

# Operációs rendszerek

ELTE IK. BSC.

Dr. Illés Zoltán

# Miről beszéltünk korábban...

- Operációs rendszerek kialakulása
  - Sz.gép – Op.rendszer generációk
- Op. Rendszer fogalmak, struktúrák
  - Kliens-szerver modell, ...
  - Rendszerhívások
- Fájlok, könyvtárak, fájlrendszerek
  - Fizikai felépítés
  - Logikai felépítés
  - FAT, UNIX, NTFS,...

# Mi következik ma...

- Folyamatok- Processes
  - Létrehozása, befejezése- Creating, ending
  - Folyamat állapotok- States of processes
- Folyamatok kommunikációja- Process communication
  - Versenyhelyzetek, kritikus szekciók- Race situation
  - Szemaforok, mutexek, monitorok
- Klasszikus IPC problémák

# Folyamatok modellje

- Program – folyamat különbsége
- Folyamat( process): futó program a memóriában (kód+I/O adatok+állapot)
- Egyszerre hány folyamat működik?
  - Single Task – Multi Task
  - Valódi Multi Task?
- Szekvenciális modell
- Processzek közti kapcsolás: multiprogramozás
- Egy időben csak egy folyamat aktív.

# Rendszer modell

- 1 processzor + 1 rendszer memória + 1 I/O eszköz = 1 feladatvégrehajtás
- Interaktív (ablakos) rendszerek, több program, több processz fut
  - Környezetváltásos rendszer: csak az előtérben lévő alkalmazás fut
  - Kooperatív rendszer: az aktuális processz bizonyos időközönként, vagy időkritikus műveletnél önként lemond a CPU-ról (pl: Win 3.1)
  - Preemptív rendszer: az aktuális processztől a kernel bizonyos idő után elveszi a vezérlést, és a következő várakozó folyamatnak adja.
  - Real time rendszer

# Folyamatok létrehozása

- Ma tipikusan preemptív rendszereket használunk (igazából a valós idejű is az)
- Több folyamat él, aktív.
- Folyamat létrehozás oka lehet:
  - Rendszer inicializálás
  - Folyamatot eredményező rendszerhívás
    - Másolat az eredetiről (fork)
    - Az eredeti cseréje (execve)
  - Felhasználói kérés (parancs&)
  - Nagy rendszerek köteget feladatai
- Előtérben futó folyamatok
- Háttérben futó folyamatok (démonok)

# Folyamatok kapcsolata

- Szülő – gyermek kapcsolat
- Folyamatfa:
  - egy folyamatnak egy szülője van
  - Egy folyamatnak több gyermeke lehet
  - Összetartozó folyamatcsoport
  - Pl: Init, /etc/rc script végrehajtása
    - Az init id-je 1.
  - Fork utasítás...vigyázat a használatával
- Reinkarnációs szerver
  - Meghajtó programok, kiszolgálók elindítója.
  - Ha elhal az egyik, akkor azt újraszüli, reinkarnálja.

# Folyamatok befejezése

- Folyamat elindulása után a megadott időkeretben végzi (elvégzi) a feladatát.
- A befejezés okai:
- Önkéntes befejezések
  - Szabályos kilépés (exit, return stb.)
  - Kilépés valamilyen hiba miatt, amit a program felfedez (szintén pl. return utasítással)
- Önkéntelen befejezések
  - Illegális utasítás, végzetes hiba (0-val osztás, nem létező memória használat, stb)
  - Külső segítséggel. Másik processz, netán mi „lőjük” ki az adott folyamatot.

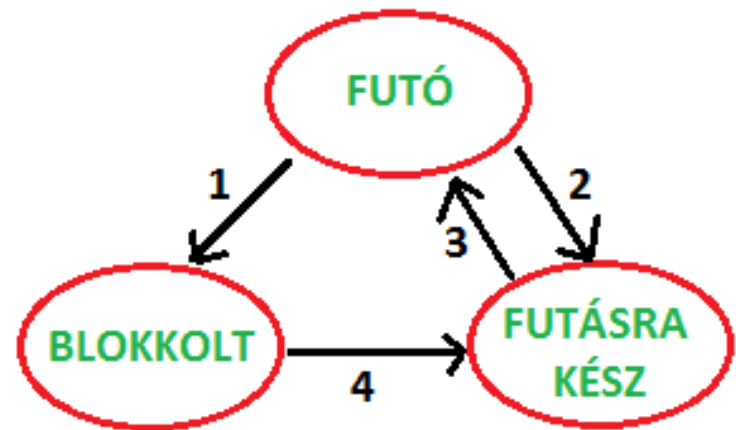


# Folyamatok állapota

- Folyamat: önálló programegység, saját utasításslámlálóval, veremmel stb.
- Általában nem függetlenek a folyamatok
  - Egyik-másik eredményétől függ a tevékenység
- Egy folyamat három állapotban lehet:
  - Futó
  - Futásra kész, ideiglenesen leállították, arra vár, hogy az ütemező CPU időt adjon a folyamatnak.
  - Blokkolt , ha logikailag nem lehet folytatni a tevékenységet, mert pl. egy másik eredményére vár. (cat Fradi.txt | grep Fradi | sort, grep és sort blokkolt az elején...)

# Állapotátmenetek

1. Futó -> Blokkolt
  - Várni kell valamire
2. Futó ->Futásra kész
3. Futásra kész ->Futó
  - Ezekről az ütemező dönt, a folyamatok nem nagyon tudnak róla.
4. Blokkolt->Futásra kész
  - A várt adat megérkezett



# Folyamatok megvalósítása

- A processzor „csak” végrehajtja az aktuális utasításokat (CS:IP)
- Egyszerre egy folyamat aktív.
- Folyamatokról nem tud.
  - Ha lecseréljük az aktív folyamatot a következőre, mit kell megőrizni, hogy visszatérhessünk a folytatáshoz?
  - Mindent....utasítás számlálót, regisztereket, lefoglalt memória állapotot, nyitott fájl infókat, stb.
  - Ezeket az adatokat az un. Folyamat leíró táblában tároljuk (processz tábla, processz vezérlő blokk)
- I/O megszakításvektor

# Folyamatok váltása

- Mi kezdeményezi?
  - Időzítő, megszakítás, esemény, rendszerhívás kezdeményezés.
- Ütemező elmenti az aktuális folyamat jellemzőket a folyamatleíró táblába
- Betölti a következő folyamat állapotát, a processzor folytatja a munkát.
- Nem lehet menteni a gyorsító táraikat
  - Gyakori váltás - többlet erőforrást igényel
  - A folyamat váltási idő „jó” megadása nem egyértelmű.

# Folyamatleíró táblázat – Process Control Block (PCB)

- A rendszer inicializáláskor létrejön
  - 1 elem, rendszerindító már bent van mikor az rendszer elindul.
- Tömbszerű szerkezet(PID alapon) - de egy-egy elem egy összetett processzus adatokat tartalmazó struktúra.
- Egy folyamat fontosabb adatai:
  - Azonosítója (ID), neve (programnév)
  - Tulajdonos, csoport azonosító
  - Memória, regiszter adatok
  - Stb.

# Szálak

- Tipikus helyzet: Egy folyamat – egy utasítássorozat – egy szál
- Néha szükséges lehet, hogy egy folyamaton belül „több utasítássorozat” legyen
  - Szál: egy folyamaton belüli különálló utasítás sor
  - Gyakran „lightweight process”-nek nevezik
- Szálak: Egy folyamaton belül több egymástól „független” végrehajtási sor.
  - Egy folyamaton belül egy szál
  - Egy folyamaton belül több szál-Ha egy szál blokkolódik, a folyamat is blokkolva lesz!
  - Száltáblázat
- Folyamatnak önálló címtartománya van, szálnak nincs!

# Folyamatok-Szálak jellemzők

- Csak folyamatnak van:
  - Címtartománya
  - Globális változók
  - Megnyitott fájl leírók
  - Gyermek folyamatok
  - Szignálkezelők, ébresztők
  - ...
- Szálnak is van:
  - Utasításszámlálók
  - Regiszterek, verem

# Szálproblémák

- Fork- Biztos, hogy a gyerekekben kell több szál, ha a szülőben több van! (Igen)
- Fájlkezelés- Egy szál lezár egy fájlt, miközben a másik még használná!
- Hibakezelés- errno globális értéke
- Memóriakezelés-...
- Lényeges: A rendszerhívásoknak kezelni kell tudni a szálakat (thread safe)



# Folyamatok kommunikációja

- IPC – Inter Process Communication
- Három területre kell megoldást találni:
  - Két vagy több folyamat ne keresztezze egymást kritikus műveleteknél.
  - Sorrend figyelembevétel (bevárás). Nyomtatás csak az adatok előállítása után lehetséges.
  - Hogy küldhet egy folyamat információt, üzenetet egy másiknak.
- Szálakra mindhárom terület ugyanúgy érdekes, csak az információküldés, az azonos címtartomány miatt egyszerű.

# „Párhuzamos” rendszerek

- Ütemező a folyamatok gyors váltogatásával „teremt” párhuzamos végrehajtás érzetet.
- Többprocesszoros rendszerek
  - Több processzor egy gépben
  - Nagyobb teljesítmény
  - Megbízhatóságot általában nem növeli
- Klaszterek (Fürt, Cluster)
  - Megbízhatóság növelése elsősorban
- Kulcskérdés: a közös erőforrások használata

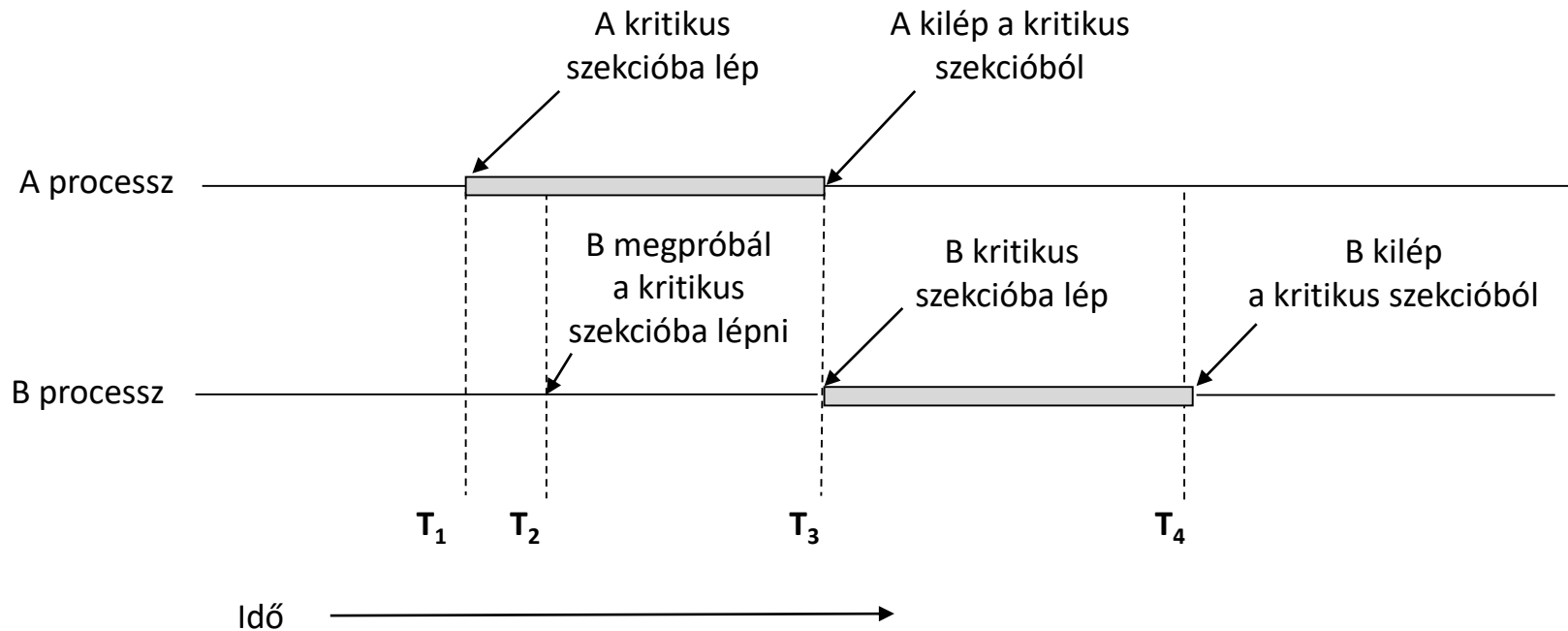
# Közös erőforrások

- Avagy, amikor két folyamat ugyanazt a memóriát használja...
  - Közös ló ...
  - Pl: 2 folyamat nyomtatása, közös nyomtatósor
- Versenyhelyzet: két vagy több folyamat közös memóriát ír vagy olvas, a végeredmény a futási időpillanattól függ!
  - Nehezen felderíthető hibát okoz.
- Megoldás: Módszer ami biztosítja, hogy a közös adatokat egyszerre csak egy folyamat tudja használni

# Kölcsönös kizárás

- Kritikus programterület, szekció, az a rész mikor a közös erőforrást (memóriát) használjuk.
- A jó kölcsönös kizárás az alábbi feltételeknek felel meg:
  - Nincs két folyamat egyszerre a kritikus szekciójában.
  - Nincs sebesség, CPU paraméter függőség.
  - Egyetlen kritikus szekción kívül levő folyamat sem blokkolhat másik folyamatot.
  - Egy folyamat sem vár örökké, hogy a kritikus szekcióba tudjon belépni.

# A megkívánt kölcsönös kizárás viselkedése



# Kölcsönös kizárás megvalósítások I.

- Megszakítások tiltása (összes)
  - Belépéskor az összes megszakítás tiltása
  - Kilépéskor azok engedélyezése
  - Ez nem igazán jó, mivel a felhasználói folyamatok kezében lenne a megszakítások tiltása...persze a kernel használja.
- Osztott, un. zárolás változó használata
  - 0 (senki) és 1 (valaki) kritikus szekcióban van
  - Két folyamat is kritikus szekcióba tud kerülni!
    - Egyik folyamat belép a kritikus szekcióba, de éppen az 1-re állítás előtt a másik folyamat kerül ütemezésre.

# Kölcsönös kizárás megvalósítások II.

- Szigorú változtatás:
  - Több folyamatra is általánosítható.
  - A kölcsönös kizárás feltételeit teljesíti a 3 kivétellel, ugyanis ha pl 1 folyamat a lassú, nem kritikus szekcióban van, és a 0 folyamat gyorsan belép a kritikus szekcióba, majd befejezi a nem kritikus szekciót is, akkor ez a folyamat blokkolódik mert a `kovetkezo=1` lesz! (Saját magát blokkolja!)
  - 0. folyamat
  - 1.folyamat

```
while(1)
{
    while(kovetkezo!=0) ;
    kritikus_szekcio();
    kovetkezo=1;
    nem_kritikus_szekcio();
}
```

```
while(1)
{
    while(kovetkezo!=1) ;
    kritikus_szekcio();
    kovetkezo=0;
    nem_kritikus_szekcio();
}
```

# G.L.Peterson javítása

- 1981, a szigorú változtatás javítása
- A kritikus szekció előtt minden folyamat meghívja a belépés, majd utána kilépés fv-t.

```
#define N 2
int kovetkezo;
int akarja[N];
/* a módosított folyamat*/
while(1)
{
    belepes(processz);
    kritikus_szekcio();
    kilepes(processz);
    nem_kritikus_szekcio();
}
```

```
void belepes(int proc)
{
    int masik;
    masik=1-proc; //mivel N=2...
    // masik=(proc+1) % N;
    akarja[proc]=1; //processz futni akar
    kovetkezo=proc;
    while( kovetkezo==proc &&
           akarja[masik]);
}
void kilepes(int proc)
{
    akarja[proc]=0; //hamis
}
```



# Kis Peterson „javítás”- nagy hiba

- Tegyük fel  $proc=0$ !
- A jelölt ütemezés váltásnál a  $proc=1$  belépése jön.
- Mivel  $akarja[0]$  értéke 0, ezért az 1-es process belép a kritikus szakaszba!
- Ekkor újra váltson az ütemező,  $akarja[1]=1$ , a következő értéke szintén 1, így a  $kovetkezo==proc$  hamis, azaz a 0. proc is belép a kritikus szakaszba!

```
void belepes(int proc)
{
    int masik;
    masik=1-proc; //mivel N=2...
    // masik=(proc+1) % N;
    kovetkezo=proc; //két sor csere
    /* itt van ütemező váltás
    akarja[proc]=1; //proc futni akar
    while( kovetkezo==proc &&
           akarja[masik]);
}
```

```
void kilepes(int proc)
{
    akarja[proc]=0; //hamis
}
```

# Tevékeny várakozás gépi kódban

- TSL utasítás – Test and Set Lock
  - Atomi művelet (megszakíthatatlan)

belepes:

```
TSL regiszter, LOCK ; LOCK a regiszterbe kerül  
                      ; és LOCK=1  
                      ; TSL alatt a CPU zárolja a  
                      ; memóriasínt!!!
```

```
cmp regiszter,0  
jne belepes      ; ha nem 0, ugrás  
ret
```

;

Kilepes:

```
mov LOCK,0  
ret
```

# Tevékeny várakozás

- A korábbi Peterson megoldás is, a TSL használata is jó, csak ciklusban várakozunk.
- A korábbi megoldásokat, tevékeny várakozással (aktív várakozás) megoldottnak hívjuk, mert a CPU-t „üres” ciklusban járatjuk a várakozás során!
- A CPU időt pazarolja...
- A CPU pazarlása helyett jobb lenne az, ha a kritikus szekcióba lépéskor blokkolna a folyamat, ha nem szabad belépnie!

# Alvás - ébredés

- Az aktív várakozás nem igazán hatékony
- Megoldás: blokkoljuk(alvás) várakozás helyett a folyamatot, majd ha megengedett ébresszük fel.
  - sleep –wakeup, down-up, stb.
  - Különböző paraméter megadással is implementálhatók.
  - Tipikus probléma: Gyártó-Fogyasztó probléma

# Gyártó-Fogyasztó probléma

- Korlátos tároló problémaként is ismert.
- PL: Pék-pékség-Vásárló háromszög.
  - A pék süti a kenyeret, amíg a pékség polcain van hely.
  - Vásárló tud venni, ha a pékség polcain van kenyér.
  - Ha tele van kenyérrel a pékség, akkor „a pék elmegy pihenni”.
  - Ha üres a pékség, akkor a vásárló várakozik a kenyérre.

# Gyártó-Fogyasztó probléma egy megvalósítása

- Pék folyamat

## Vásárló folyamat

```
#define N 100
int hely=0;
void pék()
{
    int kenyér;
    while(1)
    {
        kenyér=új_kenyér()
        if (hely==N) alvás();
        polcra(kenyér);
        hely++;
        if (hely==1)
            ébresztő(vásárló);
    }
}
```

```
void vásárló()
{
    int kenyér;
    while(1)
    {
        if (hely==0) alvás();
        kenyér=kenyeret();
        hely--;
        if (hely==N-1)
            ébresztő(pék);
        megesszük(kenyér);
    }
}
```

# Pék-Vásárló probléma

- A „hely” változó elérése nem korlátozott, így ez okozhat versenyhelyzetet.
  - Vásárló látja, hogy a hely 0 és ekkor az ütemező átadja a vezérlést a péknek, aki süt egy kenyeret. Majd látja, hogy a hely 1, ébresztőt küld a vásárlónak. Ez elveszik, mert még a vásárló nem alszik.
  - Vásárló visszakapja az ütemezést, a helyet korábban beolvasta, az 0, megy aludni.
  - A pék az első után megsüti a maradék  $N-1$  kenyeret és ő is aludni megy!
- Lehet ébresztő bittel javítani, de több folyamatnál a probléma nem változik.

# Szemaforok I.

- E.W. Dijkstra (1965) javasolta ezen új változótípus bevezetését.
- Ez valójában egy egész változó.
- A szemafor tilosat mutat, ha értéke 0.
  - A folyamat elalszik, megáll a tilos jelzés előtt.
- Ha a szemafor  $>0$ , szabad a pálya, beléphetünk a kritikus szakaszra.
- Két művelet tartozik hozzá:
  - Ha beléptünk, csökkentjük szemafor értékét. (down)
  - Ha kilépünk, növeljük a szemafor értékét. (up)
  - Ezeket Dijkstra P és V műveletnek nevezte.



# Szemaforok II.

- Elemi művelet: a szemafor változó ellenőrzése, módosítása, esetleges elalvás, oszthatatlan művelet, nem lehet megszakítani!
- Ez garantálja, hogy ne alakuljon ki versenyhelyzet.
- Ha a szemafor tipikus vasutas helyzetet jelöl, azaz 1 vonat mehet át csak a jelzőn, a szemafor értéke ekkor 0 vagy 1 lehet!
  - Bináris szemafor
  - Ezt MUTEX-nek (Mutual Exclusion) is hívjuk, kölcsönös kizárásra használjuk.

# Szemafor megvalósítások

- Up, Down műveleteknek atominak kell lenni.
  - Nem blokkolhatók!
- Hogyan?
  - Op. Rendszerhívással, felhasználói szinten nem biztosítható.
  - Művelet elején például letiltunk minden megszakítást.
  - Ha több CPU van akkor az ilyen szemafort védeni tudjuk a TSL utasítással
- Ezek a szemafor műveletek kernel szintű, rendszerhívás műveletek.
- A fejlesztői környezetek biztosítják.
  - Ha mégsem gáz van...

# Gyártó-fogyasztó probléma megoldása szemaforokkal I.

- Gyártó (pék) függvénye

```
typedef int szemafor;
szemafor szabad=1; /*Bináris szemafor,1 mehet tovább, szabad a jelzés*/
szemafor üres=N, tele=0; /* üres a polc, ez szabad jelzést mutat*/
void pék()          /* N értéke a „kenyerespolc” mérete */
{
    int kenyér;
    while (1)
    {
        kenyér=pék_süt();
        down(&üres);          /* üres csökken, ha előtte>0, mehet tovább*/
        down(&szabad); /* Piszkálhatjuk-e a pékség polcát? */
        kenyér_polcra(kenyér); /* Igen, betesszük a kenyeret. */
        up(&szabad);          /* Elengedjük a pékség polcát. */
        up(&tele);            /* Jelezzük vásárlónak, van kenyér. */
    }
}
```

# Gyártó-fogyasztó probléma megoldása szemaforokkal II.

- Fogyasztó (Vásárló) függvénye.

```
void vásárló()      /* vásárló szemaforja a tele */
{
    int kenyér;
    while (1)
    {
        down(&tele);      /*tele csökken, ha előtte>0, mehet tovább*/
        down(&szabad); /*Piszkálhatjuk-e a pékség polcát? */
        kenyér=kenyér_polcáról(); /* Igen, levesszük a kenyeret. */
        up(&szabad);      /* Elengedjük a pékség polcát. */
        up(&üres);        /* Jelezzük péknek, van hely, lehet sütni. */
        kenyér_elfogyasztása(kenyér);
    }
}
```

# Szemafor példa összegzés

- Szabad: kenyér polcot (boltot) védi, hogy egy időben csak egy folyamat tudja használni (vagy a pék, vagy a vásárló)
  - Kölcsönös kizárás
  - Elemi műveletek (up, down)
- Tele, üres szemafor: szinkronizációs szemaforok, a gyártó álljon meg ha a tároló tele van, illetve a fogyasztó is várjon ha a tároló üres.

# Szemafor példa

- Unix környezetben:
  - semget: szemafor létrehozása(System V)
  - semctl: szemafor kontrol, kiolvasás, beállítás
  - semop: szemafor operáció (up,down művelet)
  - sembuf struktúra
  - Gyakorlaton részletesen szerepel
  - sem\_open,sem\_wait,sem\_post,sem\_unlink (Posix)
- Most nézzünk egy C# példát szemaforokra.
  - VS 2008.
  - Szemafor-pék-vásárló példa.

Köszönöm a figyelmet!

[zoltan.illes@elte.hu](mailto:zoltan.illes@elte.hu)