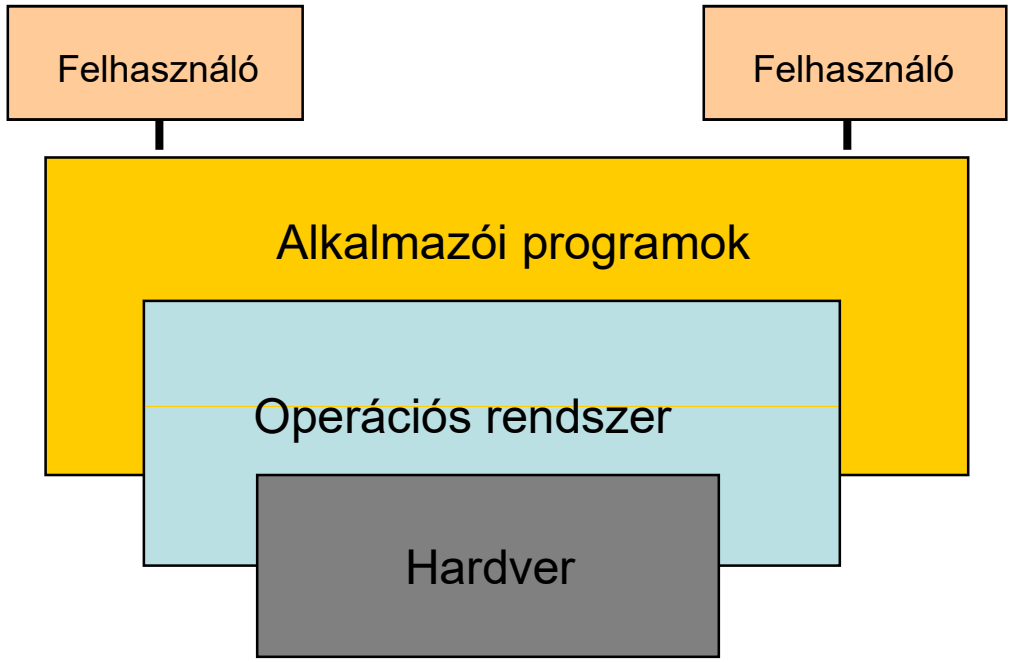
BMEVIIIAB04

Operációs rendszerek



Operációs rendszerek

Számítógép rendszer



Operációs rendszer
kapcsolat a hardver és a felhasználó között

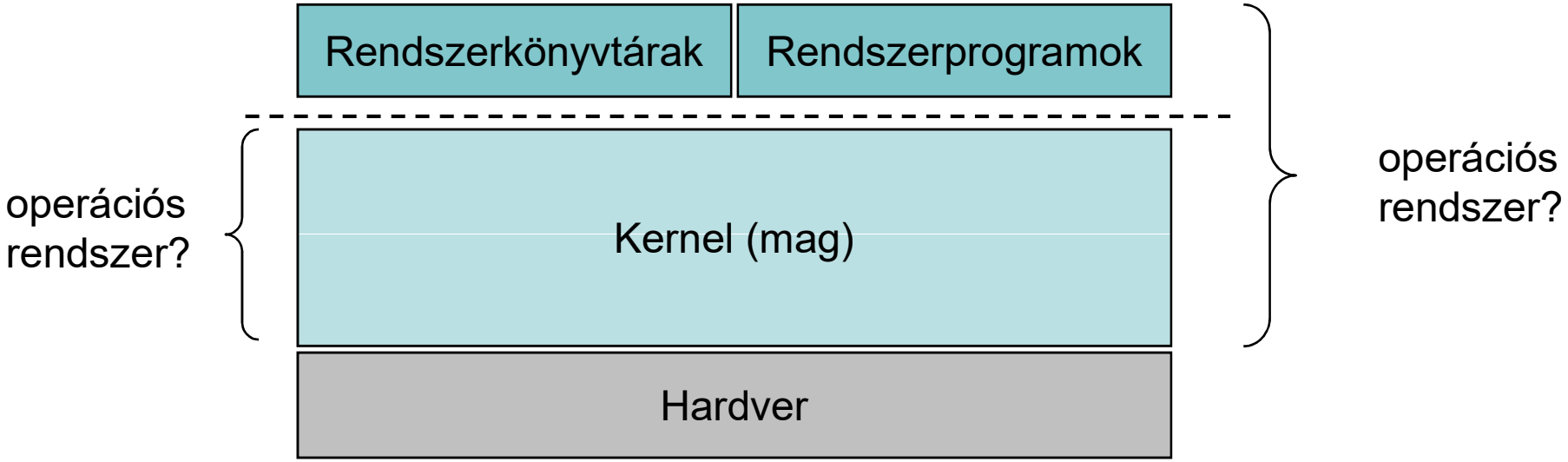
Cél
Hatékony hardver kihasználás
A felhasználó kényelme

Operációs rendszer

Környezet a felhasználó számára
Program, amely a hardver erőforrásait kiosztja
Program, amely vezérli a számítógép működését
Felhasználói programok közös műveleteinek gyűjteménye
…

Programok, amelyek vezérlik a hardvert és lehetővé teszik azon felhasználói alkalmazások végrehajtását

Operációs rendszer



Operációs rendszerek

Korai rendszerek: OPEN-SHOP, CLOSED-SHOP, pufferelés, SPOOLING

Multiprogramozás

BATCH - több munka párhuzamosan

Time-sharing – több felhasználó párhuzamos kiszolgálása (interaktív)

Real-time – több, külső eseményre véges időn belül reagáló feladat

Beágyazott – környezebe ágyazott, feladat specifikus programok

Folyamatok elnevezése

BATCH - JOB

Time-sharing – Processz

Real-time – Task

Folyamat - végrehajtás alatt álló program (utasítás sorozat)

BATCH - JOB

- program sorozat
- JOB lépés: egy JOB következő programjának végrehajtása
- Multiprogramozott környezetben: Ha egy program várakozni kényszerül elindítható egy másik program vagy JOB
- A JOB-ok látszólagosan párhuzamosan futnak

Time-sharing (interaktív) - Process

- Több felhasználó kiszolgálása
- (általában) a felhasználók indítják

Real-time - Task

- Reakció valamilyen külső eseményre
- Az események aszinkron módon következnek be

Folyamatok

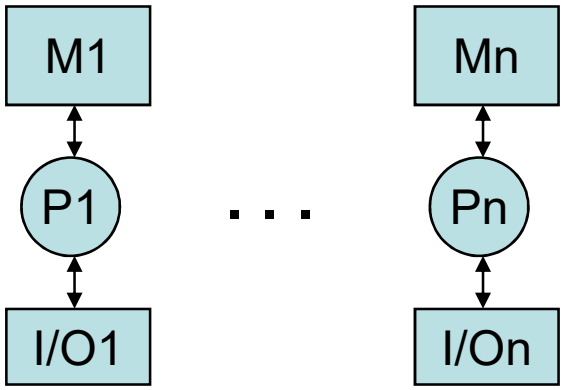
- Egymással párhuzamosan futnak
- Van kezdetük és végük
- Betöltésük az operációs rendszer feladata

  (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

Folyamatok



Van saját

- Processzoruk
- Memóriájuk
- I/O kapcsolatuk

Logikai processzor

Utasítás készlet - egy utasítást oszthatatlannak tekintünk

(a logikai processzor nem tartalmazza a privilegizált utasításokat, de az operációs rendszer szolgáltatásait igen. A be- és kivitelt a logikai processzor utasításkészletébe értjük)

Regiszterek (PC, adat, cím, …)

Memória

RAM-ként viselkedik

Program, adat, stack

I/O kapcsolat

Egységes interfész a logikai eszközökhöz

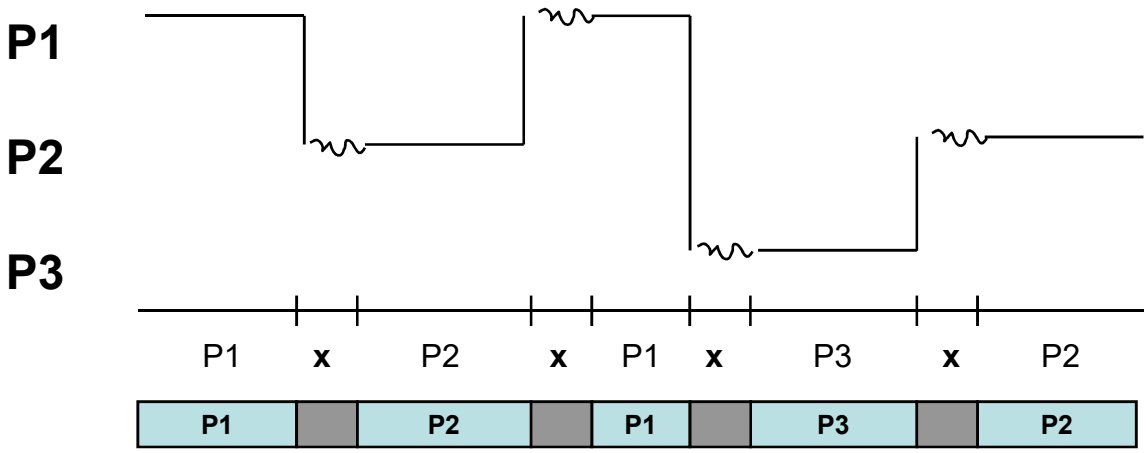
A folyamat állapottere

- A memória tartalma (programkód, változók)
- A processzor állapota (PC, regiszterek, flagek)

Memória – RAM modell

- Tároló-rekeszekből áll
- Egy dimenzióban, rekeszenként címezhető
  (Lineáris címzés)
- Csak rekeszenként érhető el
- Az írás a teljes megcímzett rekeszt felülírja
- Az olvasás nem változtatja meg a rekesz értékét

Folyamat váltás

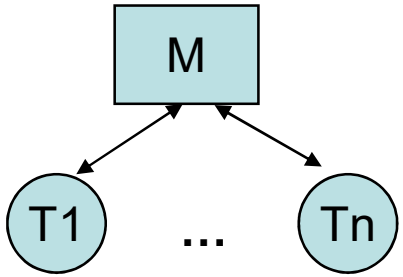


x: átkapcsolás (sajnos nem 0 ideig tart)

Pi processz állapot mentés (processzor állapota, memória)

Pj processz állapot visszaállítás (processzor állapota, memória)

Szál (Thread)



Párhuzamosan végrehajtódó, de közös memóriát használó programok egy folyamaton belül

Saját logikai processzor

Ugyan úgy versenyeznek érte mint a folyamatok

Közös memória

A kód és az adatterület is közös!

→ Gyorsabban lehet váltani a szálak között

Csak a processzor állapotát kell visszaállítani
A védelem megoldása a programozó feladata!

Folyamatok viszonya egymáshoz

Független folyamatok

- Végrehajtásuk aszinkron
- Nem ismerik egymást
- Egymás működését nem befolyásolják
- Egymáshoz viszonyított sebességükről semmit nem mondhatunk

Versengő folyamatok

- Nem ismerik egymást
- Közös erőforrásokon osztoznak
- Nem kell tudniuk, hogy multiprogramozott környezetben futnak
- Helyes és hatékony futtatásuk az operációs rendszer feladata
  (memória kezelés, erőforrás kiosztás)

Folyamatok viszonya egymáshoz

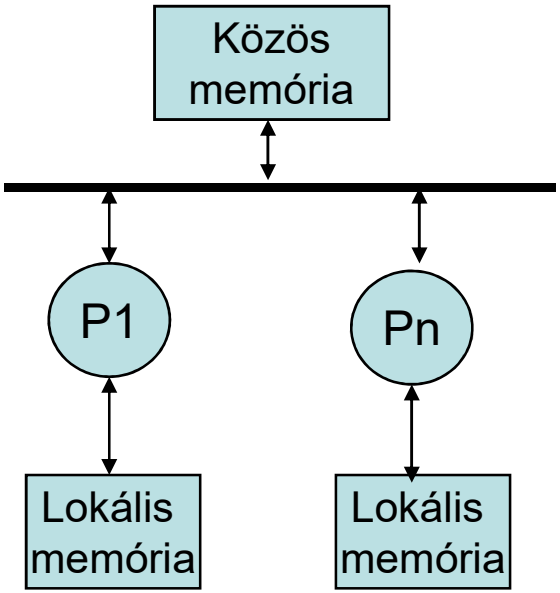
Együttműködő folyamatok

Ismerik egymást
Információt cserélnek
Az együttműködés műveletei a programkódban is megjelennek
(a logikai utasítások között kell, hogy legyen ilyen utasítás – pl. üzenetküldés)

Az együttműködés lehetőségei

- Közös memória használat
- Üzenetküldés

Együttműködés közös memórián



Közös memória: RAM modell elegendő?

Kezelendő működések

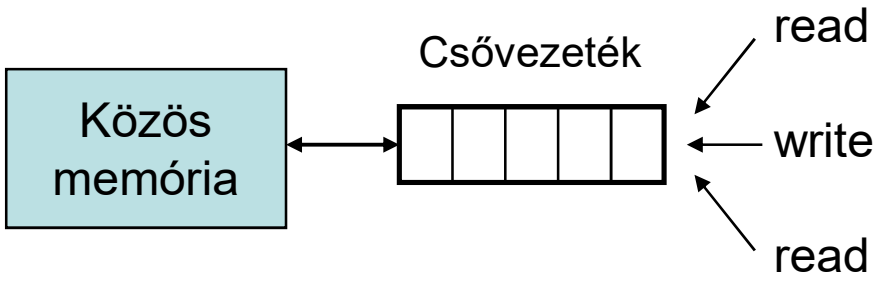
P1	P2	
read	**read**	**- egyszerre olvasnak**
read	**write**	**- egyik olvas, másik ír**
write	**write**	**- egyszerre írnak**

Lokális memória: RAM modell
write – felülírja az eredeti tartalmat
read – visszaadja az aktuális tartalmat
(amit előzőleg beírtunk)
A read művelet nem rombol, ismétlődő művelet mindig ugyan azt az eredményt adja

PRAM modell (Pipelined RAM)

P1	P2	
read	**read**	- P1 és P2 ugyan azt az eredményt kell, hogy kapja
read	**write**	- a memória felülíródik, P1 vagy az előző, vagy az új értéket kapja, mást nem kaphat
write	**write**	- a memória felülíródik, értéke vagy P1 vagy P2 által írt érték lesz, de más nem lehet

Az egyidejű műveletek nem interferálnak



A műveletek érkezési sorrendje nem definiált

Egyszerre csak egy művelet hajtódik végre

Példa

Legyen két folyamat
P1 – beolvas adatot a külvilág felől és elhelyezi egy pufferbe
P2 – kiírja a puffer tartalmát

Kommunikáció a közös memóriaterületen keresztül: puffer a közös memóriában

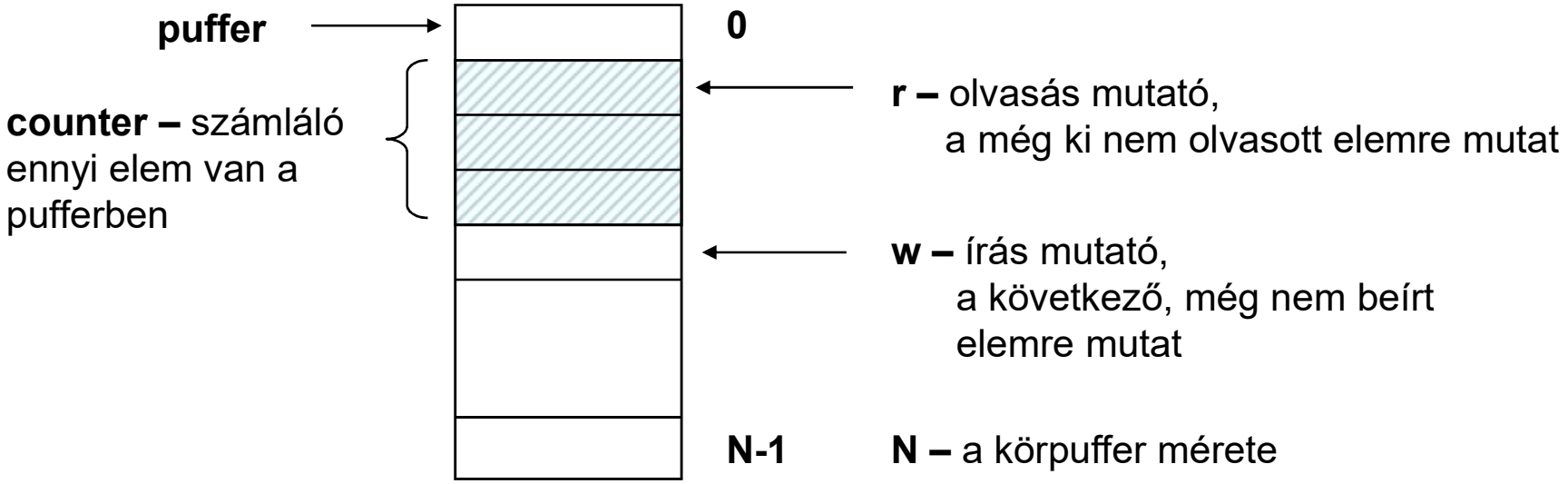
Megvalósítandó funkciók:
P1
`void put_char(char c)` // elhelyez egy karaktert a pufferbe, visszatér az elhelyezés után

P2
`void get_char(char *p)` // kivesz egy karaktert a pufferből

A puffer mérete véges
- P1 csak akkor írhat bele, ha nincs tele
- P2 csak akkor olvashat belőle, ha van benne adat

Megvalósítás körpufferrel

Körpuffer




```
char puffer[N]; // közös memóriában
int counter=0;  // közös memóriában
```

P1: `int w = 0;` // lokális memóriában
P2: `int r = 0;` // lokális memóriában

Mutatók kezelése
lépés a következő elemre
körpuffer
w = (w+1) MOD N

P1

```
void put_char( char c )
{
static int w = 0;
   while (counter == N)…;
   puffer[w] = c;
   w = (w + 1) % N;
   counter++;
}
```

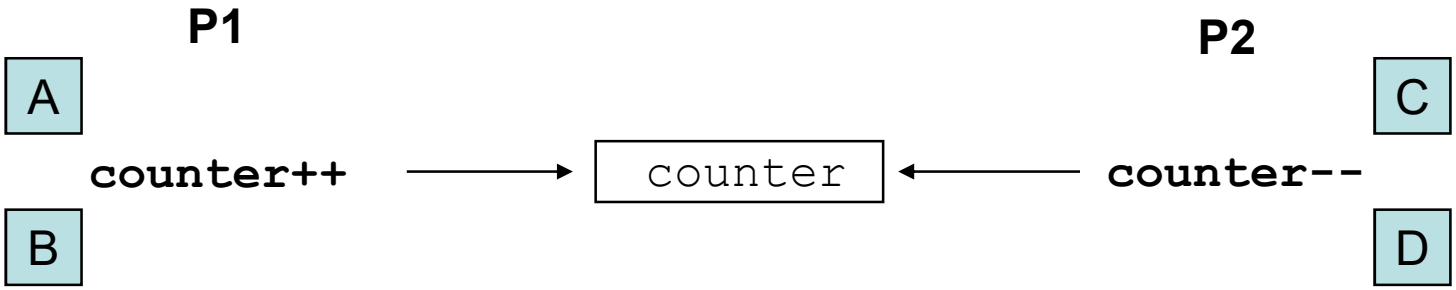
P2

```
void get_char( char *p )
{
static int r = 0;
   while (counter == 0)…;
   *p = puffer[r];
   r = (r + 1) % N;
   counter--;
}
```

Hol lehet gond?

Növelés/csökkentés megvalósítása

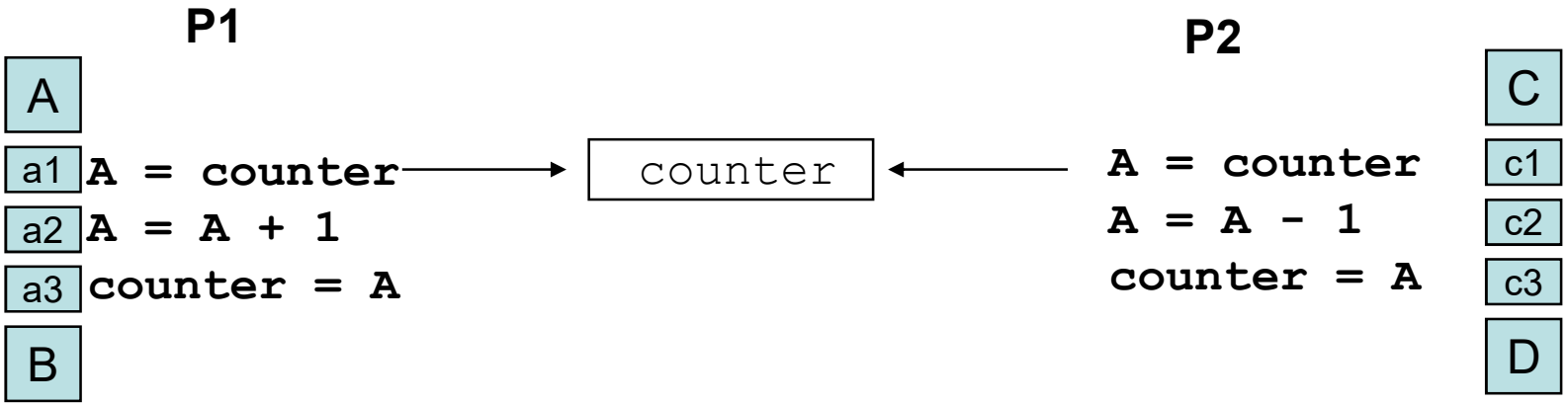
- INR counter / DCR counter – oszthatatlan utasítás
- A = counter; A = A + 1; counter = A; - utasítás sorozat



PRAM modell, **`counter++`** és **`counter--`** oszthatatlan műveletek

Lehetséges végrehajtási sorrendek:

counter	5		6		6		5	
	A		B		C		D	**5**
	5		5		6		5	
	A		C		B		D	**5**
	5		5		4		5	
	A		C		D		B	**5**
	5		4		4		5	
	C		D		A		B	**5**
	5		5		4		5	
	C		A		D		B	**5**
	5		5		6		5	
	C		A		B		D	**5**



Lehetséges végrehajtási sorrendek:

5	—	5	—	6	—	6	—	6	—	5	**5**
a1		a2		a3		c1		c2		c3	
5	—	5	—	5	—	4	—	5	—	6	**6**
a1		c1		c2		c3		a2		a3	
5	—	5	—	5	—	5	—	6	—	4	**4**
a1		c1		a2		c2		a3		c3	
5	—	5	—	4	—	4	—	4	—	5	**5**
c1		c2		c3		a1		a2		a3	

Biztosítani kell, hogy P1 A – B , P2 C – D szakaszra kölcsönösen kizárja egymást

Egy működő megoldás

```
char puffer[N]; // közös területen
int r = 0, w = 0; // közös területen
```

P1

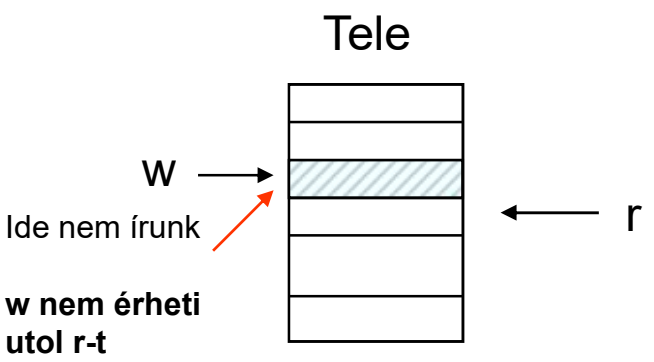
```
void put_char( char c )
{
    while ((w + 1)% N == r)…;
    puffer[w] = c;
    w = (w + 1) % N;
}
```

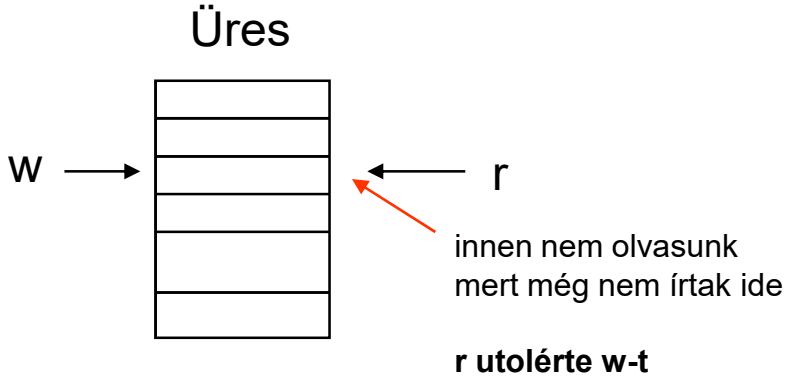
*P1 **w**-t írja, **r**-t olvassa*

P2

```
void get_char( char *p )
{
    while (w == r)…;
    *p = puffer[r];
    r = (r + 1) % N;
}
```

*P2 **r**-t írja, **w**-t olvassa*





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**

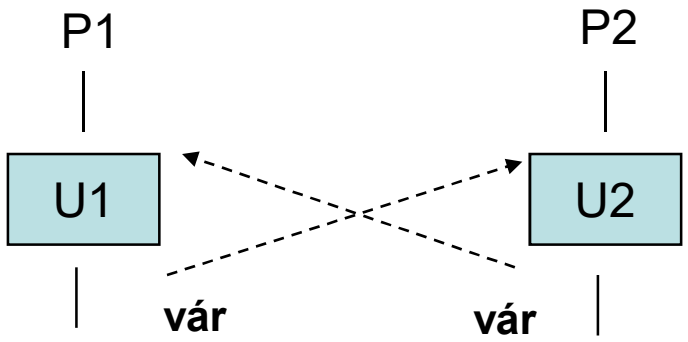
Szinkronizáció alapesetei

- Egyidejűség (randevú)
- Precedencia
- Kölcsönös kizárás

Egyidejűség (randevú)

Két folyamat U1 és U2 utasítása egyidejűleg hajtódhat végre

→ Egyik folyamat sem léphet túl az egyidejű utasításán, amíg a másik el nem kezdte a sajátját

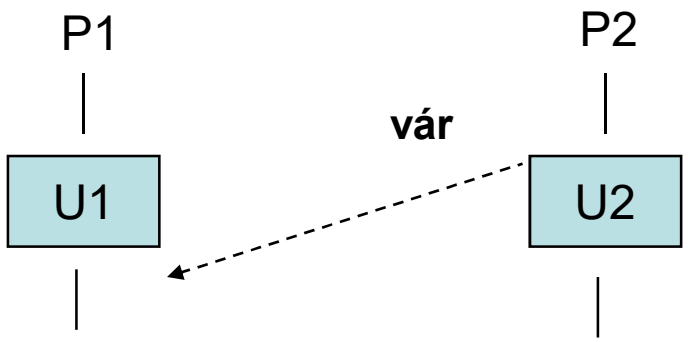


Egy folyamat két utasítása között **nem lehet** egyidejűség
(szekvenciális utasítás-végrehajtás)

Precedencia

P1 folyamat U1 utasítása hamarabb kell, hogy befejeződjön, mint hogy P2 folyamat U2 utasítása elkezdődik

(P2 csak akkor használhatja P1 adatát, ha az már rendelkezésre áll)



Egy folyamat két utasítása között **mindig** precedencia van
(szekvenciális utasítás-végrehajtás)

Kölcsönös kizárás

P1 folyamat U1 utasítása és P2 folyamat U2 utasítása közül egyszerre csak az egyik hajtódhat végre

Egy folyamat két utasítása között **mindig** kölcsönös kizárás van
(szekvenciális utasítás-végrehajtás)

Kölcsönös kizárás

N együttműködő folyamat mindegyikében van egy-egy olyan szakasz, amelyet egyszerre csak az egyik hajthat végre → **kritikus szakasz**

A kritikus szakaszokra **kölcsönös kizárást** kell biztosítani

Ha egy folyamat belépett a kritikus szakaszába, a többi addig nem léphet be (a sajátjába), amíg az ki nem lép

(1) Biztosítsuk a kölcsönös kizárást
(2) (egyidejű kérés esetén) Véges időn belül döntsünk, hogy ki léphet be (csak a versengők közül válasszunk)
(3) Véges időn belül minden belépni szándékozó beléphessen

Feltételezzük, hogy a folyamatok végrehajtása véges (≠ 0) sebességgel folyik
(3) bizonyos rendszerekben nem feltétel

Kölcsönös kizárás - két folyamatra

Belépés - **Entry**

Kilépés – **Exit**

1. Foglaltság jelző flag a közös memóriában

```
int flag = False;
```

P0

// Entry

```
while (flag == True) …;
flag = True;
```

// Exit

```
flag = False;
```

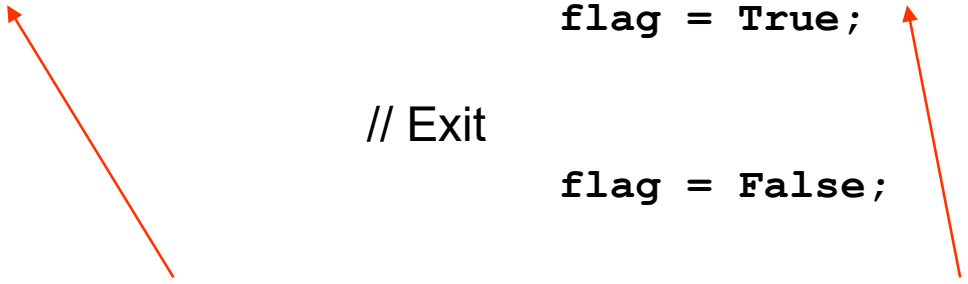
P1

// Entry

```
while (flag == True) …;
flag = True;
```

// Exit

```
flag = False;
```



Nem jó, ha egyszerre olvasnak, mindketten beléphetnek

2. Változó, amely kijelöli ki léphet be

```
int johet = 0;
```

P0

// Entry

```
while (johet != 0) …;
```

// Exit

```
johet = 1;
```

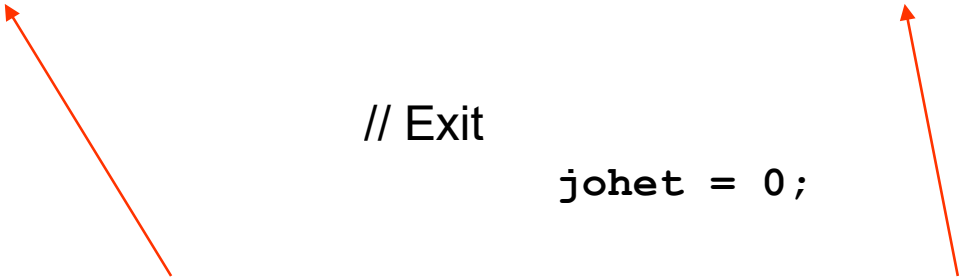
P1

// Entry

```
while (johet != 1) …;
```

// Exit

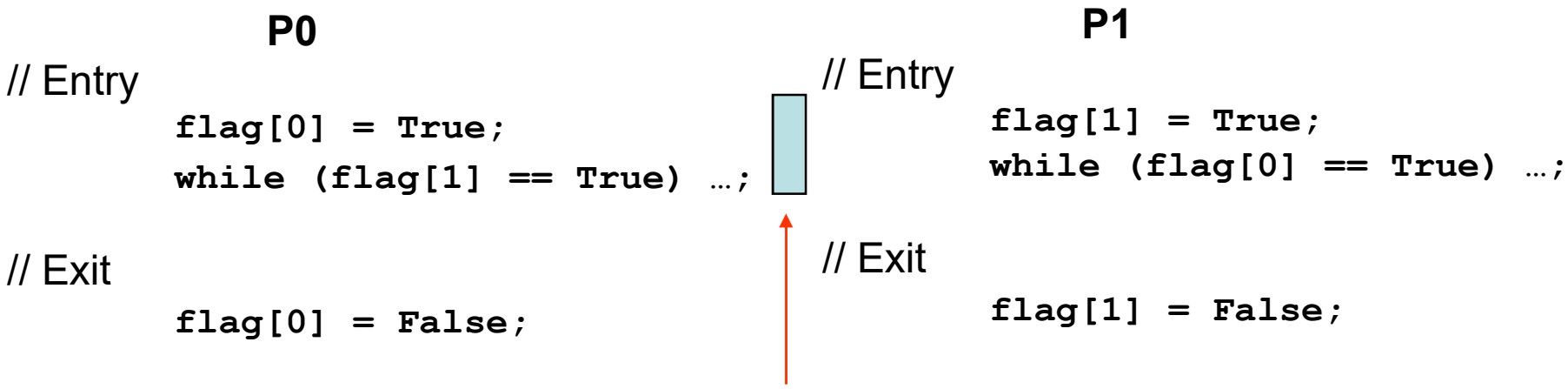
```
johet = 0;
```



Alternál, csak egymást felváltva léphetnek be

3. Mindenkinek flag a bent tartózkodásra

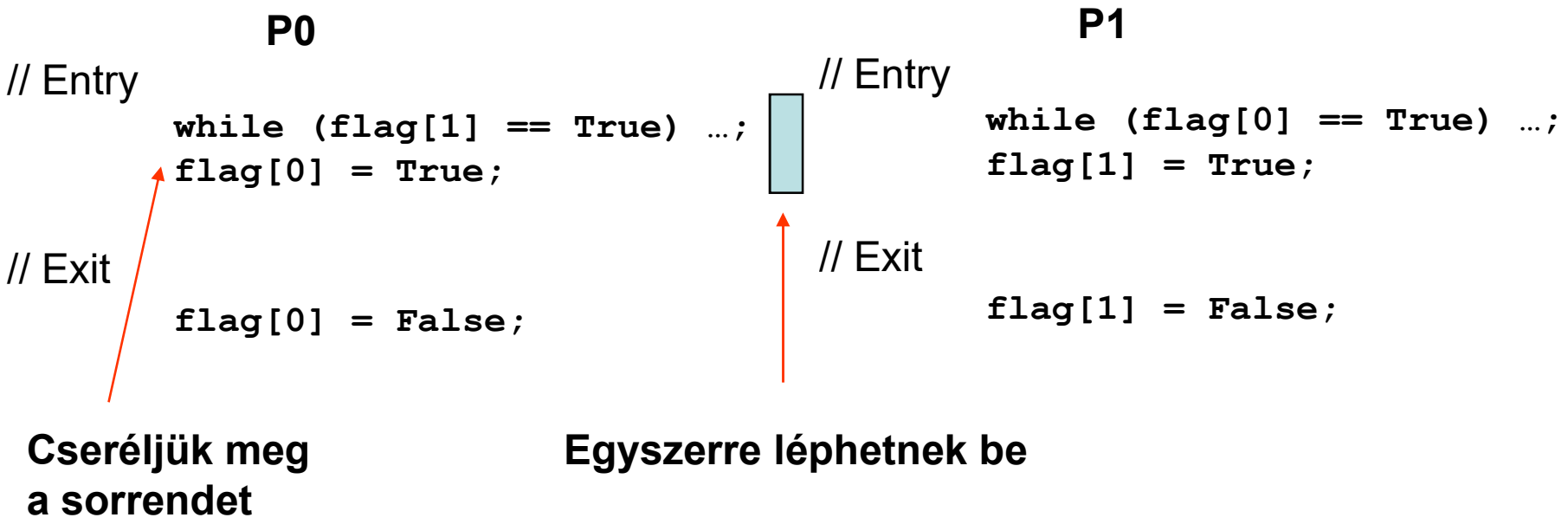
```
int flag[2] = { False, False };
```



Ha egyszerre indulnak: holtpont

3. Mindenkinek flag a bent tartózkodásra

```
int flag[2] = { False, False };
```



4. Folyamatonként flag + alternáló kapcsoló (Peterson 1981)

```
int flag[2] = { False, False }; int johet;
```

P0

// Entry

```
flag[0] = True;
johet = 1;
while ((flag[1] == True) &&
       (johet == 1)) …;
```

// Exit

```
flag[0] = False;
```

P1

// Entry

```
flag[1] = True;
johet = 0;
while ((flag[0] == True) &&
       (johet == 0)) …;
```

// Exit

```
flag[1] = False;
```

P0 beléphet, ha

P1 kilépett -- flag[1] == False

vagy

P1 belépésre vár -- johet == 0

2 folyamatra működik, n folyamatra jóval bonyolultabb