Operációs rendszerek:

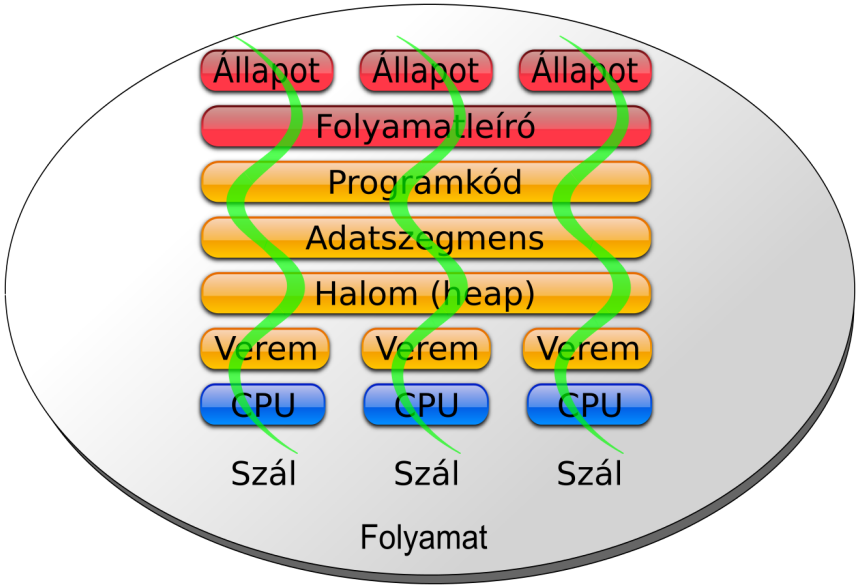
taszkok szinkronizációja

Mészáros Tamás
http://www.mit.bme.hu/~meszaros/

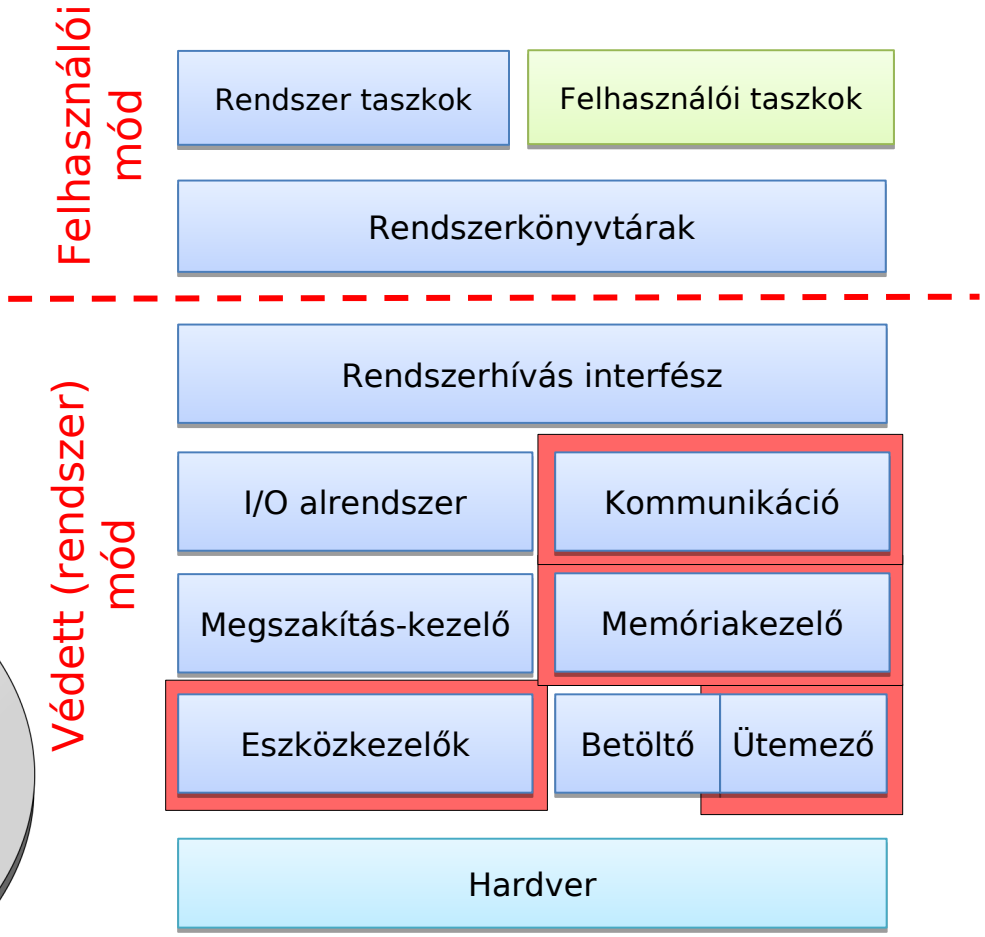
Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

Az eddigiekben történt...

- Feladat – taszk összerendelés
  - együttműködés, kommunikáció
- Kommunikáció
  - PRAM modell (versenyhelyzet)
- Kernel erőforrás-kezelés
  - virtuális gép ← erőforrások



Az OS felépítése

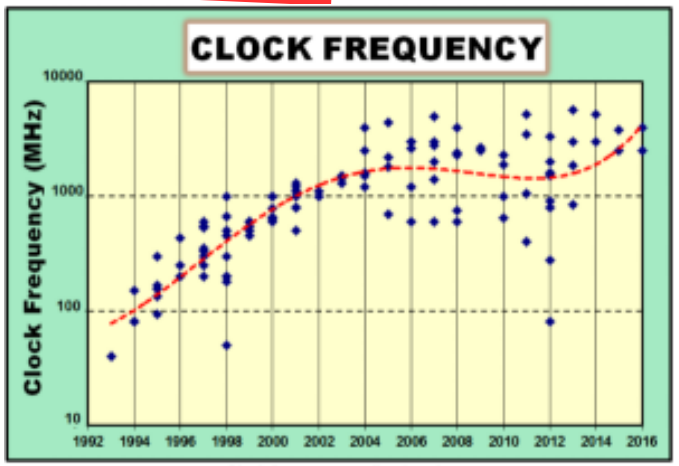


Az egyszerű teljesítménynövelés alkonya

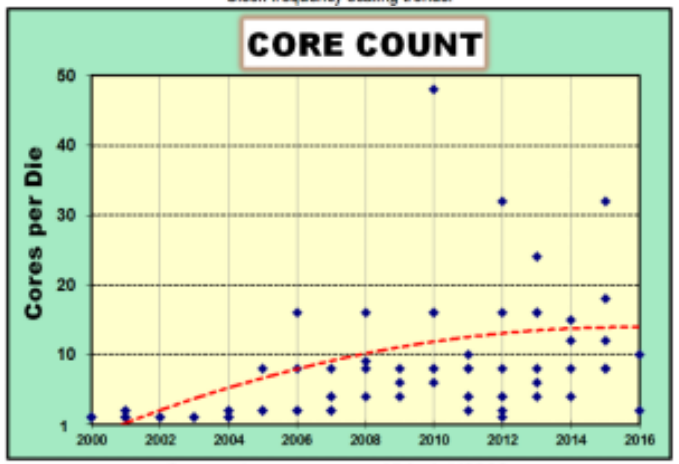
- ~~„Ha nem elég gyors a program, akkor vegyél erősebb hardvert”~~

- Egyre kevésbé járható út
  - az órajelnövelés egyre nehezebb
  - a processzormagok száma sem nő dinamikusan
  - többszálú alkalmazások (lásd Szoftvertechnikák)
  - és terjednek a heterogén rendszerek

- Más szemléletű programozás
  - munkamegosztás
    - adott feladatra illeszkedő
    - párhuzamosított megoldás
      - pl. nagy adathalmaz többszálú feldolgozása
  - konkurens programozás
    - programkódok átlapolódó végrehajtása
    - közös erőforrások koordinált használata
      - pl. osztott memória (adathalmaz) és szálak



Clock frequency scaling trends.



Core counts on processors published at ISSCC.

„Vége az ingyen ebédnek, fordulat a párhuzamosítás felé a szoftverekben” (2006)

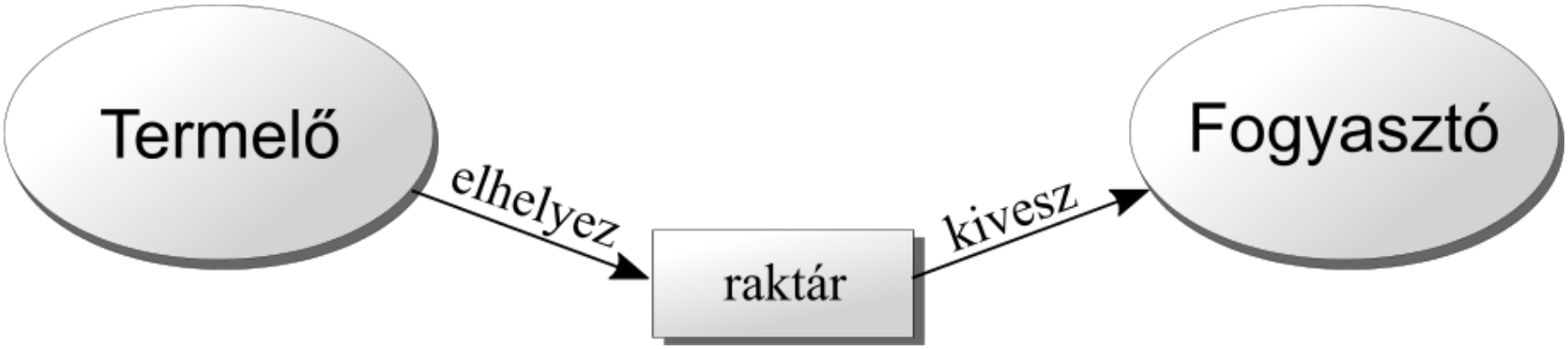
Taszkok együttműködése és versengése

- Együttműködő taszkok
  - részekre bontott feladat
  - a taszkok **kooperálnak** A programozó valósítja meg.
    - adatcsere, végrehajtási függőségek
    - versenyhelyzetek alakulhatnak ki
  - az OS védelmi mechanizmusai nehezítik a megvalósítást → kommunikáció

- Független taszkok
  - (nincs szükségük egymásra)
  - aszinkron végrehajtás
  - de **multiprogramozott rendszer**t alkotnak Az OS valósítja meg.
    - közös erőforrás-használat
    - versenyhelyzetek alakulhatnak ki
    - végrehajtási függőségek keletkezhetnek
  - az OS erőforrás-kezelési mechanizmusai könnyítik a megvalósítást

- Egyprocesszoros rendszerekben van versenyhelyzet?
  - **Igen**, hiszen egy taszk futása félbeszakadhat, és másik taszk kezdhet el futni.

Egyszerű példa: a termelő-fogyasztó probléma



- Párhuzamosan működnek
  - különböző ütemben
  - eltérő sebességgel

konkurens programozás

- Megoldandó feladatok
  - a raktár konzisztenciája (PRAM modell)
  - a Fogyasztó blokkolása üres raktár esetén
  - a Termelő blokkolása tele raktár esetén

versenyhelyzet (race condition)

Taszkok szinkronizációja

- Összehangoljuk a működésüket
  időlegesen megállítjuk valamelyik működését

*A **szinkronizáció** a taszkok működésének összehangolása a művelet-végrehajtás időbeli korlátozásával*

- Céljai
  - versenyhelyzetekben: konzisztencia
    - pl. közös memória védelme (hogyan?)
  - együttműködésben: műveleti sorrend
    - előidejűség (precedencia) biztosítása (hogyan?)

Milyen példát láttunk eddig szinkronizációra?

A szinkronizáció „ára”

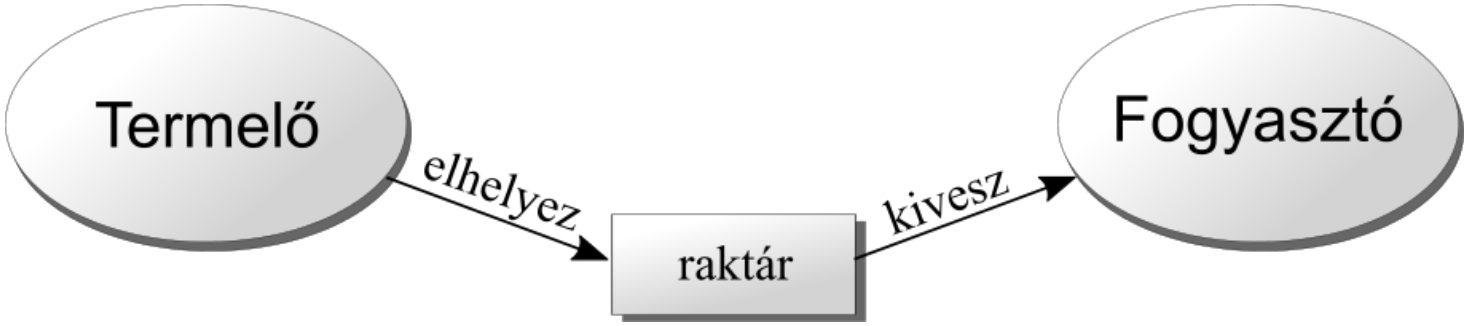
- Korlátozom a végrehajtást → csökken a teljesítmény
  - a megállított taszk
    - nem hajt végre utasítást
    - nem vár I/O műveletre stb.
  - lásd pl. ezt a tesztet
- Tervezés
  - alulterv
    - he
  - felülter
    - fel
    - ro

Method	OSX-1	OSX-2	Linux-1	Linux-2	Linux-4
Single	1.4 / 1.4	N/A	1.2 / 1.1	N/A	N/A
Lock-free	3.8 / 3.8	2.2 / 4.2	3.6 / 3.6	3.7 / 7.3	2.5 / 9.9
Spin	5.8 / 5.7	5.1 / 9.5	5.1 / 5.1	20.7 / 41.2	32.7 / 130
Pthread spin	N/A	N/A	3.8 / 3.8	21.1 / 42.2	53.8 / 206
Pthread mutex	7.5 / 7.5	182 / 271	6.0 / 5.9	31.0 / 45.1	97.2 / 284
Semaphore	85 / 85	51 / 96	5.5 / 5.4	46.0 / 68.9	133 / 418
Buffer+spin	1.3 / 1.3	0.7 / 1.3	1.2 / 1.2	0.7 / 1.4	0.5 / 2.0
Buffer+mutex	1.3 / 1.3	0.7 / 1.4	1.2 / 1.2	0.7 / 1.4	0.5 / 1.5

A szinkronizáció alapvető formái

- Kölcsönös kizárás (mutual exclusion)
  - taszkok egymást kizáró működése
  - cél: erőforrás-védelem, versenyhelyzetek kezelése
  - pl.: osztott memória védelme, közös erőforrások használata
- Egyidejűség (randevú)
  - a taszkok megadott műveletei egyszerre kezdődjenek el
  - cél: összehangolt működés
  - pl.: blokkoló, nem pufferelt üzenetküldés
- Előírt végrehajtási sorrend (precedencia)
  - taszkok adott műveletei meghatározott sorrendben hajtódjanak végre
  - cél: műveleti sorrend betartása
  - pl. termelő-fogyasztó együttműködés

Szinkronizáció a termelő-fogyasztó probléma esetén



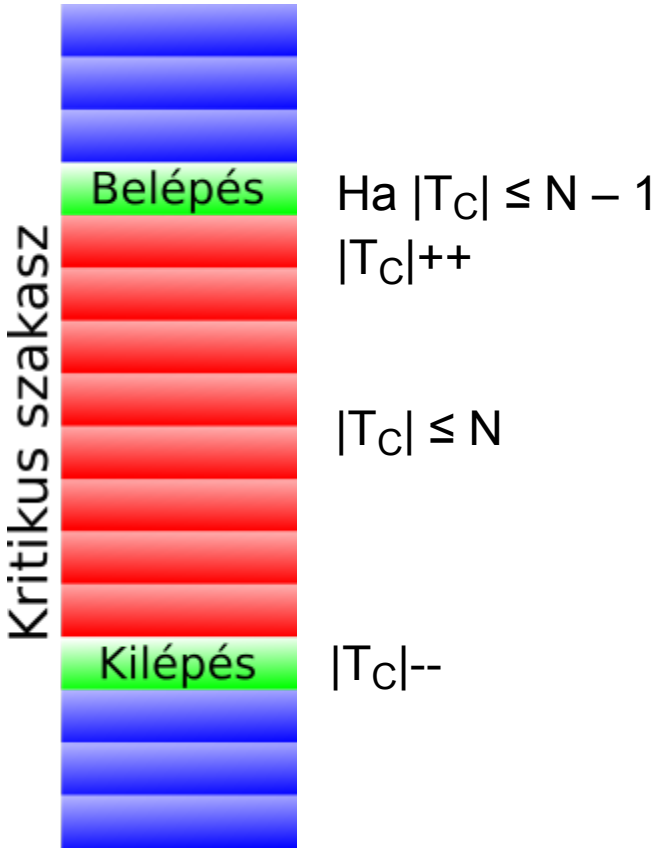
- Kölcsönös kizárás
  - a raktár konzisztenciája
    - egyszerre csak egy (termelő / fogyasztó) módosíthatja
- Precedencia
  - termék elkészítése → felhasználása
    - a fogyasztó várjon addig, amíg a raktár üres
  - termékkivétel tele raktárból → termék elhelyezése a raktárban
    - a termelő várjon addig, amíg a raktár tele van

A szinkronizáció megvalósítása

A kölcsönös kizárás megoldása

*A **kritikus szakasz** (critical section) utasítások egy olyan sorozata, amelyet egy időben a taszkoknak csak egy korlátozott halmaza* (T_C) hajthat végre, azaz $|T_C| \leq N$.

- Működési szabályok
  - ha $|T_C| < N$ akkor a Belépésre váró taszkok közül egy beléphet a kritikus szakaszba
  - egy Belépésre váró taszkot csak véges számú másik előzhet meg a belépésben
  - a kritikus szakaszban levő taszk csak véges számú utasítást hajthat végre



- Alkalmazási példák
  - osztott memória írása
    - N = 1
  - K db erőforrás közös használata
    - N = K K db taszknak jut erőforrás

A szinkronizáció hardver támogatása

- Egyszerű megoldás: a megszakítások letiltása a kritikus szakaszban
  - nincs átütemezés, nincs taszkváltás
  - csak egyprocesszoros esetben működhet
  - fontos eseményekről maradhat le a rendszer

- Jó megoldás: atomi adatműveletek

  atomi = nem szakítható meg

  - **test-and-set lock** **`TSL(zár)`**
    - beállítja egy bit (zár) új értéket `IGAZ`-ra és visszaadja a régit
    - alkalmazása:

```
while ( TSL(zár) ) { }
```

Hogyan működik?

  - **compare-and-swap** **`CAS(zár, kívánt-érték, új-érték)`**
    - ha egy változó (zár) meghatározott értékű (a), akkor módosítja (b-re)
    - a változó régi értékével (a) tér vissza
    - alkalmazása:

Hogyan működik?

```
while ( CAS(zár, a, b) != a ) { }
```

„spinning lock”: a zárra várva végtelen ciklusban „teker” a program

Kritikus szakasz megvalósítása TSL és CAS operátorral

Test-and-set lock (TSL)

```
// nem védett programrész
while(TSL(lock)) { }
// ez a kritikus szakasz
lock = FALSE;
// további, nem védett rész
```

Compare-and-swap (CAS)

```
// nem védett programrész
while(CAS(lock, 0, 1) != 0) { }
// ez a kritikus szakasz
lock = 0;
// további, nem védett rész
```

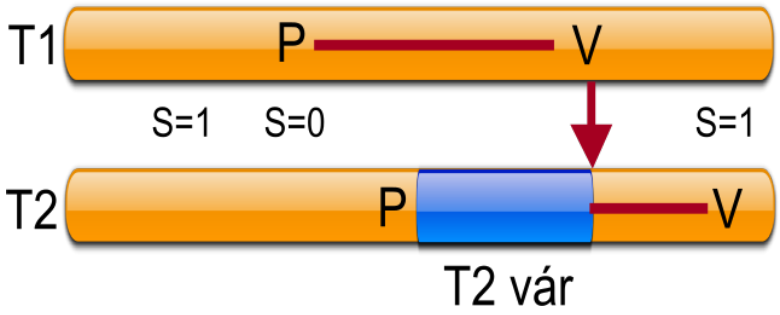
- Mi a gond ezekkel?
  - a „spinning”: folyamatosan fut a taszk („busy waiting”)
  - jobb lenne egy blokkoló művelet, hogy a taszk várakozó állapotba kerüljön.
- Mikor / miért jó mégis?
  - a hardver támogatja (x86: XCHG és CMPXCHG)
  - ha tudjuk, hogy nem kell sokat várni (pl. a kernelben)

Zárolási eszközök áttekintése

- Lock bit
  - egybites, oszthatatlan művelettel rendelkező zárolási eszköz, pl. a TSL
- Mutex (mutual exclusion lock)
  - egy kritikus szakasz védelmére alkalmazott zárolási eszköz (pl. lock bit)
  - jellemzően blokkolja a taszkot, azaz kontextusváltással jár
- Szemafor
  - atomi műveletekkel rendelkező változó
  - várakozás (P) és továbbengedés (V)
- Spinlock (spinning lock)
  - olyan lock, mutex vagy szemafor, amely aktívan várakozik („busy waiting”)
  - pl. a TSL és a CAS egy while() { } ciklusban
  - rövid kritikus szakaszok esetén kiváló (megspórolja a kontextusváltás költségét)
- ReaderWriterLock
  - tetszőleges számú olvasó beléphet a kritikus szakaszba (reader lock)
  - ha író lépne be (writer lock), akkor blokkolódik, míg az összes olvasó ki nem lép
- RecursiveLock
  - aki a zárat birtokolja, az bármikor megkaphatja újra a zárat blokkolás nélkül
  - rekurzív programok készítéséhez hasznos zárolási típus

A szemafor

- Szemafor (S)
  - két atomi művelettel támogatott adattípus
- S értékkészlete
  - bináris szemafor (mutex): {0, 1}
  - (számláló típusú) szemafor: nemnegatív egész számok {0, 1, …}
- Műveletei
  - P(S) művelet: vár, amíg a szemafor értéke eggyel csökkenthető lesz (>0)
    - lehet blokkoló (várakozó állapotba kerül a taszk)
  - V(S) művelet: eggyel növeli a szemafor értékét
    - nem blokkoló művelet (nincs mire várni)



A szemafor megvalósítása `TSL()` segítségével

- Adatstruktúrák

```
int count           // a szemafor értéke
queue_t waiting     // P() várakozási sor
lock_t lock         // a count változó védelme
```

P(S)

```
while (TSL(lock)) { }
if (count == 0) {
  fifo_add(waiting, T);
  sched_block(T);
} else {
  count--;
}
lock = 0;
```

V(S)

```
while (TSL(lock)) { }
if (is_empty(waiting)) {
  count++;
} else {
  T2 = fifo_get_first(waiting);
  sched_wakeup(T2);
}
lock = 0;
```

POSIX szemafor alkalmazási példa

- Bináris, szálak által használt szemafor megvalósítása
- globális változó (az összes szál által látható)

```
sem_t mysem;
```

- inicializálás bináris szemaforként

```
sem_init(&mysem, 0, 1);
```

- zárolás / zárolás időkorláttal

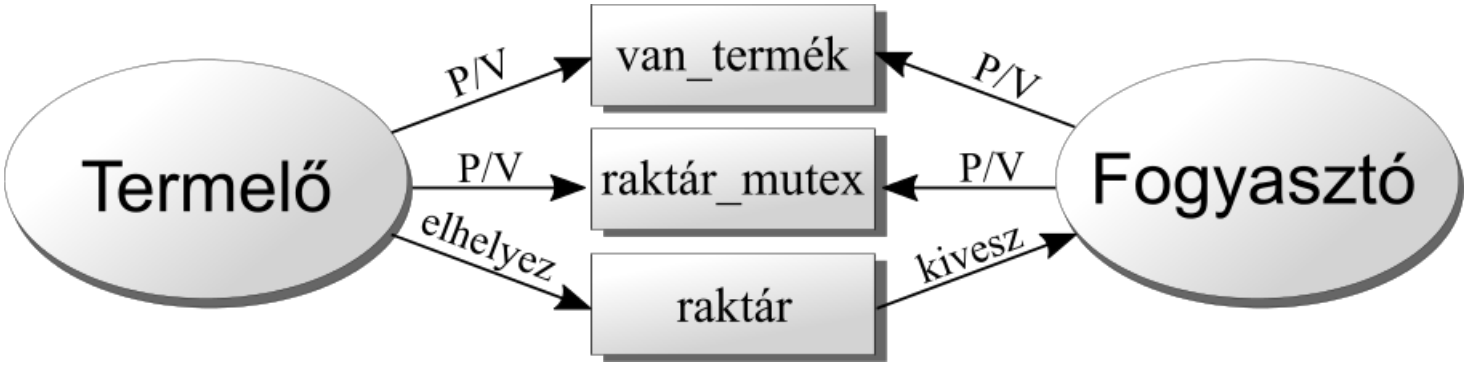
```
sem_wait(&mysem);
sem_timedwait(&mysem, timeout);
// itt lép be a kritikus szakaszba
...
```

- feloldás

```
// itt lép ki a kritikus szakaszból
sem_post(&mysem);
```

A termelő-fogyasztó probléma megoldása szemaforral

- Kölcsönös kizárás és precedencia biztosítása



Termelő

```
while () {
  T = termék_előállítás();
  P(raktár_mutex);
  elhelyez(raktár, T);
  V(raktár_mutex);
  V(van_termék);
}
```

Fogyasztó

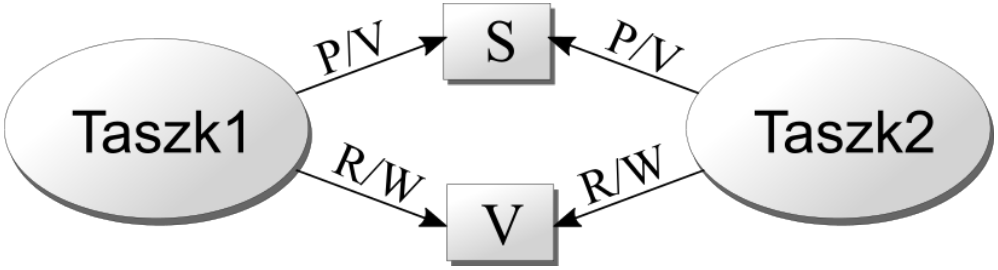
```
while () {
  P(van_termék);
  P(raktár_mutex);
  T = kivesz(raktár);
  V(raktár_mutex);
  termék_felhasználás(T)
}
```

Otthoni gyakorlás: szemaforral hogyan kezelhető a raktár túlcsordulásvédelme?

A szinkronizáció klasszikusai: író-olvasó probléma

- A probléma
  - közösen használt változó (V) konzisztenciájának biztosítása (PRAM)
  - egypéldányos erőforrás védelme

- A megoldás



Taszk1

```
P(S);
// változó írása, olvasása
V(S);
```

Taszk2

```
P(S);
// változó írása, olvasása
V(S);
```

Mi a helyzet, ha túl sok taszk van? Például: sok olvasó és néhány író.

A többszörös olvasók problémája

- A probléma
  - sok olvasó kevés író esetén a szemafor túl sok várakozást eredményez
  - felesleges blokkolni az olvasókat, ha a változó értéke nem módosul

- A megoldás
  - ne blokkoljuk az olvasókat, ha nincs folyamatban írás

Olvasás kezdete

```
P(reader_mutex);
++readerCount;
if (readerCount == 1) {
  P(writerLock);
}
V(reader_mutex);
```

Olvasás befejezése

```
P(reader_mutex);
--readerCount;
if (readerCount == 0) {
  V(writerLock);
}
V(reader_mutex);
```

Írás

```
P(writerLock);
// változó írása
V(writerLock);
```

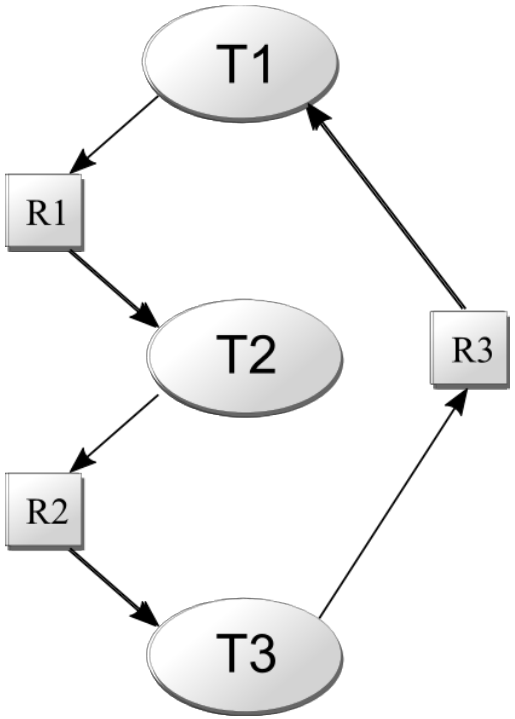
Így működik a ReaderWriterLock szemaforok segítségével.

Összetettebb kölcsönös kizárási példa

- A probléma: több erőforrás együttes védelme

```
T1
    P(R3);
    P(R1);
    V(R1);
    V(R3);
T2
    P(R1);
    P(R2);
    V(R2);
    V(R1);
T3
    P(R2);
    P(R3);
    V(R3);
    V(R2);
```

Az erőforrás-foglalási gráf



T1, T2 és T3 egyszerre kezdenek futni...

Mi történik?

Holtpont alakul ki.

A holtpont (deadlock)

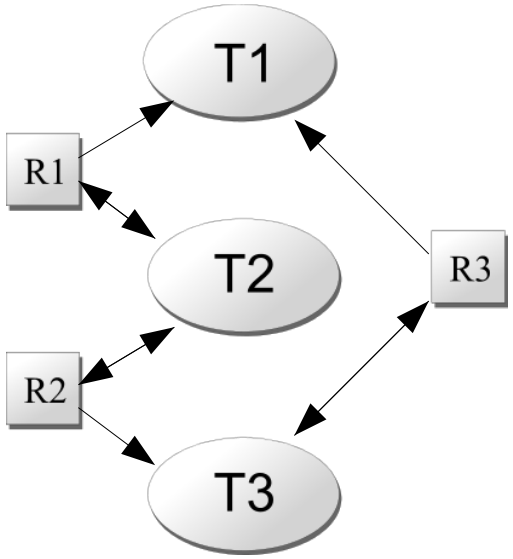
Taszkok egy $\mathcal{H}$ halmazában található valamennyi taszk olyan eseményre vár, amelyet csak $\mathcal{H}$-n belüli taszkok idézhetnek elő.

Minden lefutás sorrend esetén kialakul a holtpont?

Pl. először T1 lefut, T2 és T3 csak később indul

Mi történik, ha más sorrendben vannak az utasítások?

Pl. T1 fordítva foglalja le és szabadítja fel az erőforrásokat



T1

```
P(R3);
P(R1);
V(R1);
V(R3);
```

T2

```
P(R1);
P(R2);
V(R2);
V(R1);
```

T3

```
P(R2);
P(R3);
V(R3);
V(R2);
```

Holtpont kialakulásának feltételei

1. kölcsönös kizárás

legyenek kizárólagosan használható erőforrások

2. foglalva várakozás

valamelyik taszk egy erőforrást foglalva másikra várakozik

3. nincs erőszakos erőforrás-elvétel

a taszkok önszántunkból mondanak le erőforrásról, nem veszik el tőlük

4. körkörös várakozás

Létezik taszkoknak egy olyan T_1 a T_N sorozata, amelyre igaz az, hogy

T_i a T_{i+1} által birtokolt erőforrásra vár ($1 \leq i < N$), és T_N a T_1 által foglaltra vár

A rendszer állapotát az **erőforrásfoglalási-gráffal** modellezhetjük.

Mit kezdhetünk a holtponttal?

- Nem veszünk róla tudomást (strucc „algoritmus”)
  - ha nagyon kicsi a holtpont esélye
  - és nem okoz kritikus hibát

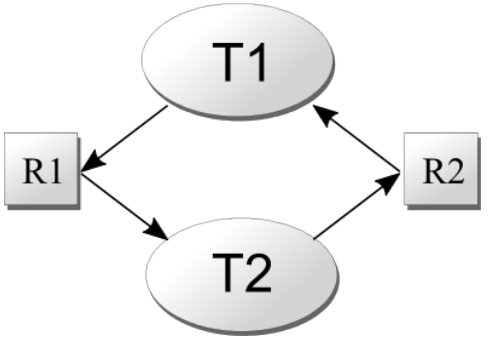
- Észrevesszük és megpróbáljuk kezelni
  - detektáljuk a holtpontot
  - feloldjuk
    - csak erőszakosan megy: leállítunk taszko(ka)t, elveszünk erőforrás(oka)t
    - ki? hogyan? mit eredményez?
    - a programozó talán tudja...

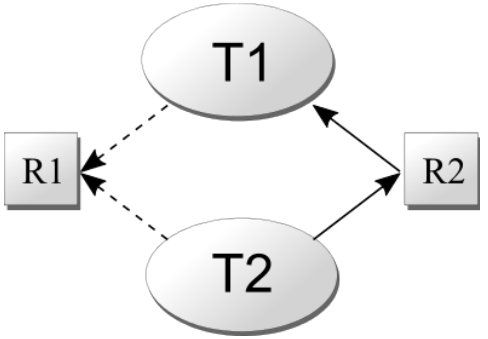
- Védekezünk ellene
  - **holtpontmentesre tervezzük**
    - valamelyik feltételt kizárjuk (melyiket lehet?) ← Ez a helyes út.
  - futásidőben ellenőrizzük a foglalásokat
    - még kialakulása előtt detektáljuk
    - **biztonságos állapot**: holtpont nélkül erőforrást allokálhatunk

Holtpont kialakulásának detektálása / megelőzése

- A rendszer kiindulási állapota biztonságos
  - nincs erőforrás-foglalás, nincs holtpont
  - az előrejelzéshez bekérjük a jövőbeli foglalásokat is

- Egypéldányos erőforrások esetén
  - kört keresünk az erőforrás-allokációs gráfban $O(N^2)$
  - **detektálás**
    - a jövőbeli foglalásokat nem vizsgáljuk
    - kör alakult ki → holtpont van
  - **előrejelzés**
    - a jövőbeli igényeket is vizsgáljuk
    - adott igény esetén (pl. R1 → T2 él)
      - kör alakulna ki → az állapot nem biztonságos ekkor az igényt elutasítjuk

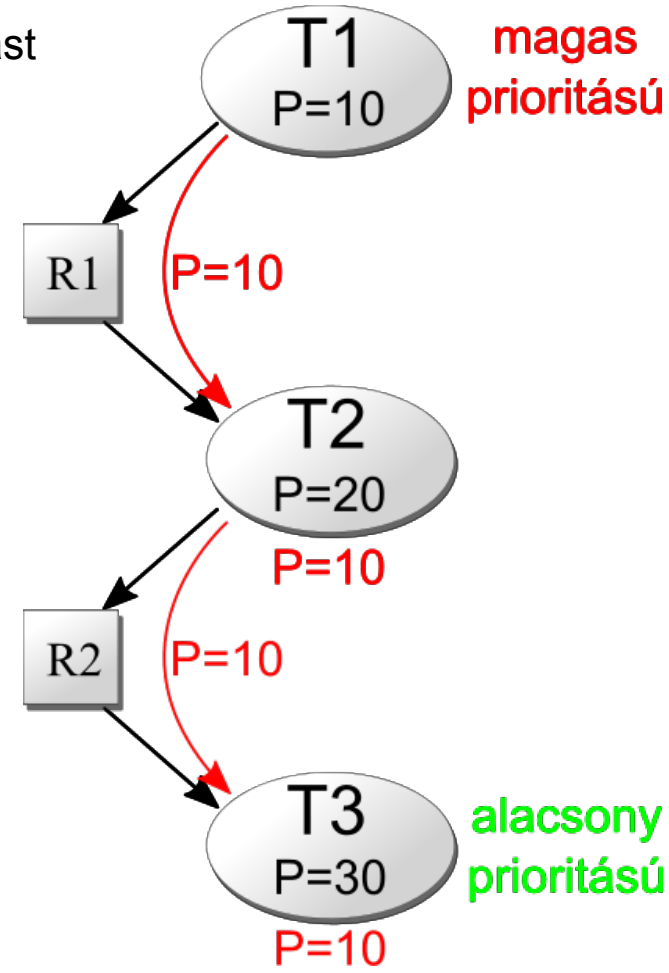




- Többpéldányos erőforrásokra
  - **bankár algoritmus** (lásd KK tankönyv)
  - $M * N^2$ komplexitású, ahol M az erőforrástípusok száma

A zárolás további problémái

- **Prioritásinverzió**
  - egy alacsony prioritású taszk birtokol egy erőforrást
  - egy magasabb prioritású várakozik
  - feloldása: örökölt prioritásokkal (lásd ütemezés)



- **Kiéheztetés** foglalással
  - egy taszk folyamatosan foglal egy erőforrást
  - az erőforrásra várók blokkolódnak
  - feloldása: a hibás működés javítása
- **Kiéheztetés** várakoztatással
  - nem FIFO várakozás esetén
    - pl. kérések prioritásos rendezése
  - a várakozási sorban „ragad” egy taszk
  - feloldása: pl. öregítéssel
- A szinkronizáció okozta teljesítményromlás (lásd korábban)
  - kezelése: optimista zárolás, zárolás- és várakozásmentes szinkronizáció

Optimista zárolás

- **Pesszimista zárolás:** mindig véd
- **Optimista zárolás:** nem zárol, de detektálja és korrigálja a hibát
- Tranzakció-alapú megvalósítása
  - BEGIN: feljegyzi a kiinduló állapotot
  - MODIFY: végrehajtja a műveleteket
  - VALIDATE: ellenőrzi, hogy valami meghiúsítja-e a műveletek konzisztenciáját
  - COMMIT: ha nincs gond, akkor zárja a műveleteket
  - ROLLBACK: ha problémát észlel, akkor visszalép a kiinduló állapotba
- Értékelése
  - kevés konfliktus esetén javul a teljesítmény
  - sok hiba esetén jelentősen romlik
- Megvalósítási példák
  - tranzakciós memória (szoftver és hardver, pl. Intel Haswell TSX és RTM + példák)
  - adatbázis-kezelők, programozási nyelvek (pl. Java, C/C++) konstrukciói stb.

Zárolás- és várakozásmentes algoritmusok (haladó)

- A **zárolásmentes** (lock-free) erőforrás-használat
  - az erőforráson (CPU, adat stb.) mindig történik valamilyen „hasznos” művelet
  - **garantálja az erőforrás teljes kihasználtságát**, nincs teljesítményvesztés

- **Várakozásmentes** (wait-free) erőforrás-használat
  - zárolásmentes + minden művelet véges időn belül végrehajtható
  - sokkal nehezebb megvalósítani

Kizárják a holtpont lehetőségét. Miért?

- Az algoritmusok átírhatók ily módon (lásd cikk)
  - gyakorlatilag azonban a várakozást esetenként jóval meghaladó költséggel
  - a kihívás hatékony algoritmusok kifejlesztése, adott adatstruktúrákra szabva:
    pl. várakozási sorok (ring buffer fifo) és alkalmazása, láncolt lista, lockless cache

- A gyakorlati megvalósítás kihívásai
  - Nem-atomi műveletek megvalósítása (pl. a `var++` három lépésből áll).
    Az optimista zárolás segíthet, akár CPU támogatással is.
  - A műveletek sorrendje sem garantált, CPU és a fordító is átrendezheti azokat.
    Instrukciókat kell adni számukra, hogy korlátozzák az átrendezést.

A szinkronizáció további formái

Lásd KK tankönyv elosztott rendszerekkel foglalkozó fejezetei

- Egyidejűség (randevú)
  - a taszkok megadott műveletei egyszerre kezdődjenek el
  - **kooperációs** séma, a részfeladatok végrehajtásának összehangolása kívánhatja
- Előírt végrehajtási sorrend (precedencia)
  - taszkok adott műveletei meghatározott sorrendben hajtódjanak végre
  - **kooperációs** séma, a részfeladatok végrehajtásának előírt sorrendje kívánja meg

Taszkok szinkronizációja (összefoglalás)

- PRAM modellhez kapcsolódó szinkronizáció
  - szálak között sokféle eszközzel lehetséges (lásd Szoftvertechnikák)
    - pl. egyszerű mutex a közös címtérben
  - folyamatok között szűkebb a kínálat
    - pl. szemaforok
- Közös erőforrások védelme
  - jellemzően számláló típusú szemaforokkal
- Kernel adatstruktúrák védelme
  - rövid idejű zárolásokra spinlock
- **Gondos tervezést igényel**
  - többféle hibaforrás (végtelen foglalás, holtpont)
- Teljesítményromlást okoz(hat)
  - ügyes tervezéssel csökkenthető
  - megfontolható az optimista zárolás és a zárolásmentes szinkronizáció