

Beágazott real-time operációs rendszerek

Naszály Gábor
<naszaly@mit.bme.hu>



Méréstechnika és
Információs Rendszerek
Tanszék

Fogalmak

Beágyazott operációs rendszerek

- A beágyazott rendszerek operációs rendszerei... ☺

Real-time operációs rendszerek

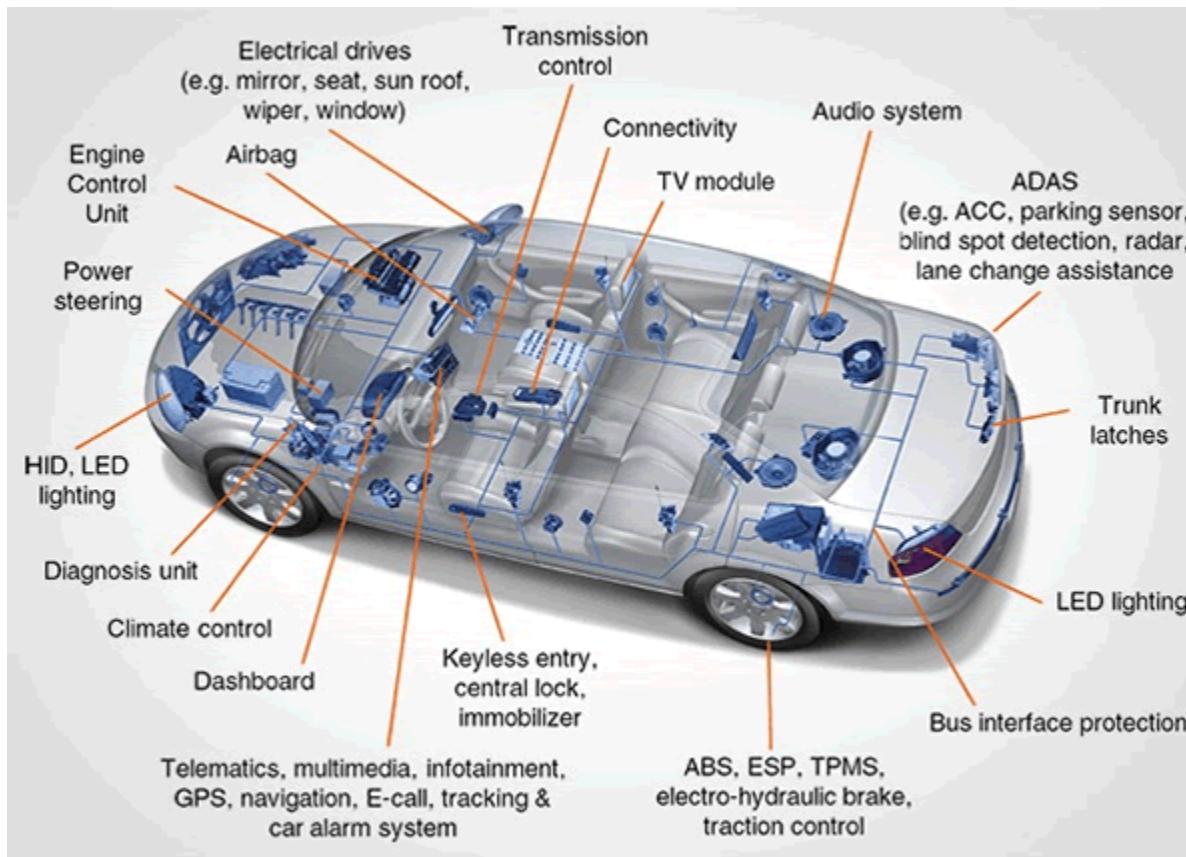
- A real-time rendszerek operációs rendszerei... 😊

Beágyazott rendszerek

- Pongyola definíció:
 - „Minden olyan számítógépes rendszer, ami nem egy hagyományos értelemben vett számítógép.”

Beágyazott rendszerek

■ Alkalmazási területek – Autóiparipar



Beágyazott rendszerek

- Alkalmazási területek – Szórakoztató elektronika



Beágyazott rendszerek

- Alkalmazási területek – Orvosi berendezések



Beágyazott rendszerek

■ Alkalmazási területek – Ház tartási berendezések



Beágyazott rendszerek

■ Alkalmazási területek – Mérőberendezések



Beágyazott rendszerek

- Alkalmazási területek – Hálózati berendezések



Beágyazott rendszerek

■ Alkalmazási területek – Úrkutatás



Beágyazott rendszerek

- Pontosabb definíció:
- Olyan **speciális számítógépes rendszerek**, amelyeket egy jól meghatározott feladatra találtak ki
- Ezen feladat ellátása érdekében a **külvilággal intenzív információs kapcsolatban állnak**
 - Található bennük valamilyen **feldolgozó egység**
 - Érzékelik a külvilág bizonyos paramétereit (**szenzorok**), és gyakran be is avatkoznak abba (**beavatkozók**)
 - A gépi komponensek között különféle **kommunikációs interfészek**, protokollokon keresztül áramlik az információ
 - Biztosítanak valamilyen **felhasználói felületet** (humán operátor számára kijelzők, kezelő szervek)

Beágyazott rendszerek

- A klasszikus értelemben vett beágyazott rendszerek általában nem személyi számítógépek.
- De vannak kivételek...

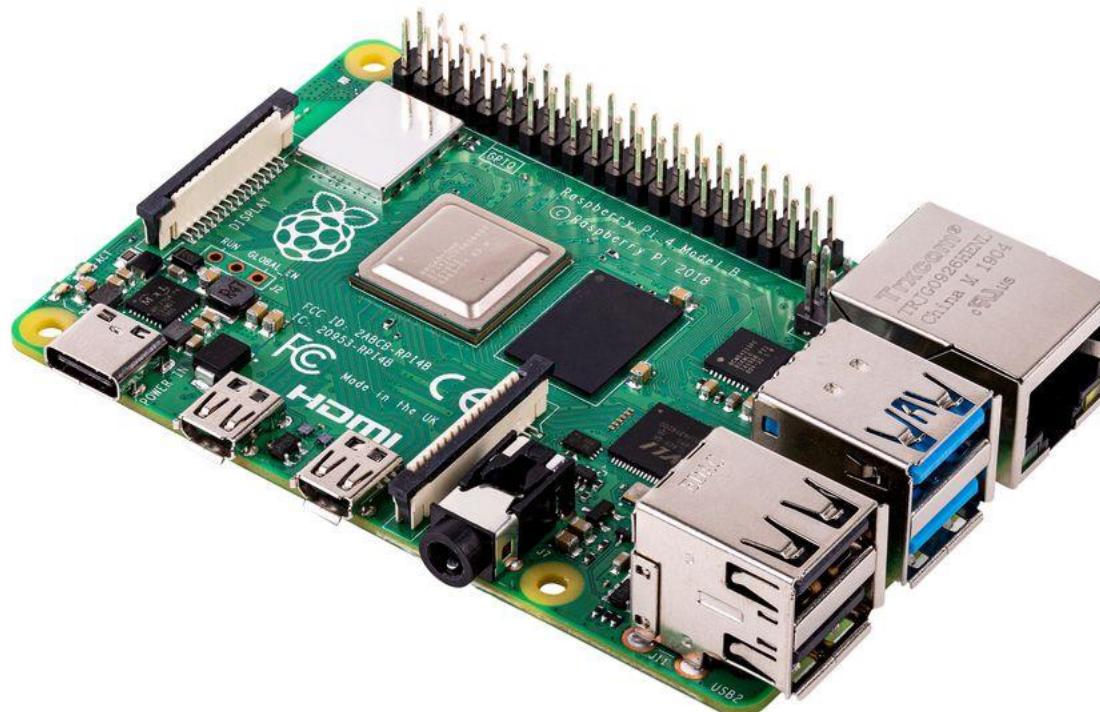
Beágyazott rendszerek

- Nem klasszikus értelemben vett (high-end) beágyazott rendszerek – Ipari PC-k



Beágyazott rendszerek

- Nem klasszikus értelemben vett (high-end) beágyazott rendszerek – Kártya méretű PC-k



Beágyazott rendszerek

- Nem klasszikus értelemben vett (high-end) beágyazott rendszerek – Okostelefon?

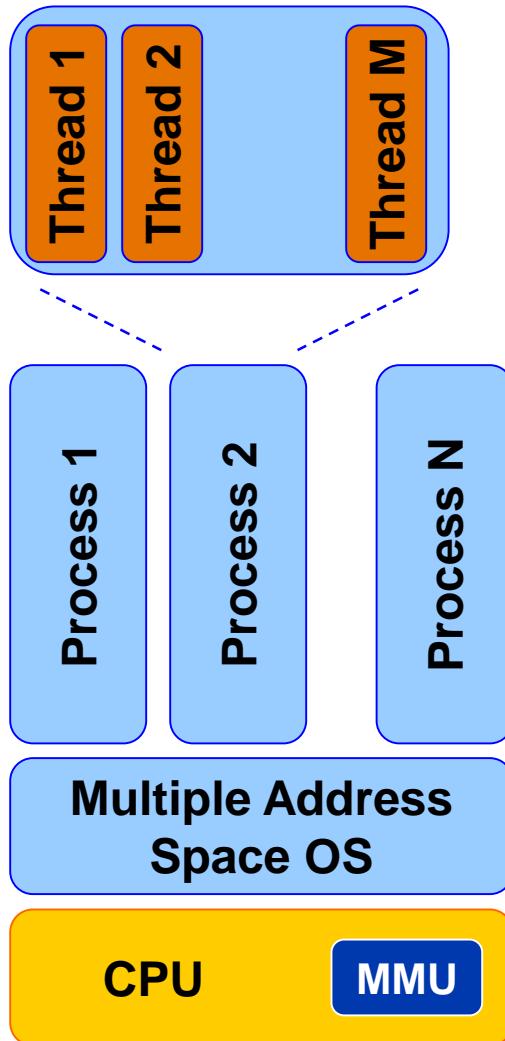


Beágyazott rendszerek

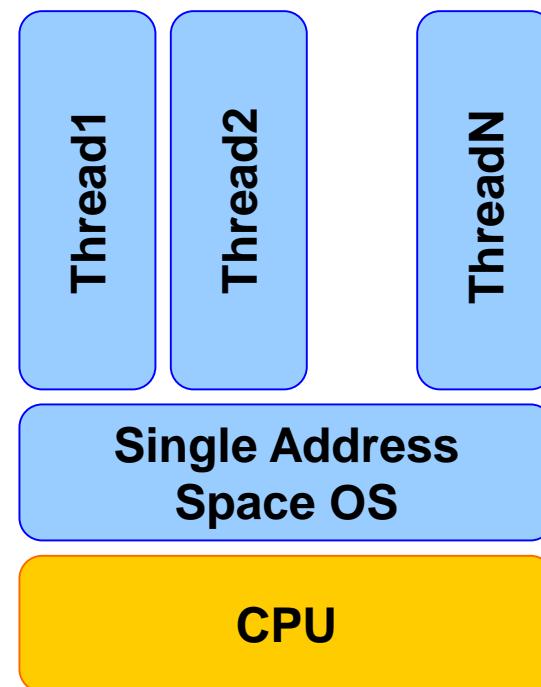
	Klasszikus	High-end
Processzor órajel	~10 – x100 MHz	~ xGHz
Processzor védelmi szintek	Nincs	Van
MMU (virtuális tárkezelés)	Nincs	Van
Folyamatok, szálak?	Csak szálak	Folyamatok (és bennük szálak)
Linux futhat rajtuk?	Nem	Igen

Beágyazott rendszerek

Virtuális memória MMU-val



Csak fizikai memória
MMU nélkül



Beágyazott rendszerek

- **Rengeteg belső periféria a CPU-n kívül:**
 - Memóriák: program (flash), adat (SRAM)
 - Analóg-Digitál, Digitál-Analóg átalakítók (ADC, DAC)
 - Időzítő áramkörök (timers)
 - Kommunikációs interfészek
 - Egyszerűbbek: U(SART), I2C, SPI
 - Bonyolultabbak: USB (device, host), Ethernet (100M)
 - Terület specifikus: CAN, LIN, FlexRay (ezek autóipari buszok)
 - Általános célú I/O (General Purpose I/O, GPIO)
 - Egyszerű beavatkozások (pl. LED kigyújtása, nyomógomb beolvasása)
 - HW-ből nem támogatott kommunikáció SW-es megvalósítása

Real-time rendszerek

- A rendszer adott eseményre adott időn belül válaszol
- Az adott időnek nem feltétlenül kell kicsinek lennie (bár gyakran az)

Real-time rendszerek

- Csoportosításuk (szigorú módszer)
 - Egy rendszer vagy real-time (azaz betartja a határidőket), vagy nem, nincs középút. (Lásd jog: pl. nem szabad elvenni más tulajdonát, nincs olyan, hogy „csak kicsit loptam”.)

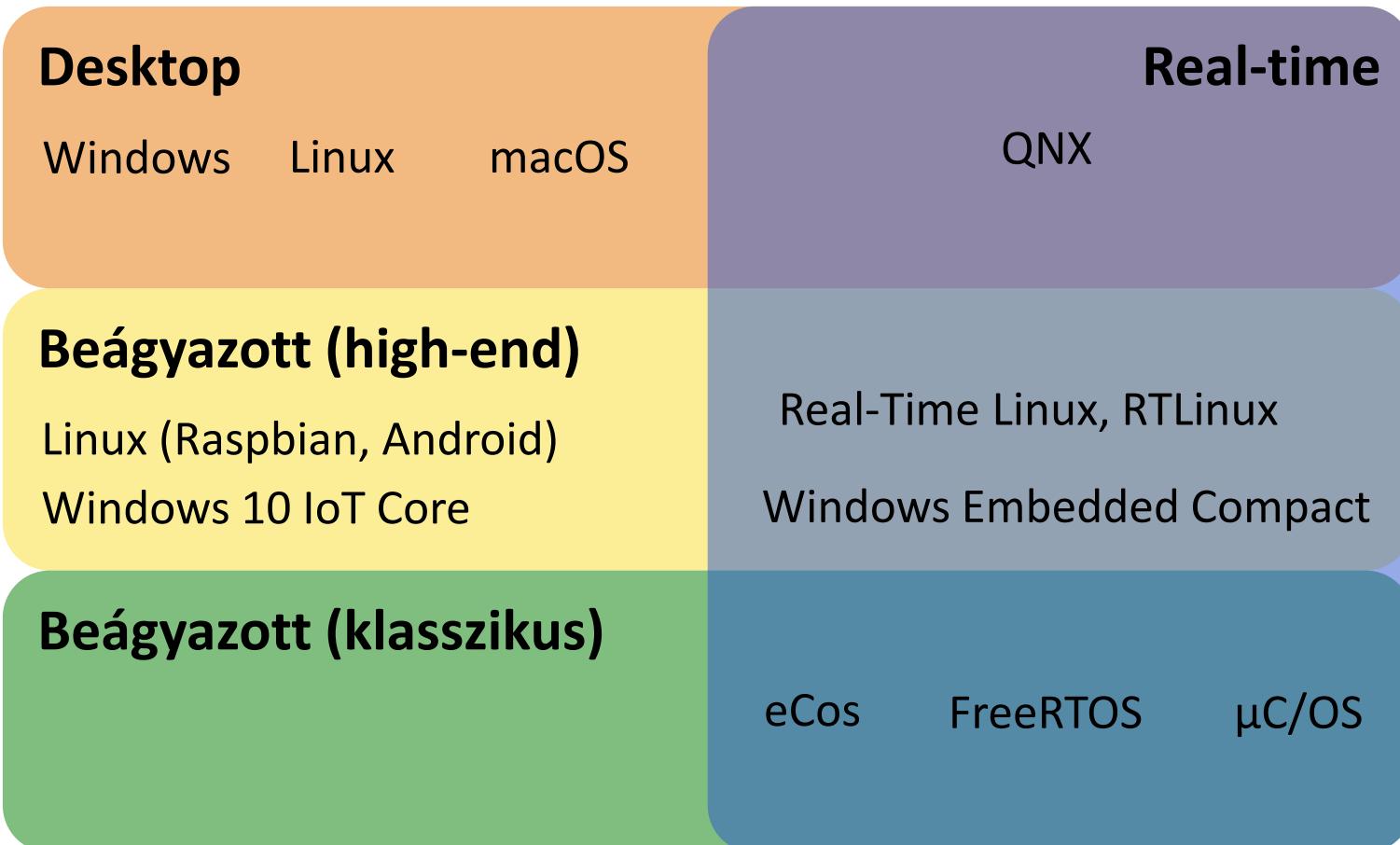
Real-time rendszerek

- Csoportosításuk (megengedőbb módszer)
 - Hard real-time rendszer
 - Mindent meg kell tennünk, hogy a rendszer a határidőket 1 valószínűsséggel tudja tartani.
 - A határidő elmulasztása katasztrófát okozhat (lásd légitöröklikédés).
 - Soft real-time rendszer
 - Próbáljuk úgy megvalósítani a rendszert, hogy közel 1 valószínűsséggel tudjon határidőn belül válaszolni a kérésekre.
 - A határidő elmulasztása káros, de nem végzetes. (Pl. az 5 másodperces azonnali átutalás 1 óra múlva teljesül.)

Beágyazott és real-time rendszerek

- A legtöbb (klasszikus értelemben vett) beágyazott rendszer real-time követelményeknek kell, hogy eleget tegyen.
- Így a két fogalom metszete nagy.

Operációs rendszerek csoportosítása



Real-time operációs rendszerek

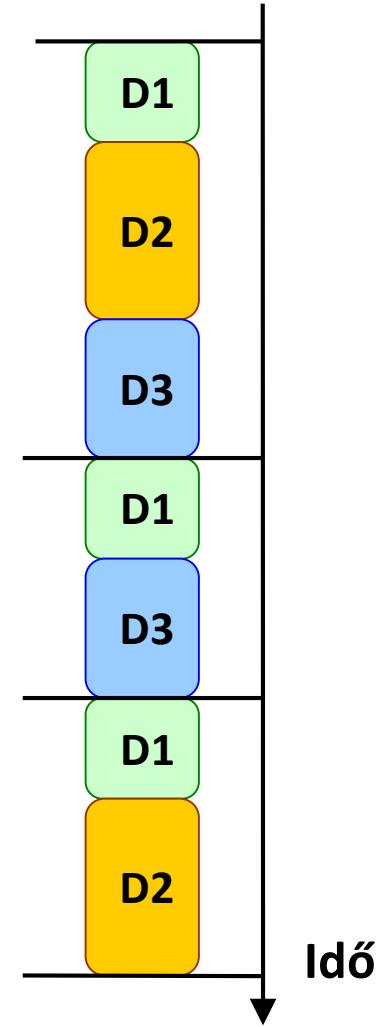
A klasszikus értelemben vett beágyazott rendszerekhez

Út az RTOS (mint szoftver architektúra) felé

- Kezdetben assembly nyelven programozták a beágyazott rendszereket
- Később a C vált uralkodójává (csak az maradt assemblyben, amit nem lehetett máshogy megvalósítani, vagy így volt hatékonyabb)
- Kezdetben nem használtak RTOS-t (ma sem mindig). A feladat bonyolultsága és egyéb követelmények szabják meg, végül milyen SW architektúrát használunk.
- Nézzünk meg egy párat a legegyszerűbbtől kezdve

Ciklikus programszervezés

```
void main(void) {  
  
    while(1) {  
  
        if ( !! Device 1 needs service ) {  
            !! Handle Device 1 and its data  
        }  
  
        if ( !! Device 2 needs service ) {  
            !! Handle Device 2 and its data  
        }  
  
        if ( !! Device 3 needs service ) {  
            !! Handle Device 3 and its data  
        }  
  
        ...  
    }  
}
```



Ciklikus programszervezés

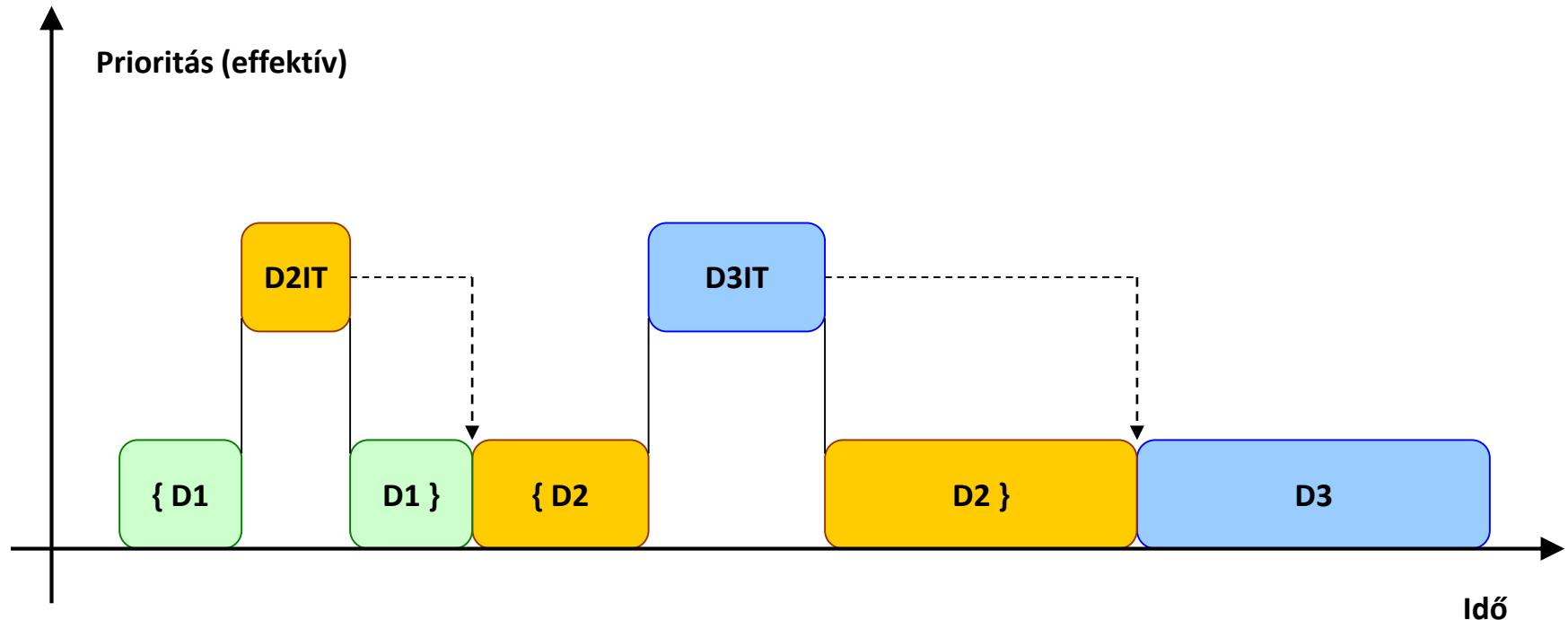
- A legegyszerűbb (nincs megszakítás, nincs közös erőforrás probléma)
- Nagy a szórása a válaszidőnek
- Worst-case válaszidő: a feladatok válaszidejeinek összege (azaz lineárisan nő a taszkok számával)
- Egyes feladatok tekintetében javítható a válaszidő, ha többször kérdezzük le őket a főhurokban
- „Törékeny” az architektúra (új taszk hozzáadása felboríthat minden)

Ciklikus programszervezés (IT-kkel)

```
BOOL Device1_flag = FALSE;  
BOOL Device2_flag = FALSE;  
BOOL Device3_flag = FALSE;  
  
void interrupt Device1_ISR (void) {  
    !! Handle Device 1 time critical part  
    Device1_flag = TRUE;  
}  
  
void interrupt Device2_ISR (void) {  
    !! Handle Device 2 time critical part  
    Device2_flag = TRUE;  
}  
  
void interrupt Device3_ISR (void) {  
    !! Handle Device 3 time critical part  
    Device3_flag = TRUE;  
}
```

```
void main (void)  
{  
    while(1)  
    {  
        if ( Device1_flag ) {  
            Device1_flag = FALSE;  
            !! Handle Device 1 and its data  
        }  
  
        if (Device2_flag ) {  
            Device2_flag = FALSE;  
            !! Handle Device 2 and its data  
        }  
  
        if (Device3_flag ) {  
            Device3_flag = FALSE;  
            !! Handle Device 3 and its data  
        }  
        ...  
    }  
}
```

Ciklikus programszervezés (IT-kkel)



Ciklikus programszervezés (IT-kkel)

- Kicsit jobban kezeli az időkritikus részeket (a megszakítások magasabb effektív prioritáson futnak)
- Ha lehetséges interruptokhoz prioritást rendelni, még tovább lehet finomítani az architektúrát
- Jelentkezhet a közös erőforrások problémája
- Nagy a szórása a válaszidőnek
- Worst-case válaszidő: az összes feladat válaszideje + a megszakítások (arányos a taszkok számával)
- Egyes feladatok tekintetében javítható a válaszidő, ha többször kérdezzük le őket a főhurokban
- „Törékeny” az architektúra (új taszk hozzáadása felboríthat minden)

Függvény-sor alapú ütemezés

```
!! Define queue of function pointers

void interrupt Device1 (void)
{
    !! Handle Device 1 time critical part
    !! Put Device1_func to call queue
}

void interrupt Device2 (void)
{
    !! Handle Device 2 time critical part
    !! Put Device2_func to call queue
}

void interrupt Device3 (void)
{
    !! Handle Device 3 time critical part
    !! Put Device3_func to call queue
}
```

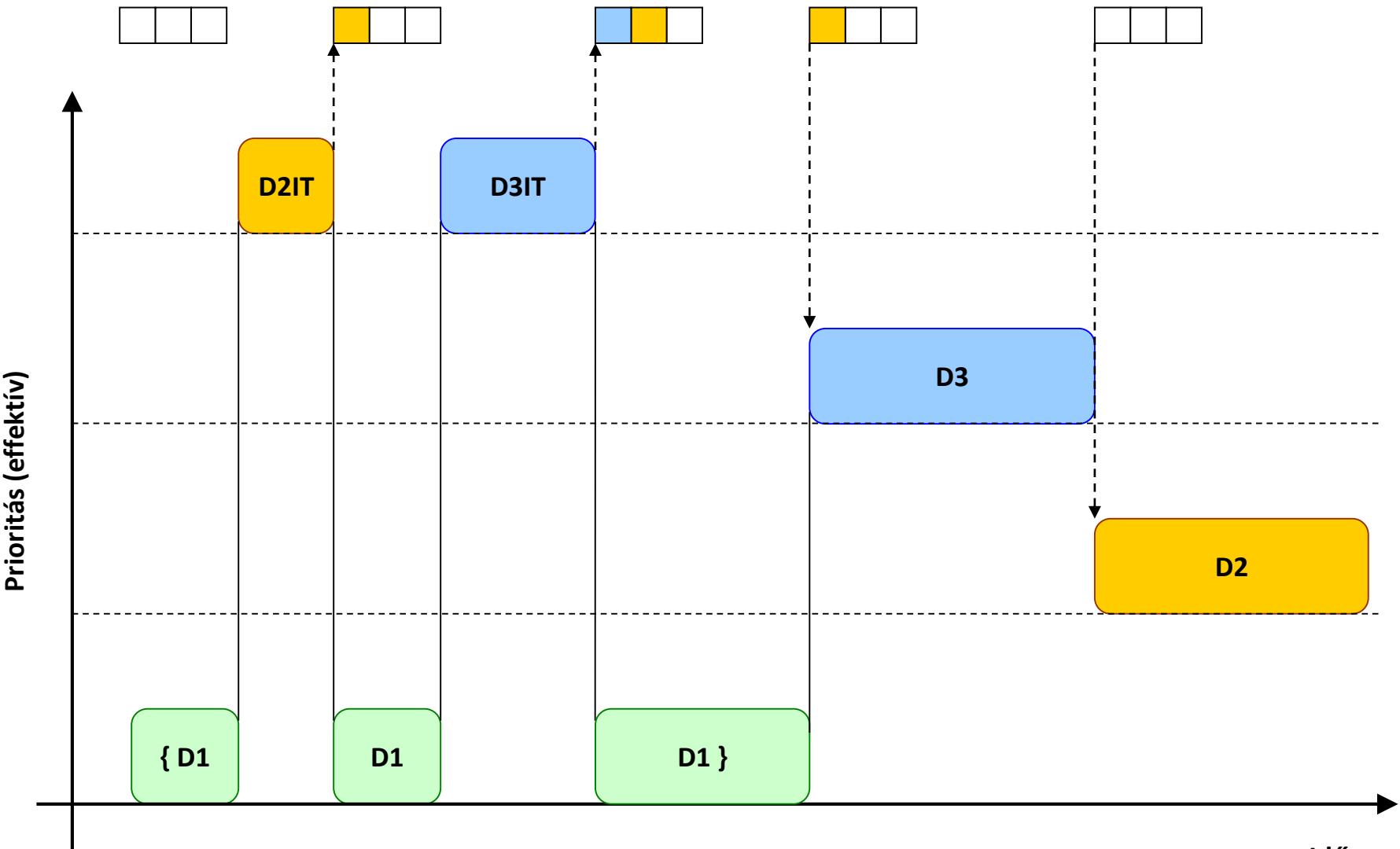
```
void main (void)
{
    while(1)
    {
        while(!! Function queue not empty)
            !! Call first from queue
    }
}

void Device1_func (void)
{   !! Handle Device 1 }

void Device2_func (void)
{   !! Handle Device 2 }

void Device3_func (void)
{   !! Handle Device 3 }
```

Függvény-sor alapú ütemezés



Függvény-sor alapú ütemezés

- Képes a prioritások kezelésére (mind taszk, mind IT szinten)
- Jelentkezhet a közös erőforrások problémája az IT és a főprogram között
- Worst-case válaszidő (a legmagasabb prioritású feladatra) = a leghosszabb taszk válaszideje + IT
- A worst-case válaszidő nem nő lineárisan a taszkok számával
- A válaszidő jóval kevésbé szór
- Egy új feladat nem borítja fel az eddigi időzítést
- Nem preemptív

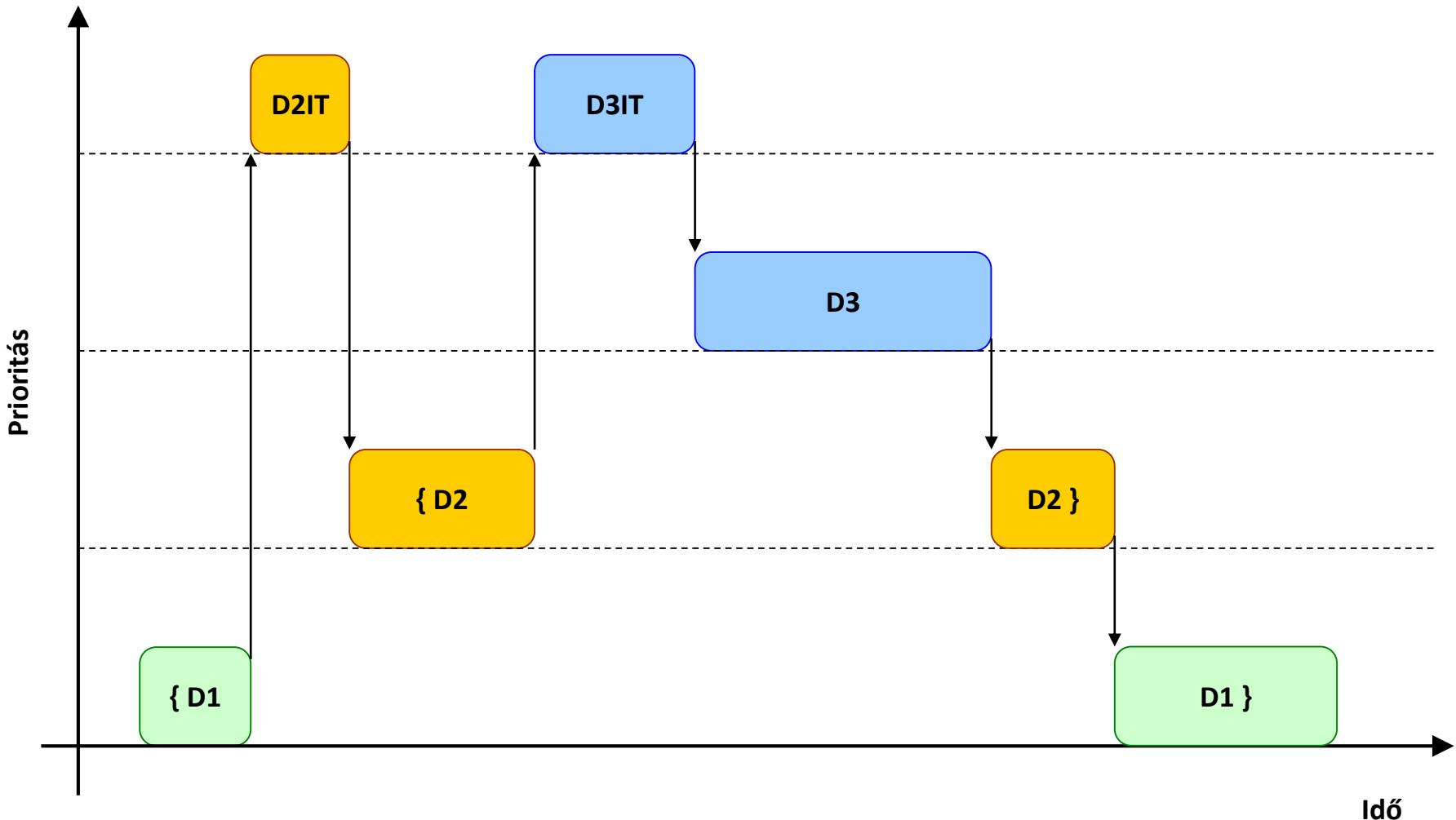
RTOS architektúra (preemptív)

```
void interrupt Device1 (void) {  
    !! Handle Device 1 time critical part  
    !! Set signal to Device1_task  
}  
  
void interrupt Device2 (void) {  
    !! Handle Device 2 time critical part  
    !! Set signal to Device2_task  
}  
  
void interrupt Device3 (void) {  
    !! Handle Device 3 time critical part  
    !! Set signal to Device3_task  
}
```

```
void Device1_task (void) {  
    while (1) {  
        !! Wait for signal to Device1_task  
        !! Handle Device 1  
    }  
}  
  
void Device2_task (void) {  
    while (1) {  
        !! Wait for signal to Device2_task  
        !! Handle Device 2  
    }  
}  
  
void Device3_task (void) {  
    while (1) {  
        !! Wait for signal to Device3_task  
        !! Handle Device 3  
    }  
}
```

```
void main (void) {  
    !! Initialize OS  
    !! Start OS scheduler  
}
```

RTOS architektúra (preemptív)



RTOS architektúra (preemptív)

- Képes a prioritások kezelésére (mind taszk, mind IT szinten)
- Jelentkezhet a közös erőforrások problémája az IT és a főprogram, valamint az egyes taszkok között is
- Worst-case válaszidő (a legmagasabb prioritású feladatra) = a taszk váltási idő + IT (mindkettő kicsi)
- A worst-case válaszidő nem nő az új feladatok hozzáadásával
- A válaszidő szórása nagyon alacsony
- Kód overhead

RTOS architektúra (preemptív)

- Tipikus képviselők ebben a kategóriában
 - FreeRTOS (eredetileg Real Time Engineering Ltd., ma Amazon Web Services)
 - μC/OS-II, μC/OS-III (eredetileg Micrium, ma Silabs)
 - eCOS
 - Keil RTX
 - Freescale/NXP MQX
 - Express Logic ThreadX
 - TI DSP/BIOS és TI-RTOS
 - és sokan mások...

FreeRTOS

Története

- A FreeRTOS >15 éves múltra tekint vissza:
 - Eredetileg **Richard Barry** kezdte el fejleszteni 2003 körül
 - Később Richard cége, a **Real Time Engineers Ltd.** folytatta a fejlesztését
 - 2017-ben a projektet az **Amazon Web Services (AWS)** vette át
 - Richard továbbra is dolgozik a projekten az azt gondozó AWS csapat tagjaként



Licencelés

- MIT Open Source License:
 - Ingyenes és nyílt forrású
 - Kereskedelmi alkalmazásban használható
 - Jogdíj mentes
 - Technikai támogatás van, de fizetős
 - Garancia nincs
 - Jogi védelem nincs
- Megjegyzés:
 - A 10-es verzió előtt a GNU GPLv2 licence egy módosított (könnyítést tartalmazó) verziója volt érvényben.

A FreeRTOS család tagjai

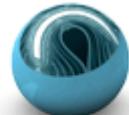


Amazon Web Services:

Amazon FreeRTOS (a:FreeRTOS)



HighIntegritySystems:



OPENRTOS®

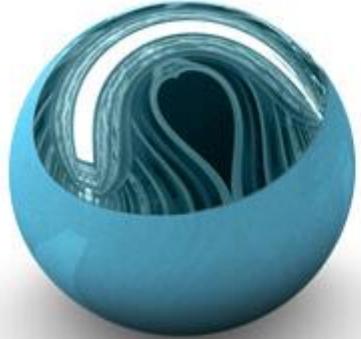


SAFERTOS®



SAFERTOS CORE

A FreeRTOS család tagjai



OPENRTOS®

- A WITTENSTEIN high integrity systems áll mögötte
- A FreeRTOS egy kereskedelmi licencelésű változata
- Nem ingyenes
- Cserébe adnak garanciát és IP védelmet

A FreeRTOS család tagjai



SAFERTOS®

- A WITTENSTEIN high integrity systems áll mögötte
- A FreeRTOS funkcionális modelljén alapszik, de nem egyezik meg vele.
- Teljes mértékben átdolgozták és az alapoktól építették fel biztonság kritikus elveket szem előtt tartva. Nincs dinamikus memória foglalás, alaposabb paraméter ellenőrzés OS hívásoknál.

A FreeRTOS család tagjai



Amazon FreeRTOS (a:FreeRTOS)

- Az Amazon Web Services fejleszti
- FreeRTOS kernel + egyéb szoftver csomagok:
 - Kommunikáció (pl. TCP/IP, MQTT)
 - Titkosítás (pl. TLS)
- Cél: a beágyazott eszközök számára könnyű és biztonságos legyen akár az egymás közti, akár egy felhő felé a kommunikáció, illetve a távoli frissítés.
- MIT licenc

A FreeRTOS főbb jellemzői

- Forráskódja könnyű portolhatóságra lett tervezve (jelenleg hivatalosan több mint 35 beágyazott architektúrára támogatott)
- Tervezésénél fontos szempont volt a kis méret, egyszerűség és könnyű használat
- A forráskódhoz rengeteg demó alkalmazást mellékelnek a kezdeti lépések könnyebbé tételehez

A FreeRTOS főbb jellemzői

- Az ütemezés egysége:
 - Taszk (a legelterjedtebb)
 - Ko-rutin (taszknál egyszerűbb eszköz, legacy)
 - Hibrid (taszk + ko-rutin)
- A taszkok ütemezése:
 - Prioritásos
 - Preemptív (alapértelmezett)
 - Round-robin time-slicing segítségével az azonos prioritási szinteken (ez az alapértelmezett)

Fontosabb OS szolgáltatások

- **taszkok** (tasks)
- **bináris szemaforok** (binary semaphores)
- **mutexek** (mutexes)
- **sorok** (queues)
- **esemény jelző bitek** (event groups /or event flags/)
- **szoftver időzítők** (software timers)
- **memória kezelés** (memory management)

Ritkábban használt OS szolgáltatások

- **ko-rutinok** (co-routines)
- **óraütés nélküli üresjárás** (tickless idle mode)
- **számláló szemaforok** (counting semaphores)
- **rekurzív mutexek** (recursive mutexes)
- **közvetlen taszk értesítések** (direct to task notifications)
- **bájt folyam** és változó méretű **üzenet pufferek** (stream buffers és message buffers)
- **várakozás egyszerre több RTOS objektumra** (blocking on multiple RTOS objects)

Ritkábban használt OS szolgáltatások

- **kicsatolások az RTOS kódjából** (hook functions)
- **TLS** (thread local storage pointers)
- **verem túlcsordulás detektálás** (stack overflow detection)
- **nyomkövetés** (trace features)
- **futás idejű statisztikák** (run time statistics)
- **MPU támogatás** (memory protection support)

A FreeRTOS felépítése

Alkalmazás

FreeRTOSConfig.h

FreeRTOS: architektúra független kód

list.c	<i>stream_buffer.c</i>	list.h	<i>stream_buffer.h</i>
tasks.c		task.h	<i>message_buffer.h</i>
queue.c		queue.h	semphr.h
<i>event_groups.c</i>		<i>event_groups.h</i>	
<i>timers.c</i>		<i>timers.h</i>	
<i>croutine.c</i>		<i>croutine.h</i>	FreeRTOS.h

FreeRTOS: architektúra függő kód

port.c portmacro.h

FreeRTOS: mem. man.

heap_(1-5).c

Szoftver

CPU

Időzítő

Hardver

Konfigurációs lehetőségek

- **FreeRTOSConfig.h** fejléc fájlban számos **#define** sor található
 - Vannak ki-/bekapcsolható funkciók, pl.:
 - **#define configUSE_PREEMPTION** (1)
 - **#define configUSE_MUTEXES** (1)
 - **#define configIDLE_SHOULD_YIELD** (0)
 - ...
 - Vannak számszerűsíthető konfigurációs opciók is, pl.:
 - **#define configTICK_RATE_HZ** (100)
 - **#define configMAX_PRIORITIES** (6)
 - ...

Konfigurációs lehetőségek

- Az OS kódja használja a konfigurációs definíciókat:

```
/* A task being unblocked cannot cause an immediate
context switch if preemption is turned off. */
#if ( configUSE_PREEMPTION == 1 )
{
    /* Preemption is on, but a context switch should
only be performed if the unblocked task has a
priority that is equal to or higher than the
currently executing task. */
    if( pxTCB->uxPriority >= pxCurrentTCB->uxPriority )
    {
        xSwitchRequired = pdTRUE;
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }
}
#endif /* configUSE_PREEMPTION */
```

Részlet a tasks.c forrás fájlból

Konfigurációs lehetőségek

- Az OS kódja használja a konfigurációs definíciókat:

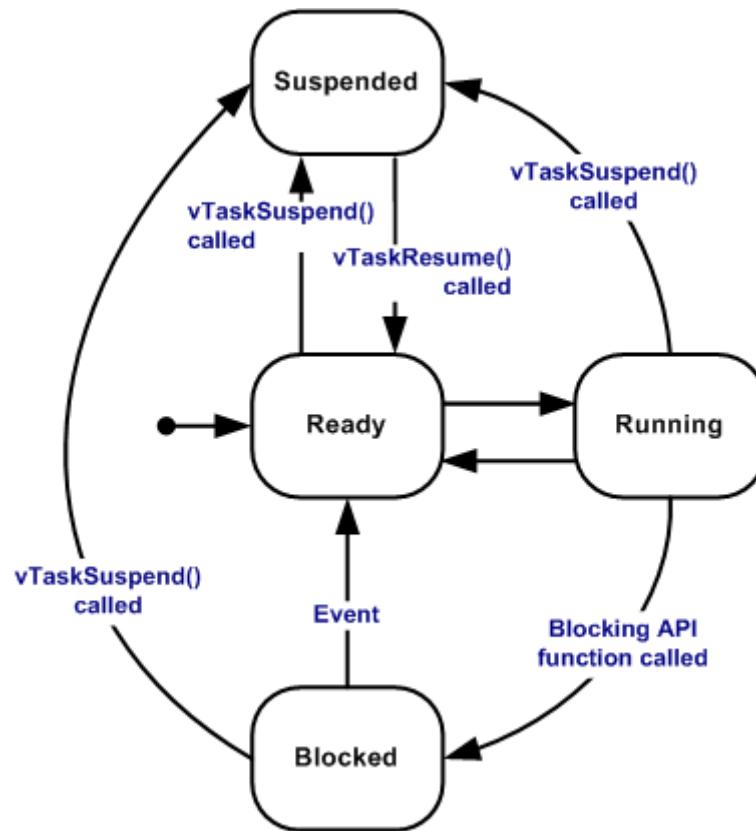
```
/* Setup the systick timer to generate the tick interrupts at the
 * required frequency. */
__attribute__(( weak )) void vPortSetupTimerInterrupt( void )
{
    /* Calculate the constants required to configure the tick interrupt */

    /* Stop and clear the SysTick. */
    portNVIC_SYSTICK_CTRL_REG = 0UL;
    portNVIC_SYSTICK_CURRENT_VALUE_REG = 0UL;

    /* Configure SysTick to interrupt at the requested rate. */
    portNVIC_SYSTICK_LOAD_REG =
        ( configSYSTICK_CLOCK_HZ / configTICK_RATE_HZ ) - 1UL;
    portNVIC_SYSTICK_CTRL_REG =
        ( portNVIC_SYSTICK_CLK_BIT |
        portNVIC_SYSTICK_INT_BIT |
        portNVIC_SYSTICK_ENABLE_BIT );
}
```

Részlet a /portable/GCC/ARM_CM3/port.c forrás fájlból (az olvashatóság kedvéért picit átszerkesztve)

Task állapotok



Tipikus alkalmazás

■ **main():**

- Inicializálás
- Taszkok létrehozása → xTaskCreate()
- Az ütemező elindítása → vTaskStartScheduler()
 - Ez a függvény vissza már nem tér a main()-be. Az OS minden futásra kész állapotban, akkor a beépített idle taszk fog futni.

Tipikus alkalmazás

■ Taszkok

- Megvalósítás függvényekben
- Kétféle taszk szervezés lehetséges:
 - Egyszeri lefutású:
 - A taszk egyszer fut le
 - Futása során kiválthat olyan eseményeket, amelyek más taszkok futását előidézik
 - A futása végén törli magát → vTaskDelete()
 - Végtelen ciklusú
 - Az elején lehet egy opcionális inicializációs szakasz
 - Utána egy végtelen ciklus található
 - Amiben el kell helyezni olyan OS hívást, aminek hatására várakozó állapotba kerül (hogy az alacsonyabb prioritásúakat ne éheztesse ki)

Egyszerű FreeRTOS példa alkalmazás

Demo

A megvalósított funkció

- Két taszk: egy magas és egy alacsony prioritású
- Végtelen ciklus szervezésűek
- A cikluson belül:
 - Kiírnak egy rövid szöveget a standard kimenetre („Hi” illetve „Lo”)
 - Majd időre várakozó állapotba teszik magukat (a magas prioritású 1, az alacsony fél másodpercre)

Az alkalmazás kódja

```
int main(void) {
    [...] // Init stdio: use UART0

    xTaskCreate(
        prvTaskHi,
        "Hi",
        mainTASK_HI_STACK_SIZE,
        NULL,
        mainTASK_HI_PRIORITY,
        NULL);

    xTaskCreate(
        prvTaskLo,
        "Lo",
        mainTASK_LO_STACK_SIZE,
        NULL,
        mainTASK_LO_PRIORITY,
        NULL);

    vTaskStartScheduler();
    return 0; // Never reached
}
```

```
static void prvTaskHi(void *pvParam) {
    while (1) {
        printf("Hi\n");
        vTaskDelay(configTICK_RATE_HZ);
    }
}

static void prvTaskLo(void *pvParam) {
    while (1) {
        printf("Lo\n");
        vTaskDelay(configTICK_RATE_HZ / 2);
    }
}
```

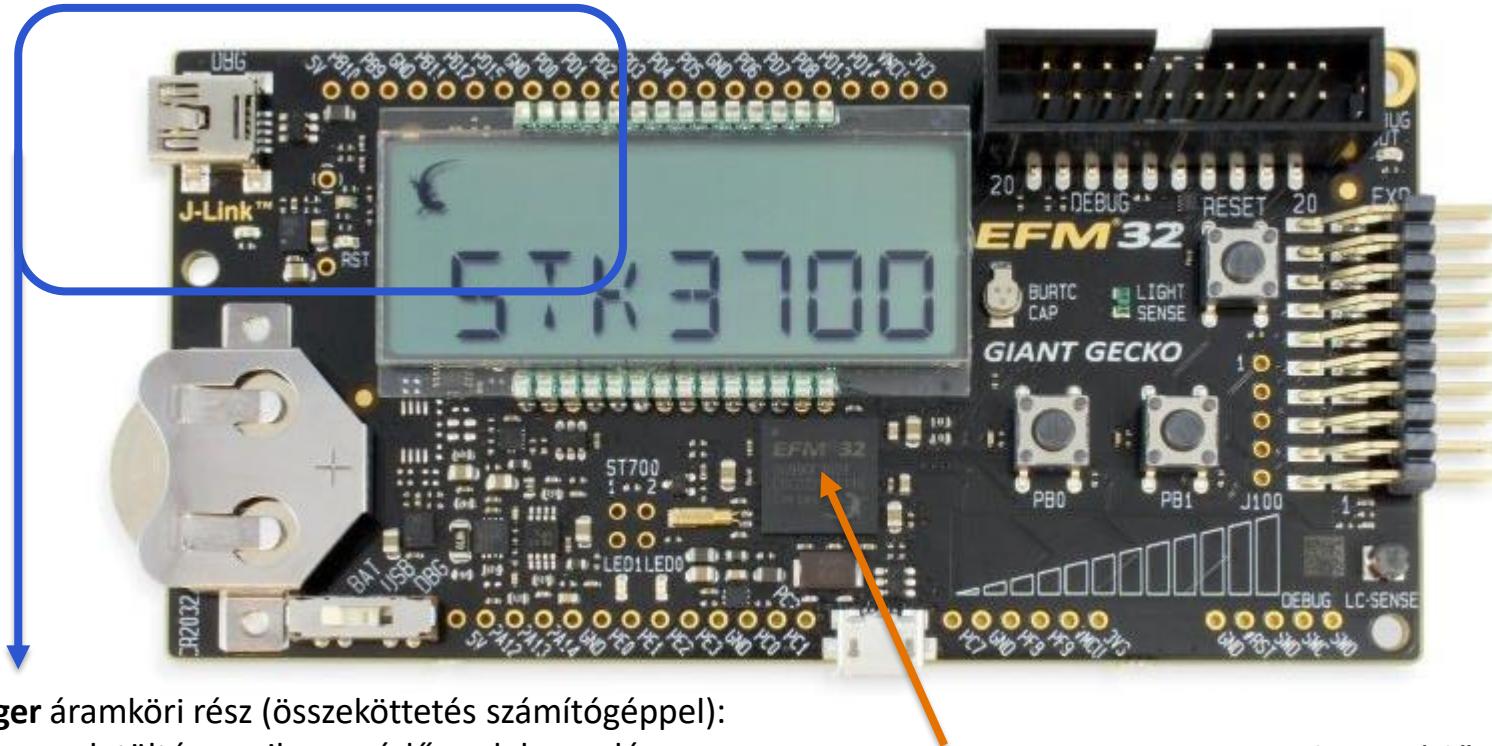
prvTaskHi, prvTaskLo: Függvény pointerek a taszkok kódjának otthonat adó függvényekre.

mainTASK_HI_STACK_SIZE, mainTASK_LO_STACK_SIZE: A taszkok vermének méretét megadó konstansok. Fontos, hogy elegendően nagyok legyenek. Beágyazott környezetben a **printf()** relatíve nagy verem fogyasztó lehet...

mainTASK_HI_PRIORITY, mainTASK_LO_PRIORITY: A taszkok prioritása: az alacsonyé 1, a magasé 2 (az idle taszk prioritása 0).

A felhasznált hardver

- A felhasznált eszköz a laborokon is szereplő Silicon Labs EFM32 Giant Gecko Starter Kit (STK3700):



Debugger áramköri rész (összeköttetés számítógéppel):

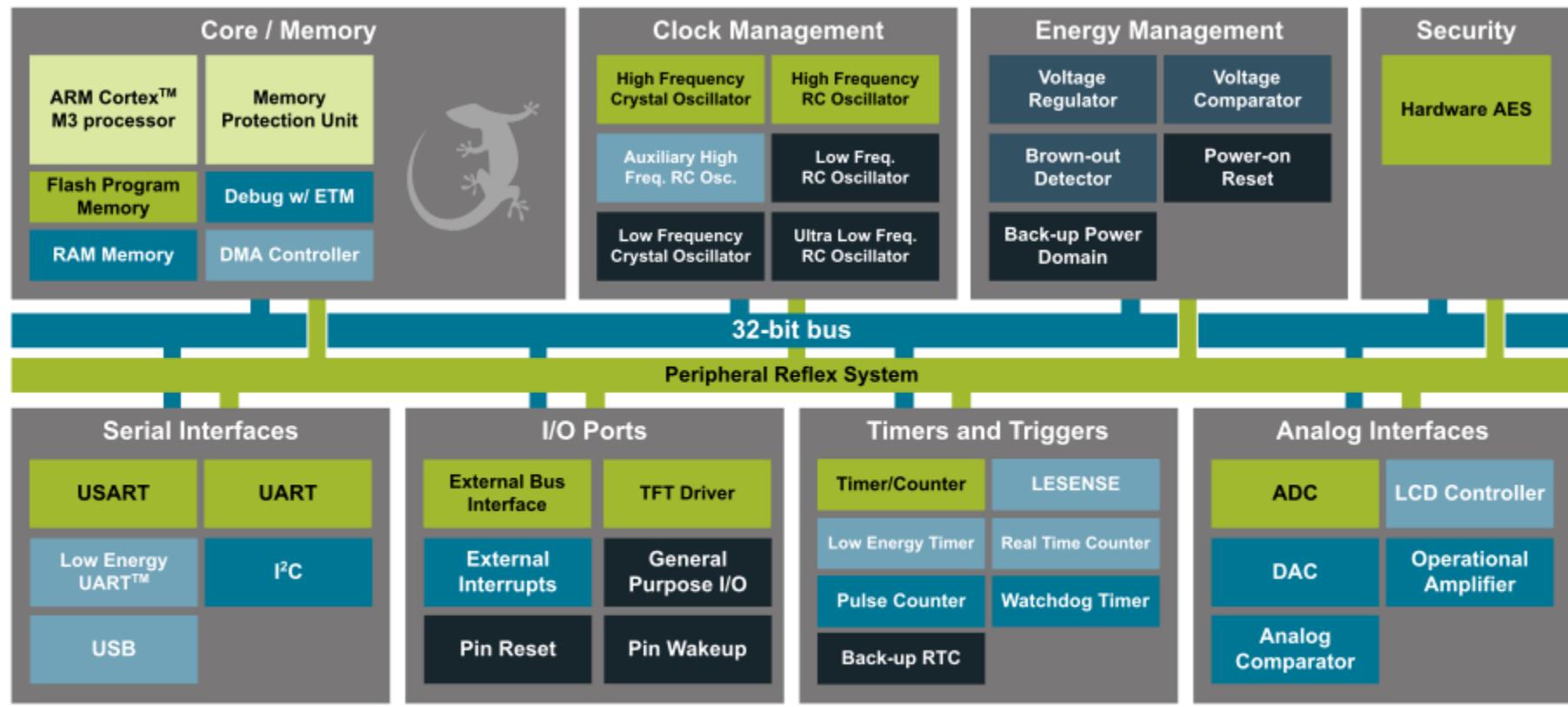
- Program letöltés a mikrovezérlőre, debuggolás
 - A mikrovezérlő egyik UART perifériájának kommunikációját ráülteti az USB kapcsolatra (virtuális soros port a számítógépen)
 - A mikrovezérlő egyik lehetséges táp forrása. Ha innét tápláljuk, megfigyelhető a felvett áram.

EFM32GG990F1024 mikrovezérlő:

- 32-bites mag (ARM Cortex-M3)
 - 1 MiB program memória (flash)
 - 128 KiB adat memória (SRAM)

A felhasznált hardver

- A kártyán szereplő mikrovezérlő egyszerűsített blokk diagramja:



A felhasznált hardver

- A példa alkalmazás az alábbiakat használja a mikrovezérlőből:
 - ARM Cortex-M3 (CPU) → program végrehajtás
 - Flash Memory → program memória a kódnak
 - RAM Memory → adat memória a változóknak
 - UART0 → egyszerű soros kommunikáció a C stdio számára (a printf() ide ír ki)
 - System Timer → az idő műlásának követésére (a FreeRTOS kezeli megszakításos módon; ez az egység az előző blokk diagramon közvetlenül nem látszik, az ARM Cortex-M3 beépített időzítője)

A felhasznált hardver

- A printf() útja a számítógép felé:
 - A beágyazott alkalmazás kimenete a mikrovezérlő egyik UART perifériáját használja karakterek küldésére.
 - Ez a debugger áramköri részhez érkezik, ami továbbítja azt a számítógép felé USB kapcsolaton keresztül.
 - A számítógépen egy driver virtuális soros portot hoz létre (Windows alatt COM<#> néven).
 - A COM portokra ún. terminál programok (mint pl. a PuTTY) képesek csatlakozni. Ezáltal a vett karaktereket megjeleníteni, és a legépelt karaktereket elküldeni (ezzel most a példa alkalmazásunk nem kezd semmit).

Az alkalmazás által generált kimenet

Hi
Lo
Lo
Látrehozás után mindenki taszk futásra kész.
Láthatóan az ütemező helyesen a magasabb prioritásúval kezd.

Látszik, hogy az alacsony prioritású taszk kétszer gyakrabban fut,
lévén fele annyi ideig várakozik egy ciklusban mint a magas prioritású.

Az alkalmazás nyomon követése (trace)

- Számos RTOS (így a FreeRTOS is) lehetőséget biztosít a futásának megfigyelésére → trace (nyomkövetés) szolgáltatások
- Ez a gyakorlatban azt jelenti, hogy az OS számos pontján trace makrók találhatóak, pl.:
 - Amikor egy taszk elhagyja a FUT állapotot
 - Amikor egy taszk FUT állapotba kerül
 - Amikor megtörtént egy óraütés (System Timer megszakítás)
 - Amikor egy taszk elkezd várni egy szemaforra
 - ...

Az alkalmazás nyomon követése (trace)

- Ezek a makrók alapértelmezetten üresek:

```
#ifndef traceTASK_SWITCHED_IN
    /* Called after a task has been selected to run.
       pxCurrentTCB holds a pointer to the task control block
       of the selected task. */

#define traceTASK_SWITCHED_IN()

#endif
```

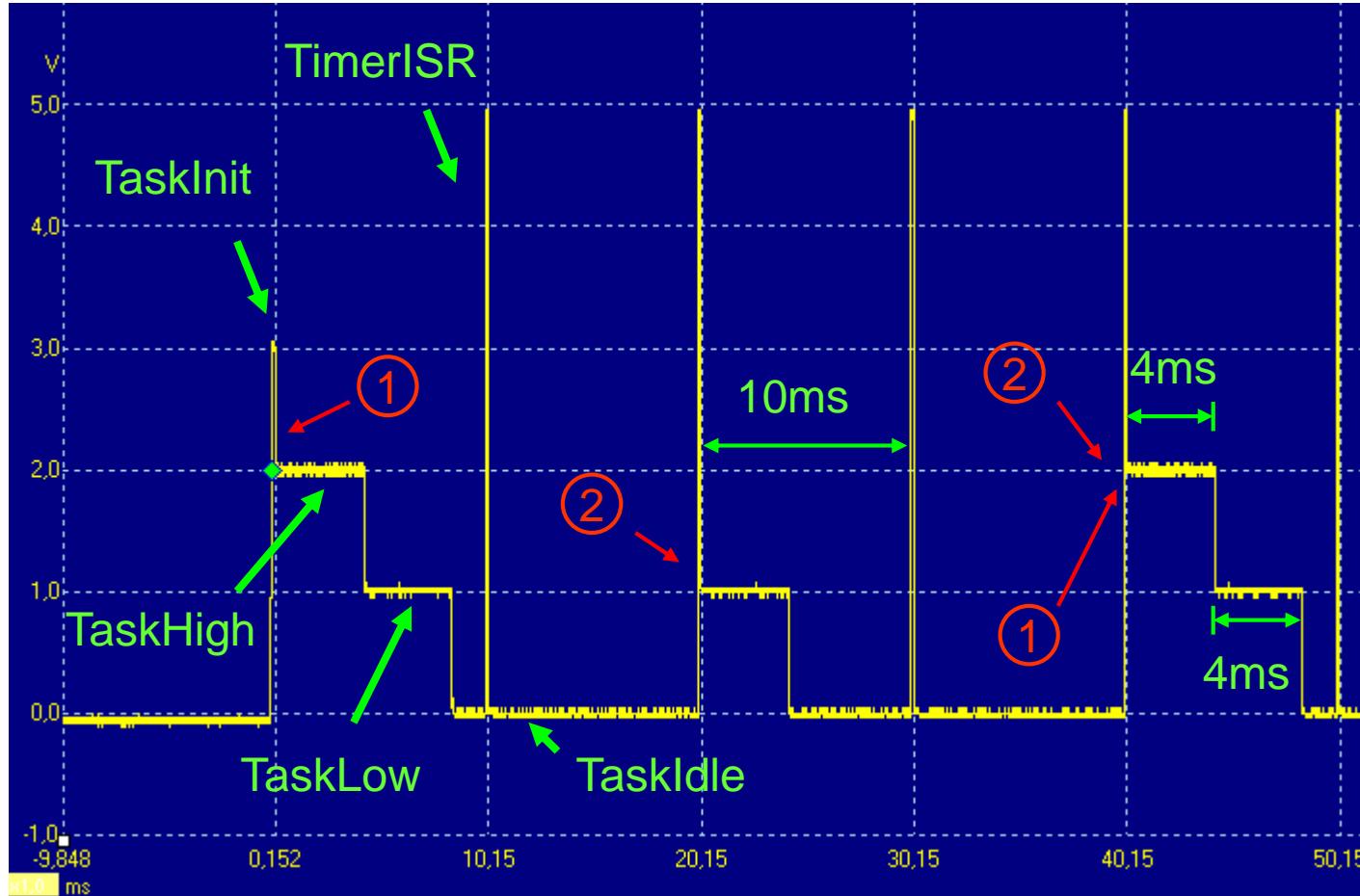
Részlet a FreeRTOS.h fejléc fájlból

- Látszik azonban, hogy felül lehet definiálni őket

Az alkalmazás nyomon követése (trace)

- Az egyik legegyszerűbb esetben a trace makrókat úgy lehet felüldefiniálni, hogy az alkalmazás egy analóg kimenetet mindenig egy – az éppen futó taszk prioritásával arányos – feszültség szintre állítson be.
- Kiegészítés gyanánt a megszakításokhoz (mint pl. amilyen a legtöbb RTOS-ben lévő óraütés) is rendelhető egy – a taszkokhoz rendelteknel magasabb – feszültség szint
- A kimenetet egy oszcilloszkóppal megfigyelve követhető az alkalmazás futása

Az alkalmazás nyomon követése (trace)



Ez az ábra régebben készült egy másik (de hasonló) példa alkalmazással egy másik (de hasonló) operációs rendszer alatt (μ C/OS-II). Az alkalmazás a mostanitól annyiban különbözik, hogy van benne egy harmadik, egyszeri lefutású taszk is (TaskInit). Ennek van a legnagyobb prioritása, és ez hozza létre a másik kettőt, majd töri magát.

- 1: olyan időpont, amikor több taszk is futásra kész, és az OS helyesen a legnagyobb prioritású ütemez
 - 2: olyan időpont, amikor a megszakítás nem oda tér vissza, ahova beütött, hanem az OS átüzemel

Az alkalmazás nyomon követése (trace)

- Léteznek kifinomultabb megoldások is egy analóg kimeneti láb feszültségének állítgatásán felül
- A trace makrókat megvalósíthatjuk úgy is, hogy az egyes eseményeket naplózzák, gyűjtsenek be róluk egyéb hasznos kiegészítő információkat.
- Majd ezeket valamelyen csatornán keresztül juttassák el egy számítógépnek, ahol egy analizátor alkalmazással ezeket jóval hatékonyabb módon elemezni tudjuk

Az alkalmazás nyomon követése (trace)

- Egy lehetséges ilyen tool a SEGGER SystemView



A beágyazott eszközön futó kód:

- Hozzá kell fordítani a saját alkalmazásunkhoz
- Minimális overhead-et jelent
- Feladata az eseményekről az információk begyűjtése, majd eltárolása egy bufferbe a RAM-ban
- A buffer tartalmát debug csatornán keresztül elérhetővé teszi egy számítógép számára

A számítógépen futó alkalmazás:

- Feladata a megfigyelt események megjelenítése

Az alkalmazás nyomon követése (trace)

- Példa egy SystemView által megvalósított FreeRTOS trace makróra:

```
#define traceTASK_SWITCHED_IN()
    if(prvGetTCBFromHandle(NULL) == xIdleTaskHandle) {
        SEGGER_SYSVIEW_OnIdle();
    } else {
        SEGGER_SYSVIEW_OnTaskStartExec((U32)pxCurrentTCB);
    }
```

Részlet a SEGGER_SYSVIEW_FreeRTOS.h fájlból (újra tördelve)

A SystemView által biztosított
függvény

A FreeRTOS által biztosított makró,
változó

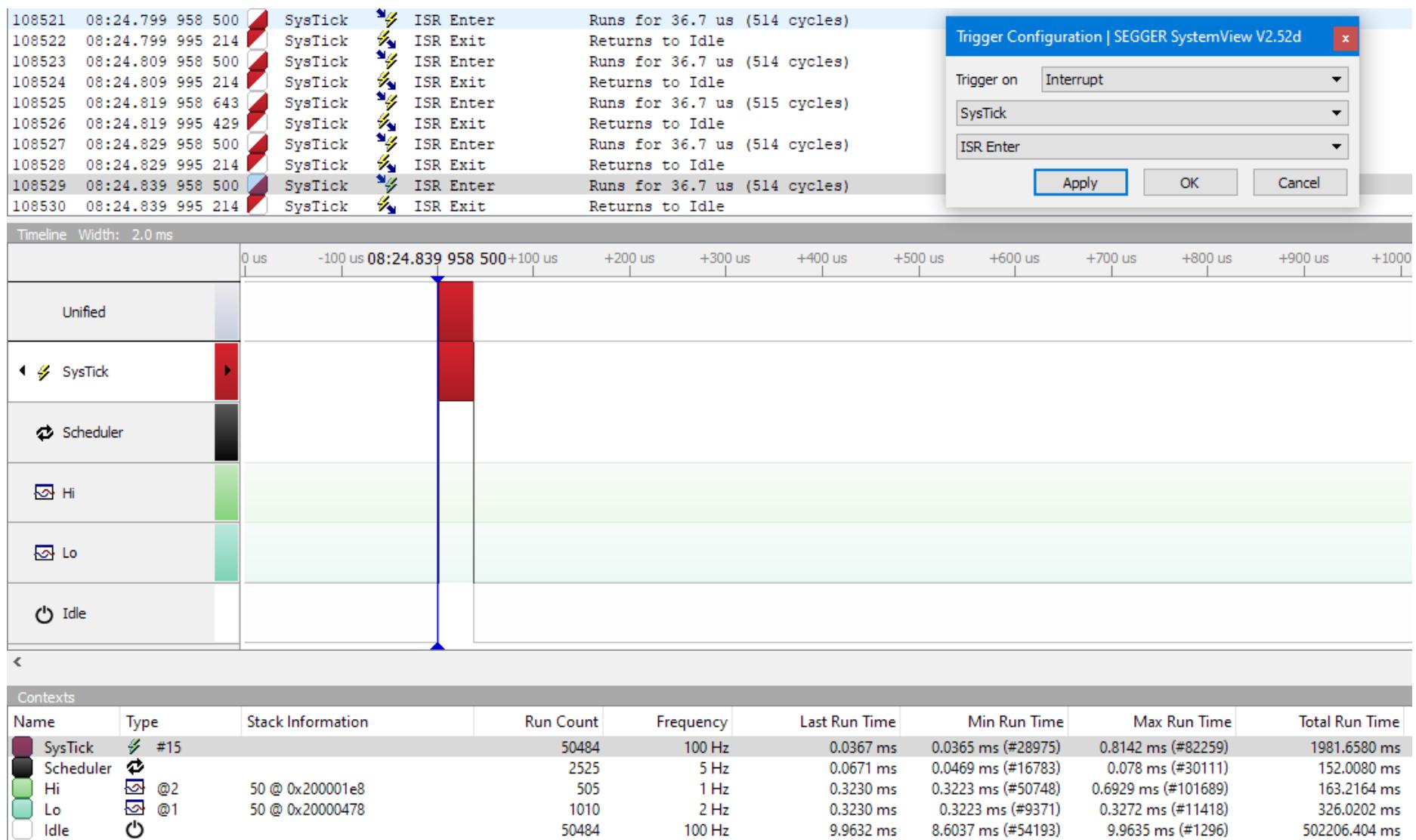
Az alkalmazás nyomon követése (trace)

- Kövessük nyomon a példa alkalmazásunk futását a SEGGER SystemView segítségével!
- A következő diákon a futás különböző fázisait fogjuk megnézni.

Az alkalmazás nyomon követése (trace)

- Az alkalmazásunk az ideje nagy részét az Idle szálban tölti, mivel a saját szálaink 1 ill. fél másodpercenként aktívak csak egy rövid időre.
- Az Idle szálat a System Timer által generált megszakítás kérések hatására periodikusan lefutó megszakítás kezelő rutin (SysTick ISR) szakítja meg
- Az ISR-t az operációs rendszer valósítja meg. Így követi az idő mülását, és ha kell, átütemez.
- A mi esetünkben általában nem lesz átütemezés, az Idle szál megszakítása után oda is térünk vissza.

Az alkalmazás nyomon követése (trace)

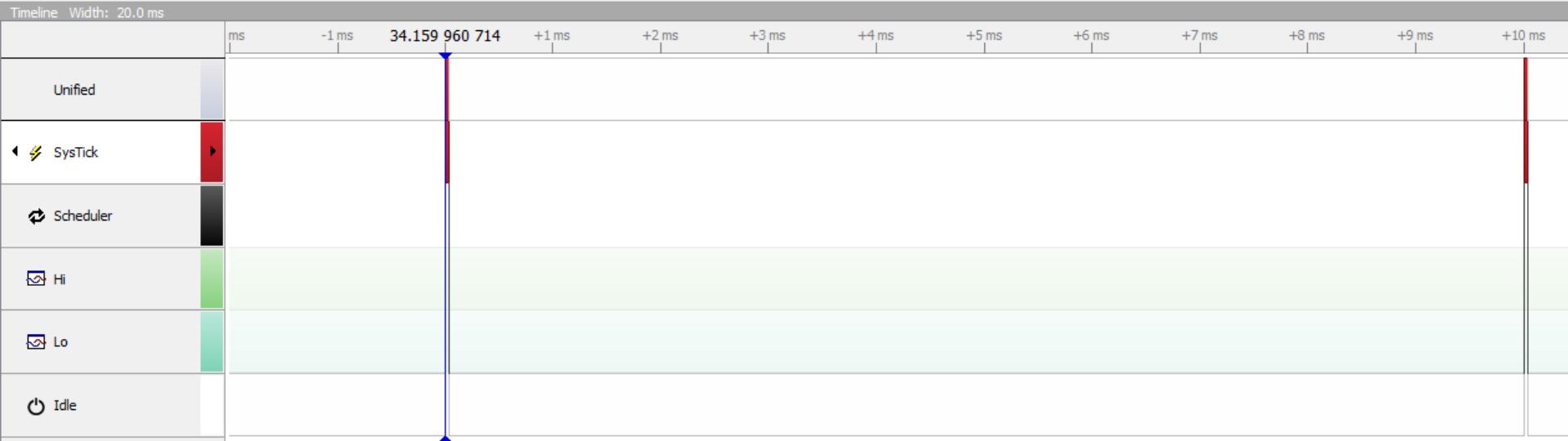
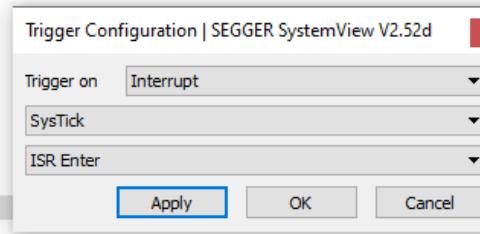


Az alkalmazás nyomon követése (trace)

- Kicsit nagyobb időszkeletet ábrázolva látható, hogy jelenleg a rendszer úgy van felkonfigurálva, hogy 10ms-es periódussal kövessék egymást a SysTick megszakítások

Az alkalmazás nyomon követése (trace)

7352	34.119	960	643	SysTick	ISR Enter	Runs for 36.7 us (514 cycles)
7353	34.119	997	357	SysTick	ISR Exit	Returns to Idle
7354	34.129	960	643	SysTick	ISR Enter	Runs for 36.7 us (514 cycles)
7355	34.129	997	357	SysTick	ISR Exit	Returns to Idle
7356	34.139	960	643	SysTick	ISR Enter	Runs for 36.7 us (514 cycles)
7357	34.139	997	357	SysTick	ISR Exit	Returns to Idle
7358	34.149	960	643	SysTick	ISR Enter	Runs for 36.7 us (514 cycles)
7359	34.149	997	357	SysTick	ISR Exit	Returns to Idle
7360	34.159	960	714	SysTick	ISR Enter	Runs for 36.7 us (514 cycles)
7361	34.159	997	429	SysTick	ISR Exit	Returns to Idle
7362	34.169	960	643	SysTick	ISR Enter	Runs for 36.7 us (514 cycles)



Contexts										
Name	Type	Stack Information	Run Count	Frequency	Last Run Time	Min Run Time	Max Run Time	Total Run Time	Run Time/s	
SysTick	⚡ #15		3416	100 Hz	0.0367 ms	0.0366 ms (#458)	0.8123 ms (#611)	135.1815 ms	3.8333 ms	0.38%
Scheduler	♻️		172	5 Hz	0.0670 ms	0.0469 ms (#3358)	0.0738 ms (#1641)	10.296 ms	0.3007 ms	0.03%
Hi	⌚ @2	50 @ 0x200001e8	34	1 Hz	0.3230 ms	0.3224 ms (#1858)	0.3271 ms (#5294)	10.6643 ms	0.3230 ms	0.03%
Lo	⌚ @1	50 @ 0x20000478	69	2 Hz	0.3230 ms	0.3223 ms (#6904)	0.3273 ms (#6262)	21.9715 ms	0.6461 ms	0.06%
Idle	⚡		3416	100 Hz	9.9633 ms	9.0206 ms (#5299)	9.9635 ms (#4083)	33971.1935 ms	994.8967 ms	99.49%



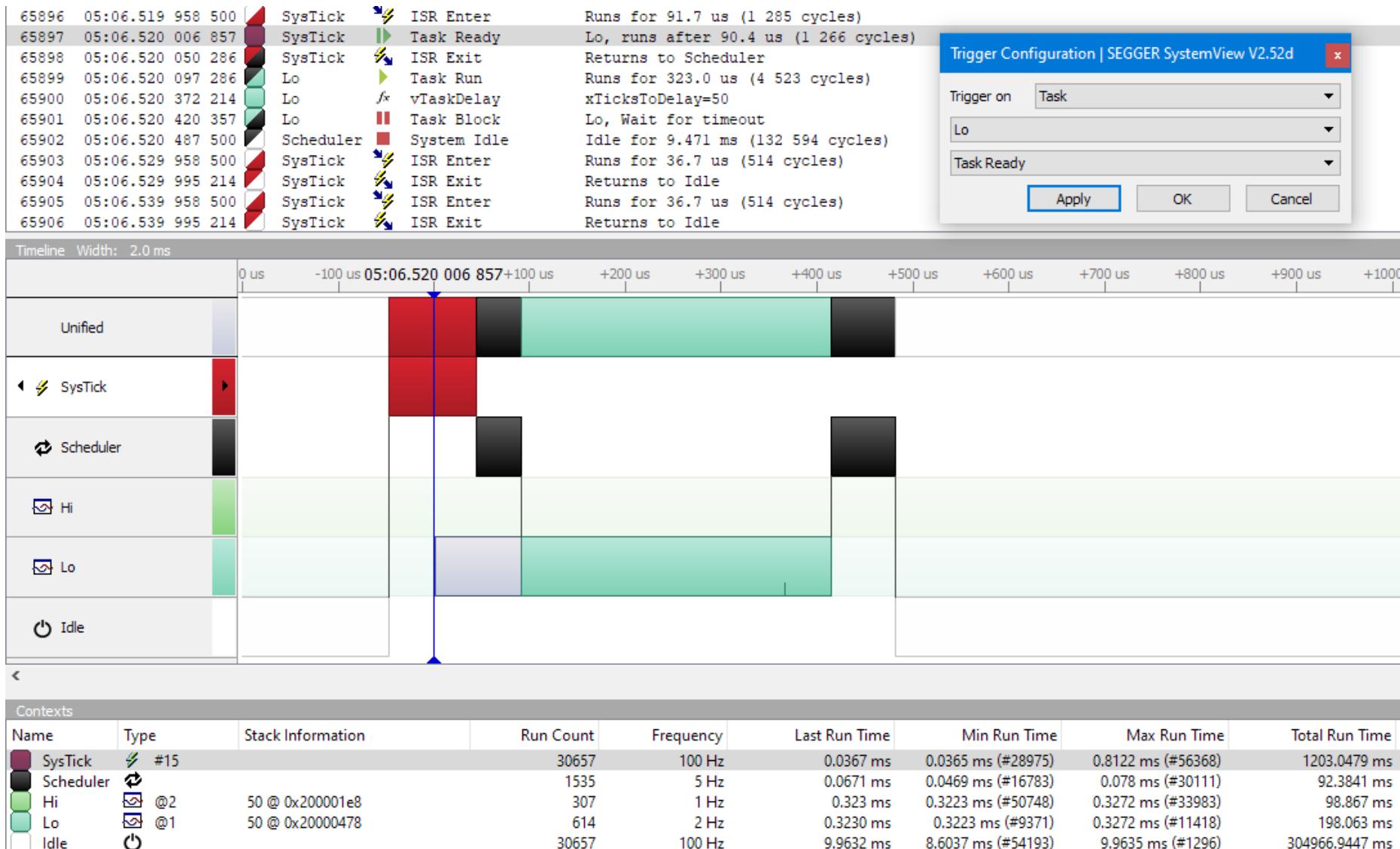
Az alkalmazás nyomon követése (trace)

- Kicsit érdekesebb az az eset, amikor valamelyik taszk felébred.
- Az egyszerűség kedvéért elsőre most egy olyan időpontot nézünk meg, amikor csak az alacsony prioritású szál ébred fel.
- A SysTick ISR-ben az OS most futásra kész állapotba teszi az alacsony prioritású szálat (ezt jelzi a szürke sáv, ekkor még nem fut).
- Az ISR után most az ütemező (fekete) fog futni, ami az Idle szálról átvált az alacsony prioritásúra

Az alkalmazás nyomon követése (trace)

- Miután az alacsony prioritású szál kiírta az üzenetét, ismét elteszi magát időre való várakozó állapotba (ezt az OS hívást szemlélteti a pici függőleges zöld vonalka)
- Az OS hívás hatására tehát az éppen futó szál várakozó állapotba kerül. Ezért az OS ebből a hívásból nem tér vissza azonnal az alacsony prioritású szálba, hanem az ütemező algoritmus fog futni, ami visszaadja a vezérlést az immár egyedüli futásra kész szálnak (Idle)

Az alkalmazás nyomon követése (trace)



Az alkalmazás nyomon követése (trace)

- A következő dia az egyszerű példa alkalmazásunk legbonyolultabb futási szekvenciáját mutatja. Azt, amikor mindenki saját szálunk egyszerre ébred fel.
- Látható, hogy most az ISR mindenki saját szálunkat futásra kész állapotba helyezi, majd az ISR-ből történő kilépés után az ütemező fut, ami helyesen a magas prioritású szál fogja elsőként ütemezni.
- Ezt követi később az alacsony prioritású szál, miután a magas elment várakozni, majd az Idle szál, miután az alacsony is elment várakozó állapotba.

Az alkalmazás nyomon követése (trace)

123082	09:32.589	960	429	█	SysTick	⚡	ISR Enter	Runs for 142.7 us (1 999 cycles)
123083	09:32.590	008	786	██████	SysTick	▶	Task Ready	Hi, runs after 141.4 us (1 980 cycles)
123084	09:32.590	061	214	██████	SysTick	▶	Task Ready	Lo, runs after 485.6 us (6 799 cycles)
123085	09:32.590	103	214	██████	SysTick	⚡	ISR Exit	Returns to Scheduler
123086	09:32.590	150	214	███	Hi	▶	Task Run	Runs for 323.0 us (4 523 cycles)
123087	09:32.590	425	143	███	Hi	✗	vTaskDelay	xTicksToDelay=100
123088	09:32.590	473	286	███	Hi	■	Task Block	Hi, Wait for timeout
123089	09:32.590	546	857	███	Lo	▶	Task Run	Runs for 323.0 us (4 523 cycles)
123090	09:32.590	821	786	███	Lo	✗	vTaskDelay	xTicksToDelay=50
123091	09:32.590	869	929	███	Lo	■	Task Block	Lo, Wait for timeout
123092	09:32.590	936	143	███	Scheduler	■	System Idle	Idle for 9.0242 ms (126 340 cycles)

Trigger Configuration | SEGGER SystemView V2.52d

Trigger on Task

Hi

Task Ready

Apply OK Cancel



Contexts									
Name	Type	Stack Information	Run Count	Frequency	Last Run Time	Min Run Time	Max Run Time	Total Run Time	
SysTick	#15		57263	100 Hz	0.0367 ms	0.0365 ms (#34248)	0.8125 ms (#122470)	2251.7629 ms	
Scheduler	↻		2865	5 Hz	0.0662 ms	0.0469 ms (#25931)	0.0779 ms (#27868)	172.4773 ms	
Hi	▣ @2	50 @ 0x200001e8	573	1 Hz	0.3230 ms	0.3223 ms (#14324)	0.3274 ms (#108899)	184.8201 ms	
Lo	▣ @1	50 @ 0x20000478	1146	2 Hz	0.3230 ms	0.3223 ms (#142)	0.3272 ms (#12825)	369.9553 ms	
Idle	⌚		57263	100 Hz	9.9632 ms	8.6045 ms (#13473)	9.9636 ms (#112961)	569640.2910 ms	



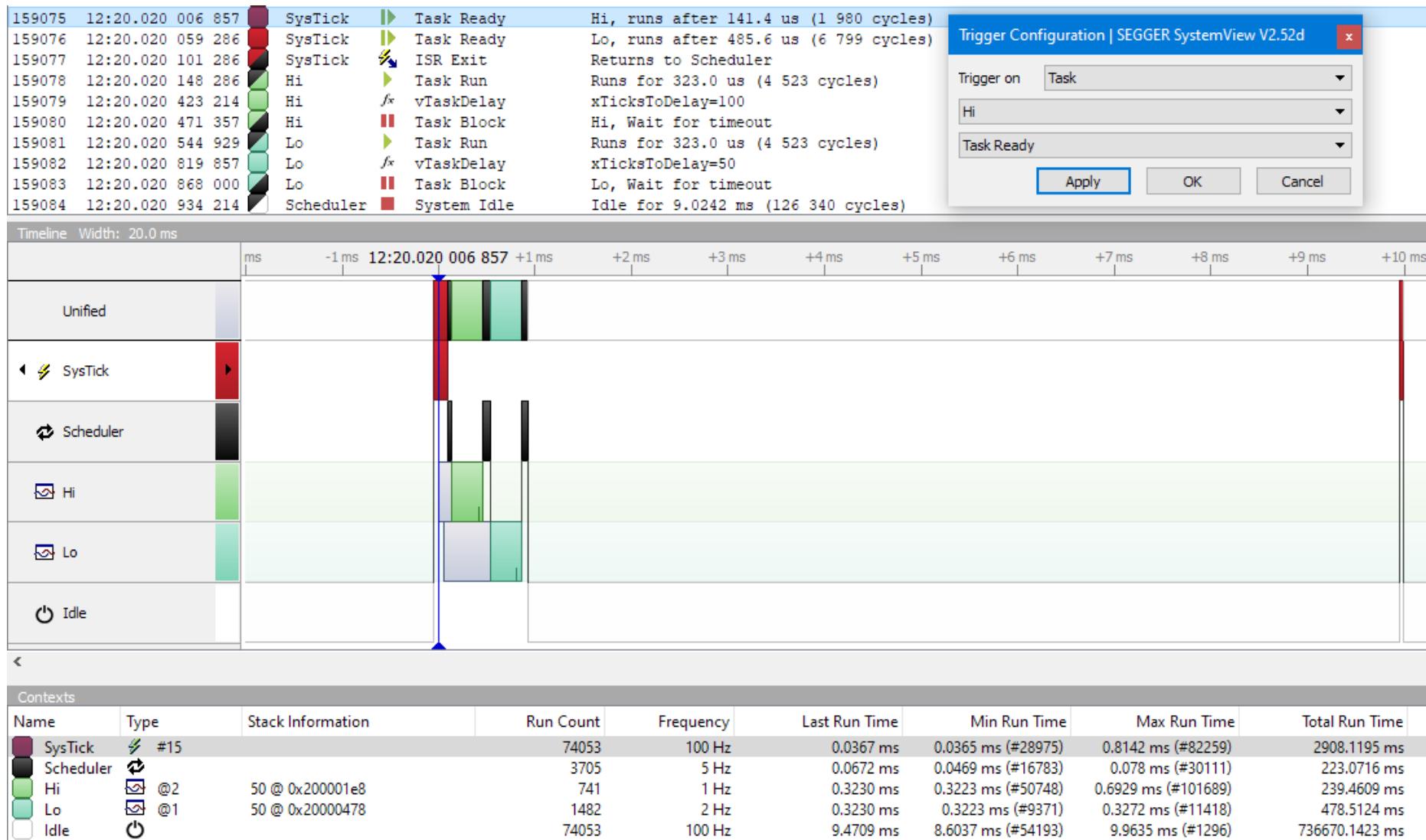
Az alkalmazás nyomon követése (trace)

- Végezetül az időzítési viszonyok szemléltetésére nézzük meg, hogy viszonyul az előbbi eset futási ideje a SysTick ISR-ek gyakoriságához
- A taszkjaink futási idejét leginkább az UART kommunikáció szabja meg. A beállított sebesség 115200 b/s (tekintsük az egyszerűség kedvéért 100000-nek). A 8 bites karakterek 2 segéd bittel egészülnek ki az átvitel során (Start ill. Stop), így egy karakter átviteléhez 10 bit kell, tehát a sebesség 10000 char/s. Ezerrel egyszerűsítve ez: 10 char/ms

Az alkalmazás nyomon követése (trace)

- Mindkét taszk egy két betűből álló szót küld el („Hi” ill. „Lo”) plusz az „új sor” karaktert (\n), amihez az API automatikusan beszúrja a másik sorvég karaktert („kocsi vissza”, \r).
- Így tehát, ha minden taszkunk egyszerre ébred fel, összesen 8 karaktert fognak elküldeni. Ez a 10 char/ms sebességgel számolva kicsit kevesebb mint 1 ms időt vesz igénybe.
- Az OS hívások (vTaskDelay()), a SysTick ISR és az ütemező futási ideje ehhez képest rövidebb, de nem nulla.

Az alkalmazás nyomon követése (trace)



Operációs rendszerek: felépítés és alapműködés

Mészáros Tamás

<http://www.mit.bme.hu/~meszaros/>

Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

Az előadásfóliák legfrissebb változata a tantárgy honlapján érhető el.

Az előadásanyagok BME-n kívüli felhasználása és más rendszerekben történő közzététele előzetes engedélyhez kötött.

Hogyan építsünk fel egy operációs rendszert?

Az operációs rendszer

azon **programok** összessége,
amelyek vezérlik a számítógép hardverének működését,
és lehetővé teszik azon felhasználói feladatok végrehajtását.

Elvárások

- egyszerre több feladat kiszolgálása (több program futtatása)
- megbízható
- biztonságos

Programok

- megoldják a feladatainkat
- különféle forrásokból származnak (OS, alkalmazásból, sw repo, web stb.)

Megbízhatunk-e a szoftverekben?



Architekturális megfontolások: multiprogramozás



Architekturális megfontolások: fennhatóság

AN x64 PROCESSOR IS SCREAMING ALONG AT BILLIONS OF CYCLES PER SECOND TO RUN THE XNU KERNEL, WHICH IS FRANTICALLY WORKING THROUGH ALL THE POSIX-SPECIFIED ABSTRACTION TO CREATE THE DARWIN SYSTEM UNDERLYING OS X, WHICH IN TURN IS STRAINING ITSELF TO RUN FIREFOX AND ITS GECKO RENDERER, WHICH CREATES A FLASH OBJECT WHICH RENDERS DOZENS OF VIDEO FRAMES EVERY SECOND

BECAUSE I WANTED TO SEE A CAT JUMP INTO A BOX AND FALL OVER.



forrás: [xkcd](#)



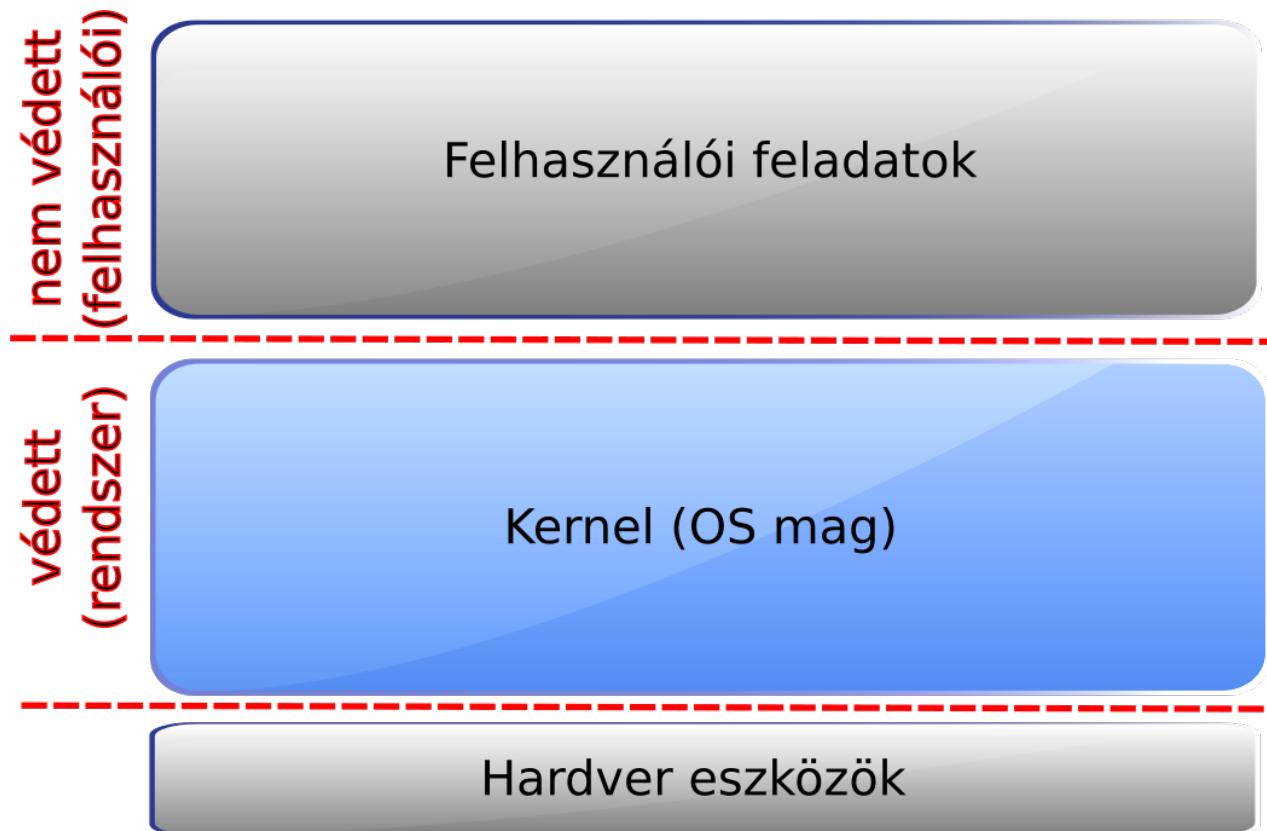
forrás: [youtube](#)

A fennhatóság megvalósítása

- Hogyan felügyelheti egy program egy másik működését?
 - Ismétlés: CPU védelmi szintek (**SZGA**, hardver alapok)
 - legalább két eltérő működési mód
 - 0. privilegizált avagy védett mód (bármi megtehető)
 - más üzemmódban a CPU korlátozza
 - utasítások végrehajtását, memóriaterületek elérését, perifériák hozzáférését
 - Az OS programjának egy része **védett módban** fut
 - ez a rész fennhatóságot gyakorol minden más program felett
 - szabályozza az életciklusukat (keletkezés, működés, megszűnés)
- A **kernel** az operációs rendszer védett módban működő programja, amely felügyeli a felhasználói módú programok működését, és biztosítja hozzáférésüket a rendszer erőforrásaihoz.*
- minden más program **felhasználói módban** működik
 - hardveresen betartatott korlátozásokkal

A védett módban működő kernel

Vezérlőprogram



Erőforrás-allokátor

A kernel

- Vezérlőprogramként felügyeli más programok végrehajtását
 - életciklus-menedzsment (létrehozás, működés, megszűnés)
 - működési események kezelése, kézbesítése
 - szolgáltatásokat nyújt számukra
- Menedzseli az erőforrásokat
 - eszközök előkészítése a felhasználásra
 - kezelésekkel kapcsolatos közös funkciók biztosítása
 - működésekkel kapcsolatos események kezelése, illetve továbbítása
 - párhuzamos kérések kiszolgálása, szeparációja, konfliktusok feloldása
- A megbízhatóság és biztonság szem előtt tartása
 - az erőforrások védelme a hibás vagy kártékony felhasználástól
 - a futó programok szeparációja, külső védelme
 - biztonsággal kapcsolatos közös funkciók biztosítása a programok számára

Mire lehet még szükségünk?

- Erőforrás-menedzsment ✓
- Felügyelet ✓
- ... ???

(Gondoljunk a felhasználói szerepkörökre!)

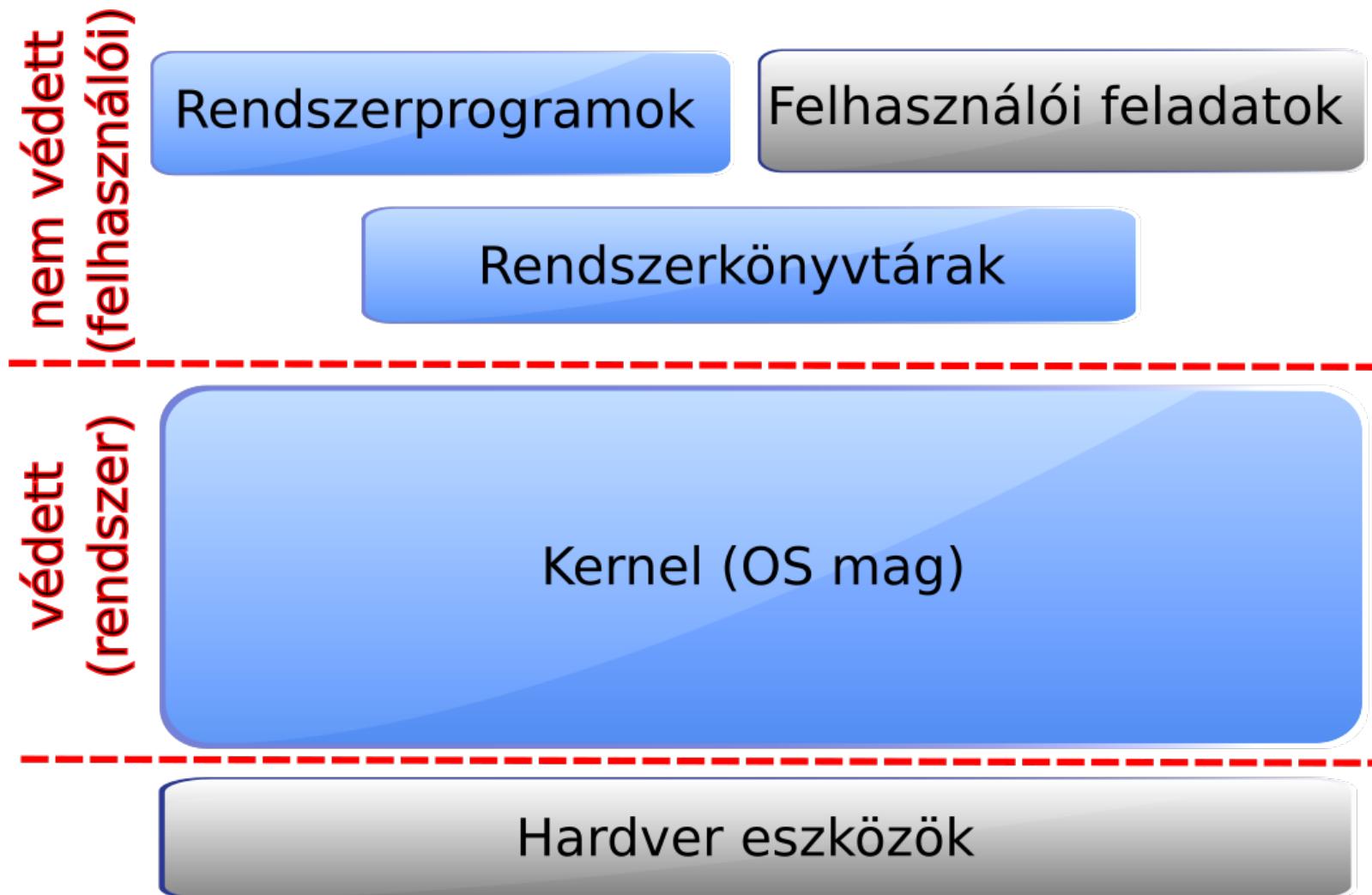
Az OS további részei

Rendszerkönyvtárnak nevezzük az operációs rendszer részét képező programkönyvtárakat, amelyeket a programok felhasználhatnak működésük során.

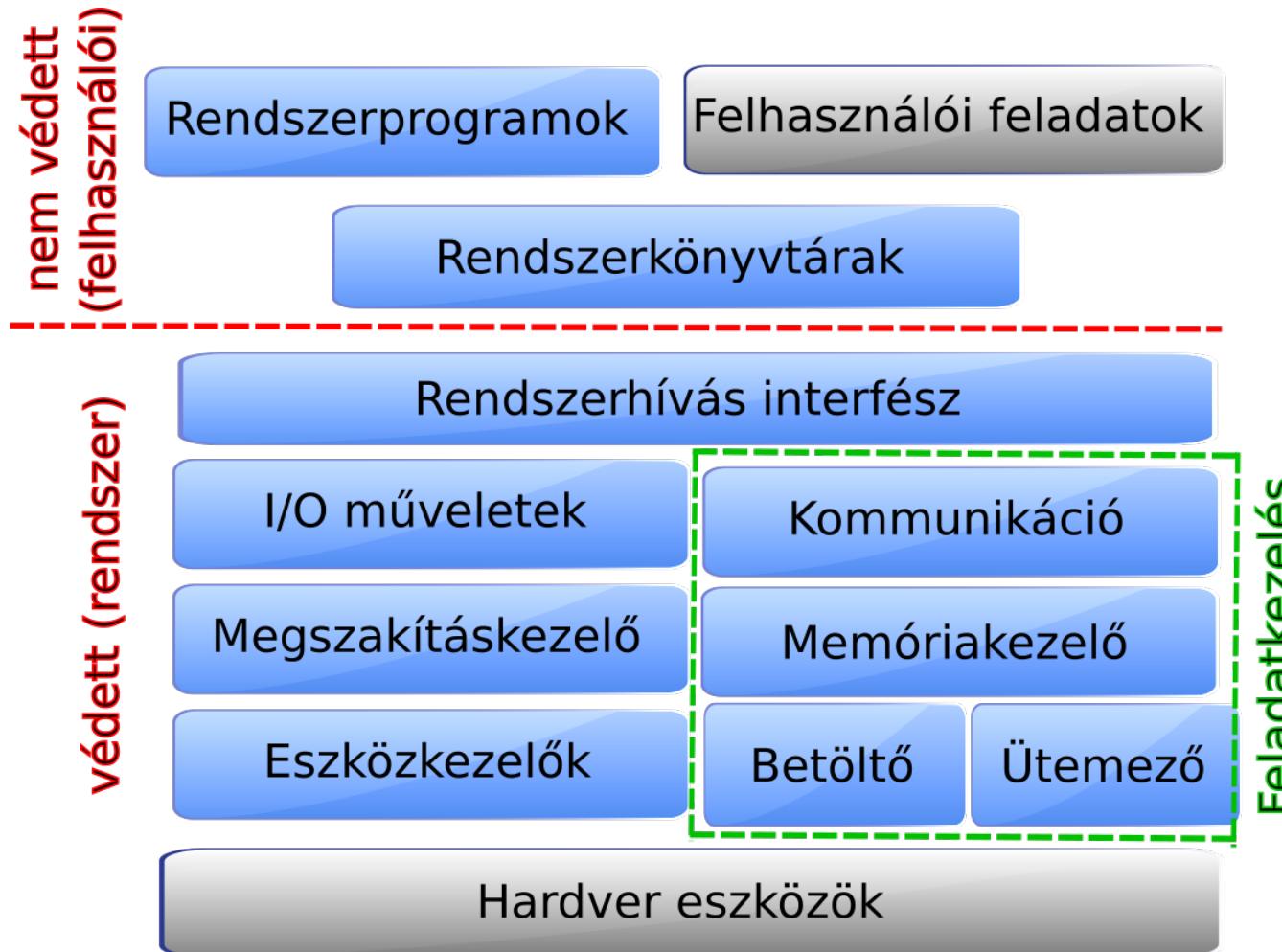
A **rendszerprogram** az operációs rendszer részét képező, működésével kapcsolatos feladatokat megoldó program.

A **rendszerszolgáltatás** az operációs rendszer által kezelt, folyamatosan elérhető funkciókat nyújtó program.

Az OS további részei



A kernel vázlatos felépítése



Az OS, mint feladat-végrehajtó rendszer

Áttekintés...

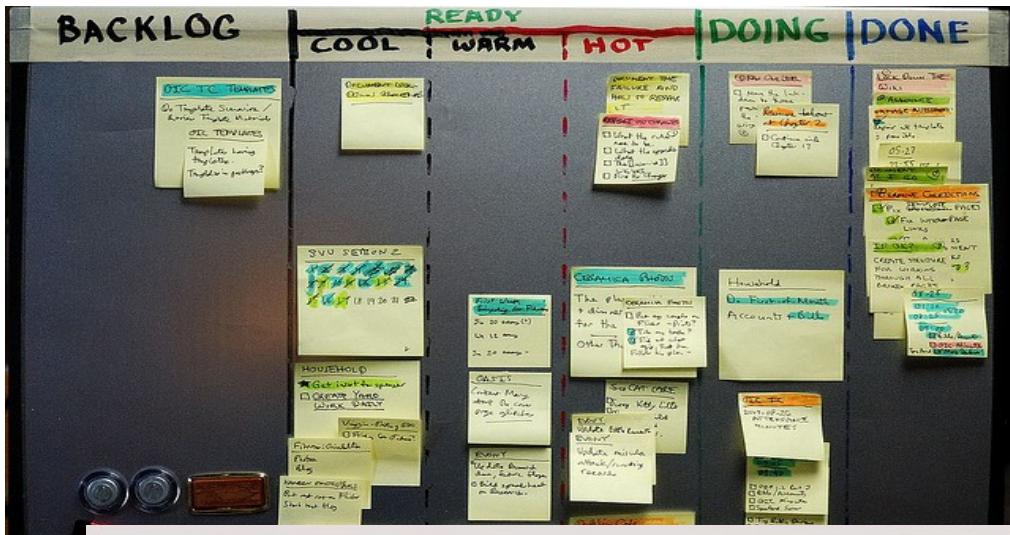
- Az operációs rendszer
 - feladatok végrehajtása
 - **vezérlőprogram**
 - **erőforrás-alkotátor**
- Elvárások
 - feladatok egyidejű kiszolgálása
 - megbízható működés
 - esetenként valósidejűség



Az OS felépítése

- Kialakulása
 - kötegelt rendszerek
 - **multiprogramozott**
 - **időosztásos**
 - beágyazott

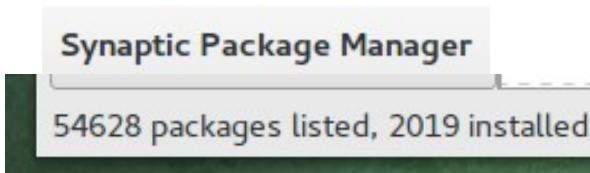
Hogyan kezeljük a feladatokat?



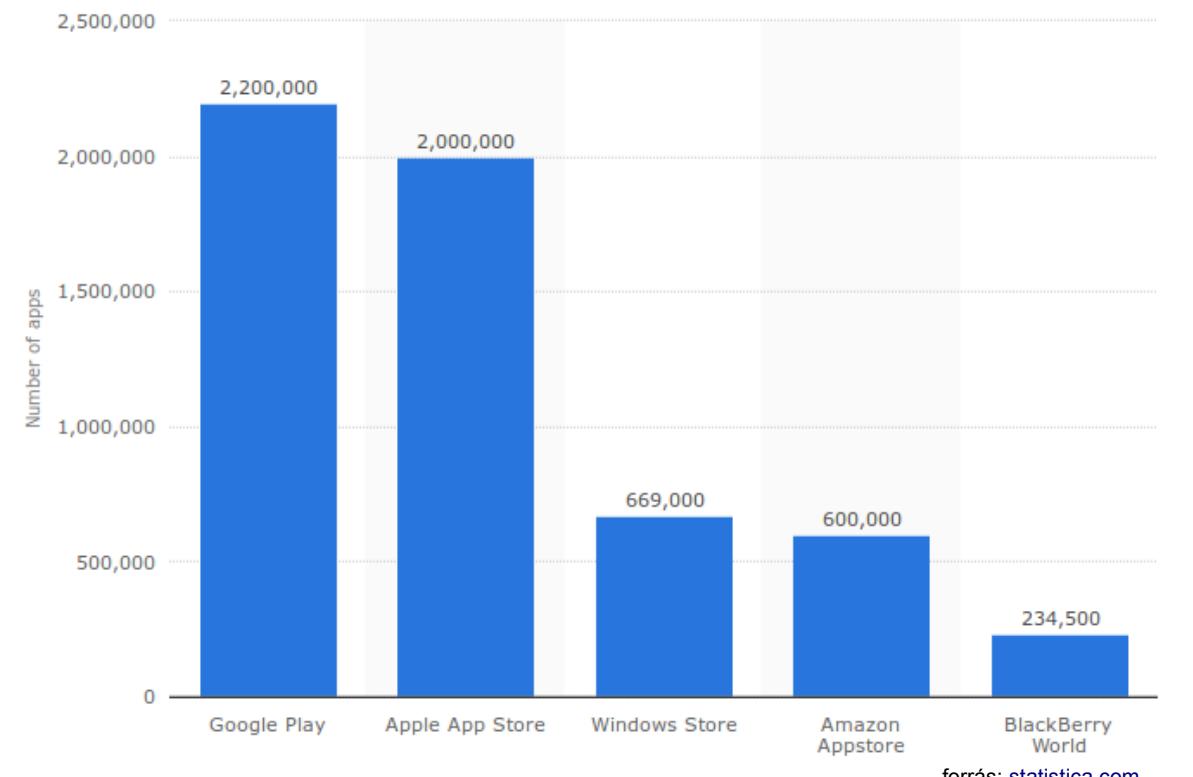
Milyen feladatokat futtatunk?

- A felhasználói feladatok sokszínűsége
- Az operációs rendszerek osztályozása (kliens, szerver, beágyazott...)
- Alkalmazások

Number of apps available in leading app stores as of June 2016



```
> yum list all | wc -l  
22747
```



A feladatok jellege

- I/O-intenzív feladatok
 - idejük nagy részét várakozással töltik (adatbetöltés, adatkiírás)
 - kevés processzoridőre van szükségük
 - pl.: fájlszerver, webszerver, email kliens és szerver stb.
- CPU-intenzív feladatok
 - idejük nagy részét a processzoron szeretnék tölteni
 - ehhez képest (relatíve) kevés I/O műveletre van szükségük
 - pl.: titkosítási és matematikai műveletek, összetett adatfeldolgozás stb.
- Memória-intenzív feladatok
 - egy időben nagy mennyiségű adat elérésére van szükségük
 - ha van elég, akkor CPU-intenzívek, ha nincs, akkor I/O-intenzív feladattá válnak
 - pl. nagy mátrixok szorzása, keresési indexek építése és használata stb.
- Speciális igények
 - valósidejű működés
 - filmnézés
 - ...

Elvárásaink

- Kevés várakozás

várakozási idő (waiting time)

körülfordulási idő (turnaround time)

válaszidő (response time)

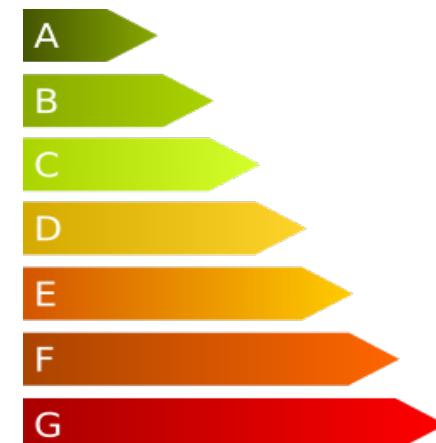
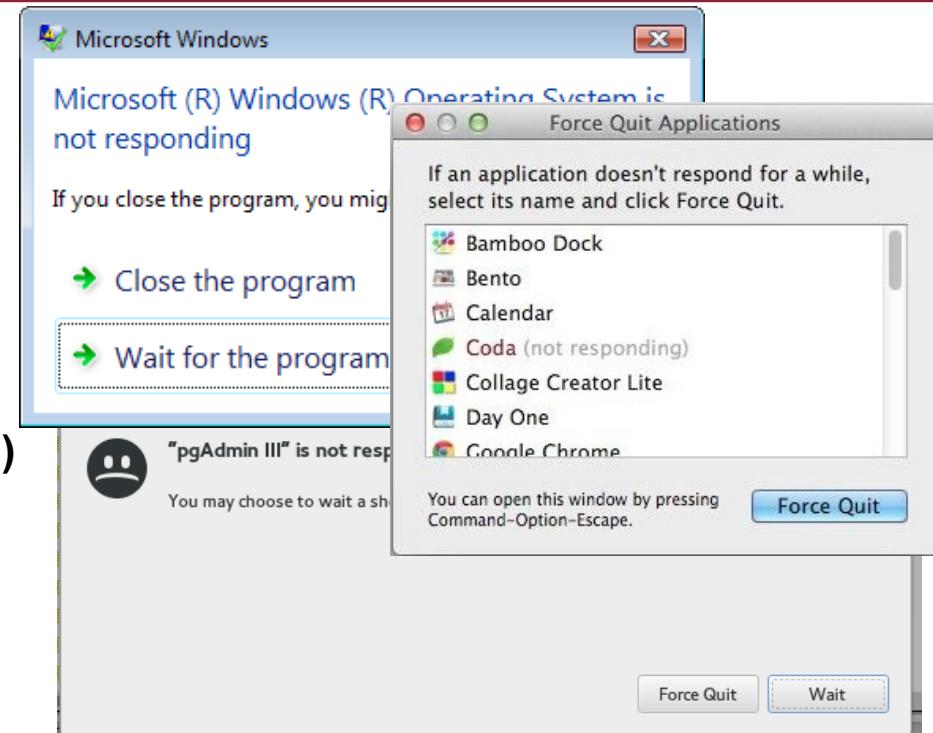
- Hatékonyság

CPU-kihasználtság (CPU utilization)

átbocsájtó képesség (throughput)

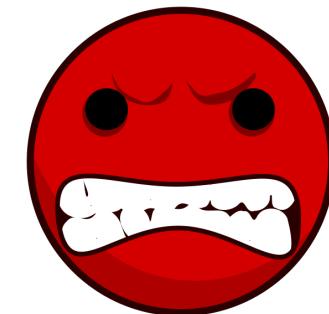
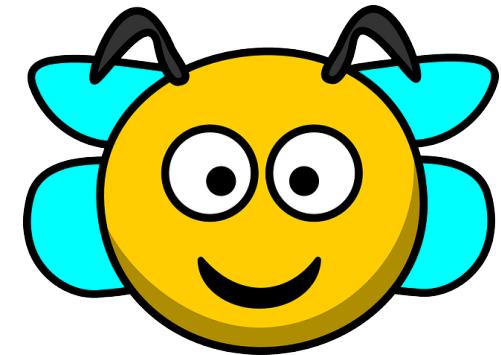
rezsiköltség (overhead)

- Jósolhatóság, determinisztikusság



Az optimális feladat-végrehajtó rendszer

- A naiv felhasználó elvárásai
 - biztosítja feladatai végrehajtását
 - minimalizálja a várakozási és válaszidőt
 - az erőforrásokat (CPU, I/O) maximális kihasználja
 - minél kisebb rezsiköltséggel dolgozik
- Mit tapasztal a rendszer használata során?
 - egyes programok „lassan” futnak
 - mások ok nélkül „lefagynak”
 - „feleslegesen” erőforrásokat foglalnak
 - akadozik a filmnézés
 - gyorsan merül az akkumulátor
 - néha mintha az egész rendszer leállna
 - nem tudja fogadni a hívást
 - ...



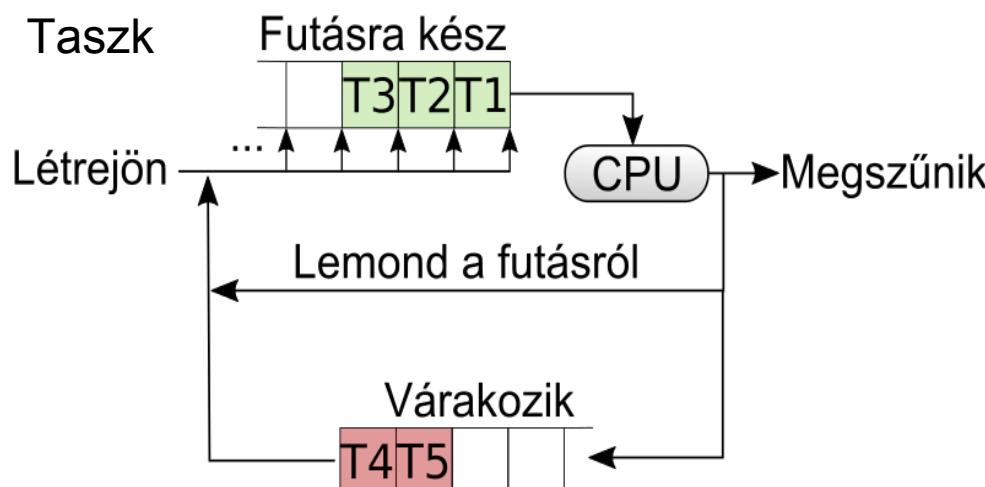
Mi okozza a nehézségeket?

- Az OS nem lát a jövőbe
 - milyen feladatok jönnek
 - milyen jellegűek
- Sok a feladat
 - különböző elvárások
 - más az optimalitási kritérium
 - néha túl sok, „vergődik” a rendszer
- A feladatok hatással vannak egymásra
 - együttműködnek
 - versenyeznek
- Hibák
 - programozói
 - hardver

Az OS, mint erőforrás-allocátor

Áttekintés...

- Az operációs rendszer
 - felhasználói feladatok támogatása
 - vezérlőprogram
 - **erőforrás-alkotátor**
- Elvárások
 - feladatok egyidejű kiszolgálása
 - megbízható működés
 - erőforrások optimális kihasználása

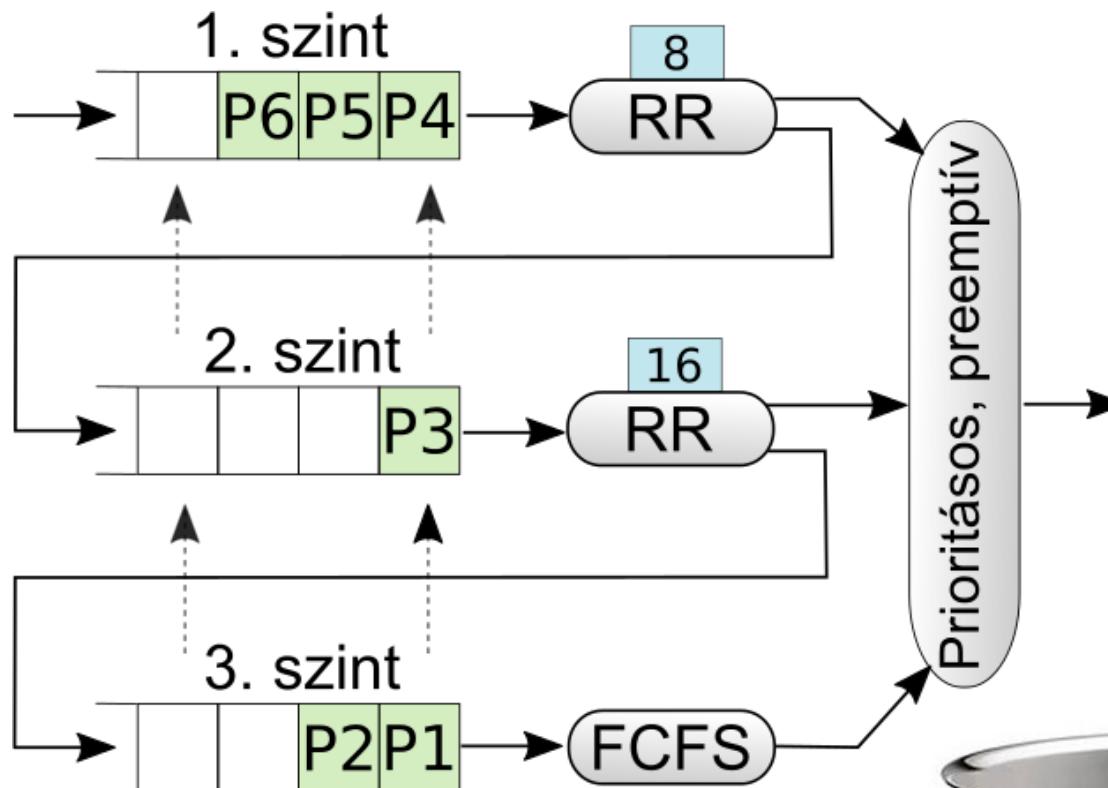


Az OS felépítése

- Erőforrások
 - processzor (CPU)
 - központi memória
 - tárolórendszerek
 - perifériák
 - egyéb hardvereszközök

Többszintű visszacsatolt sorok ütemező

multilevel feedback queue (MFQ)



Taszkok szintlépései:

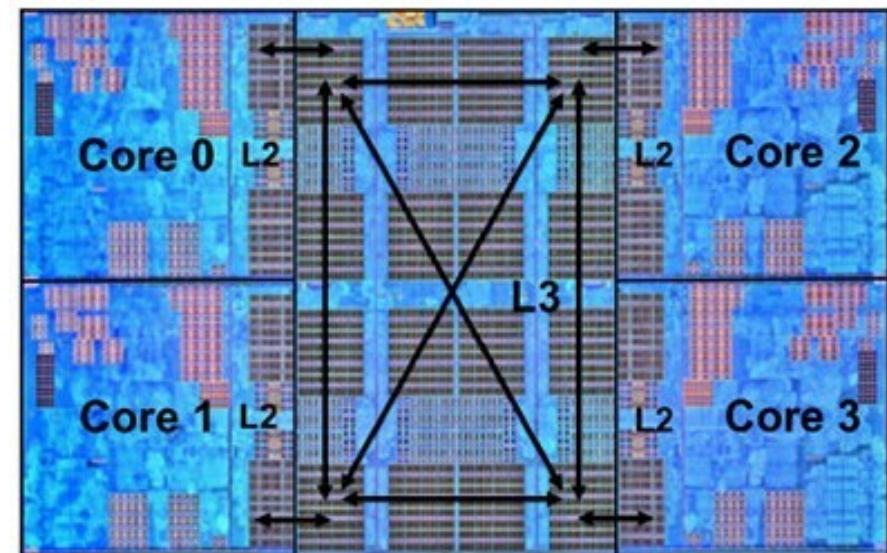
- amelyik kihasználja az időszeletét, az lentebb lép
- várakozó állapotba kerülő taszkok fentebb lépnek



Turing díj járt érte

Esettanulmány: AMD Ryzen

- Egyetlen tokban 8 CPU mag
 - 2 „Core Complex” (CCX)
 - Infinity Fabric összekötőelem
 - L3 tekintetében ~NUMA
- CCX
 - 4 CPU mag (SMP)
 - 8MB L3 cache

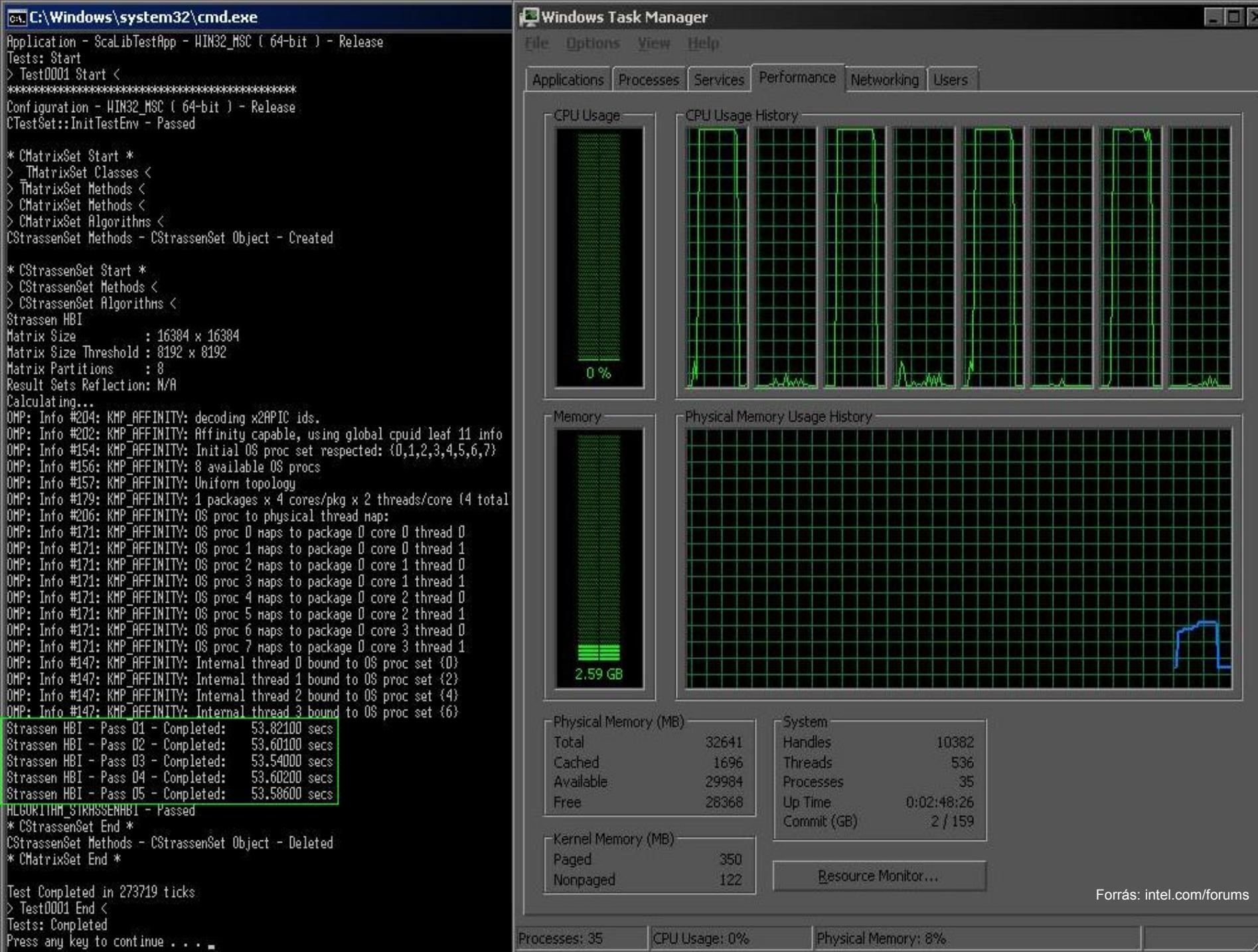


- CPU mag
 - 2 szál (SMT)
 - 512K L2 cache, 64K+32K L1 cache

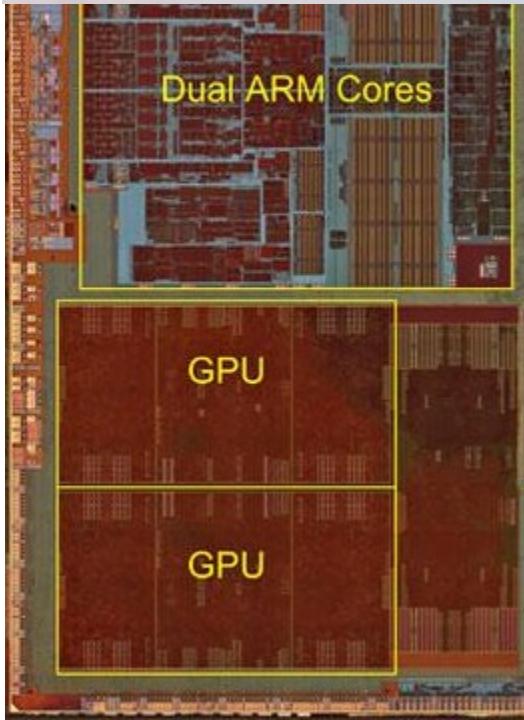
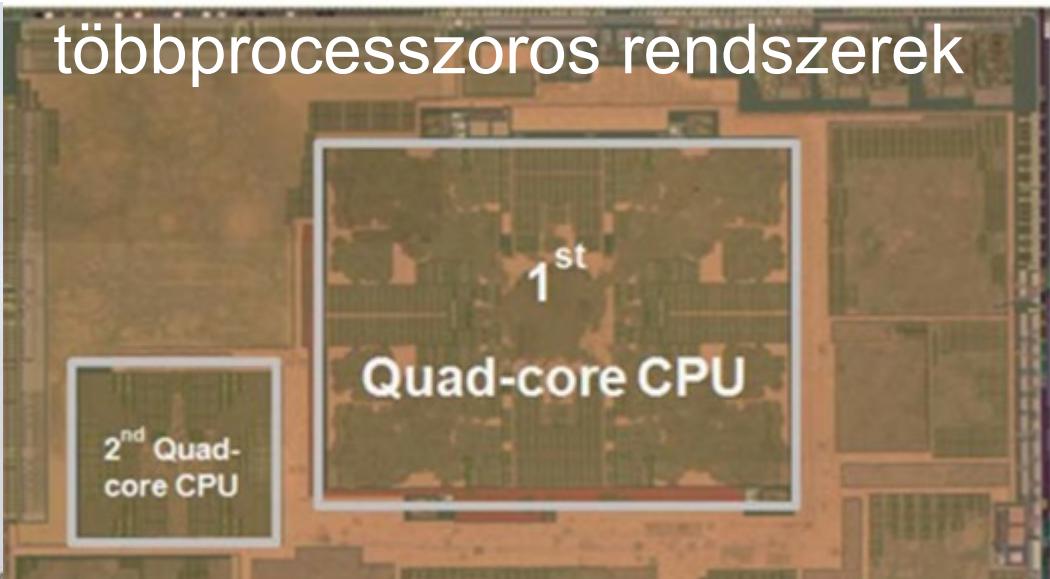


Forrás: AMD

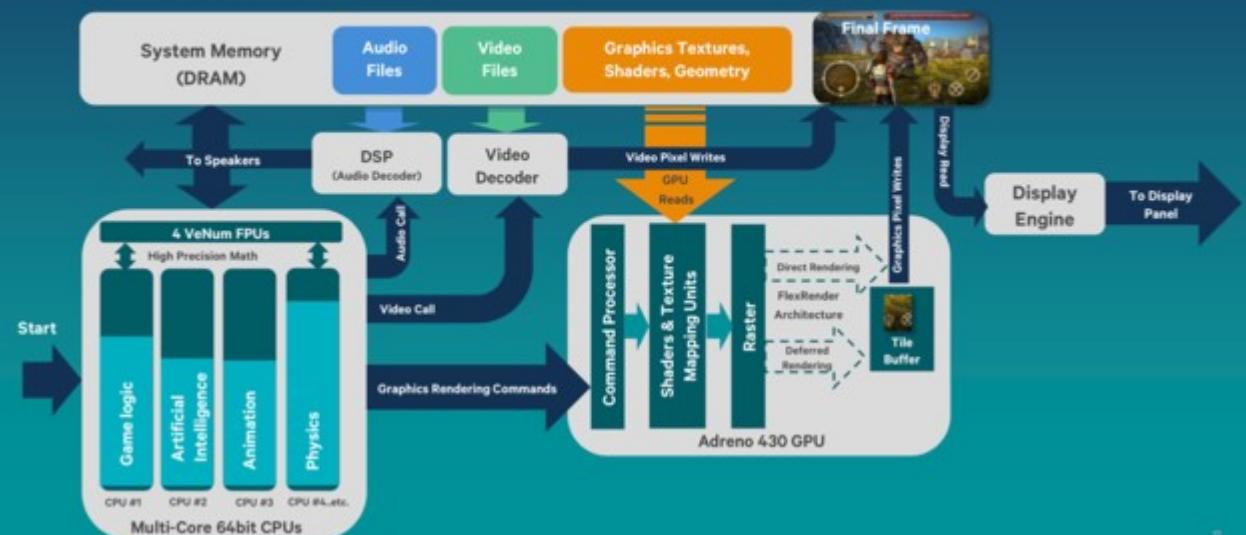
- Szempontok az ütemező / programozók számára
 - a CCX ismerete
Mozoghat egy taszk a CCX-ek között? Az L3 adat elveszik, a teljesítmény esik.
 - többszálú végrehajtás
8 mag 16 szála között hogyan osszuk el a taszkokat?



Heterogén többprocesszoros rendszerek

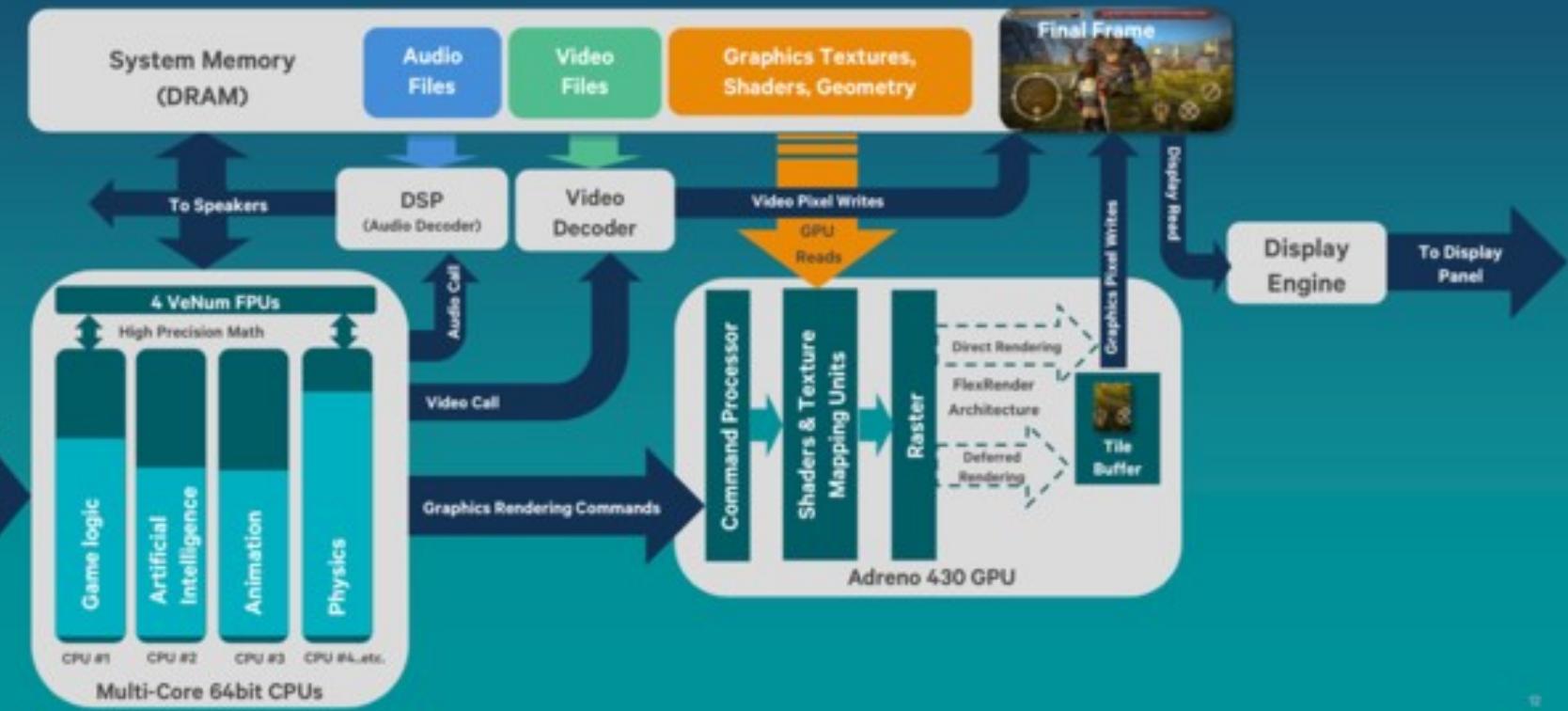


Advantages of heterogeneous architecture for gaming use cases
Heterogeneous hardware blocks and data flow



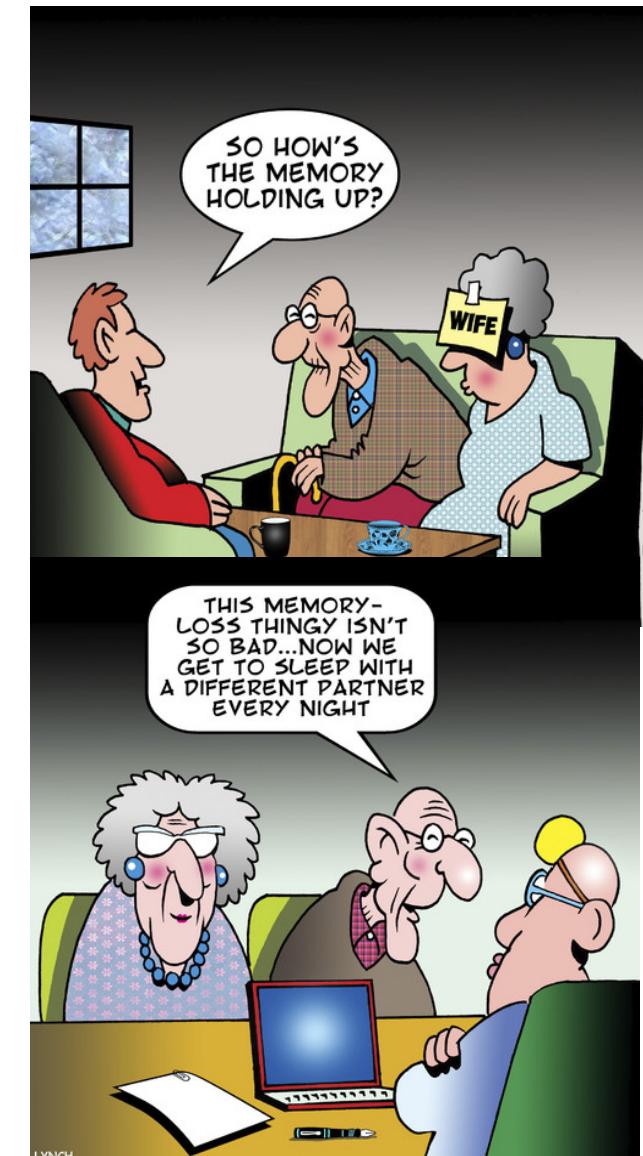
Feladatok – taszkok – végrehajtó egységek

Advantages of heterogeneous architecture for gaming use cases
Heterogeneous hardware blocks and data flow



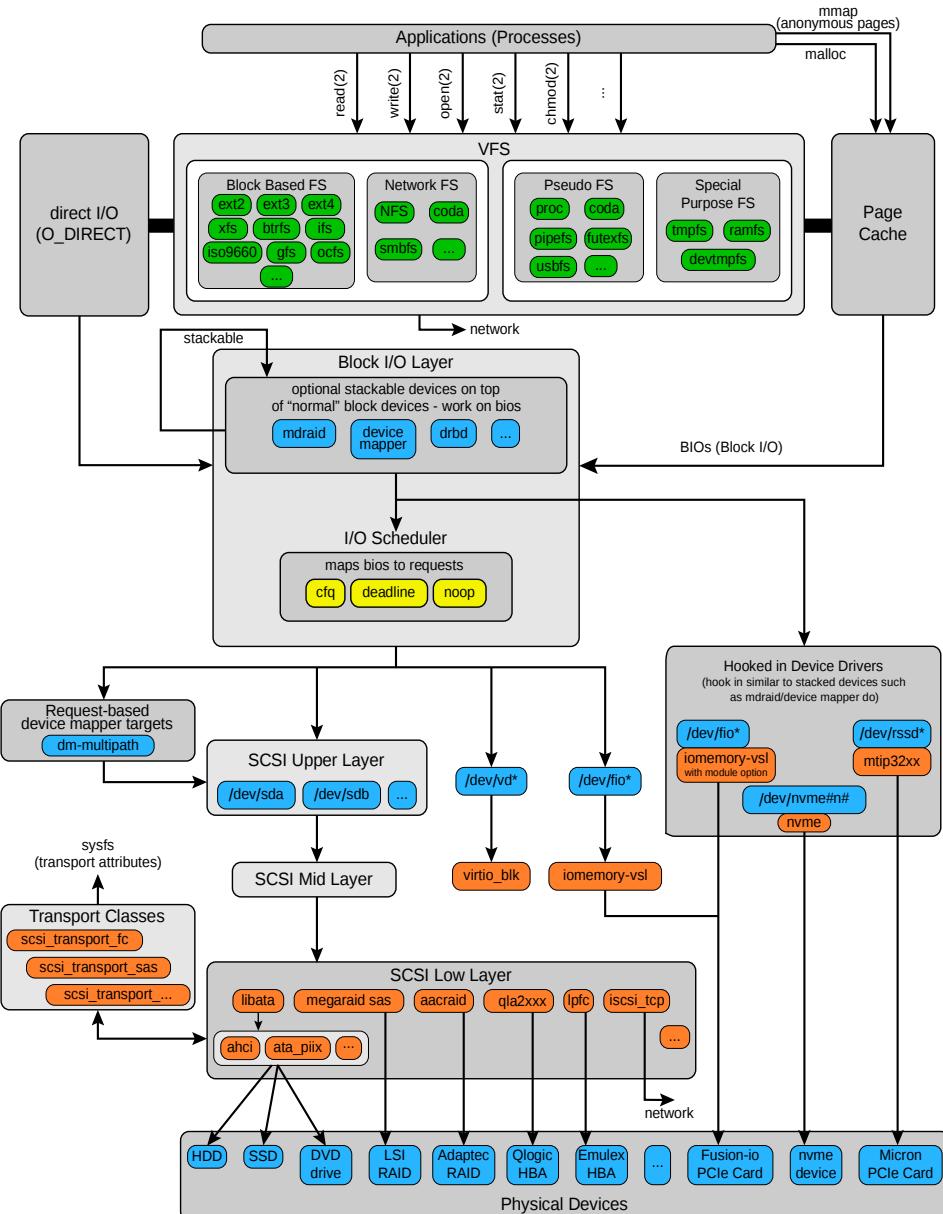
Mivel foglalkozik a memóriakezelés?

- Kiosztja az erőforrást
 - erőforrás: fizikai memória
 - igénylők: taszkok és kernel
- Elhelyezi a taszkok adatait
 - programkód + statikus
 - dinamikusan allokált
- Elhelyezi a kernel adatait
 - programkód
 - adminisztratív adatok
- Biztosítja a védelmet
 - szeparáció
 - hibák
- Támogatja a kommunikációt
 - adatcsere taszkok között



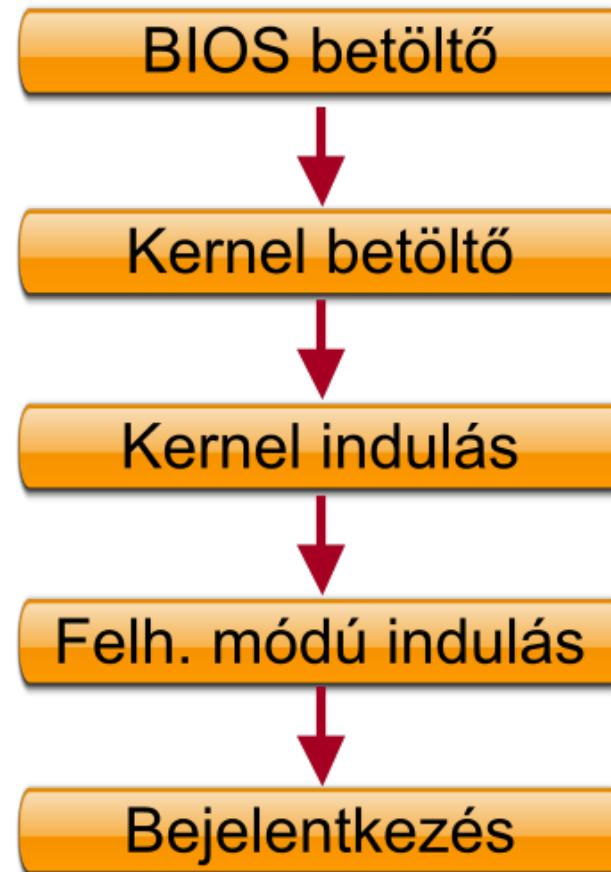
Fájl- és tárolórendszerek

- Felhasználói szemmel...
 - végfelhasználó
 - adminisztrátor
 - programozó
- Belső működés
 - fájlrendszer interfések
 - kernel adatstruktúrák
 - a háttértár szervezése
 - virtuális fájlrendszerök
- Adattárolás
 - fizikai tárolók (HDD, SSD)
 - I/O ütemezés
 - tárolórendszer-virtualizáció:
 - helyi (RAID, LVM)
 - hálózati (SAN, NAS)
 - elosztott fájl- és tárolórendszerek



Az operációs rendszerek működése

Rendszerindulás



Az operációs rendszer indulása (betöltő szint)

- Bekapcsoljuk a gépet
 - elindul a rendszerórajel, ami a processzor inicializációját indukálja
 - a processzor végrehajt egy fix címen kezdődő programot (ROM betöltő)
- 0. szintű (ROM) betöltő (BIOS, boot ROM)
 - hardverinicializálási feladatok (POST)
 - betöltőeszköz meghatározása
- 1. szintű (RAM) betöltő (**MBR** vagy **GPT**)
 - ismeri a háttértár felépítését (partíciók),
 - betölti a következő szintű betöltőt az aktív partícióról
- 2. szintű (OS) betöltő (Boot loader @ PBR / VBR)
 - már ismeri az OS-t (pl. fájlrendszer felépítése, kernel betöltése stb.)
 - betölthet további programrészeket (Windows: **Bootmgr**, Linux: **Grub stage2**)
 - rendelkezhet felhasználói felülettel (az indítandó OS és opcionális megadására)
 - betölti a kernel kódját és elindítja
- Elindul a kernel

A Linux kernel indulása

- Inicializálás nem védett módban
 - alap memóriakezelés, kernel verem, megszakítások stb.
 - indulási paraméterek átvétele
 - alapvető hardverek (pl. konzol / videokártya) beállítása
- Védett (és 64 bites) módba váltás
- Inicializálás védett módban
 - teljes memória, védett módú memóriakezelés
 - megszakításvektorok kezelőfüggvényei
 - a feladatkezelés adatstruktúrái
 - rendszerindulási eszközmeghajtók (initrd: initial ram disk)
 - ütemező
 - további architektúrafüggő feladatok
 - az első feladat, az **init** programkódjának betöltése és elindítása
- Fut az első felhasználói módú folyamat, az **init**

A Windows kernel indulási folyamata

- **Bootmgr** – 2. szintű betöltő
 - védett módba vált
 - még a BIOS eszközkezelőkre támaszkodik
 - megjeleníti a boot menüt
- **Winload** – a kernel betöltője
 - 32/64 bites védett módban működik
 - betölti az Ntoskrnl.exe-t és függőségeit, valamint az induláskori eszközkezelőket
 - átadja a rendszerindulási paramétereket az Ntoskrnl-nek
- **Ntoskrnl** és **HAL** – a kernel és a hardverkezelő
 - 32/64 bites védett módban működik
 - **0. fázis** (phase 0) – inicializálás letiltott megszakításokkal
 - boot processzor, kernel adatstruktúrák, zárolási táblák stb.
 - megszakításvezérlők és -kezelők
 - a memóriamenedzsment és a feladatkezelés
 - **1. fázis** (phase 1) – további inicializálás a normál feladatkezelés keretében
 - az összes CPU beállítása, megszakítások engedélyezése
 - videó (folyamatjelző sáv), I/O és több tucatnyi más alrendszer
 - kernel ütemező
 - elindul a munkamenet-kezelő (SMSS)

A Windows felhasználói módú kritikus folyamatai

- **SMSS** – Munkamenet-kezelő (session manager)
 - a felhasználói módú működés alapvető beállítása
 - fájlrendszerek ellenőrzése
 - környezeti változók
 - lapozófájlok
 - registry
 - Wininit indítása (S0 InitialCommand)
 - munkamenetek (CSRSS) és bejelentkezés-kezelő (Winlogon) indítása (S1+)
 - A Winlogon kilépéséig fut (vár)
- **Wininit** – további felhasználói inicializálási lépések (Session 0)
 - pl. Service Control Manager (services.exe)
- **CSRSS** – Win32 alrendszer indítása (Session 1+)
 - további hardver-inicializálás (pl. teljes grafikus felbontás)
- **Winlogon** – felhasználói bejelentkezés (Session 1+)
 - bejelentkezási képernyő megjelenítése (LogonUI)

A Unix rendszerek felhasználói módú indulása

- Az init indítja...
 - meghatározza és fenntartja a rendszer működési szintjét (**/etc/inittab**)
 - elindítja/leállítja a szükséges OS szolgáltatásokat
- **Futási szint** (runlevel)
 - az OS állapotleírása
 - a működési aktuális módja (karbantartás, többfelhasználós, grafikus stb.)
 - rendszerszolgáltatások köre
 - jellemzően számmal (0-6), vagy betűvel jelölik
 - **0**: teljes leállás
 - **1** vagy **S**: single-user: egyfelhasználós (adminisztrátori) mód
 - **2-5**: többfelhasználós üzemmódok (GUI, ha van)
 - jellemzően az **5** az alapértelmezett teljes felhasználói mód
 - **6**: újraindítás
 - A rendszergazda válthatja: telinit, init, shutdown, halt, reboot
 - Lekérdezhető: who -r
- Az android-alapú rendszerek **indulása** hasonló

A Unix rendszerszolgáltatások kezelése

- Az init működése
 - konfiguráció: /etc/init.d/ és /etc/rc?.d/ (? a runlevel)
 - a parancsfájlok
 - nevüknek megfelelő sorrendben futnak le
 - beállítják a rendszert
 - pl. fájlrendszer ellenőrzése és csatolása
 - elindítják, illetve leállítják a szolgáltatásokat
 - pl. felhasználói bejelentkezés, grafikus felület, adatbázis- és webszerver stb.
- Az adminisztrátor meghatározhatja az aktív rendszerfeladatok körét
 - a parancsfájlok manuálisan is meghívhatók

```
service <parancsfájl-neve> <start|stop|restart|...>
```

- beállítható az adott futási szinten aktív szolgáltatások, elvégzendő feladatok köre

ntsysv, tksysv, chkconfig, bum

A Sysinit alternatívái

- Mi a gond az init-tel?
 - egysíkú függőségek (fájlok nevei)
 - lassan (egyesével) indítja a szolgáltatásokat
 - hibakezelés?
- **Systemd** (RedHat, CentOS, Ubuntu 15.04+, Arch Linux, Debian stb.)
 - deklaratív szolgáltatásleírások, pontosabb függőségek
 - párhuzamos / késleltetett indítás
 - működési hibák észlelése és kezelése
 - megváltozott parancskészlet:

```
systemctl <start|stop|restart|...> <szolgáltatás/feladat>
```
- Futottak még:
 - a bevezetése főleg Debian/Ubuntu-körökben elég sok [vihart kavart](#)
- Upstart (Debian és Ubuntu korábbi változatai, Google Chrome/Chromium OS)

Hogyan áll le a Windows operációs rendszer?

- **ExitWindowsEx()** rendszerhívás (sokféle paraméterezés)
 - leállítja a futó alkalmazásokat
 - kijelentkezteti a felhasználót
 - kezdeményezi az operációs rendszer leállítását
- Explorer → Winlogon
 - Start menü – Leállítás (Explorer)
 - Értesíti a CSRSS-t a rendszer leállítási kérésről
- CSRSS (Win32 alrendszer)
 - kilépteti az összes felhasználót (session 1+) az alábbiak szerint
 - végiglépked az összes futó alkalmazáson (shutdown order)
 - és leállítja azokat
 - ha egy taszk nem áll le (HungAppTimeout), akkor jelzi ezt a felhasználó felé
 - hasonló módon leállítja a szolgáltatásokat (session 0)
 - a HungAppTimeout itt is működik, csak nincs jelzés róla
- Winlogon
 - az összes CSRSS leállítási procedura után
 - meghívja az NtShutdownSystem() rendszerhívást
 - amely a PoSetSystemPowerState() híváson keresztül a kernelszintű részek leállítását végzi

Hogyan áll le a Unix rendszer?

- Elindítjuk a leállítási folyamatot
 - pl.: shutdown +60 "System going down for regular maintenance"
- Az init (PID 1) értesül a leállítási szándékról
 - értesíti az interaktív felhasználókat a leállításról
„Broadcast message from root@localhost
The system is going down for regular maintenance in 60 seconds”
 - a szolgáltatások konfigurációi alapján leállítja azokat a megfelelő sorrendben
 - szinkronizálja majd lecsatolja a fájlrendszerket*
 - leállítja a még futó taszkokat
 - elindítja a kernel leállítási folyamatát ([reboot\(\)](#))
- A kernel leállítása
 - letiltja a felhasználói módú működést
 - értesíti a komponenseit a leállásról ([reboot_notifier_list](#))
 - leállítja a hardvereszközöket a vezérlőken keresztül
(lecsatolja a még megmaradt fájlrendszerket)
 - leállítja az alapvető funkcióit (pl. megszakításkezelés)
 - leállítja a számítógépet

A kernel belső felépítése

Mekkora a kernel forráskódja?

- Példák:
 - World of Warcraft: **5.5 millió programsor** (LOC)
 - Windows XP: **45 millió LOC** (nem csak a kernel)
 - Linux kernel: ~60 ezer fájl, ~25 millió LOC (~ fele eszközkezelés)
 - MINIX alap kernel < 1400 LOC, teljes kernel kb. 5000 LOC
- Érdekességek
 - <http://www.informationisbeautiful.net/visualizations/million-lines-of-code/>
 - <http://www.pabr.org/kernel3d/kernel3d.html>
 - <http://www.jukie.net/bart/blog/linux-kernel-walkthroughs>
 - http://en.wikiversity.org/wiki/Reading_the_Linux_Kernel_Sources
 - Linux vs. Windows kernel (videó, Mark Russinovich)
 - Minix OS fut Intel AMT mikrocsipekben (Tannenbaum levele az Intelhez)



A kernel felépítésének alapelvei

- Réteges
 - jól definiált (szabványos?) **interfészekkel**

A rendszerhívás interfész egy programozói felület, amely a kernel felhasználói módban működő programok számára nyújtott szolgáltatásait tartalmazza.

- Monolitikus
 - a kernel részei egyetlen címtérben találhatók
 - egyszerű fejlesztés vs. megbízhatóság

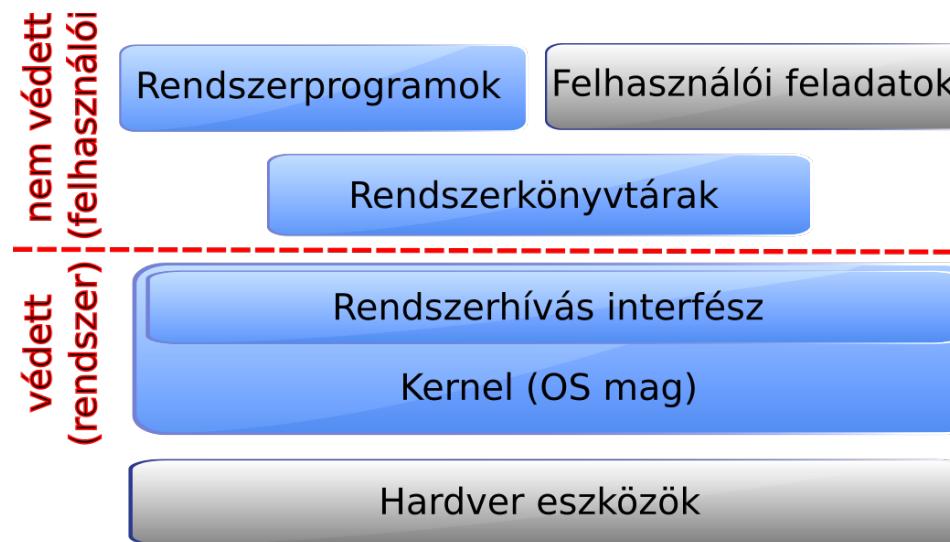
A mai kernelek jellemzően moduláris, monolitikus szoftverek

- Moduláris
 - nem érhető el minden rész
 - fordítási időben / konfiguráció során / futásidőben

*Linux: vmlinu
Windows: ntoskrnl.exe*

- Elosztott
 - önálló komponensek (külön címtérben)
 - üzenetalapú kommunikáció

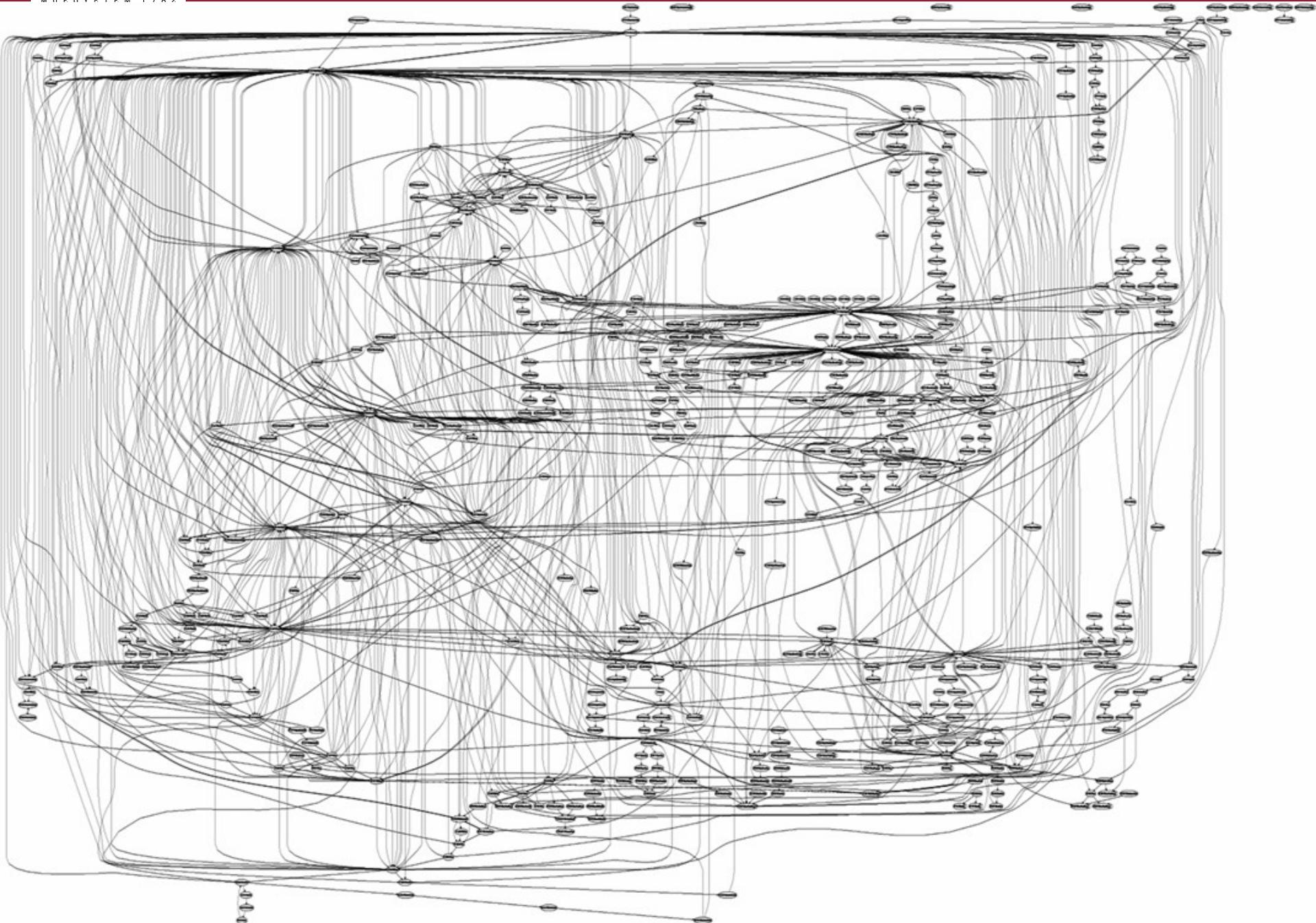
A rendszerhívások működése



- A rendszerhívás látszólag egy függvényhívás, de valójában más
 - üzemmódváltás szoftver megszakítással
trap, syscall, sysenter
 - a kernel megszakításkezelője
 - átveszi a paramétereket (nem a vermen keresztül)
 - végrehajtja a feladatokat
 - visszatér a megszakításból (iret, sysexit)

Rendszerhívások működése Unix alatt

- a rendszerhívás meghívása (`read()`, `write()` stb.)
 - klasszikus függvényhívásnak tűnik
 - a libc rendszerkönyvtár implementálja
 - csak a valódi rendszerhívás előkészítését végzi
- a `libc` kiadja a `SYSCALL` utasítást (megszakítást generál)
- a `kernel` `SYSCALL` kezelője előkészíti a rendszerhívás végrehajtását
- végrehajtódik a `kernel` módú eljárás (a tényleges rendszerhívás)
- a `kernel` visszatér a megszakításból (`iret`, `sysexit`)
 - folytatódik a `libc` segédfüggvénye, amely beállítja a visszatérési értéket
- a `libc` visszatér a folyamat által meghívott függvényből



Virtuális rendszerhívások

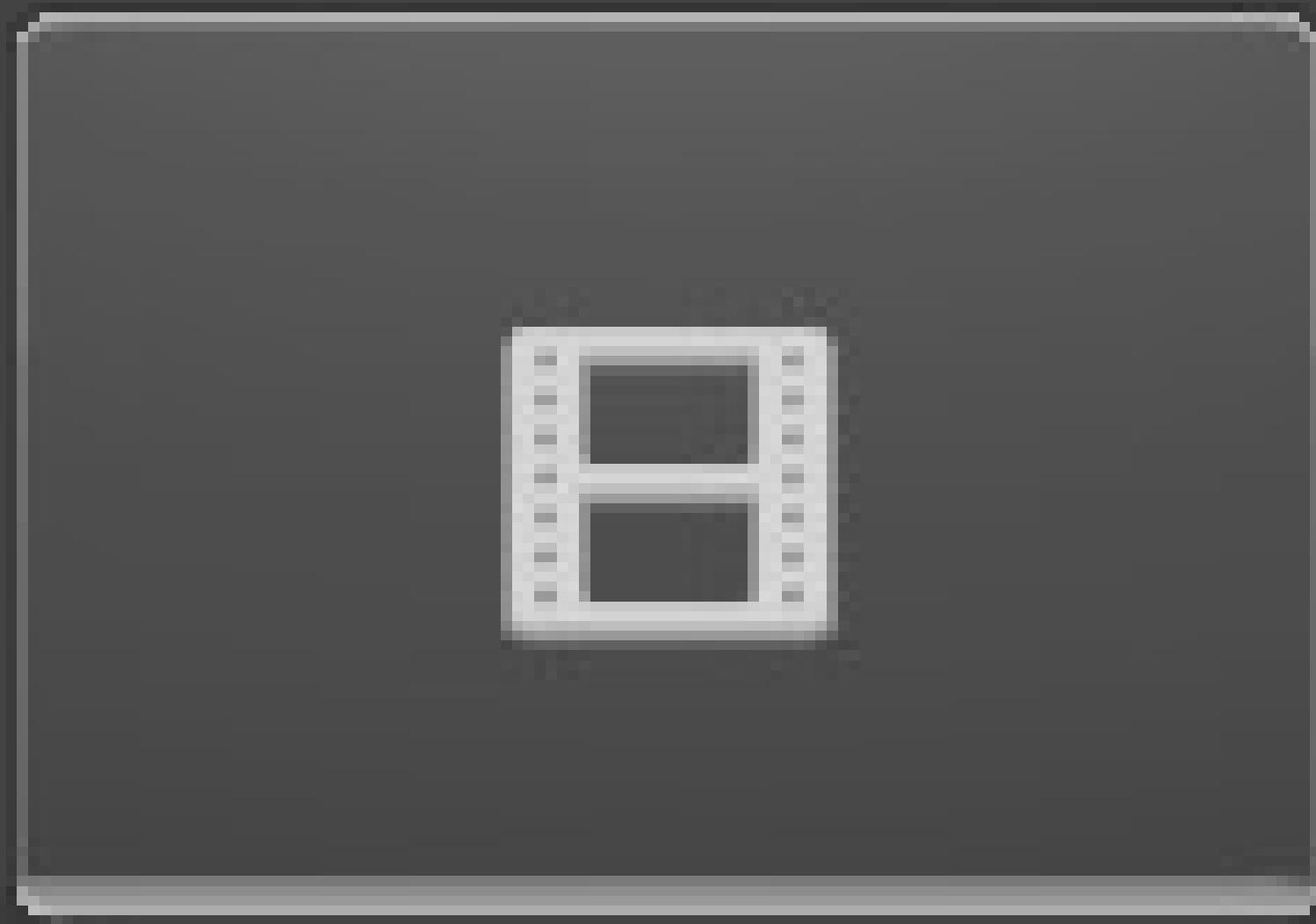
- A rendszerhívás rendkívül gyakori és költséges eljárás
 - szoftver megszakítás
 - üzemmódváltás
 - megszakítás-kezelés
- Hogyan csökkenthető a rezsiköltség?
 - ötlet: próbáljuk elkerülni a megszakítást és az üzemmódváltást
 - bizonyos kernel funkciók elérhetők a felhasználói címtérben
- Virtuális rendszerhívások (Linux)
 - speciális „kernel” memóriaterület felhasználói címtérben
 - biztonságosnak ítélt rendszerhívások érhetők el rajta
 - szoftver megszakítás és módváltás nélkül működnek
 - a felhasználók programjai nem látják a különbséget (a libc igen)

Mi a baj a kernelek felépítésével?

- Hatalmas kódbázis, és **emberek írják**
 - 1000 soronként kb. 10-100 hiba az átadott programban ([forrás](#))
 - egy mai kernel több millió programsor.....
 - ~~hibaizoláció, futásidejű javítás hibák, kártevők~~

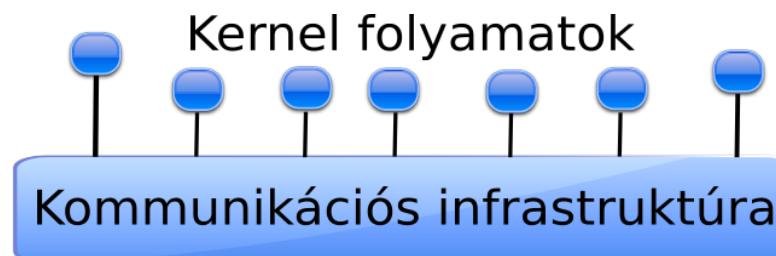


Forrás: [Linux.com](#) (2016)

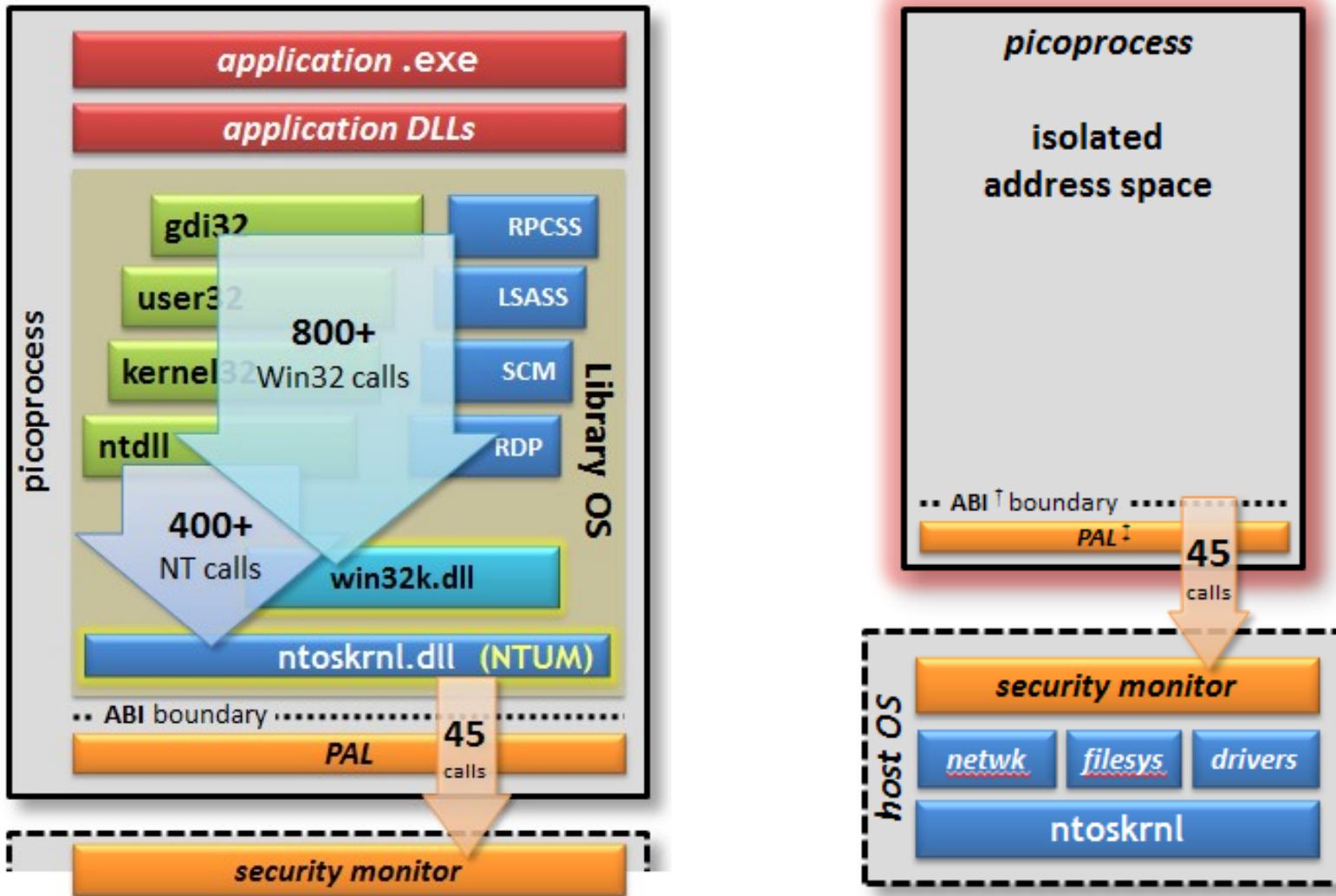


Mit tehetünk a helyzet javítása érdekében?

- Kernel sandboxing armored OS
 - a kezelőfüggvényeket „védőréteggel” látja el (hibadetektálás)
 - egy felhasználói módú helyreállító ágens kezeli a felmerült problémákat
- OS/app sandboxing KVM/vmware, Docker, MirageOS, Drawbridge
 - kisebb felületű rétegek, erősebb szeparáció
 - virtualizáció: még egy felügyeleti szint
 - konténerek: ugyanazon a kernelen független OS-ek
 - unikernel: mini kernel (library OS) + alkalmazás egy címtérben
- Kidobjuk a monolitikus felépítést
 - elosztott rendszer (feladat-végrehajtók és kommunikáció)
 - védett módban csak a legszükségesebb részek működnek



Windows Library OS és pProcess (Drawbridge koncepció)

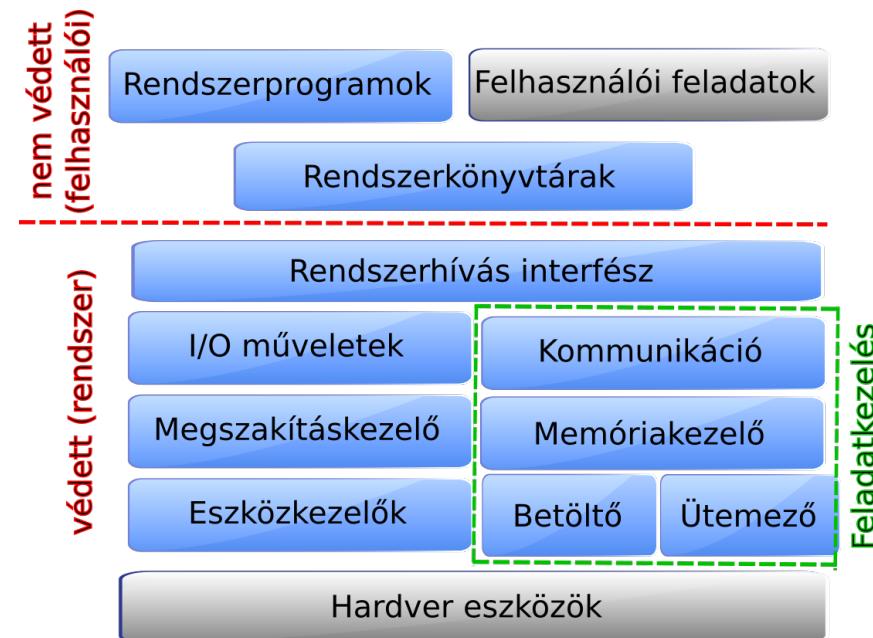


Forrás: microsoft.com

A mikrokernel

A mikrokernel egy olyan operációs rendszer kernel, amely csak az alapműködéshez feltétlenül szükséges kódrészleteket tartalmazza, minden más funkciót felhasználói módban működtet.

Mikro- vs. monolitikus kernel



Újgenerációs mikrokernelek

- L4 mikrokernel
 - akár CPU regiszterekben is átvihető az üzenet
 - 10-20-szor gyorsabbak, mint a klasszikus mikrokernel IPC
 - nagyon kevés védett módú funkció (az L4 API 7 funkcióval rendelkezik)
 - a védett módú kernel nagyon kicsi (5-15 ezer programsor)
 - speciális ütemezést alkalmaznak (sok a blokk az IPC üzenetek miatt, ezt kezelik)
 - erősen hardverfüggőek (még x86-on belül is sokféle implementáció szükséges)

a kis kernel lehetővé teszi a formális leírást és a verifikációt

- Hibrid kernelek
 - monolitikus rendszerekkel vegyített mikrokernelek
 - OS X [XNU](#) (az Apple kernele), egy Mach mikrokernel + BSD Unix hibrid kernel
 - Kísérleteztek L4 mikrokernelre épülő reinkarnációjával is, lásd [Lee & Gray, 2006](#)
 - A Windows is tartalmaz mikrokernel elemeket, de nem mikrokernel felépítésű.

Az L4 mikrokernel családfája

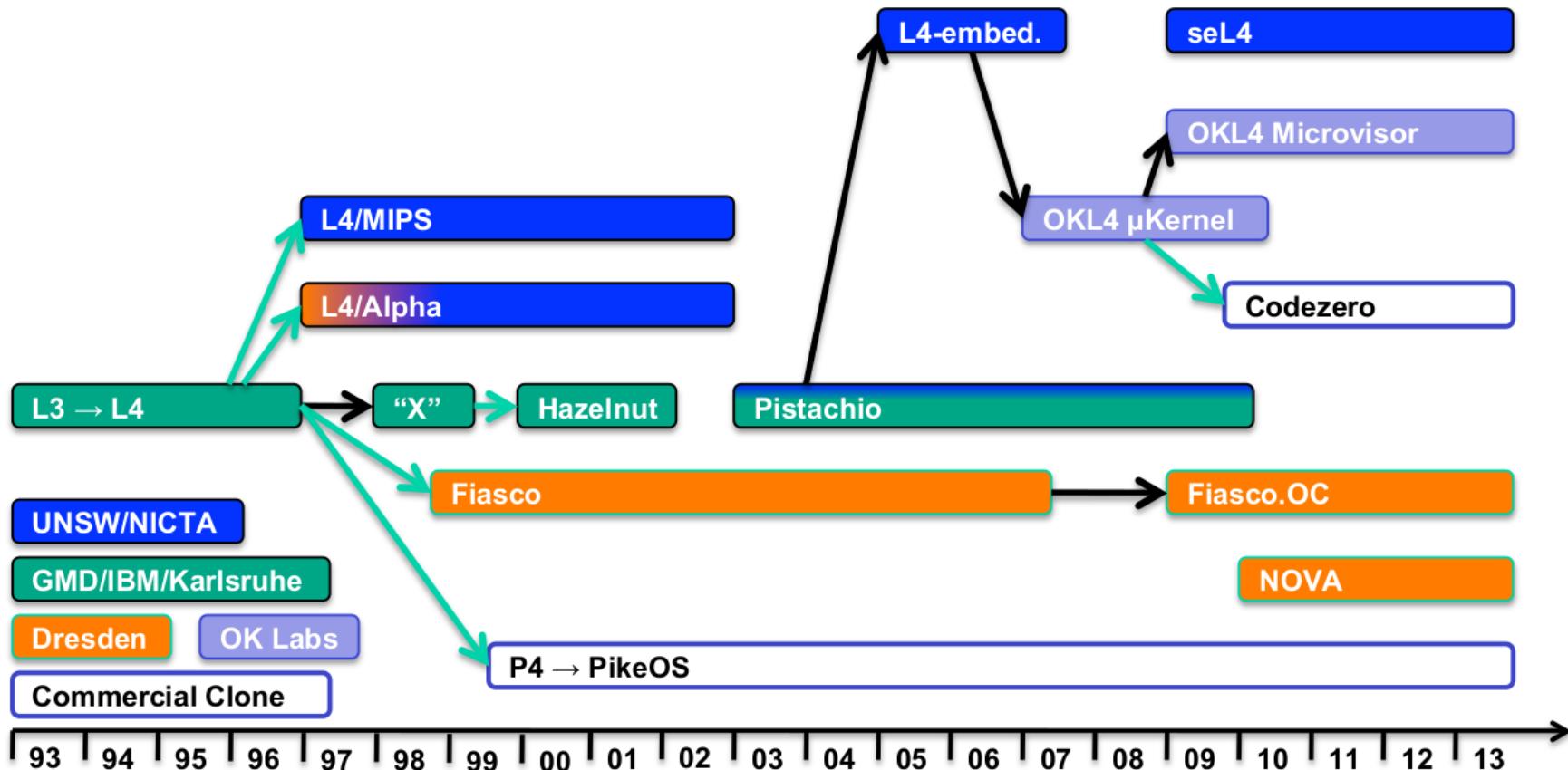


Figure 1: The L4 family tree (simplified). Black arrows indicate code, green arrows ABI inheritance. Box colours indicate origin as per key at the bottom left.

Forrás: Kevin Elphinstone , Gernot Heiser, From L3 to seL4 what have we learnt in 20 years of L4 microkernels?

Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, November 03-06, 2013, Farmington, Pennsylvania

Összefoglalás

- A kernel egy komplex program
 - jellemzően réteges, moduláris és monolitikus szerkezetű
 - ez utóbbi számos megbízhatósági és biztonsági problémával küzd
 - a mikrokernelek próbálnak ezen segíteni
- Az operációs rendszer indulása egy komplex eljárássorozat
 - ROM, RAM, OS és kernel saját betöltő
 - kernel és felhasználói módú indulás
- Az operációs rendszer működése
 - rendszerszolgáltatások
 - felhasználói munkamenetek
 - rendszerhívások
- Javasolt otthoni gyakorlatok (virtuális géppel)
 - OS telepítése, rendszerindulás, szolgáltatáskezelés, rendszerhívás-nyomkövetés

Operációs rendszerek felhasználói felületei

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

Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

Az előadásfóliák legfrissebb változata a tantárgy honlapján érhető el.
Az előadásanyagok BME-n kívüli felhasználása és más rendszerekben történő közzététele előzetes engedélyhez kötött.

Az eddigiekben...

- Az operációs rendszer
 - biztosítja a felhasználói feladatok megoldásához szükséges környezetet
 - kezeli a hardvert
- Sokszínű feladat
 - számítási, üzleti, személyi, szórakoztató, kommunikáció, „okosítás”, ...
- Sokféle környezetben
 - PC / mobil / beágyazott stb.
 - grafikus / karakteres kezelői felület
- Széles felhasználói tábor
 - informatikában nem jártas emberek
 - gyerekek, idősek
- Elvárások
 - felhasználóbarát („hordozható”), élvezetes

A feladatok és az operációs rendszer

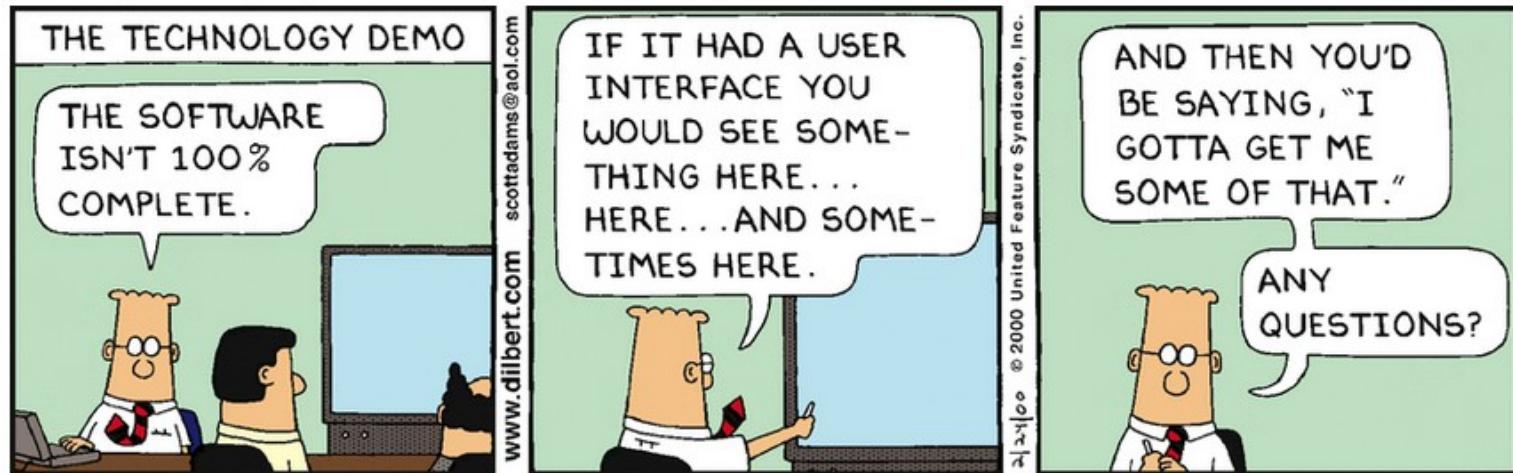
- Az OS egyszerre sok feladattal foglalkozik
 - saját feladatai és szolgáltatásai
 - felhasználói bejelentkezéshez kötődő feladatok
- Az összetartozó feladatok csoportosíthatók
 - a **munkamenet** (session) egy feladatcsoport
 - munkavezető (Session ID)
 - vezérlő terminál kapcsolódhat hozzá
 - esetenként menthető és visszaállítható (pl. X Session Manager)
 - egy vagy több folyamatcsoportból áll (pl. előtérben és háttérben futó folyamatok)
- A futó programok is összefoghatók
 - a **folyamatcsoport** (process group) taszkok összetartozó halmaza
 - csoportvezető (létrehozó)
 - azonosító (PGID)
 - bizonyos eseményekről az egész csoport értesül
 - egységként vezérelhetők



Photo by: kubliczech

Felhasználói felületek





A parancsértelmező

- Felhasználó ↔ OS ↔ programok
 - az OS kezelése (parancsok kiadása)
 - programok kezelése
- A felhasználó parancsokat ad
 - az OS **parancsértelmezője (shell)** értelmezi azokat
 - belső parancsok
 - a parancsértelmező hajtja végre
 - külső parancsok
 - rendszer- és felhasználói programok oldják meg a feladatokat
- Külső parancs végrehajtása
 - a parancsértelmező meghatározza a program elérési helyét (PATH)
 - elindítja a programot az argumentumokkal (l. argv[])
 - beállítja a környezeti változóit (l. env és environ)
 - összeköti a futó programot a felhasználóval
 - kezeli a munkamenetét
 - visszaadja a végrehajtás eredményeit és az esetleges hibákat

Felhasználói felület típusok

- Karakteres (parancssori, TTY)
 - stdin – stdout
 - minden rendszeren elérhető
 - billentyűzet és monitor, soros vonal, hálózat stb.
 - korlátos felhasználói „élmény” (demo)
 - hatékony, gyors, jól automatizálható (!!)
- Grafikus (GUI)
 - **WIMP**: Windows, Icons, Menus, Pointer
 - ablakozó rendszer (windowing system)
 - ablakkezelő (window manager)
 - kijelzőszerver (display server)
 - nem mindenhol érhető el (erőforrásigényes, felesleges)
- A WIMP-en túl
 - hangalapú
 - gesztusokkal irányítható (mobil)
 - ...

A karakteres parancsértelmező

Unix: **bash**, csh, ksh, zsh stb.

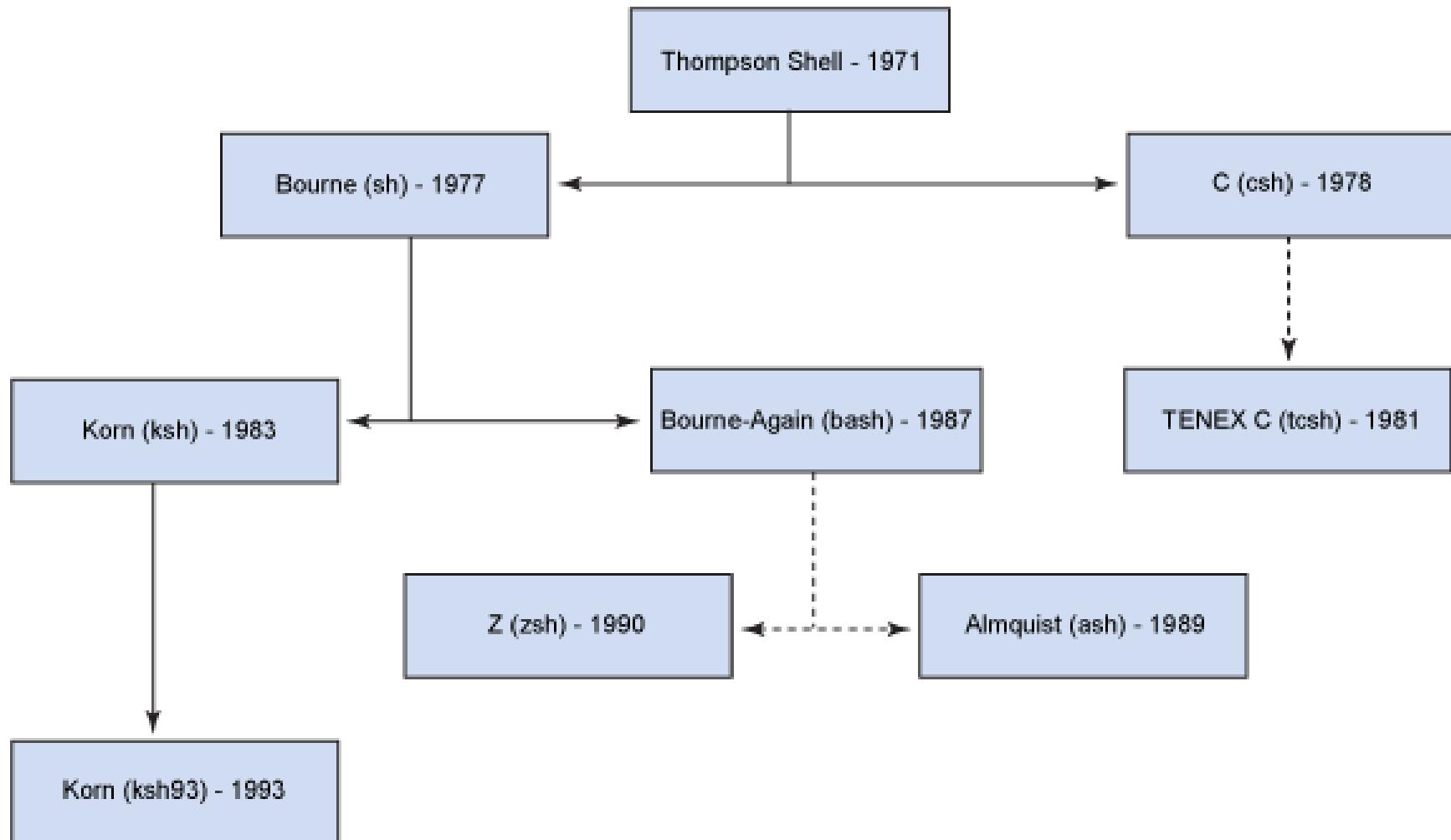
Windows: cmd.exe → **PowerShell**, illetve **bash @ WSL**

- **Belső parancsok**
 - **bash builtins**: logout, alias, echo, read, source, ulimit stb.
 - PowerShell **kulcsszavak**, Cmdlets core és külső **modulok** (sokféle **feladatra**)
- **Programozható (interpretált nyelv)**
 - OS menedzsment
 - szövegkezeléssel kapcsolatos feladatokra is kiváló
 - programozási nyelvek alapvető konstrukciói
elágazás, ciklus, eljáráshívás, makró stb.
 - külső parancsokat is használhatunk
- **Beépített súgó:**

man <parancs>
<parancs> -h vagy --help

Get-Help <parancs>
<parancs> /h vagy /?

Unix Shell családfa



Forrás: <http://www.ibm.com/developerworks/>

„Mi történik a rendszerben?” (demo)

- Futó programok listázása (Unix)
 - ps, ps -ef, ps axu, ps -u <felhasználó>, pstree, ...
 - top, atop, htop és társaik

```
$ ps -ef
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1	0	0	07:59	?	00:00:00	/sbin/init
...							

```
$ ps -eo pid,pmem,rss,comm --sort -pmem
```

PID	%MEM	RSS	COMMAND
5006	5.9	970680	firefox
3336	3.2	527936	thunderbird
2358	1.7	285004	gnome-shell
2708	1.5	250644	soffice.bin
...			

- Milyen adminisztratív adatokat látunk a programknál? (**man ps**)

A shell programozás alapjai: parancsok

- Beépített nyelvi elemek (bash, PowerShell)
 - programozási konstrukciók (ciklus, elágazás, eljárás, **változók**, kiírás stb.)

```
if [ "$1" == "az opre remek" ]; then echo OK; fi
:() { :|:& };: # értelmezés?
```
 - OS feladatok megoldása
`Stop-Service -displayname "Bluetooth service"`
- Külső parancsok
 - minden parancssori felülettel (is) rendelkező alkalmazás
 - pl. PowerShell: .NET objektumok manipulálása
 - pl. Unix: sokféle szövegfeldolgozó eszköz (demo)
`grep, sed, awk, sort, uniq`
 - Külső parancsok elérési útvonala: **\$PATH**
`C:\Program Files;C:\Winnt;C:\Winnt\System32`
`echo $PATH`
- Folyamatcsoport-vezérlés (job control)
`fg, bg, jobs (demo)`

A shell programozás alapjai: átirányítás

- Stdout / stderr - stdin

```
Get-Service | Where-Object {$_.DependentServices -ne $null}  
curl -so - https://www.kernel.org/doc/linux/MAINTAINERS | \  
grep "^\M:" | wc -l  
tr -d '\r' < dos-file.txt > unix-file.txt  
strace ls 2>&1 | less
```

- Speciális fájlok

```
strace -e trace=openat ls > /dev/null  
dd if=/dev/zero of=/root/filesystem.img bs=1k count=1000  
read password < /dev/tty
```

- Parancsok összefűzése

```
apt update ; apt upgrade  
apt update && apt upgrade  
grep '^opredemo:' /etc/passwd || useradd opredemo
```

A shell programozás alapjai: változók, regex

- Változók
 - név, érték

```
PATH=$PATH:/usr/local/bin
```
- Láthatóság
 - függvényen belül
 - folyamaton (futó programon) belül
 - globális

```
export $PATH
```
- Változókezelés beépített regex-műveletekkel (demo)

```
 ${v}  ${#v}  ${v#* }  ${v% *}  ${v/remek/kerek}
```
- Népszerű sztring-manipulátorok: grep, awk és sed

```
echo 4.3.2.1 | awk -F . '{print $4 "." $3 "." $2 "." $1 }'  
sed 's/csoki/torna/4'          # minden 4. csoki helyett torna  
ps -ef | grep "[f]irefox"
```

Grafikus felhasználói felületek



Microsoft Bob. Forrás: freewarewiki.com

Grafikus felhasználói felületek

- **WIMP:** Windows, Icons, Menus, Pointer
 - ötlet: Xerox PARC (1973)
 - elterjedés: Apple Macintosh (1984)
 - közös felhasználói elemek

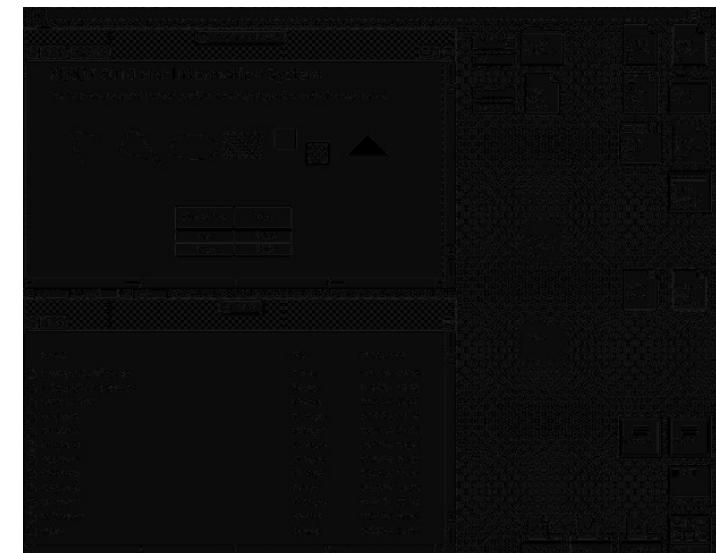
lásd [The Paradigm Project](#)

- Szabványosítás

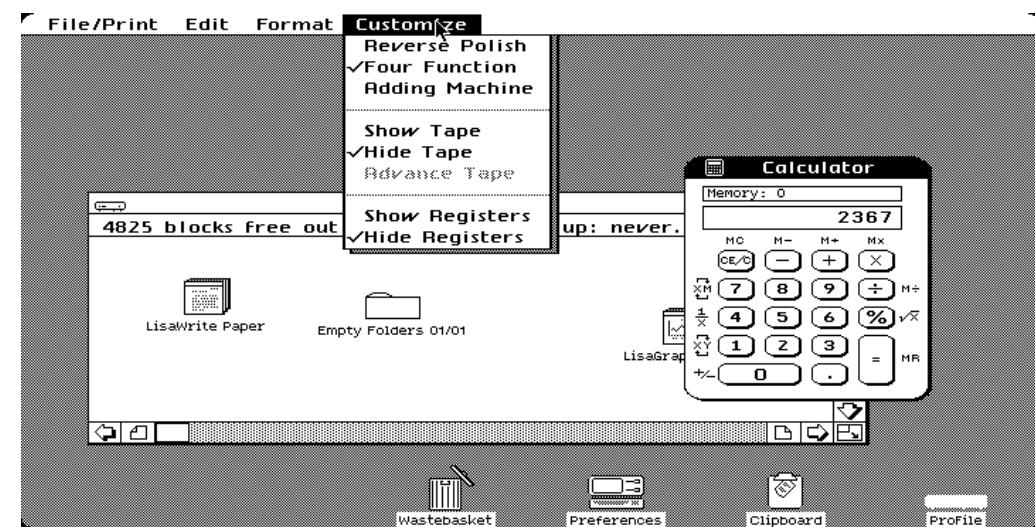
- de jure: sikertelen kísérletek
- de facto:
 - Apple Macintosh
 - [IBM Systems App Arch.](#)
 - [MIT X Window System \(X11\)](#)
 - és sokan mások

- Komplex rendszer

- ablakozó környezet
- ablakkezelő
- kijelzőszerver



Xerox 8010 Star. Forrás: [toastytech.com](#)



Apple Lisa Office System 1.0. Forrás: [GULdebook](#)

Grafikus felhasználói felület és ablakkezelő

- GUI: összetett szoftver
 - réteges, moduláris, sok nyílt interfész és protokoll
 - parancsértelmező + ablakkezelő + kijelzőszerver
 - pl.: Windows Shell, Gnome Shell, Ubuntu Unity, KDE, XFCE stb.
- Ablakkezelő (window manager, WM)
 - alkalmazásablakok elhelyezése és megjelenítése
 - + programozási felület és rendszerkönyvtárak
 - testre szabható és bővíthető
 - példák
 - Windows Desktop Window Manager (dwm.exe)
 - Unix KDE: KWin, Gnome2: Metacity, Gnome3 (új Gnome Shell): Mutter

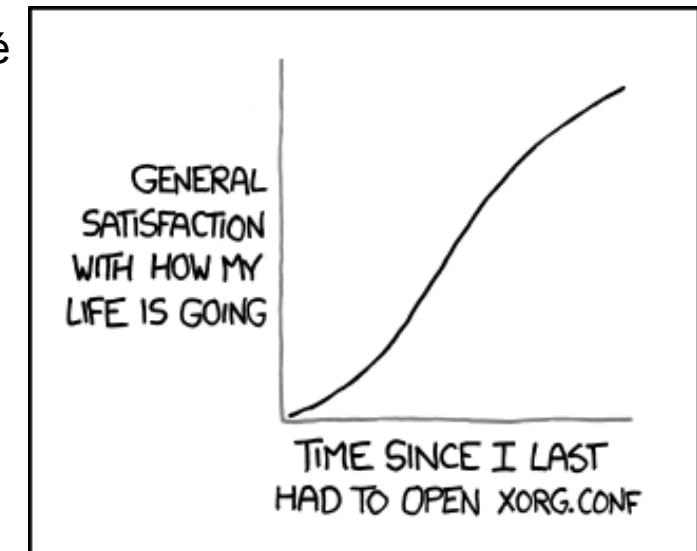
kompozíciós ablakkezelő (compositing window manager)

[demo](#)

- az alkalmazások saját kijelzőpuffert kapnak
- a végeleges vizuális képet az ablakkezelő állítja össze úgy, hogy
- közben különféle grafikus transzformációkat hajt végre a pufferek tartalmán
- jellemzően 3D hardvergyorsítást használva, látványos vizuális effektekkel

A kijelzőszerver (display server)

- A GUI és a kiszolgáló hardver kapcsolata
 - API és protokoll
 - GUI továbbítása a hardver felé
 - hardveresemények továbbítása az alkalmazás felé
 - felhasználói módban működik
- Kijelzőszerverek és protokollok
 - Unix X11
 - X.Org Server, XFree86
 - Windows alatt: Xming X Server, Cygwin/X
 - Unix Wayland
 - kompozíciós rendszer
 - terjed... (beágyazott rendszerekben is: Raspberry Pi, Tizen)
 - **Mir**: hasonló kezdeményezés, de csak Ubuntu alatt látható támogatással
 - Android SurfaceFlinger
 - kompozíciós rendszer, virtuális kijelzők megjelenítése
 - További hálózati protokollok
 - Microsoft RDP, Citrix HDX, SPICE, VNC RFB stb. (lásd még PC-over-IP)



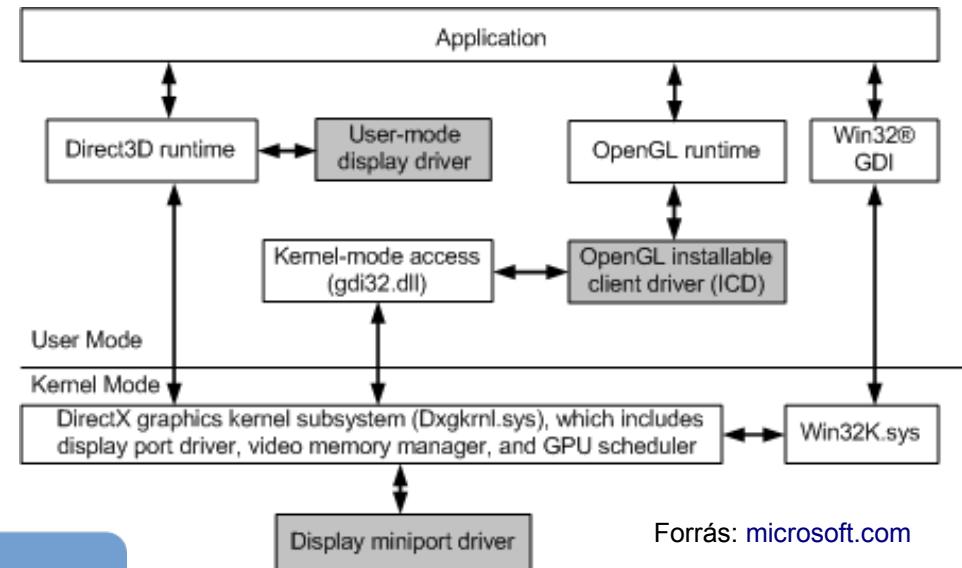
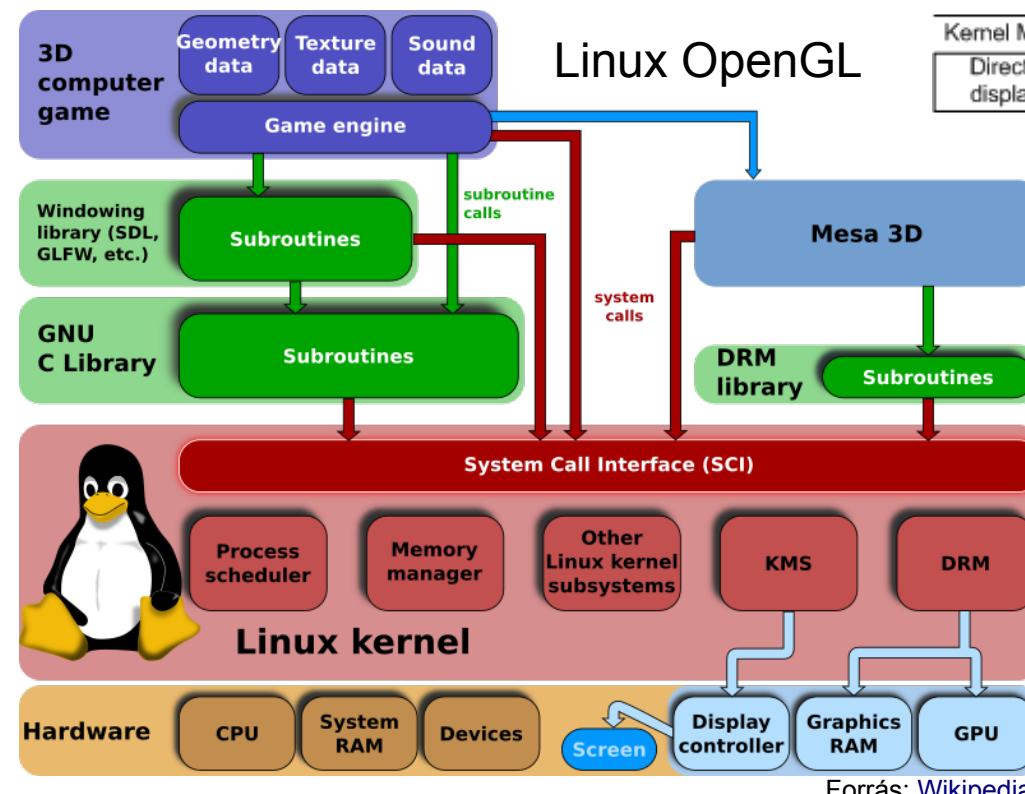
Forrás: [xkcd](#)

A Linux és Windows megjelenítők áttekintése

Linux rendering

KMS: videoüzemmód beállítása

DRM: adatküldés a videokártyának



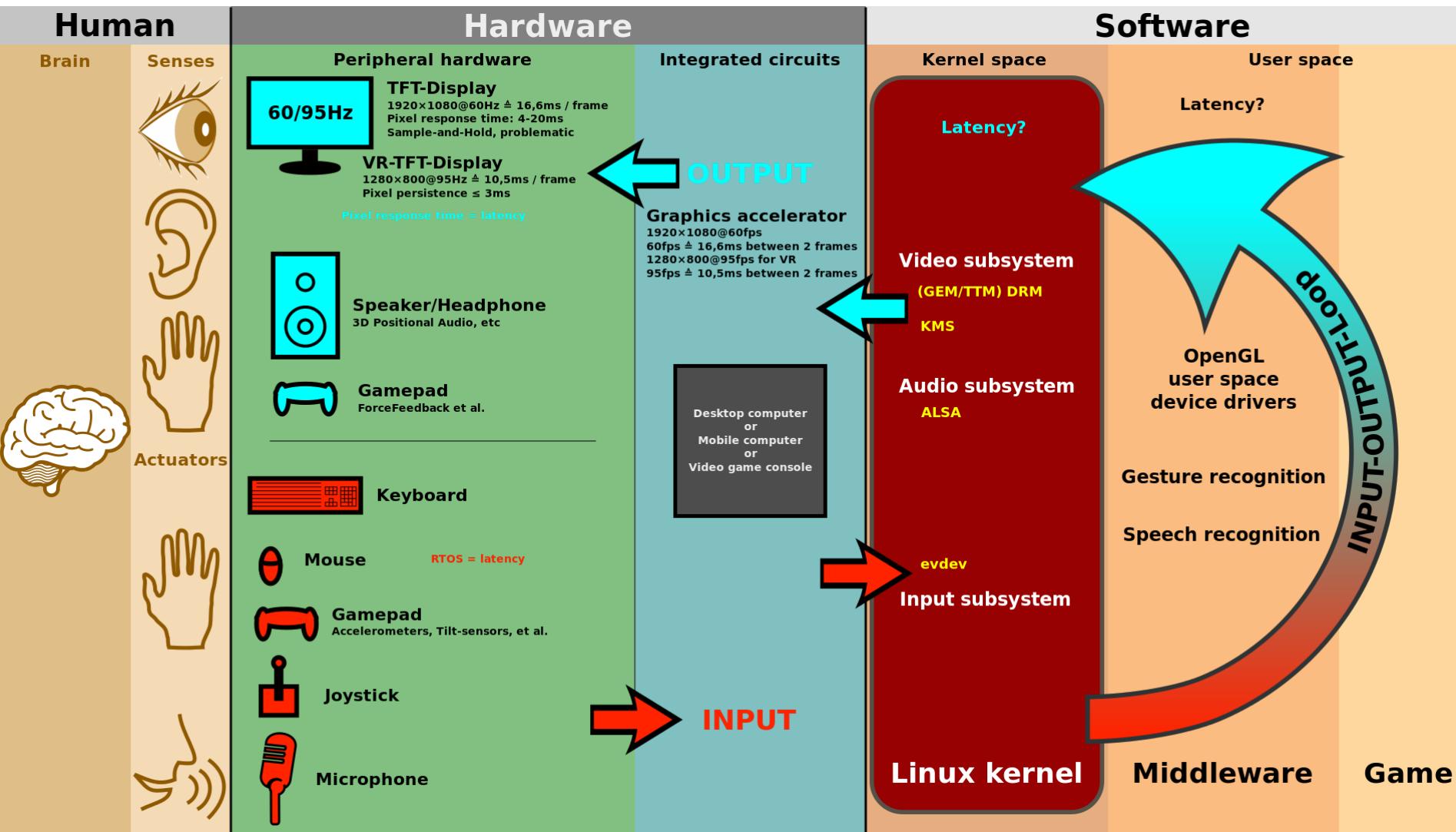
Windows rendering

GDI, Direct2D, Direct3D API-k: grafikai elemek megjelenítése

DirectX: multimédia megjelenítő API

OpenGL: platformfüggetlen API
2D és 3D vektorgrafikai elemekhez

A GUI I/O-működésének áttekintése



Forrás: Wikipedia

Az ablakokon túl ... (áttekintés)

- Sok elbukott kísérlet
 - (demók: Microsoft Bob, Solaris 1994 → 2004)
- Az emberi kommunikáció (nonverbális is) gépi alkalmazása
 - beszédalapú, kézírás-felismerő és természetes nyelvű rendszerek
 - pl. Apple Siri, Google Voice, Microsoft Cortana, Amazon Echo
 - vizuális „nyelvek” használata
 - gesztikuláció, mozgáselemzés (Wii Remote, Xbox Kinect)
- „Láthatatlan” interfések („Natural user interfaces”)
 - viselhető számítógépek
 - beágyazott rendszerek (ipari és szórakoztató elektronikai egyaránt)
- Virtuális (VR), kiterjesztett (AR) valóság és hibrid (MR) rendszerek
 - 3D imersive (belemerítő? körülölelő?) felületek (pl. Microsoft [HoloLens](#) demo)
- Közvetlen agyi interfések
 - Emotiv Eloc (témaink [korábban](#) és [most](#)), lásd [IEEE SMC 2016 BMI Hackaton](#)

témakiírásaim

Természetes nyelvű felhasználói felületek

- Az emberi kommunikáció meghatározó formája
 - mindenki ismeri és használja
 - ismeretek megosztása, tudás leírása, információk közzététele
 - befolyásolhatja a világot ([beszédaktus-elmélet](#))
- Miért nem kommunikálnak velünk a gépek természetes nyelven?
 - nagyon sok esetben egyszerűbb és jobb lenne
 - bonyolult felhasználói felületek,
 - mesterségesen létrehozott (programozási) nyelvek,
 - gyenge kifejezőerejű vagy nem létező kommunikációs interfések helyett

Látunk erre példákat, de valahogy nem mindennaposak és nem átütőek
(demo)

Miért?

Természetes nyelvű közlések gépi megértése

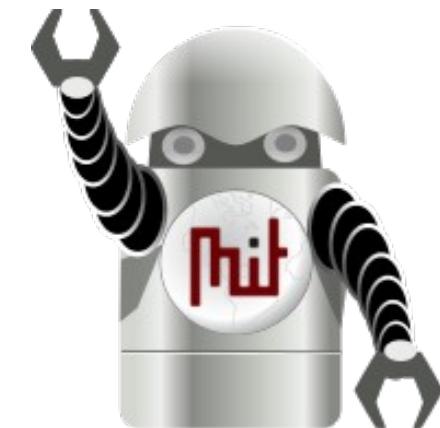
- Mi az a természetes nyelv? Mik a szabályai? Hogyan használjuk?
 - a nyelv egy **élő** doleg
 - meglehetősen összetett rendszer
 - időben és térben (beszélőről beszélőre is) változik (csurka?, kaffer?)
 - háttérteudást igényel (miről beszélünk?)
- Természetes nyelvű ember-gép interfések kialakítása
 - **számítógépes nyelvészeti** (computational linguistics)
 - több évtizedes mesterséges intelligencia (MI) kutatások
beszédfelismerés, szintaktikai elemzés, szemantikai értelmezés, párbeszédkezelés, mondatgenerálás, beszédszintézis, gépi fordítás, helyesírásellenőrzés, kérdésmegválaszoló rendszerek stb.
 - számos **könyv** és **publikáció**
 - eredményeit sokfelé látjuk az iparban és mindennapjainkban

Mitől nehéz egy ilyen rendszert készíteni?

- Beszédfelismerés (audio jelek feldolgozása, szöveggé alakítása)
 - sokféle hang, stílus, beszédhibák - néha még nekünk sem triviális
- A nyelv szintaktikai szabályainak ismerete
 - mit szabad és mit nem. A mondat részei és szerepeik. Írásjelek (?!).
- Szókincs beépítése
 - milyen szavakat használhatunk
- Értelmezés
 - szavak értelme, szavakból épített kifejezések értelmezése
 - a mondatok értelmezése, beleértve az írásjelek módosító hatását is.
- Párbeszédek kezelése
 - egy közlés értelme alapvetően függhet a korábbiaktól („Nem.”)
- Hivatkozások feloldása
 - „ő volt az, aki...”
- A válasz előállítása (mit és hogyan mondunk el)
 - karakterek összefűzése értelmes mondattá
- Az előállított válasz hangjelkké alakítása (kiejtés, hangsúly)

Mégis, mennyire nehéz egy ilyen rendszert készíteni?

- **BME Tibi:**
 - Android alkalmazás
 - mesterséges intelligenciáról és időjárásról lehet kérdezni
 - néhány opre fogalmat is ismer
- A program részei
 - beszédfelismerő: Google
 - szintaktikai elemző: regexp (elég primitív)
 - értelmező: egyszerű szabályalapú mintaillesztő
 - válasz előállító
 - saját kútföből néhány válasz
 - külső szolgáltatók lekérdezése ([MI Almanach](#), Köpönyeg, Időkép)
 - beszédszintetizátor: Android TTS + Mariska hang
- A fejlesztés menete és időigénye
 - 1. nap: Android beszédfelismerő alkalmazás
 - 2. nap: regexp elemző, szabályalapú értelmező és beszédszintetizátor
 - 3. nap: webes háttérkiszolgálók beépítése

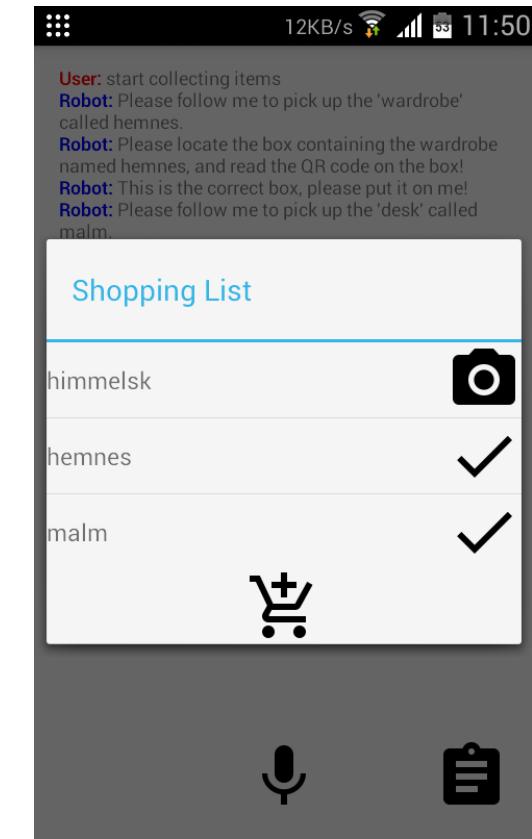


Robotirányítás természetes nyelven

Robot Operációs Rendszer (ROS)

BME MIT R5-COP projekt [WP24 demo](#)

Természetes nyelvű interfész szállítórobothoz



Agy-számítógép interfész (BCI, BMI)

- Emotiv EPOC Neuroheadset
 - 14 csatornás EEG mérőeszköz + gyorsulásmérő
- Mérési dimenziói (milyen jeleket ad)
 - emocionális állapot (hangulati jellemzők)
 - kognitív folyamatok („fordulj balra”, „előre”, „lassíts”, stb.)
 - arcmimika
- Mire használható?
 - alkalmazások irányítására (egyszerű mozgásbeviteli eszköz, a jeleket pl. billentyűleütésekkel kölcsönösen kezelni)
 - a mért agyhullámok megfigyelésére és feldolgozására (EEG jelek rögzítése és kezelése)
- Tapasztalataink...



Forrás: Emotiv



Összefoglalás

- Az OS többféle felhasználói felülettel rendelkezhet (akár egyszerre)
 - karakteres felület: elérhető, kis erőforrásigény, produktív, de nem túl barátságos
 - grafikus **WIMP**: ablakok, ikonok, menük és egér (pointer)
 - egyre többen kísérleteznek hangalapú, természetes nyelvű és más interfészekkel
- Karakteres felhasználói felület és parancsértelmező (shell)
 - jellemzően minden operációs rendszeren elérhető
 - belső és külső parancsok végrehajtására alkalmas
 - programozható (ciklus, elágazás, változók, automatizálás, szövegfeldolgozás stb.)
- Grafikus felhasználói felület (GUI)
 - összetett, réteges felépítésű rendszerek, itt-ott szabványos protokollokkal
 - főbb részei:
 - grafikus parancsértelmező (shell), pl. Windows Shell, Unity, Gnome Shell stb.
 - ablakkezelő, pl. Windows dwm.exe, KDE KWin, Gnome Metacity stb.
 - kijelzőszerver, protokollok pl. X11, Wayland
 - hálózati protokollok is elérhetők, pl.: Microsoft RDP, X11, SPICE, VNC RFB stb.

Az operációs rendszerek belső működése

Feladatkezelés

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

Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

Az előadásfóliák legfrissebb változata a tantárgy honlapján érhető el.
Az előadásanyagok BME-n kívüli felhasználása és más rendszerekben történő közzététele előzetes engedélyhez kötött.

Az eddigiekben történt...

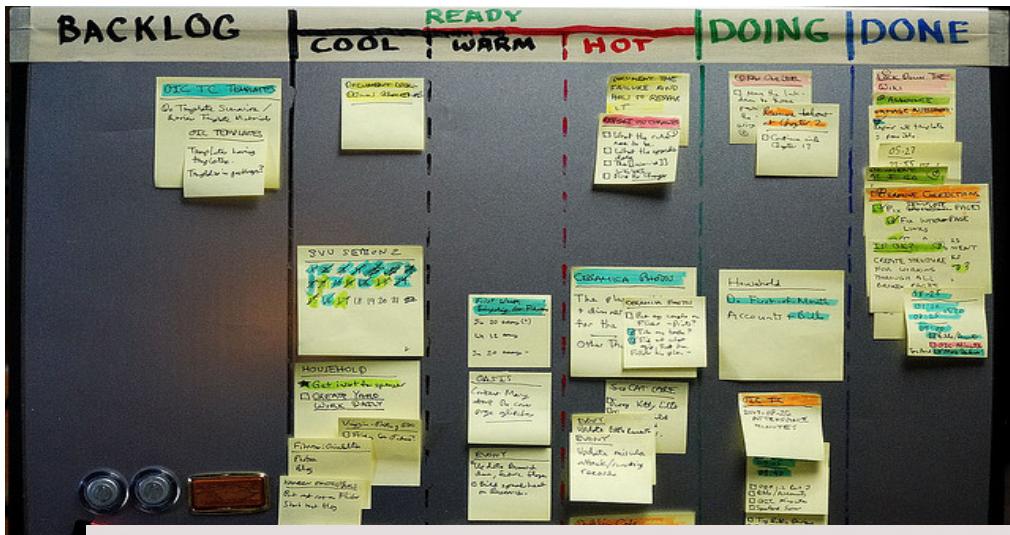
- Az operációs rendszer
 - feladatok végrehajtása
 - **vezérlőprogram**
 - **erőforrás-alkotátor**
- Elvárások
 - feladatok egyidejű kiszolgálása
 - megbízható működés
 - esetenként valósidejűség



Az OS felépítése

- Kialakulása
 - kötegelt rendszerek
 - **multiprogramozott**
 - **időosztásos**
 - beágyazott

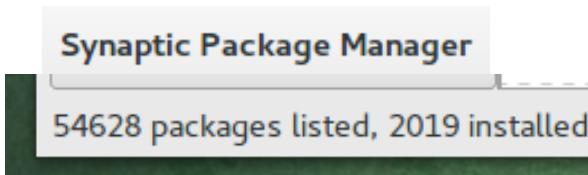
Hogyan kezeljük a feladatokat?



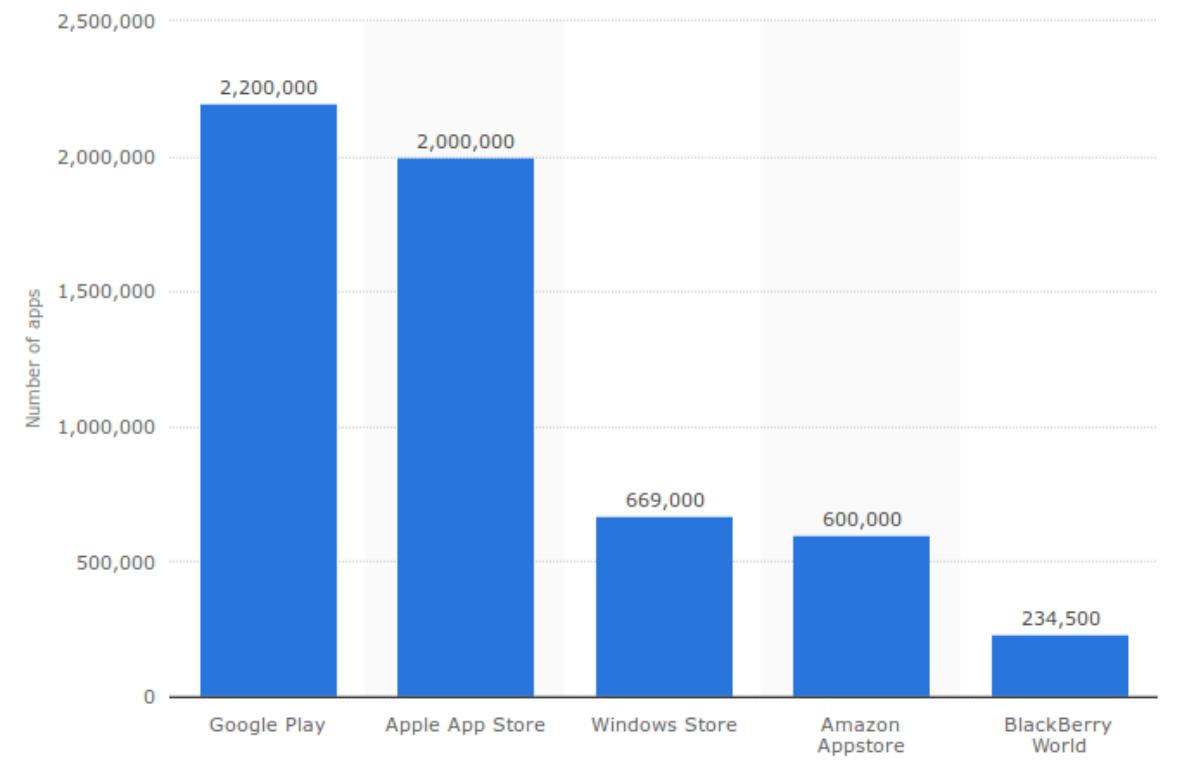
Milyen feladatokat futtatunk?

- A felhasználói feladatok sokszínűsége
- Az operációs rendszerek osztályozása (kliens, szerver, beágyazott...)
- Alkalmazások

Number of apps available in leading app stores as of June 2016



```
> yum list all | wc -l  
22747
```



forrás: statistica.com

A feladatok jellege

- I/O-intenzív feladatok
 - idejük nagy részét várakozással töltik (adatbetöltés, adatkiírás)
 - kevés processzoridőre van szükségük
 - pl.: fájlszerver, webszerver, email kliens és szerver stb.
- CPU-intenzív feladatok
 - idejük nagy részét a processzoron szeretnék tölteni
 - ehhez képest (relatíve) kevés I/O műveletre van szükségük
 - pl.: titkosítási és matematikai műveletek, összetett adatfeldolgozás stb.
- Memória-intenzív feladatok
 - egy időben nagy mennyiségű adat elérésére van szükségük
 - ha van elég, akkor CPU-intenzívek, ha nincs, akkor I/O-intenzív feladattá válnak
 - pl. nagy mátrixok szorzása, keresési indexek építése és használata stb.
- Speciális igények
 - valósidejű működés
 - filmnézés
 - ...

Elvárásaink

- Kevés várakozás

várakozási idő (waiting time)

körülfordulási idő (turnaround time)

válaszidő (response time)

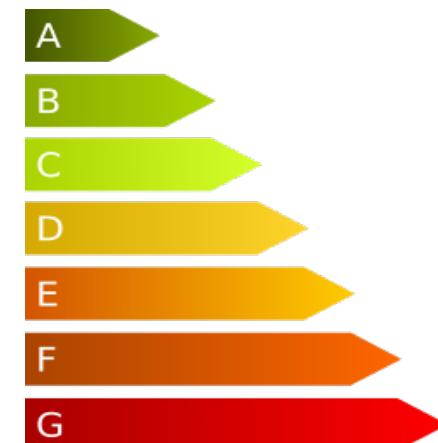
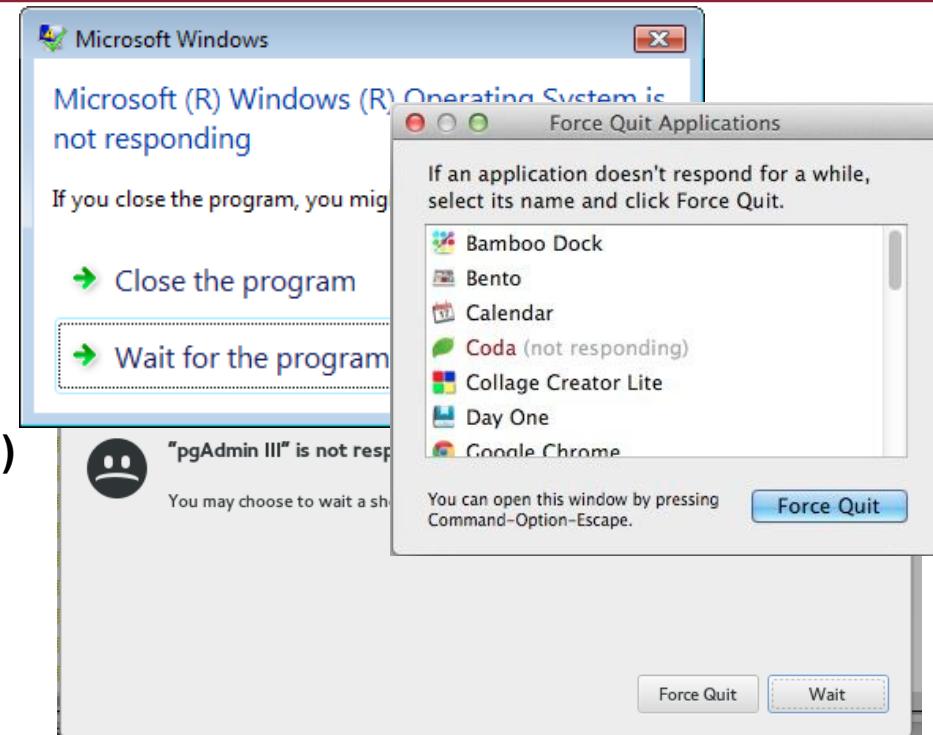
- Hatékonyság

CPU-kihasználtság (CPU utilization)

átbocsájtó képesség (throughput)

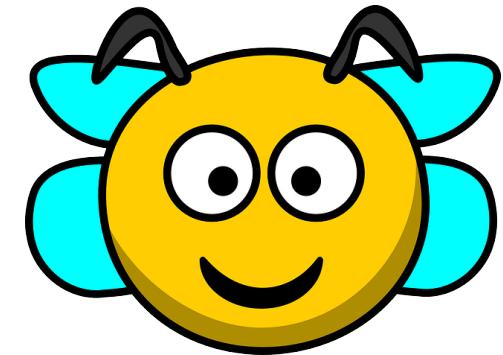
rezsiköltség (overhead)

- Jósolhatóság, determinisztikusság



Az optimális feladat-végrehajtó rendszer

- A naiv felhasználó elvárásai
 - biztosítja feladatai végrehajtását
 - minimalizálja a várakozási és válaszidőt
 - az erőforrásokat (CPU, I/O) maximális kihasználja
 - minél kisebb rezsiköltséggel dolgozik
- Mit tapasztal a rendszer használata során?
 - egyes programok „lassan” futnak
 - mások ok nélkül „lefagynak”
 - „feleslegesen” erőforrásokat foglalnak
 - akadozik a filmnézés
 - gyorsan merül az akkumulátor
 - néha mintha az egész rendszer leállna
 - nem tudja fogadni a hívást
 - ...



Mi okozza a nehézségeket?

- Az OS nem lát a jövőbe
 - milyen feladatok jönnek
 - milyen jellegűek
- Sok a feladat
 - különböző elvárások
 - más az optimalitási kritérium
 - néha túl sok, „vergődik” a rendszer
- A feladatok hatással vannak egymásra
 - együttműködnek
 - versenyeznek
- Hibák
 - programozói
 - hardver

A taszk

- A feladatainkat programok hajtják végre
elindulnak, működnek és befejeződnek

*A **taszk** (**task**) egy végrehajtás alatt álló program*

- Dinamikus entitás
 - a háttértáron tárolt program egy statikus program- és adathalmaz
 - a taszk „élő”: van működési állapota és életciklusa

állapot: adminisztratív jellemzők összessége egy adott pillanatban létrejön, végrehajtás alatt áll a processzoron, várakozik valamire, befejeződik stb.

életciklus: a létrehozástól a befejeződésig terjedő állapotváltozások az életciklus kezelése az operációs rendszer feladata

A feladat – taszk összerendelés

- Egy feladat – egy taszk

```
ps -ef
```

- Egy feladat – több taszk

```
ps -ef | wc -l
```

- néha így gyorsabban megoldható a feladat
- feladat dekompozíciója
- nagyobb teljesítmény
- jobb erőforrás-kihasználtság

- A taszkok kommunikálhatnak és együttműködhetnek

- adatokat cserélhetnek egymással
- szétoztathatnak részfeladatokat
- egyesíthetnek részeredményeket
- közös vezérlési szerkezeteket, kooperációs sémákat alakíthatnak ki

Taszkok szeparációja: az absztrakt virtuális gép

- Ideális esetben minden taszk teljesen önállóan fut
 - mintha saját gépen (erőforrásokon) futnának
- A valóságban osztoznak az erőforrásokon

absztrakt virtuális gép:

a kernel által biztosított erőforrások számítógépként elképzelt együttese.
virtuális CPU + virtuális memória

ismétlés: multiprogramozott rendszer

- M db processzor, N db taszk ($N >> M$)
- N db absztrakt virtuális gép leképezése a fizikai erőforrásokra

- A feladat-taszk összerendelés tovább bonyolítja a helyzetet
 - az absztrakt virtuális gép erős szeparációt jelent
 - ez nehezíti együttműködő taszkok kialakítását
 - adatcsere
 - vezérlési információk átadása

Taszk megvalósítások

Folyamat

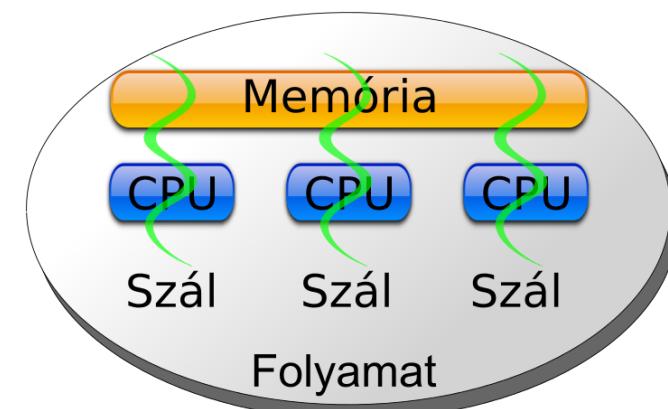
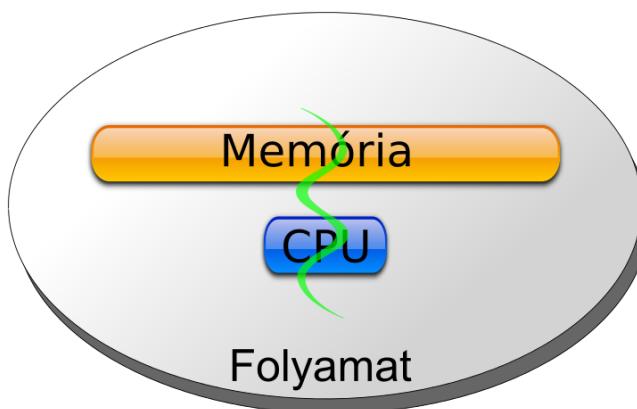
védelmi egység

önálló memóriatartománnyal rendelkező taszk, amely szálakat tartalmazhat

Szál

végrehajtási egység

szekvenciális működésű taszk, amely más szálakkal közös memóriát használhat



Folyamatok és szálak viszonya

- A szálak
 - önmagukban szekvenciálisan működnek → saját vermek van
 - **egy folyamaton belül**
 - egymással párhuzamosan működnek
 - közös memóriatartományt használnak
 - tudnak egymással adatot cserélni és kooperálni
 - nincs közöttük memóriaszeparáció
 - együttműködő taszkok megvalósítására alkalmas
- A folyamatok
 - önmagukban párhuzamosan is működhettek
 - saját memóriatartományuk van
 - nem látják más folyamatok memóriatartományát (OS védelem)
 - ezért egymással nehezebben tudnak együttműködni
- Demo: folyamatok és szálak listázása

Folyamatot vagy szálakat használjak?

- **Feladat – taszk vs. folyamat – szál**

Megkívánja a feladatom a multiprogramozást?

Hány párhuzamos végrehajtóegységre van szükségem?

Milyen gyakran?

Elérhető a szál/folyamat az adott rendszeren?

- **Szálak előnyei és hátrányai**

- kisebb az erőforrásigény
- egyszerűbb létrehozás
- egyszerű kommunikáció, ha egy folyamaton belül vannak
- nem mindenütt érhető el
- gondosabb programozást igényel (lásd később)

- **Folyamatok előnyei és hátrányai**

- kernelszintű védelem
- elterjedtebb
- nagyobb erőforrásigény
- nehézkesebb kommunikáció

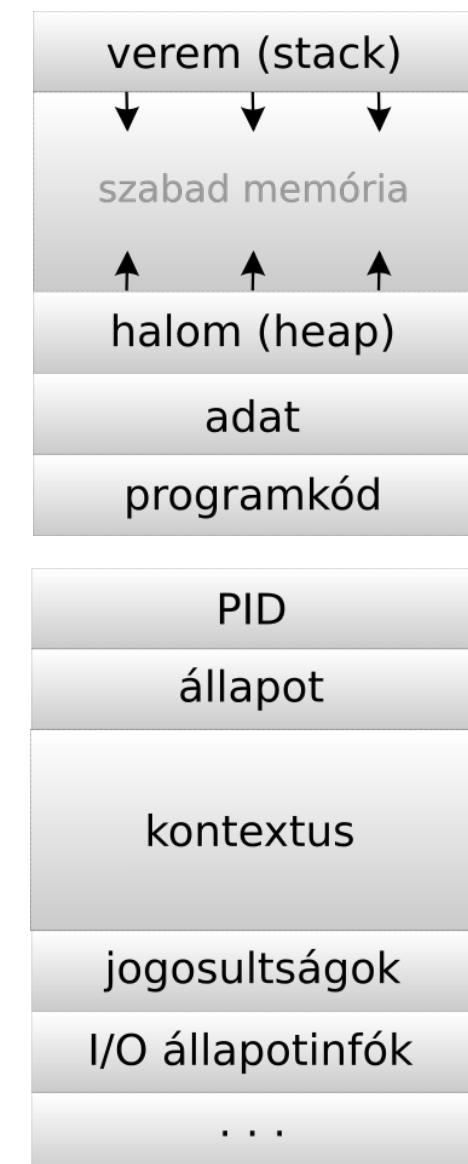
Folyamatok és szálak teljesítménye (demo)

- Apache Multi-Processing Modules (MPM)
 - folyamatalapú: „Prefork”
 - PHP modul
 - folyamat + szálak: „Worker”
 - gondban lehet, ha a szálakban hiba következik be
 - szálak: „Event”
- Teszteljük a webszerver teljesítményét és a rendszer terhelését
 - Apache Benchmarking: ab

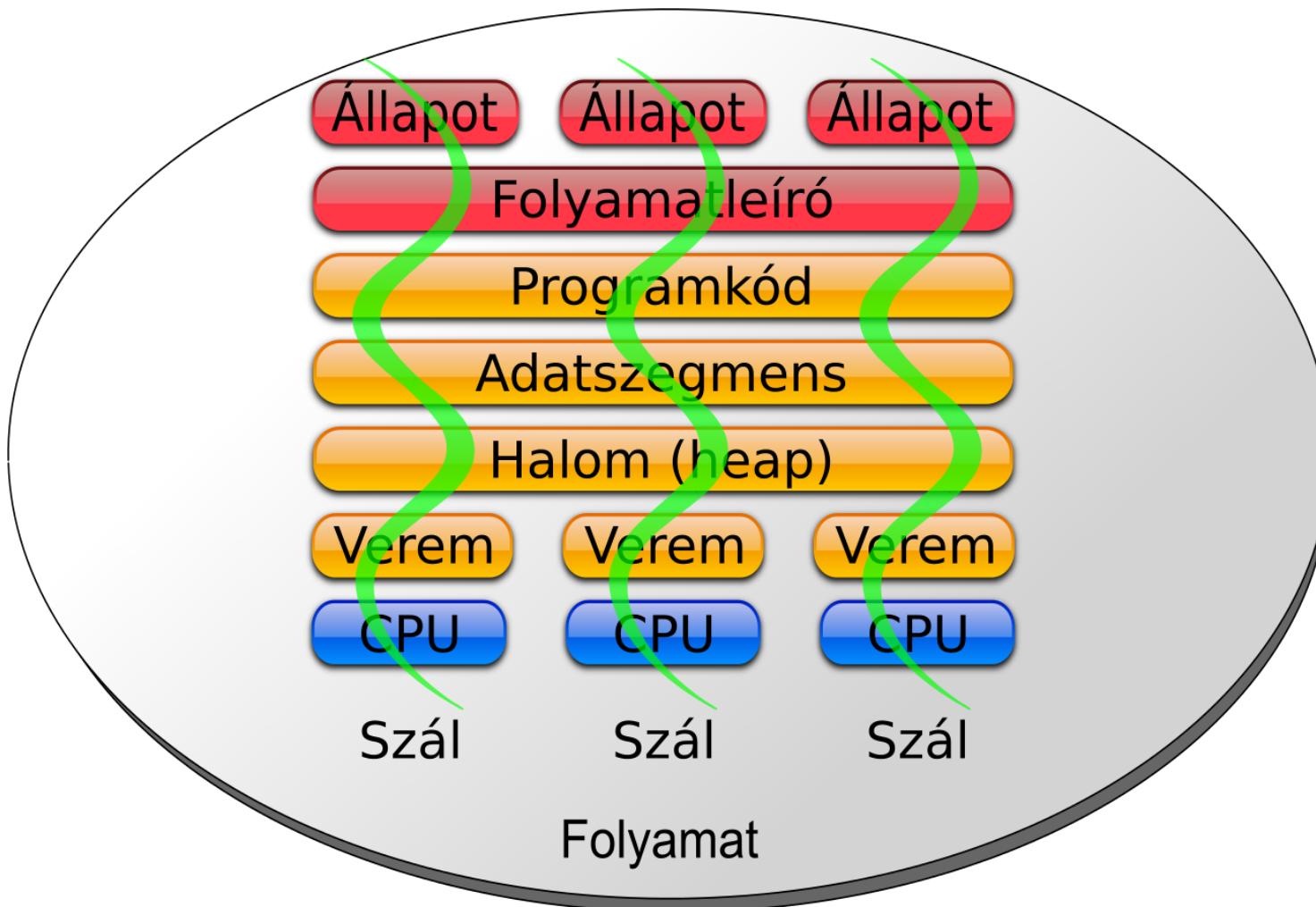
```
ab -n 5000 -c 500 <URL>
```
 - Szerver monitorozás: atop, apachetop

A taszkok adatai

- Saját
 - programkód
 - statikusan allokált adatok
 - verem, átmeneti adattár pl. függvényhívások számára
 - halom, a futásidőben, dinamikusan allokált adattár (demó)
- Adminisztratív (kernel)
 - taszk- (folyamat-, szál-) leíró**
 - egyedi azonosító (**PID**, TID)
 - **állapot** (l. később)
 - a taszk **kontextusa**: a végrehajtási állapot leírása
 - utasításszámláló (PC) és más CPU regiszterei
 - ütemezési információk
 - memóriakezelési adatok (MMU állapot)
 - tulajdonos és jogosultságok
 - I/O állapotinformációk
 - ...



Folyamat és szálak – részletesebben



Hol tároljuk a taszkok adminisztratív adatait?

- A kernel memóriatartományában?
- A folyamat címterében?
- Mikor van szükség az adatokra?
 - „gyakran”, a kernel működése során → kerüljenek a kernel címterébe
 - a folyamat működése során, ritkábban → kerüljenek a folyamat címterébe
- Az adminisztratív adatok csoportosítása
 - elsősorban a folyamat futása során szükségesek
 - hozzáférés-szabályozás adatai
 - rendszerhívások állapotai és adatai
 - IO műveletek adatai
 - számlázási és statisztikai adatak, stb.
 - elsősorban a folyamatok kezeléséhez szükségesek
 - azonosítók
 - futási állapot és ütemezési adatak
 - memóriakezelési adatak

UNIX példa
u-terület
folyamat címtér

proc struktúra
kernel címtér

A taszkok állapotai és életciklusa

- **Létrejön** (created)
 - betöltődik és elindul a taszk programja
 - a kernel létrehozza a szükséges adatstruktúrákat és bejegyzéseket
 - a taszk futásra kész állapotba lép
- Működése során
 - **Futásra kész** (ready to run)
 - **Fut** (running)
 - **Várakozik** (waiting) avagy **blokkolt** (blocked)
- **Megszűnik** (exit, terminated)
 - önszántából vagy végzetes hiba hatására



A taszkok állapotátmenetei

- Állapotváltozás rendszerhívások és megszakítások hatására
 - a rendszerhívás is megszakítást eredményez
 - az állapotváltozások megszakítások hatására következnek be
 - **a kernelek megszakítás-, azaz eseményvezéreltek**



Hogyan jön létre a taszk?

- Az init illetve a Wininit services.exe elindítja a szolgáltatásokat
- A felhasználó bejelentkezik (Logon) és elindít programokat
- Példa: Windows szálkezelés ([Szoftvertechnikák](#))
- Példa: Unix folyamatok létrehozása: fork() és exec()

```
if ((res = fork()) == 0) {           // gyerek ága  
  
    exec(...);                      // programkód betöltése  
  
} else if ( res < 0 ) {               // szülő ága, hibaellenőrzés  
    ...  
}  
  
// res = CHILD_PID (>0), szülő kódja fut tovább
```

Szálak létrehozása (példa)

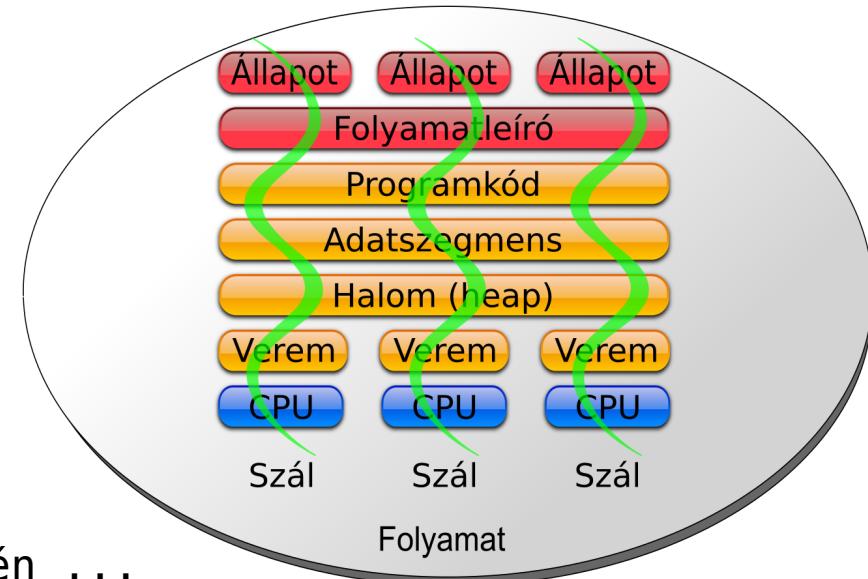
Párhuzamos adatfeldolgozás szálakkal

```
struct data_type data[N];
pthread_t *tid;

for (i=0; i < N; ++i) {
    // szálak indítása
    pthread_create(&tid[i], NULL, compute, &data[i]);
}

for (i=0; i < N; ++i) {
    // megvárjuk, míg elkészülnek
    pthread_join(tid[i], NULL);
}

compute (void *part) {
    ... műveletek a data[] *part részén ...
}
```

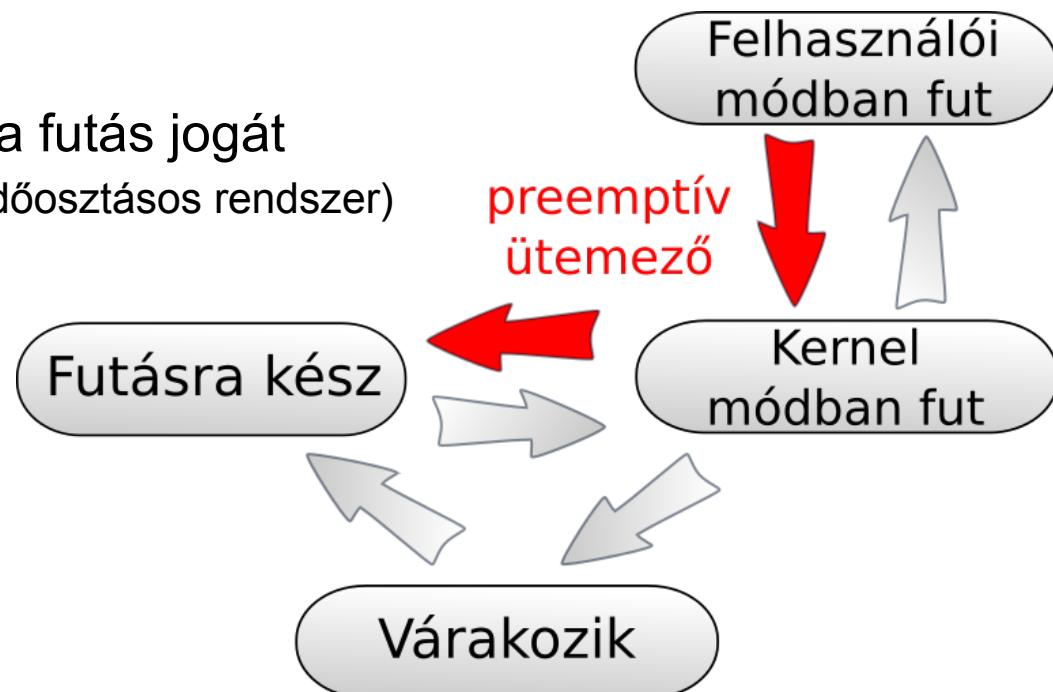


Unix folyamatok családfája

- Folyamatot csak egy másik folyamat tud létrehozni (`fork()`)
 - szülők – gyerekek – leszármazottak
 - családfa
- A szülő változhat
 - leálló folyamatok gyerekeit az init örökli
- A család fontos
 - a szülő nyilvántarthatja a gyerekfolyamatait
 - értesítést kap a gyerek folyamat leállásáról (nyugtázna kell)
- pstree demo

Mikor vált taszkot a processzor?

- A futó taszk lemond a futás jogáról
 - exit()
 - rendszerhívás
- A futó taszk elveszíti a futás jogát
 - lejár az időszelete (l. időosztásos rendszer)
 - hibás működés miatt



- Megszakítás, kivétel hatására
 - a CPU megszakítja a normál működését
 - elindul a kernel megszakítás- ill. kivételkezelője

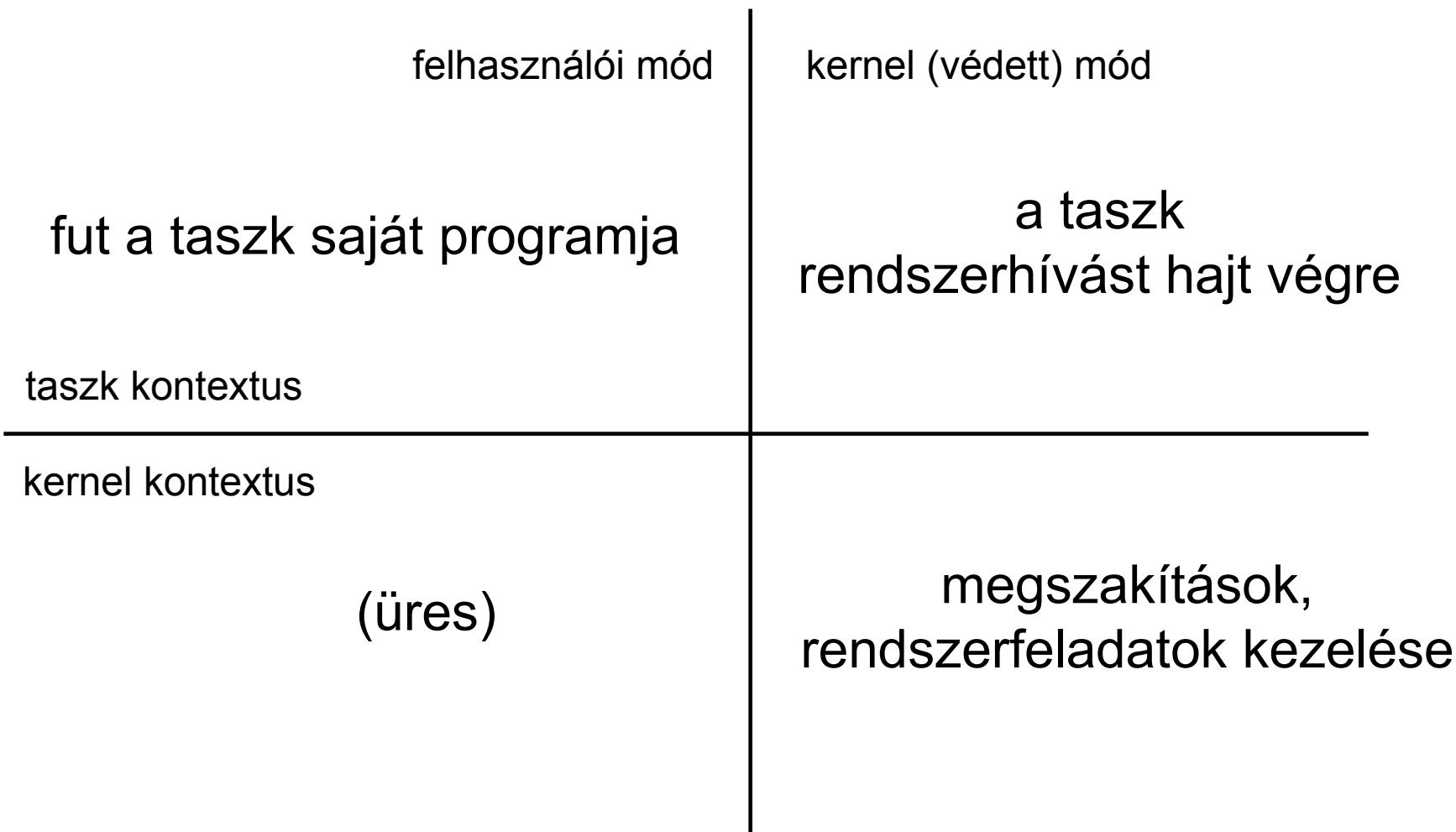
A kontextusváltás

- Kontextus: állapotleíró
 - utasításszámláló (PC), CPU és MMU állapot stb.
- Sok kontextusváltás történik az OS működése során
 - a rezsiköltség minimalizálandó
 - nem mindenkor menti a teljes kontextust
- Taszkváltás → kontextusváltás (taszk → taszk)
 - jelenlegi taszk kontextusának mentése
 - korábbi taszk kontextusának helyreállítása
- Megszakítás → kontextusváltás (taszk → kernel, üzemmódváltás!)
 - a kontextus egy kis része hardver támogatással elmentődik
 - megszakításkezelés
 - a visszatérés során visszaállítódik a korábbi végrehajtási kontextus

Kontextusváltások megfigyelése (demo)

- A kernel adattábláinak fájlrendszer interfészén keresztül lehetséges
 - a /proc/stat fájl ctxt mezője
 - a /proc/<PID>/status fájlban ctxt voluntary_ctxt_switches és nonvoluntary_ctxt_switches
- A korábbi Apache terhelésvizsgálatot megismételve
 - figyeljük meg az összes kontextusváltások számát
 - egy httpd folyamat kontextusváltásainak számát
 - Miért alacsony a nonvoluntary_ctxt_switches értéke?
 - Milyen jellegű folyamat az Apache httpd?
- CPU-intenzív folyamat kontextusváltásainak megfigyelése
 - pl.: stress -c 1
 - az általa indított gyerekfolyamat kontextusváltásait nézzük meg
 - Hogyan változik a nonvoluntary_ctxt_switches értéke?
 - Ezt megfigyelve milyen ütemezőt használ az operációs rendszerünk?
- A kísérlet [Windows alatt is elvégezhető](#).

Végrehajtási mód és kontextus



Összefoglalás

- Sokféle feladat
 - I/O-intenzív, CPU-intenzív, valósidejű, multimédia, ...
- Sokrétű elvárások
 - idő: várakozási idő, válaszidő, körülfordulási idő
 - hatékonyság: átbocsájtó képesség, CPU kihasználtság, rezsiköltség
- A feladatkezelés alapjai
 - taszk: végrehajtás alatt álló program kontextus, állapot és életciklus (létrejön – fut, FK, vár – megszűnik)
 - absztrakt virtuális gép
 - folyamat
 - szál
- Kontextusváltás
 - megszakítások hatására
 - rendkívül gyakori

Az operációs rendszerek belső működése

Ütemezés

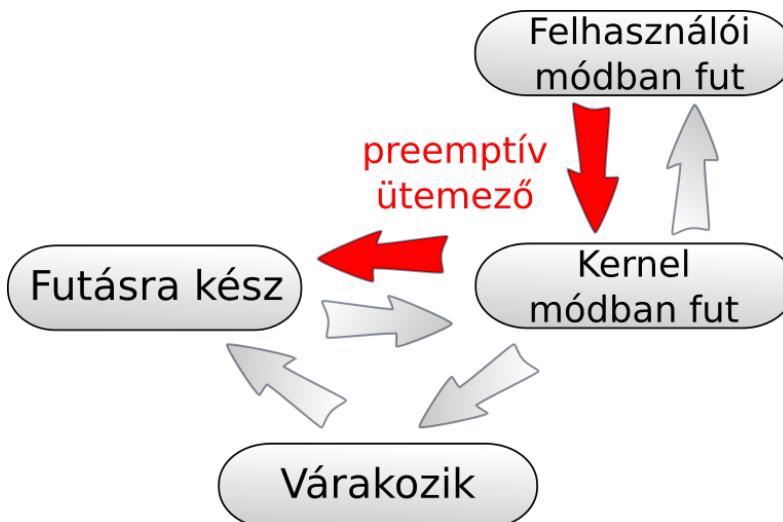
Mészáros Tamás
<http://www.mit.bme.hu/~meszaros/>

Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

Az előadásfóliák legfrissebb változata a tantárgy honlapján érhető el.
Az előadásanyagok BME-n kívüli felhasználása és más rendszerekben történő közzététele előzetes engedélyhez kötött.

Az eddigiekben történt...

- Az ütemező
 - a CPU-erőforrás allokációja
 - Futásra kész ↔ Fut állapotátmenet vezérlése
- Kihívások
 - feladatok ismeretlen jellege
 - taszkok egymásra hatása



- Elvárások
 - várakozási idő
 - körülfordulási idő
 - válaszidő
 - átbocsájtó képesség
 - kis rezsiköltség
 - jósolható
 - determinisztikus

A taszkok jellemzői

- CPU és IO műveleteket tartalmaznak

CPU-löket (CPU-burst):

a taszk processzoron végrehajtott utasításainak sorozata

IO-löket (IO-burst):

a taszk I/O műveletre vár

CPU-intenzív taszk: a CPU-löketidő dominál

IO-intenzív taszk: az IO-löketidő dominál

- Alapvető állapotaik
 - fut (**F**), futásra kész (**FK**) és várakozik (**V**)



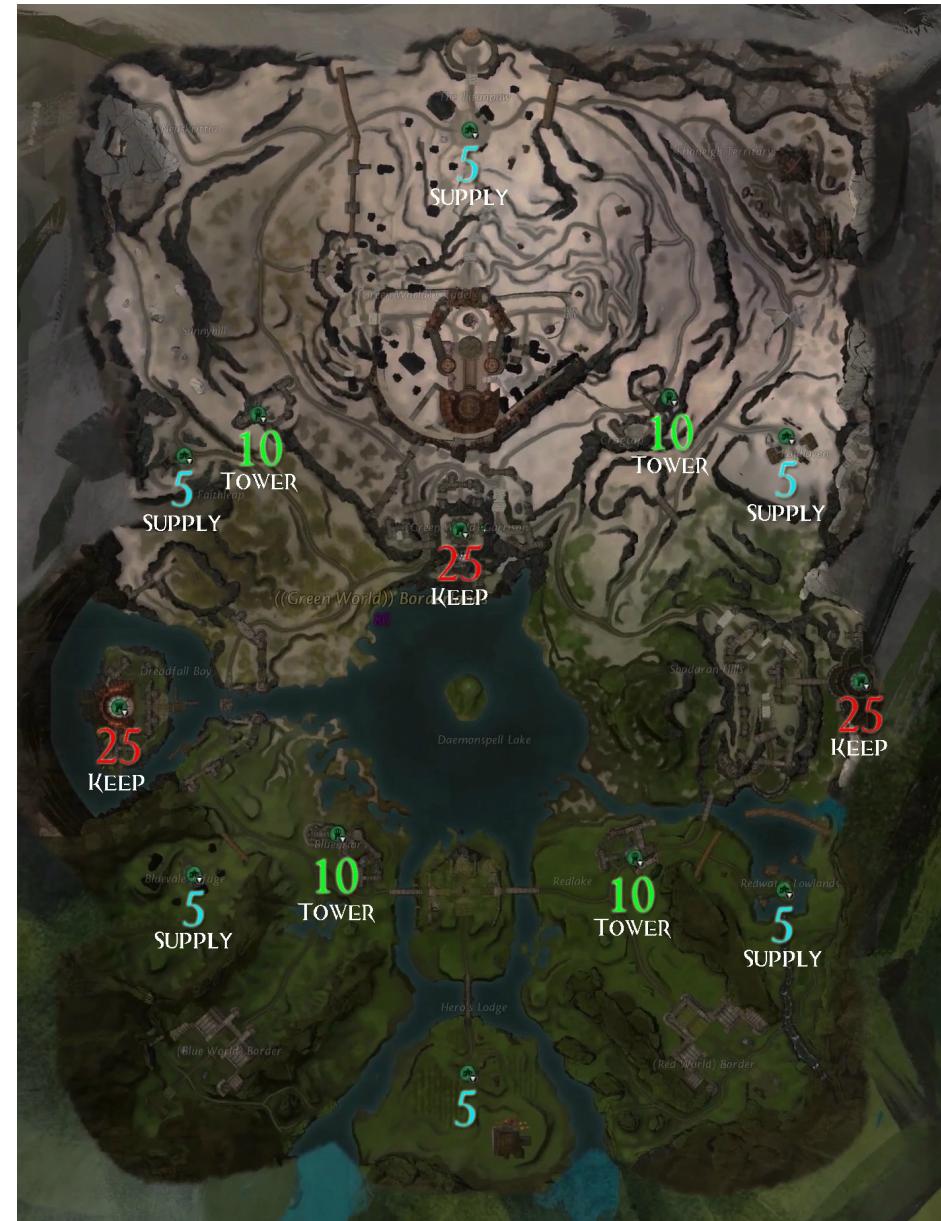
Az ütemezés feladata és időskálái

- Az ütemezés feladata eldönteni, hogy melyik taszk fog futni
- Honnan választjuk a futó taszkokat? – Az ütemezés időskálái
 - **rövid távú** (short term) avagy CPU-ütemezés
 - futásra kész állapotú taszkot választ
 - 1 – 100 ms
 - a kernel alapfeladata
 - **hosszú távú** (long term)
 - feladatot választ, taszkot indít
 - órák, napok, hetek, hónapok, ...
 - nem a kernel hatásköre (pl. Unix [cron](#), Windows [Task Scheduler](#))
 - **középtávú** (middle- vagy medium-term)
 - taszkot választ (bármilyen állapotút)
 - új állapotok: **felfüggesztve várakozik** és **felfüggesztve futásra kész**
 - percek, órák
 - a felhasználó és a kernel is kezdeményezheti
(demo)



Ütemezési példa

- feladatok
 - célok elfoglalása / védelme
 - 3 erőd, 4 vár, 6 fatelep
 - különböző pontszám („fontosság”)
- a végrehajtás fázisai
 - előkészítés (I/O-löket)
 - várunk a többiekre
 - harc (CPU-löket)
 - nem tudjuk előre a hosszát
- végrehajtó egység
 - zerg
- az ütemező
 - a zerg vezetője
- az ütemezés időskálái
 - rövid távú: most merre?
 - hosszú távú: melyik este?
 - középtávú: vacsoraszünet



Rövid távú (CPU-) ütemezés

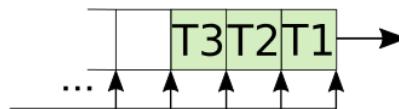
- Nagyon gyakran működésbe lép (1-100 ms)
- Kritikus hatással van az OS működésére így megítélésére
- Főbb típusai
 - egyprocesszoros
 - egyszerű
 - összetett
 - többprocesszoros
 - homogén
 - heterogén
- Fontos a rezsiköltség
 - algoritmikus komplexitás
- Időjellemzők számítása

Az ütemezés adatstruktúrái

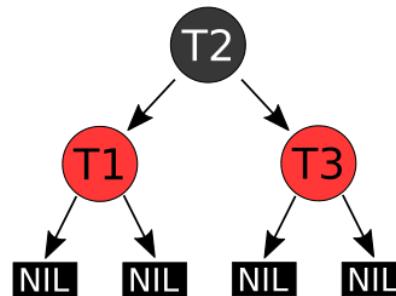
- Cél: az egyes állapotokban található taszkok nyilvántartása
- **FIFO:** a lista végéhez fűzünk, és az elejéről veszünk ki elemet



- rendezett **lista**: a beillesztés (vagy a kivétel) más ponton is lehetséges



- (bináris) **keresőfa**



pl. piros-fekete fa (lásd [algoritmuselmélet jegyzet](#))

felső korlát adható a műveleteire (valósidéjű ütemezés)

Az ütemezés műveleteinek komplexitása

- **Kritikus kérdés**

- gyakori műveletek rezsiköltsége

- az adatműveletek komplexitása legrosszabb esetben

(lásd [algoritmuselmélet](#))

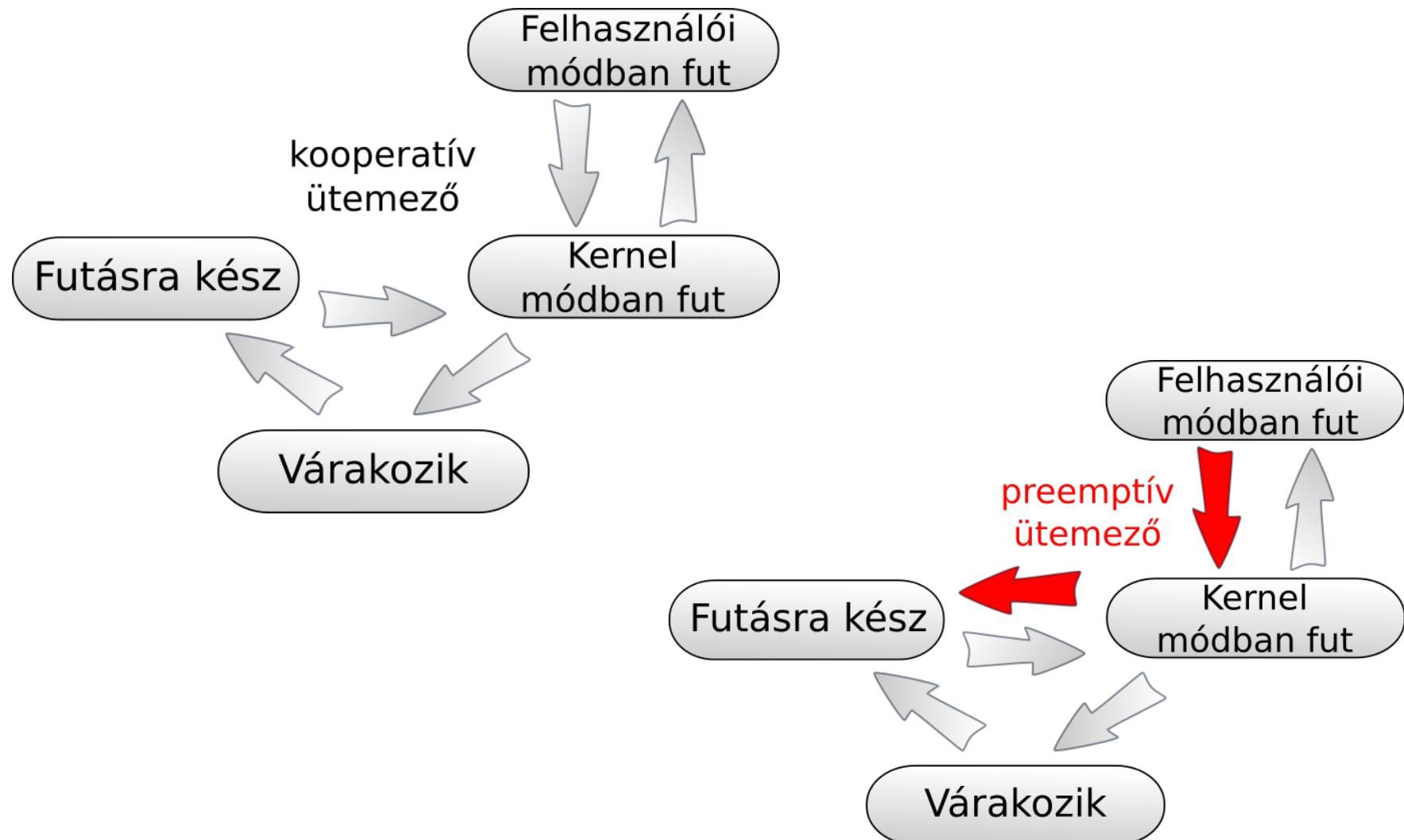
N: taszkok száma

- beszúrás FIFO végére, kivétel FIFO elejéről:
 $O(1)$ konstans
 - keresés/beszúrás rendezett láncolt listában:
 $O(N)$ lineáris
 - keresés/beszúrás piros-fekete fában:
 $O(\log N)$ logaritmikus
 - az ütemező algoritmusának egyéb részei
 $O(1)$ konstans

- elvi \leftrightarrow gyakorlati komplexitás

az elvileg jó lehet gyakorlatilag rossz, és fordítva

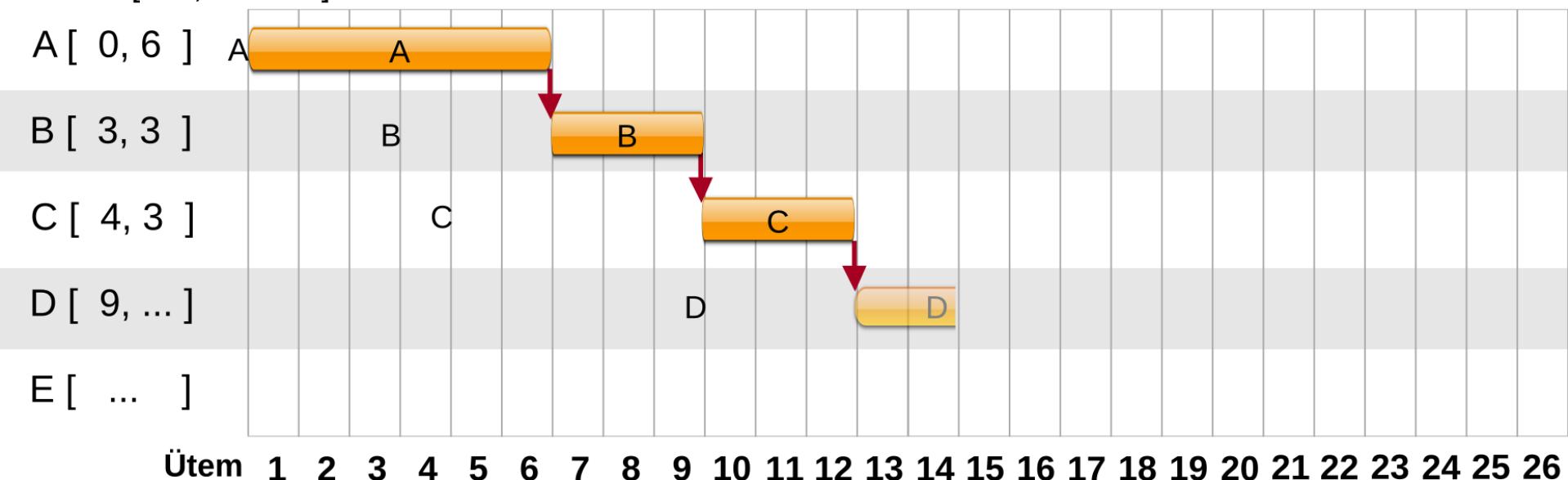
A CPU ütemezés alaptípusai



Az ütemezők működésének szemléltetése

Gantt-diagrammal

Taszkok [Start, CPU-löket]



Futó állapotú taszk:



Taszkok [Prio, Start, CPU-löket]

A [3, 0, 5]

B [4, 0, 4]

C [6, 5, 8]

D [6, 7, 8]

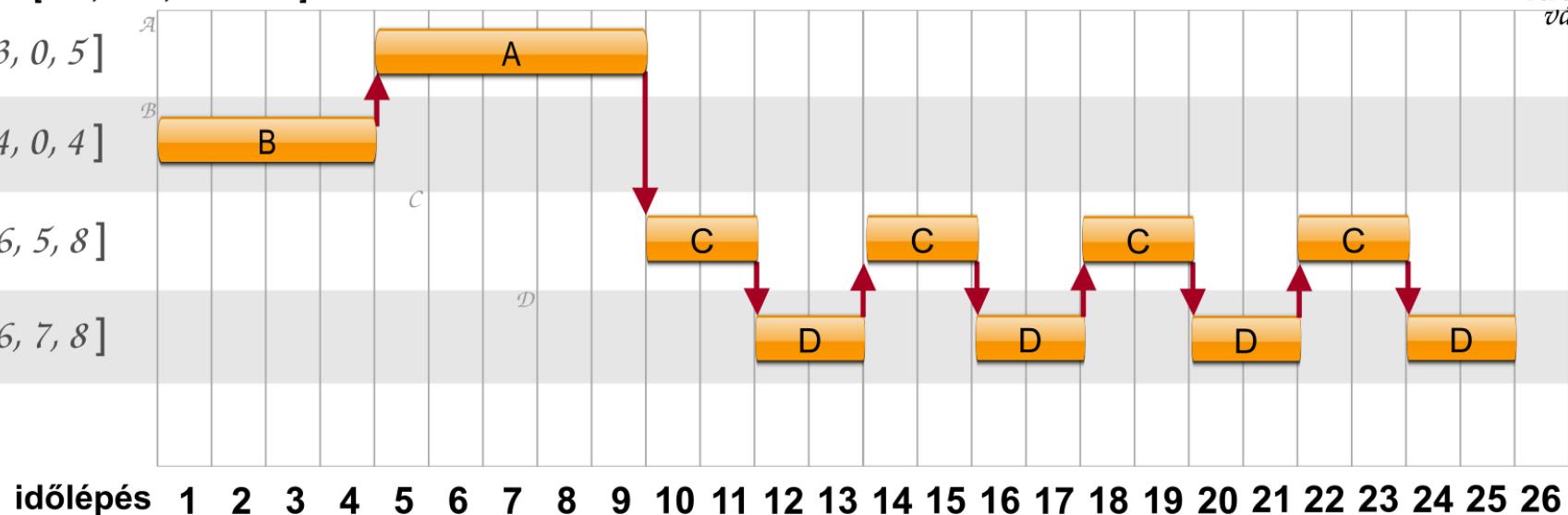
Lefutási sorrend: $\mathcal{B} \mathcal{A} \mathcal{C} \mathcal{D} \mathcal{C} \mathcal{D} \mathcal{C} \mathcal{D} \mathcal{C} \mathcal{D}$ Időérték
várakozás

4

0

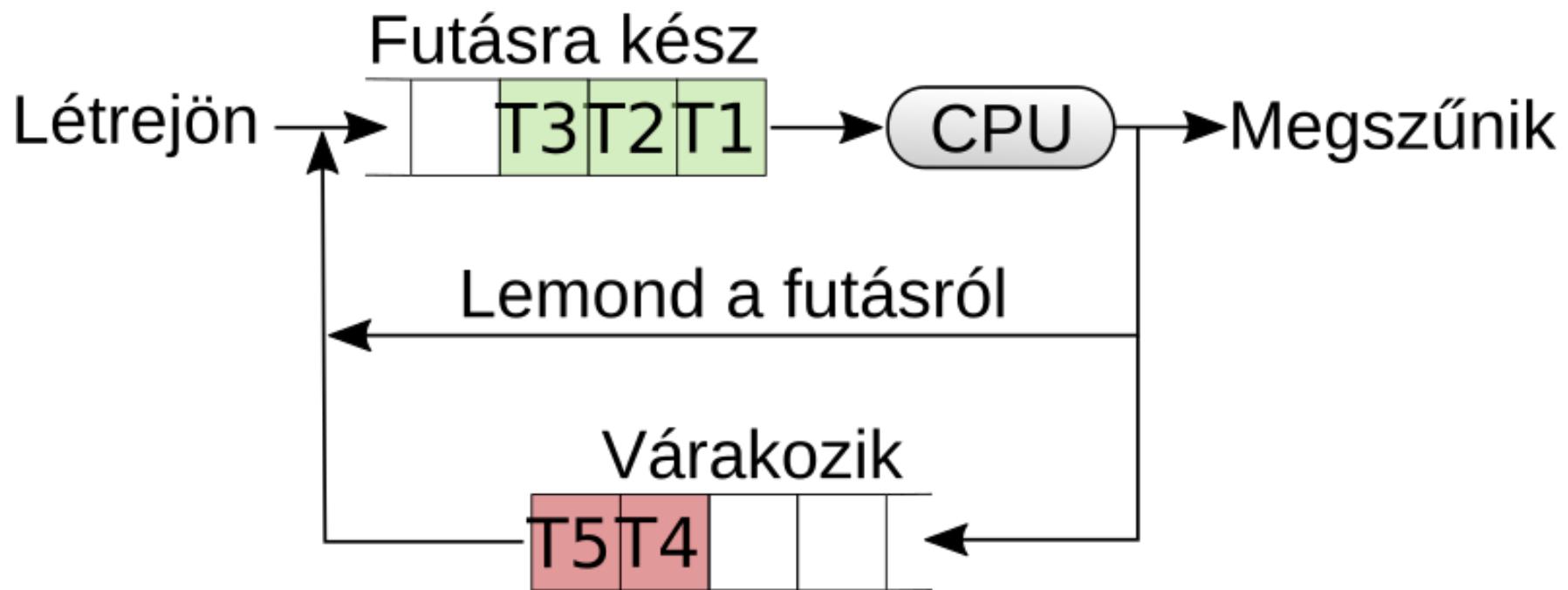
10

10

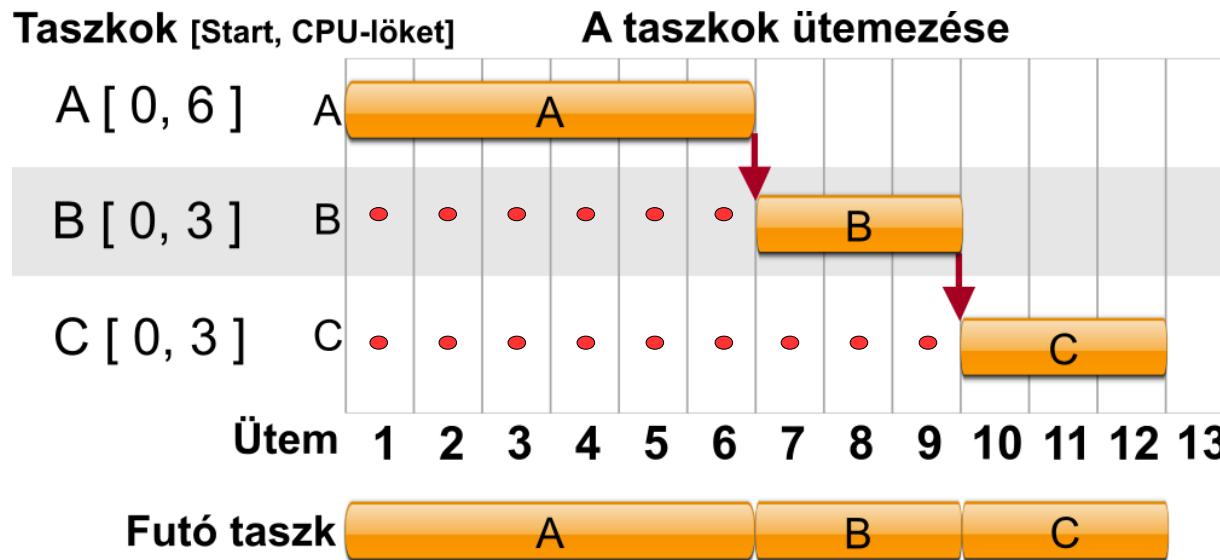


Egyszerű ütemezési algoritmusok

Legrégebben várakozó (first come, first served, FCFS)



Egyszerű FCFS ütemezési példa



Számoljuk ki a várakozási és a körülfordulási időket!
Érkezzen „A” utoljára! Hogyan változnak a várakozási idők?

Értékeljük az FCFS ütemezőt!

- Tulajdonságai
 - kooperatív
 - egyszerű adatstruktúra (FIFO) és algoritmus (FIFO)
- Az ütemező algoritmikus komplexitása (beszúrás, keresés)
 $O(1)$
- Rezsiköltség
minimális
- Minőségi jellemzők, problémák
 - Mi történik, ha egy hosszú CPU-löketű taszk fut?
Konvoj hatás: az **FK** taszkok feltorlódnak a hosszú CPU-löketű **F** taszk mögött
A feltorlódó I/O-intenzív taszkok nem kerülnek várakozó állapotba.
Az algoritmus jellemzően magas átlagos várakozási időt eredményez.
- Feladatok
 - T1...T3 sorrendben érkező taszkok, CPU-löketidők: T1: 24, T2: 3, T3: 3
 - Mennyi az átlagos várakozási idő? Ha fordított sorrendben érkeznek, mennyi lesz?
 - Csak IO-intenzív taszkok esetén hogyan értékelné az ütemezőt?

Hogyan kezelhetjük a konvoj-hatást?

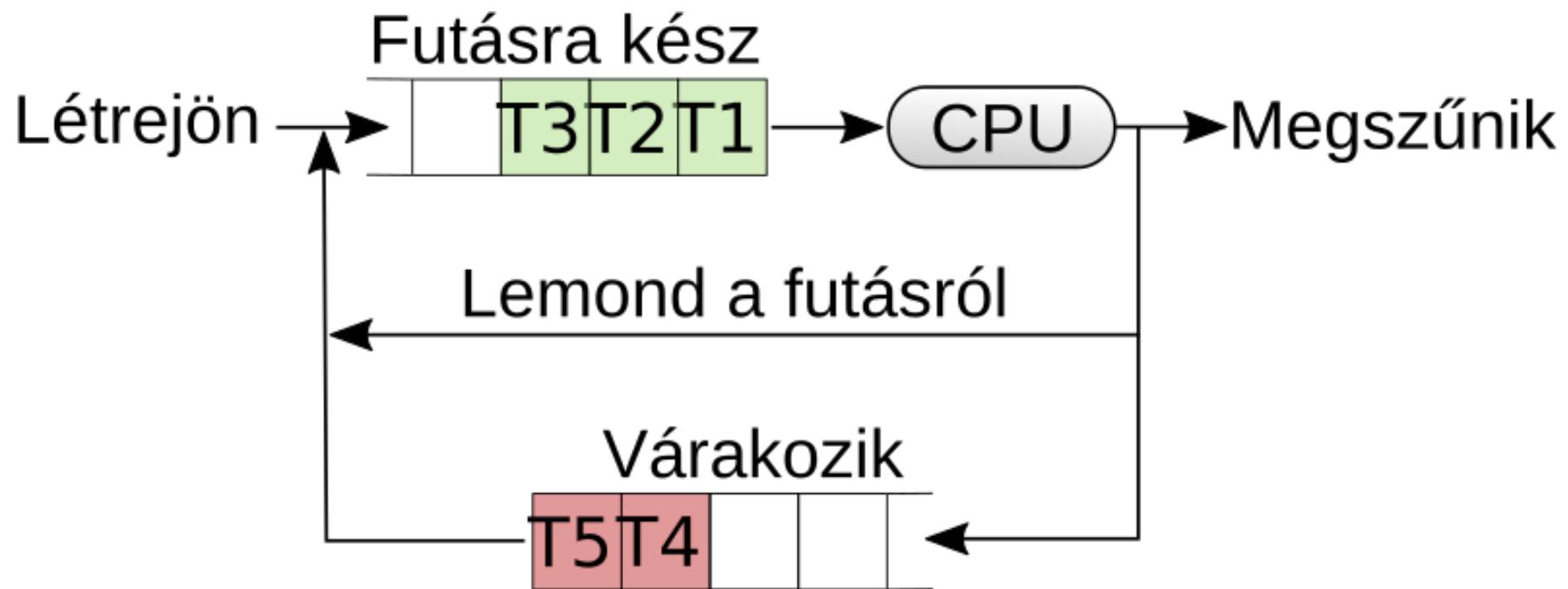
- Szakítsuk meg a sokáig futó taszkok működését
 - kooperatívból preemptívvé válik az ütemező
 - nem bonyolódik az ütemezési algoritmus

→ **Körforgó ütemezés**

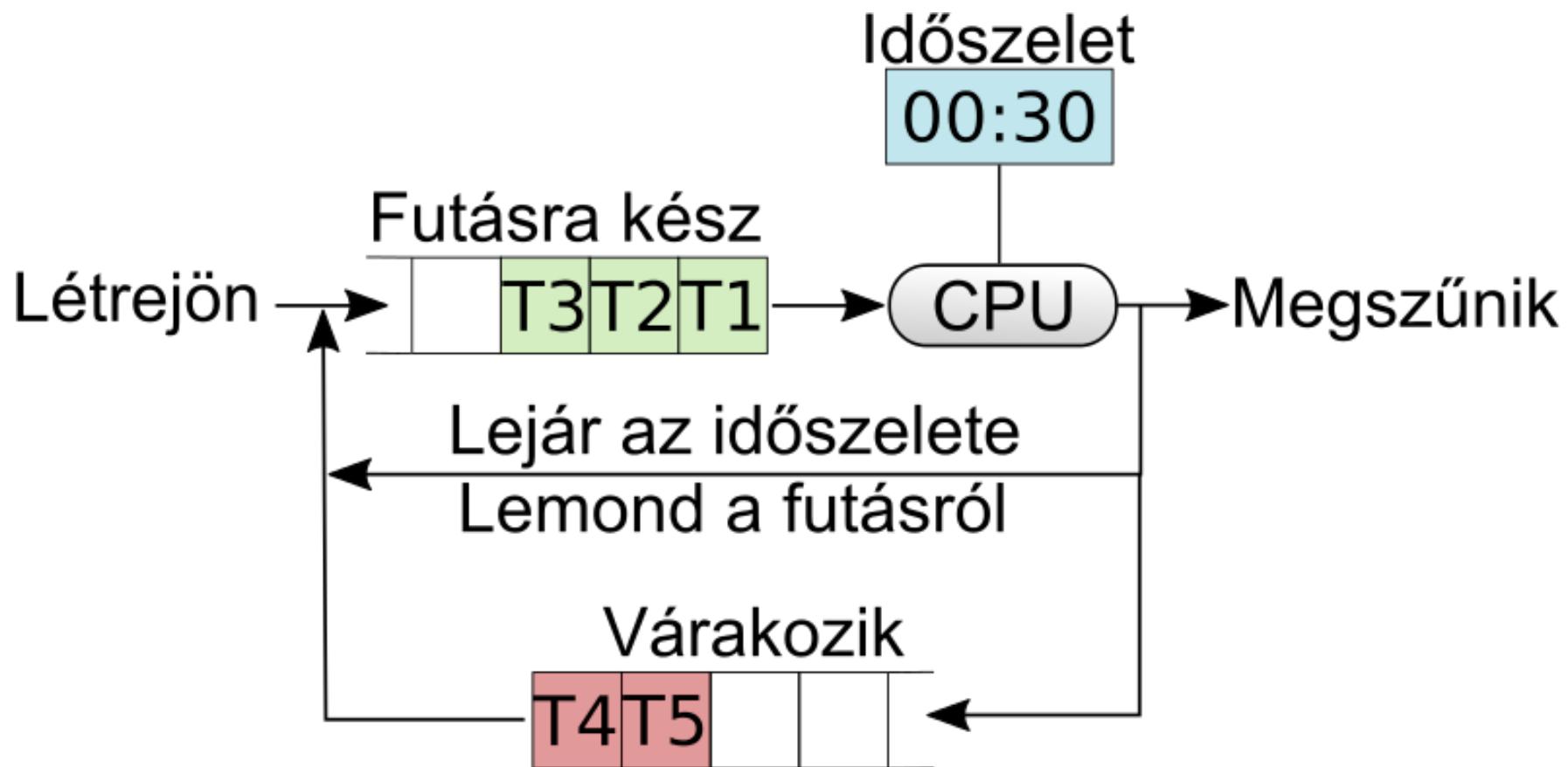
- Vegyük előre a legrövidebb CPU-löketidejű taszkot
 - gyorsan várakozó (**V**) állapotba rakhatjuk az IO-intenzív taszkokat
 - kooperatív marad az ütemező
 - kicsit bonyolódik az algoritmus

→ **Legrövidebb löketidejű ütemezés**

FCFS ütemezés (ismétlés)

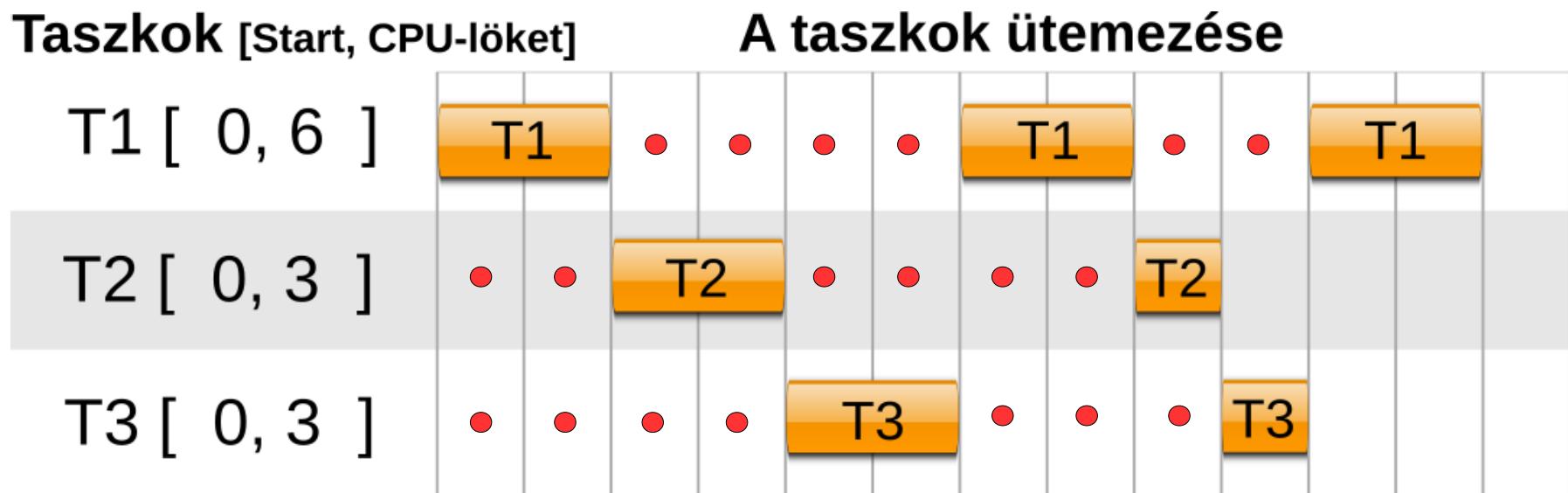


Körforgó ütemezés (Round-robin, RR)



RR példa

Taszkok [Start, CPU-löket]



A taszkok ütemezése

T1 [0, 6]



T1

T2 [0, 3]



T2

T3 [0, 3]



T3

Számoljuk ki a várakozási időket!

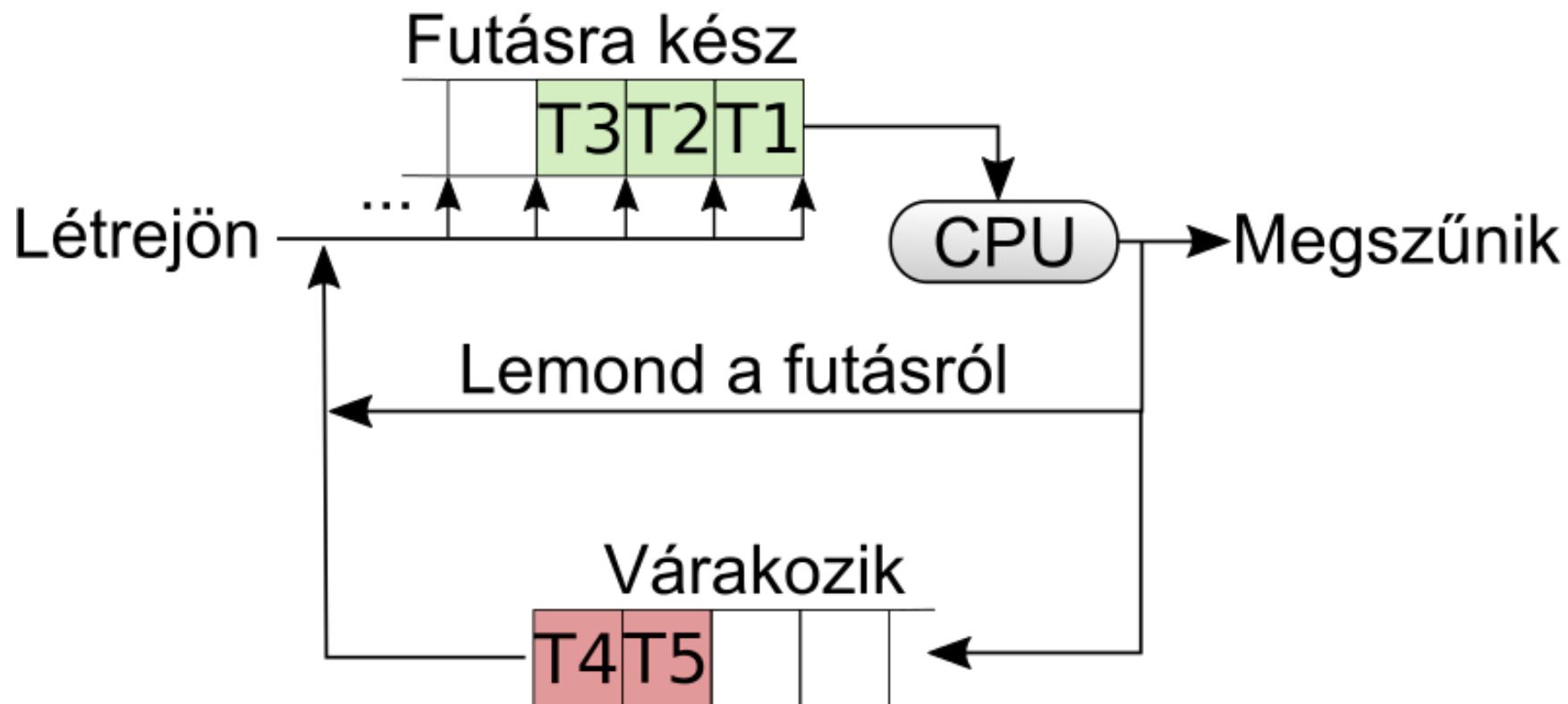
Az időszak 2 egység.

A RR ütemező értékelése

- Tulajdonságai
 - preemptív
 - nagyon egyszerű adatstruktúra (FIFO) és ütemezési algoritmus (FIFO)
- Az ütemező algoritmikus komplexitása (beszúrás, keresés)
 $O(1)$
- Rezsiköltség
 - minimális, ha jól választjuk meg az időszeletet
- Minőségi jellemzők, problémák
 - Milyen időjellemző javul leginkább?
 - Mekkora legyen az időszelet?
 - túl nagy → FCFS algoritmusba vált
 - túl kicsi → túl sok kontextusváltás lehet, nő a rezsiköltség
- Feladatok
 - T1...T3 sorrendben érkező taszkok, CPU-löketidők: P1: 24, P2: 3, P3: 3
 - Mennyi az átlagos várakozási idő, ha az időszelet 2, illetve 6? Függ a sorrendtől?
 - A körülfordulási idő számítása az időszelet függvényében.

A legrövidebb löketidejű (Shortest Job First, SJF)

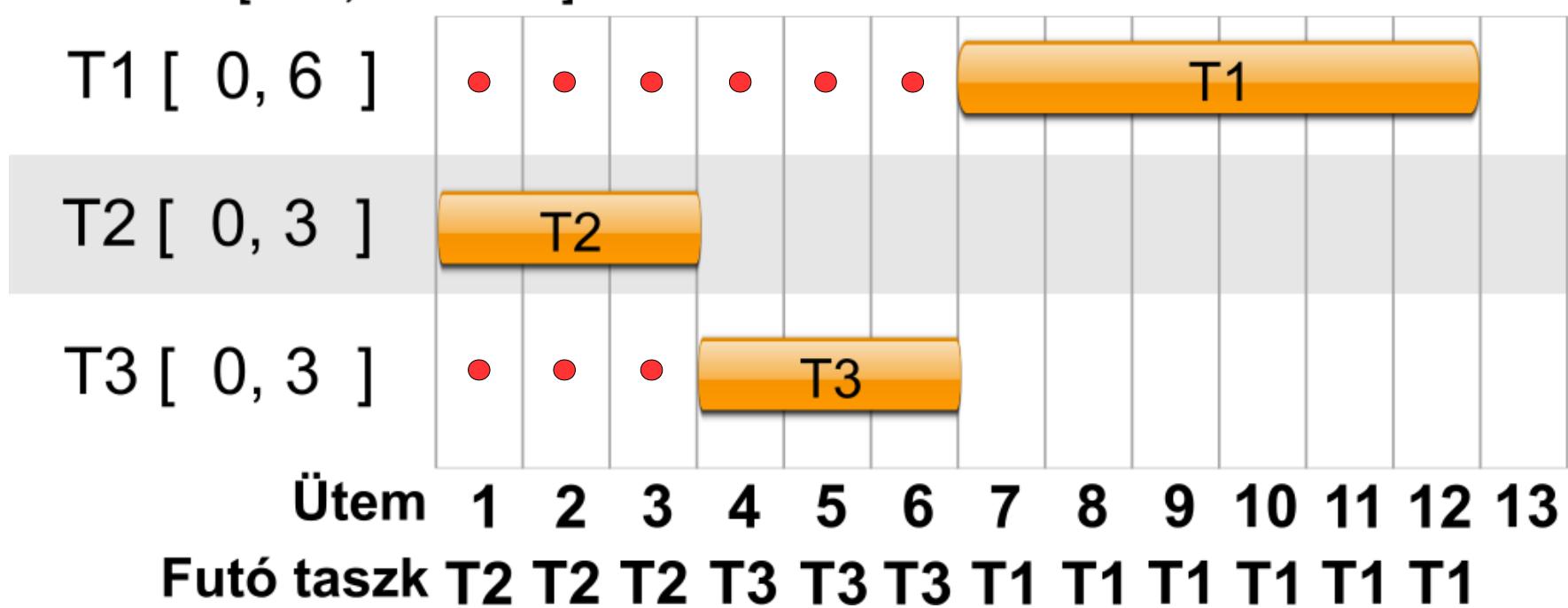
A futásra kész taszkok CPU-löketidejük szerint növekvő sorrendben vannak.



SJF példa

Taszkok [Start, CPU-löket]

A taszkok ütemezése



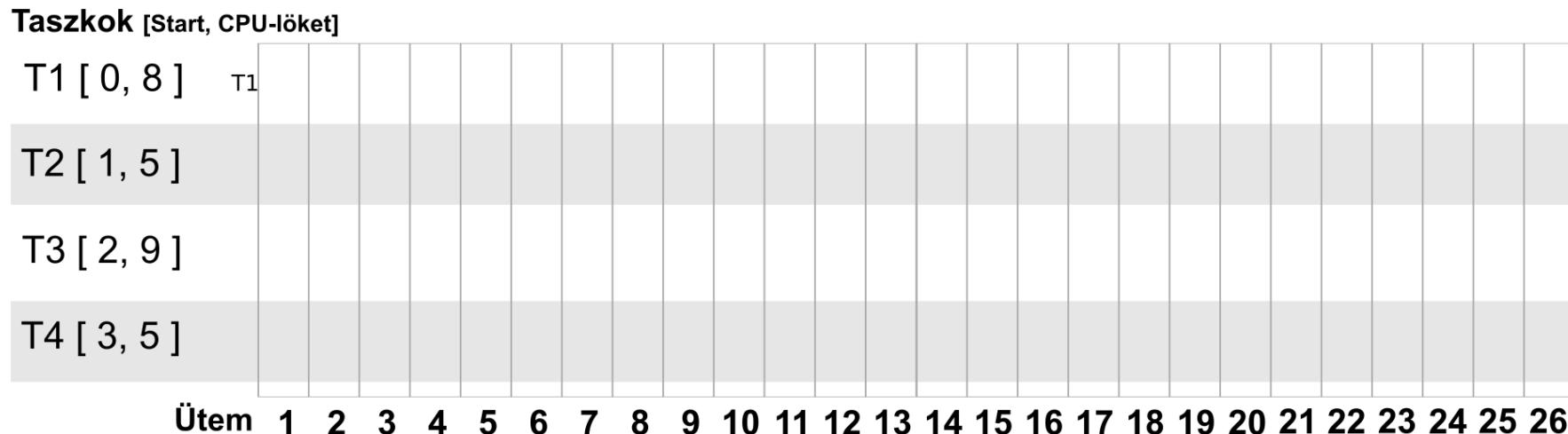
Számoljuk ki a várakozási időket!

Az SJF ütemező értékelése

- Tulajdonságai
 - kooperatív ütemező
 - bonyolultabb adatstruktúra (löketidő szerint rendezett sor)
- Az ütemező algoritmikus komplexitása (beszúrás)
 $O(N)$ – meg kell keresnünk a megfelelő helyet a sorban
- Rezsiköltség
 - nagyobb az FCFS és RR ütemezőknél (beszúrásnál keresni kell)
- Minőségi jellemzők, problémák
 - bizonyíthatóan optimális az átlagos várakozási és körülfordulási időt tekintve
 - bizonyítás indirekt módon
 - **honnán ismerjük a taszkok löketidejét?**
 - leginkább seholnan, megpróbálhatjuk megbecsülni
 - előre ismert taszkok esetén (pl. beágyazott rendszerekben) talán számítható
- Feladatok
 - T1...T3 sorrendben érkező taszkok, CPU-löketidők: T1: 24, T2: 3, T3: 3
 - Mennyi az átlagos várakozási idő? Mennyi az átlagos körülfordulási idő?
 - Számítsuk ki a várakozási időt kétféle löketidő-becsléssel!

Számítási példa

Futtassuk az eddig megismert ütemezési algoritmusokat,
és számítsuk ki az átlagos várakozási időt!

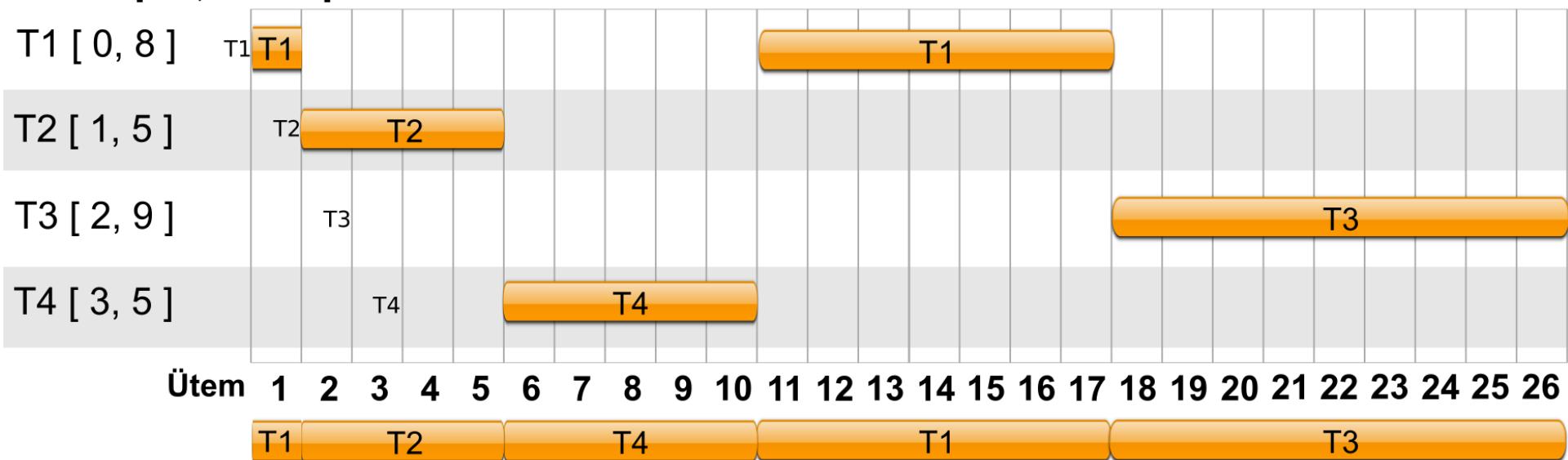


Legrövidebb hátralevő idejű (Shortest Remaining Time First, SRTF)

- A SJF preemptív változata

Ha új taszk lép FK állapotba, akkor összeveti a löketidejét a futó hátralevő idejével is, és ha az új taszk löketideje kisebb, akkor megszakítja a futó taszkot.

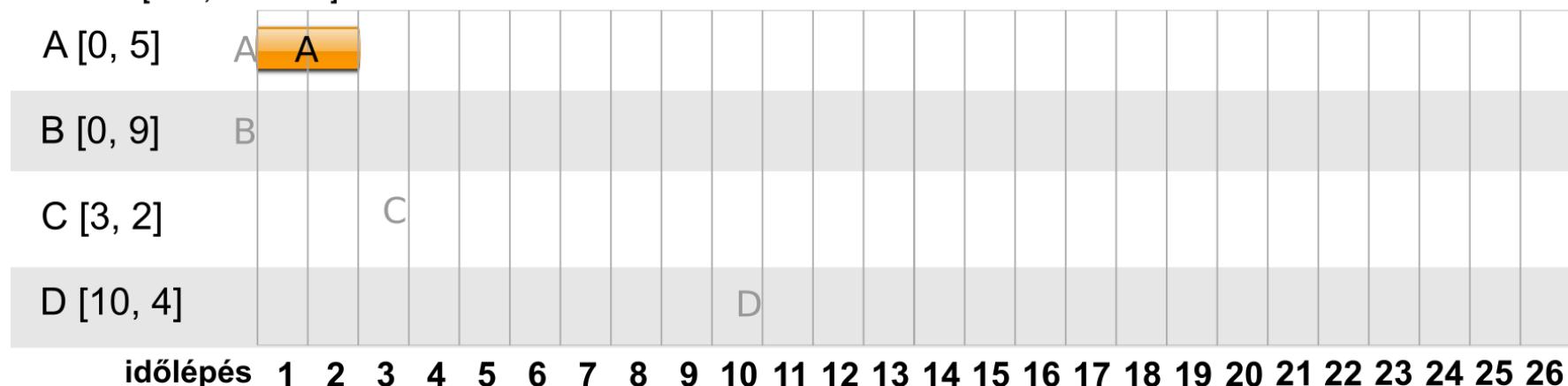
Taszkok [Start, CPU-löket]



Számítási példa

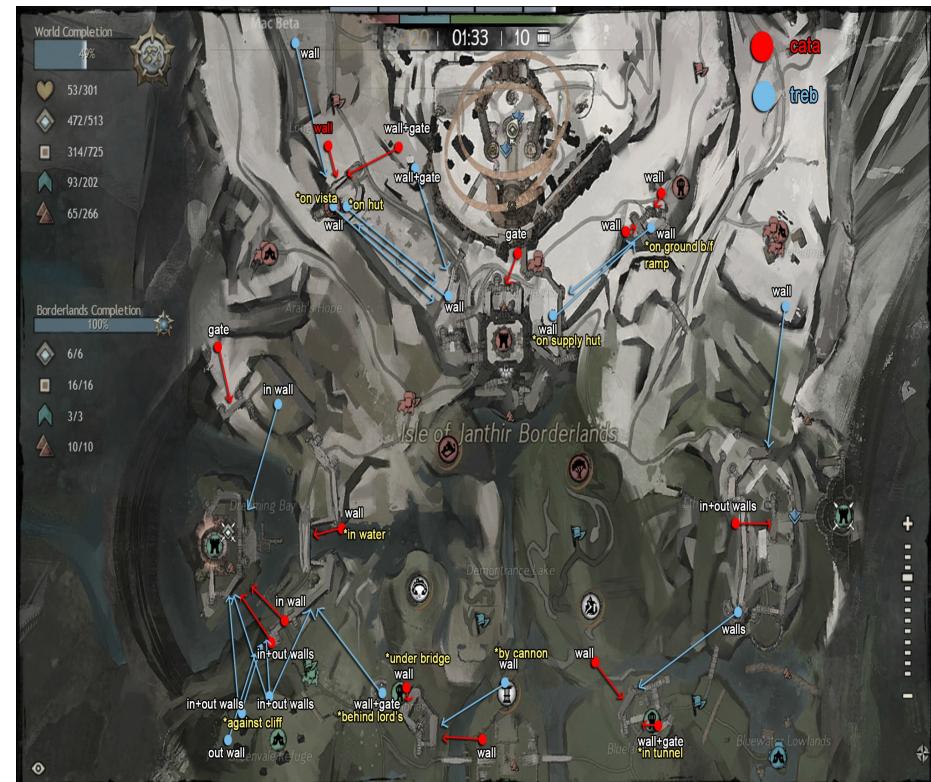
Mutassa be az alábbi taszkok ütemezését a tanult egyszerű ütemezési algoritmusokkal, és számítsa ki a taszkok várakozási és körülfordulási idejét!

Taszkok [Start, CPU-löket]



További ütemezési szempontok

- Cél: globális pontszám növelése
 - nem mindegy, melyik várat / telepet tartjuk a kezünkben
 - lehetnek más szempontok is, amit a zergvezérnek figyelembe kell vennie
klánok saját tornyai, várai, küldetések teljesítése, veszteségek mértéke stb.
 - azaz a taszkok között fontossági sorrendet is felállíthatunk
- Az eddigi ütemezők ilyen szempontokat nem vesznek figyelembe
- Hogyan vehet figyelembe az ütemező más szempontokat?
- Hogyan lássuk el az ütemezőt több információval?



Prioritásos ütemezők

- Felhasználóként beleszólhatunk-e az ütemező működésébe?
- A feladattal kapcsolatos elvárásaink hogyan jelenjenek meg?
- Megoldás: **prioritás** (priority)

a taszk jellemzője, amely végrehajtásának fontosságát fejezi ki.

Értékkészlete a pozitív egész számok részhalmaza (pl. 0-127)

„magasabb” prioritás == „fontosabb” taszk

„magasabb” prioritás /= „nagyobb” szám

Jellemzően tartományokra osztják

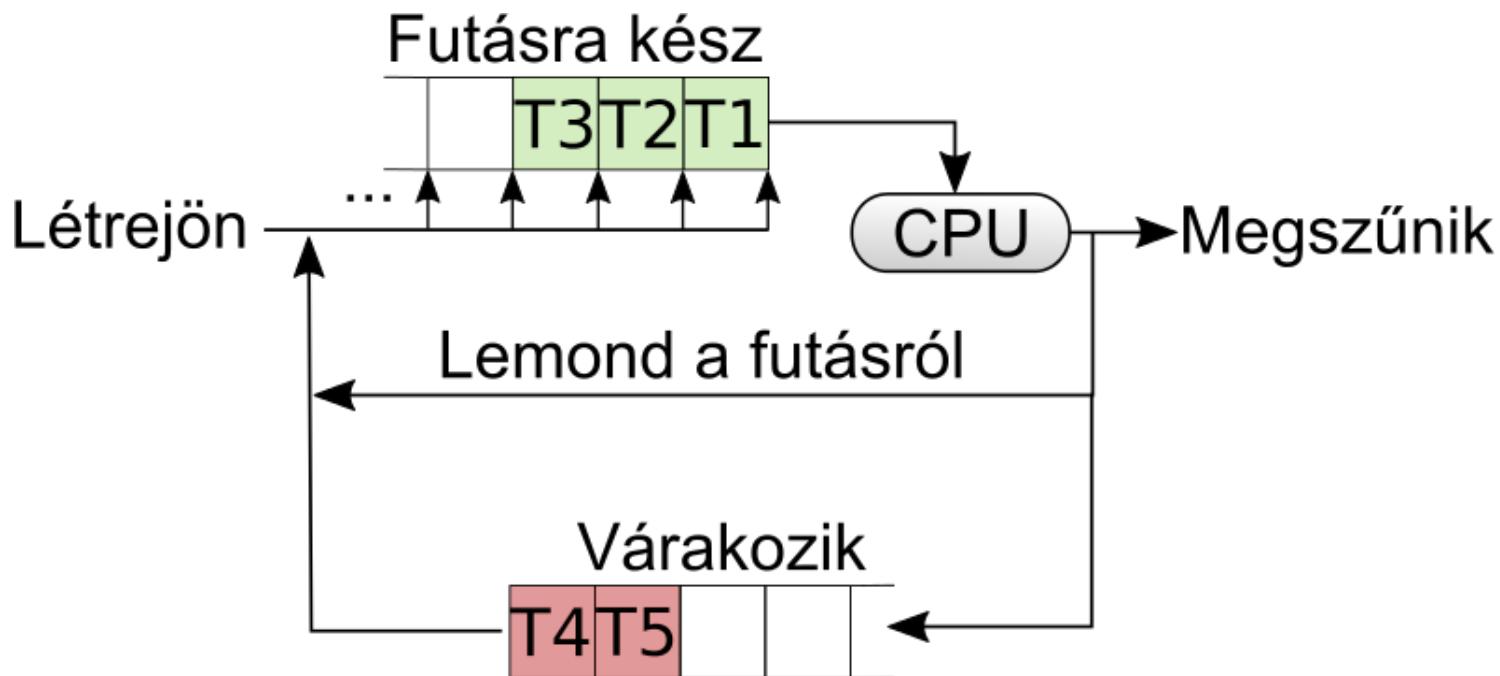
Unix: 0-49 kernel módú, 50 - 127 felhasználói módú taszkok

Linux: 0-99 „valósidejű”, 100-139 időosztásos

Windows: 16-31 statikus prioritású, 1-15 dinamikus prioritású taszkok

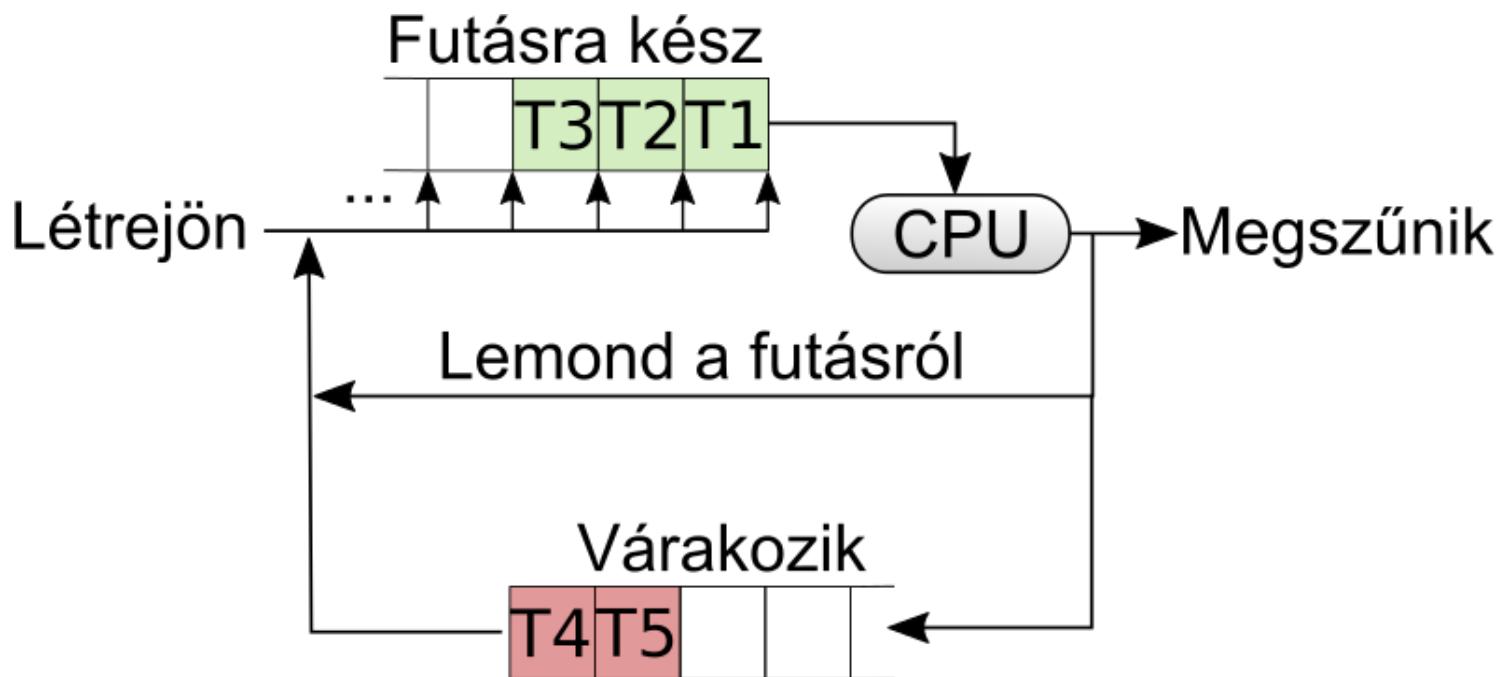
Prioritásos ütemező

A futásra kész taszkok prioritásuk szerint csökkenő sorrendben vannak.



Emlékeztető: SJF ütemező

A futásra kész taszkok CPU-löketidejük szerint növekvő sorrendben vannak.



A prioritás meghatározása

- Külső prioritás
a taszk operációs rendszeren kívül meghatározott végrehajtási fontossága

Függhet a feladat jellegétől (pl. valósidejű), felhasználói szempontoktól

A taszk és a felhasználó (korlátozottan) befolyásolhatja

növelheti vagy csökkentheti jogosultságainak megfelelő mértékben

- Belső prioritás
a taszk operációs rendszer által meghatározott végrehajtási fontossága

Az eddigi ütemezők közül melyik tekinthető prioritásosnak is?

- Statikus prioritás
a taszk elindulása előtt rögzített, életciklusa alatt állandó
- Dinamikus prioritás
a taszk életciklusa során időközönként kiszámított
→ újra kell rendezni az FK sort → no a komplexitás és így a rezsiköltség

A prioritásos ütemezők értékelése

- Adatstruktúra
 - rendezett lista vagy fa
- Tulajdonságai
 - kooperatív és preemptív ütemező is lehet
 - kicsit bonyolultabb műveletek (rendezés)
 - a dinamikus prioritást újra kell számítani → a rendezett lista/fa újraépítése
- Algoritmikus komplexitás
 - beillesztés: $O(N)$ vagy $O(\log N)$
 - teljes újraépítés: $O(N^2)$ vagy $O(N * \log N)$
- Rezsiköltség
 - prioritás számítása + beszúrás + időnkénti újrarendezés (dinamikus prioritásnál)
- Minőségi jellemzők, problémák
 - rugalmas
 - **a prioritás meghatározásának módja és számítási komplexitása?**
- Feladatok
 - Hogyan becsülhetjük a löketidőt a prioritás segítségével?
 - Hasonlítsuk össze a SJF/SRTF ütemezővel, ha a prioritás a löketidőt becsli!

Kiéheztetés: a prioritásos ütemezők alapproblémája

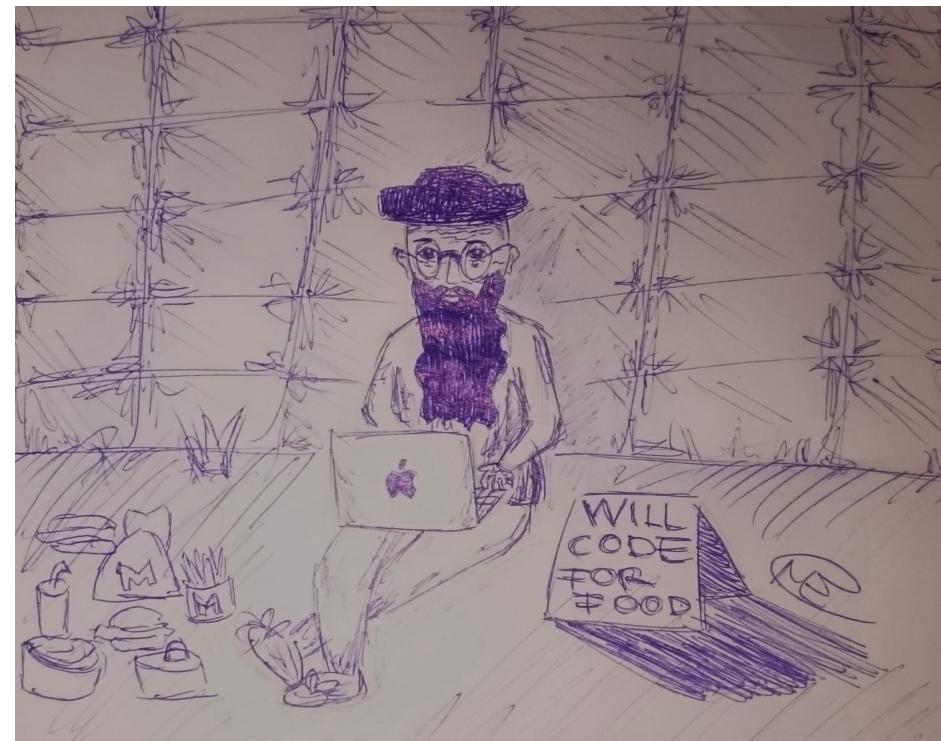
- Prioritás → nem egyenrangúak a taszkok

A **kiéheztetés** (starvation) az a jelenség, amikor egy taszkot folyamatosan megelőznek nála magasabb prioritásúak, így nem jut processzorhoz.

- Hogyan kerülhető el?

- **statikus prioritásokkal sehogy**
- emeljük fokozatosan a prioritását, míg végül futó állapotba kerül

öregítés (aging)



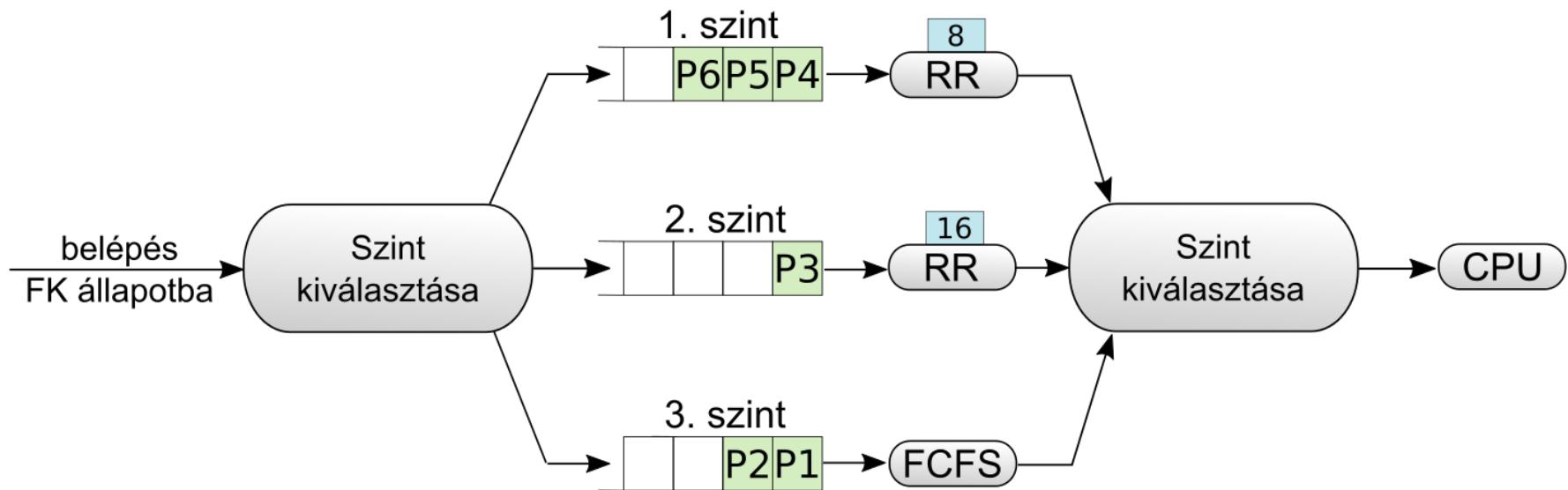
Problémák az eddigi ütemezőkkel

- A prioritás „zsúfolt” információleíró
elvesznek lényeges részletek
- A feladatokkal kapcsolatos elvárások sokrétűek
egyetlen ütemező nem képes minden elvárásnak megfelelni
- Az ütemezők erősségei különbözők
ötvözhetjük őket? ellentmondások?
- Megoldás?
 - alkalmazzunk többféle ütemezőt egyszerre

Többszintű ütemezés

- taszkok (feladatok) és elvárások kategorizálása → ütemezési szintek
- szintenként különböző ütemező

A többszintű ütemező alapműködése



Szint kiválasztása

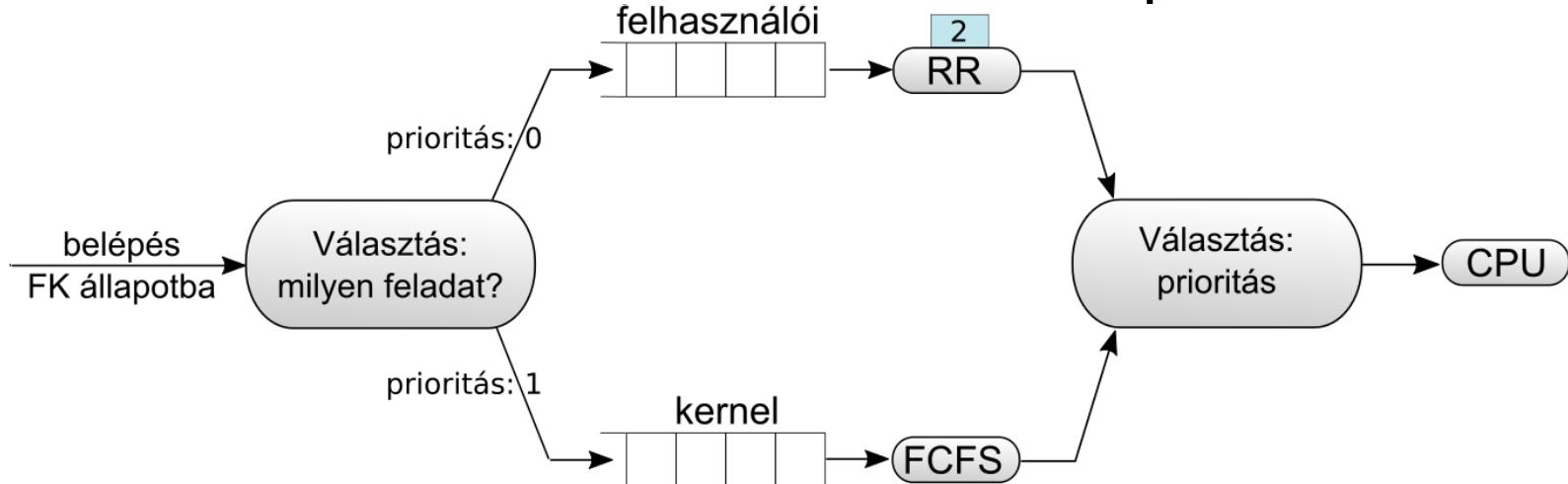
- Melyik szinten helyezzük el az FK állapotba került taszkot?
- Melyik szintről válasszuk ki a következő F taszkot?
- Ötlet?
- A taszkokat tulajdonságaik és előéletük alapján rendezhetjük
 - valósidejű, kernel feladat, CPU-intenzív, I/O-intenzív, új belépő stb.
- A szintek közül valamilyen egyszerű algoritmussal választhatunk
 - körforgó ütemezés: minden szinthez rendelhetünk egy időszeletet a fontosabb szintek (pl. valósidejű vagy kernel) nagyobb időszeletet kapnak
 - prioritásos ütemező: fontossági érték szerint sorba állítjuk jelentkezik a kiéheztetés
- Hogyan kerülhető el a kiéheztetés prioritásos szintütemező esetén?
 - megengedjük a szintek közötti váltást a „felfele” és „lefelé” léptetés számára kell egy algoritmus (mikor?)

Statikus többszintű sorok (static multilevel queues)

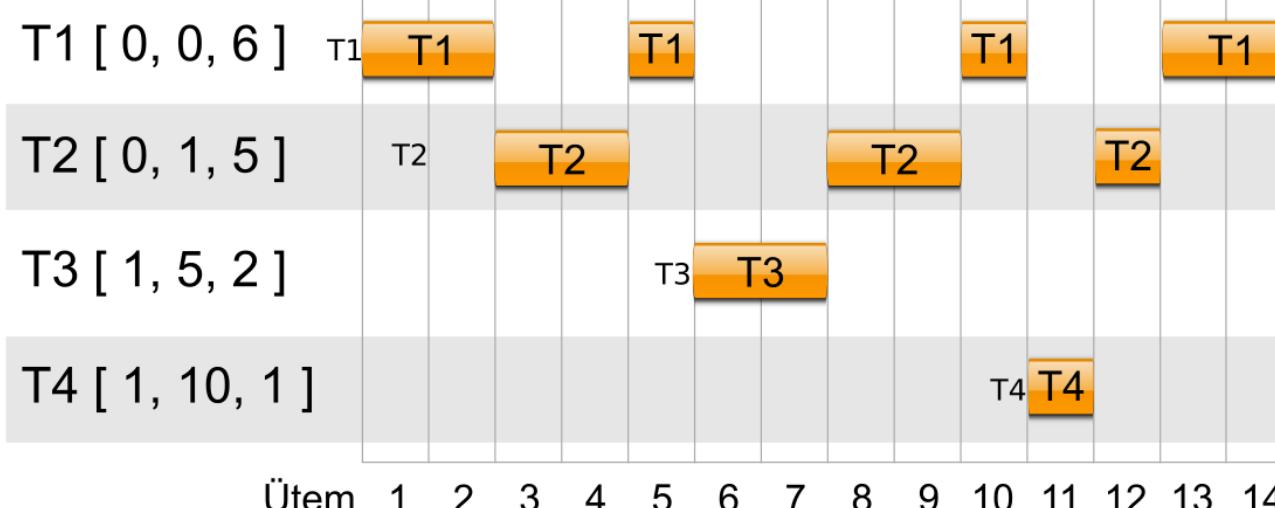
- A taszkokat statikus módon rendeljük a szintekhez (sorokhoz)
 - nem léphetnek át szintek között, pl. statikus prioritást kapnak
 - hozzákötődnek az adott ütemezési algoritmushoz
- A feladatok és az elvárások jellege szerinti alakítjuk ki a szinteket
 - valósidejű
 - rendszerfeladatok
 - interaktív
 - kötegelt (nagy CPU-löketidejű, de nem időkritikus)
- Meghatározzuk a szintek globális ütemezőjét (pl. prioritásos vagy RR)
 - preemptív ütemező
- Szintenként meghatározzuk ütemezési algoritmusokat, pl.:
 - valósidejű → FCFS, kooperatív
 - rendszerfeladatok → prioritásos, kooperatív / preemptív
 - interaktív → RR, preemptív
 - kötegelt → SJF, kooperatív

Nem gond a kooperatív ütemezés?

Statikus többszintű ütemező példa



Taszkok [Prioritás, Start, CPU-löket]

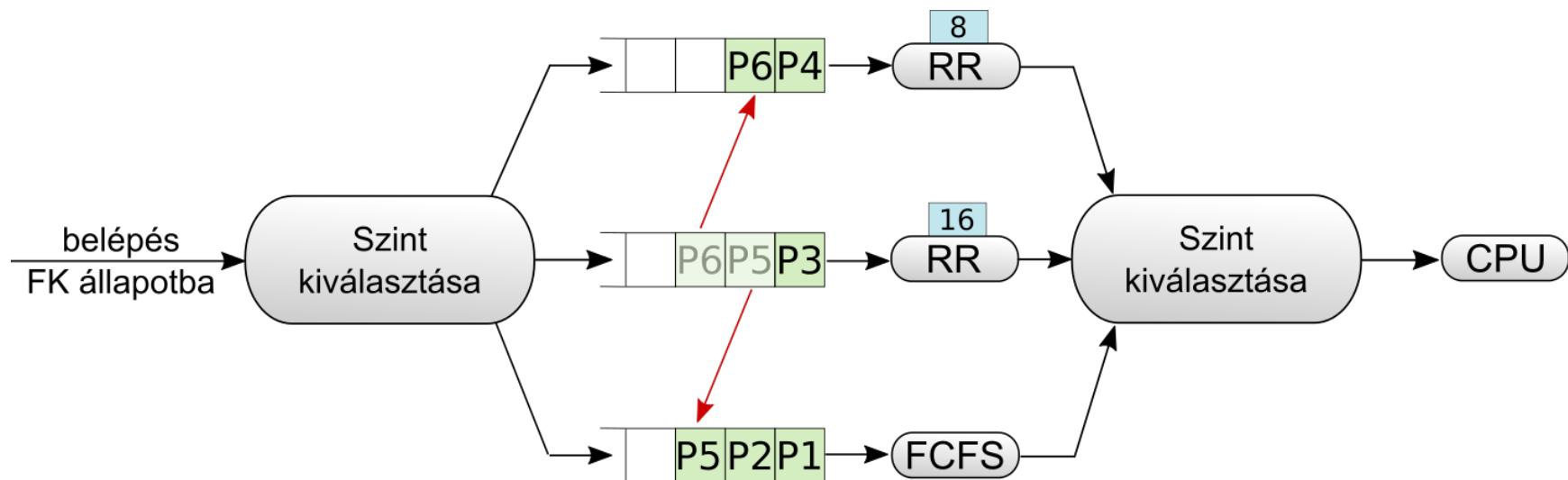


A statikus többszintű sorok értékelése

- Adatstruktúra
 - a szintek ütemezőitől függ (FIFO, lista / fa)
- Tulajdonságai
 - globálisan jellemzően preemptív ütemező
- Algoritmikus komplexitás
 - globálisan $O(1)$, emellett a szintek ütemezőitől függ
- Rezsiköltség
 - globális: alacsony, szintek: az ütemezőtől függ
- Minőségi jellemzők, problémák
 - egyszerű, többféle szint – többféle algoritmus
 - feladatoknak és elvárásoknak megfelelő ütemezés
 - statikus, jelentkezhet a **kiéheztetés**
 - a taszkok „jellemváltozása” nem kezelhető
 - pl. kötegelt feladat időnként interaktív, rövid ideig elvárt valósidejű működés stb.
- Feladatok
 - Miért nem működik az öregítés?
 - Mi történik, ha dinamikus prioritást alkalmazunk a globális döntésekben?

Dinamikus többszintű sorok (dynamic multilevel queues)

- A statikus többszintű ütemezőhöz hasonlóan működik, de ...
- Dinamikusan kezeljük a taszkok sorokhoz rendelését
 - pl. a taszk globális prioritása dinamikusan változhat dinamikusan rendeli a taszkot a szintekhez
- A taszk mozoghat a sorok között, jellemzően
 - „fentebb” lép (upgrade): magasabb prioritási szintre kerül
 - „lentebb” lép (downgrade): alacsonyabb prioritási szintre kerül

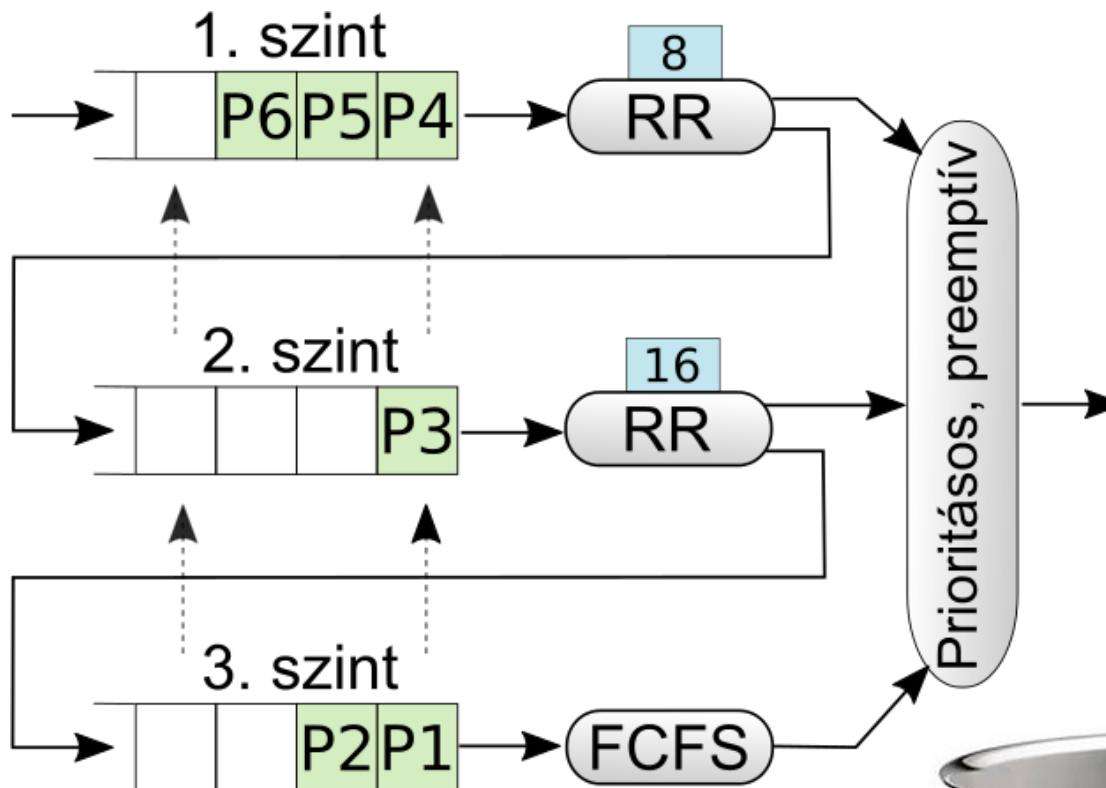


A dinamikus többszintű sorok értékelése

- A módszer előnyei
 - a statikus módszer előnyei +
 - használhatjuk az öregítést, így elkerülhető a kiéheztetés
 - adaptálódhat a taszkok változó viselkedéséhez
 - komplex, adaptív ütemezést valósíthatunk meg
- A módszer hátrányai
 - az upgrade / downgrade bonyolítja az ütemezőt
 - több számítás (dinamikus prioritások, beszúrás, átrendezés)
 - nő a számítási komplexitás (hogyan?) és a rezsiköltség
 - gondosabb tervezést igényel

Többszintű visszacsatolt sorok ütemező

multilevel feedback queue (MFQ)



Taszkok szintlépései:

- amelyik kihasználja az időszeletét, az lentebb lép
- várakozó állapotba kerülő taszkok fentebb lépnek

Turing díj járt érte

Az MFQ ütemező értékelése

- Többszintű, dinamikus ütemező
 - egyszerű megvalósítás és algoritmus
- Alapötlete: tanulás a múlt eseményeiből
 - minél többet használja egy taszk a CPU-t, annál ...
 - minél kevesebbet, annál ...

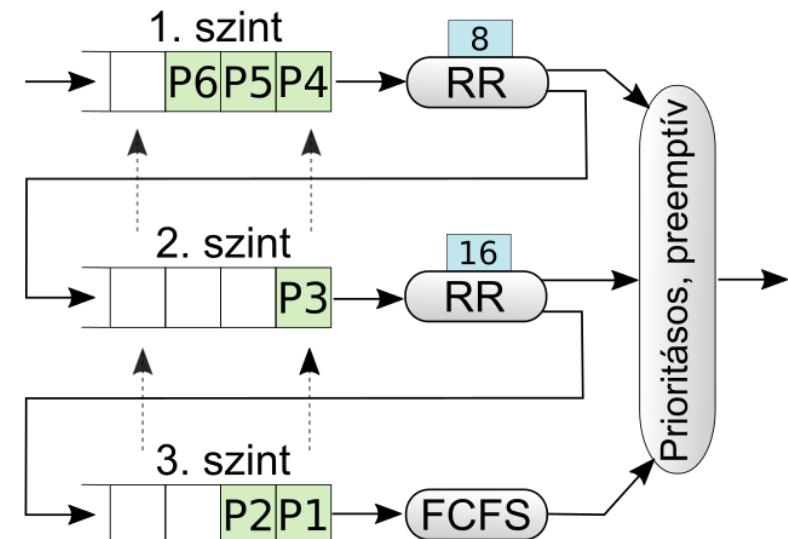
- A szintek közötti mozgás
 - a CPU-intenzív taszkok (sok CPU-idő) alacsonyabb prioritási szintekre kerülnek
 - az I/O-intenzív taszkok (kevés CPU-idő) magasabb prioritási szinteken maradnak

Milyen taszkokat részesít előnyben?

Hogyan működik az öregítés?

Milyen egyszerű algoritmust közelít a működése?

- Sok mai ütemező alapja (több Unix változat, Windows NT kernel)



Tervezzünk egy ütemezőt!

Nem lesz nehéz.

Eddigi tudásunk

- Az elvárásainkról...
 - sokféle taszkot futtatunk (CPU- és I/O intenzív, valósidejű, játék stb.)
 - szeretünk beleszólni az OS működésébe
 - kis várakozási és gyors válaszidőt szeretnénk
 - a kis rezsiköltség is elvárás
- Az OS működéséről...
 - felügyeli a taszkok működését, vezérli (ismeri) az életciklusukat
 - kernel és felhasználói mód
- Az ütemezőkről...
 - ha tudnánk előre a taszkok löketidejét, lenne optimális választás
 - sokféle alapalgoritmus van, önmagában egyik sem jó mindenre
 - a többszintű dinamikus ütemező sokféle ütemezőt tud kombinálni
 - a prioritás sokféle szempontot tud integrálni

Az ütemezőnk globális tulajdonságai

- prioritásos
 - mivel vannak fontossági szempontok minden üzemmódban
- többszintű
 - triviálisan: kernel és felhasználói mód
 - eltérő algoritmusok tűnnek célszerűnek
- dinamikus
 - változnak a taszkák tulajdonságai, pl. üzemmódot váltanak
- optimalitásra törekszik
 - löketidő becslése
 - minél egyszerűbb működés, kisebb komplexitás
 - minél kisebb rezsiköltség
- Többszintű, dinamikus prioritásos ütemező

Megfigyelések az ütemezési szintekről

- kernel mód
 - a kernel programkódja fut
 - előre ismert feladatok: kis CPU-, hosszú I/O-löketek
 - jellemzően perifériakezelésre van szükség (pl. diszk és terminál)
 - konvoj-hatás nem alakul ki
 - vannak fontosabb feladatok
 - ha egyszerre több taszk FK, akkor a futási sorrend nem mindegy
 - minél kisebb rezsiköltség a cél
- felhasználói mód
 - az alkalmazások programkódja fut
 - nem ismerjük előzetesen
 - lehetnek CPU-, I/O-intenzív és változó működésű taszkok is
 - várhatnak erőforrásokra (pl. diszk és terminál) → kernel módba lépnek
 - felléphet a konvoj hatás, ami ellen védekezni kell
 - lehetnek a feladatok fontosságára vonatkozó felhasználói preferenciák
 - az egyformán fontos feladatokat ugyanolyan esélyekkel futtassuk
 - ne legyen túl komplex

Milyen algoritmusokat válasszunk?

- kernel mód

- nincs nagy CPU-löket, ezért használjunk kooperatív ütemezőt (miért?)
- nem preemptív, ezért elég a statikus prioritás (miért?)
- nem preemptív, ezért egyszerű a kernel adatstruktúrák védelme (miért?)
- kicsi rezsiköltség

Hogyan és mikor határozzuk meg a statikus prioritást?

- felhasználói mód

- a konvojhatás veszélye miatt **preemptív** ütemező kell
- az SRTF jó lenne, ha a jövőbe látnánk, de nem
- löketidő jóslása → prioritás
- a felhasználó is beleszólna → **prioritásos ütemező**
- egyenlő esélyek → **körforgó (RR) ütemező**

Hogyan kombináljuk a prioritásos és a RR ütemezőket egy rendszerré?

Hogyan és mikor számítsuk ki a dinamikus prioritást?

Hogyan kezeljük a kiéheztetést (öregítéssel, de minden módon)?

Kernel módú prioritás

- Statikus
- Nem függ attól, hogy
 - mennyi volt a folyamat prioritása felhasználói módban (másik szinten vagyunk)
 - mennyi a löketideje (nem SJF ütemezőt használunk)
- Mitől függ? Miért van rá szükség?
A prioritást a folyamat elalvási oka határozza meg

alvási prioritás (sleep priority)

pl. 20 diszk I/O, 28 terminál I/O

- Mikor számítják ki?
 - Felhasználói mód → kernel mód?
 - Várakozó állapotban?
 - Amikor futásra késszé válik?

Felhasználói módú prioritás (p_{usrpri})

- Dinamikusan változik
- Löketidő becslése?
 - korábbi CPU-használattal (p_{cpu})
 minden óraciklusban növeljük a futó taszknál p_{pcu++}
- A felhasználó beleszólhat (p_{nice})
- Számítása a fenti két tényezőből

$$p_{pri} = P_{USER} + p_{cpu} / 4 + 2 * p_{nice}$$

a P_{USER} konstans a kernel és a felhasználói módot választja szét

$$P_{USER} = 50$$

Emiatt a p_{pri} nem lehet kisebb 50-nél.

Tapasztalatok alapján skálázzuk a két tényező hatását.

A 2-vel szorzás és a 4-el osztás egyszerű műveletek (bit shift).

A löketidő jóslása

- A p_{cpu} nem nőhet az éigig, „öregíteni” kell

$$p_{cpu} = p_{cpu} * KF \quad KF \text{ korrekciós faktor} < 1$$

- A korrekciós faktor meghatározása

$$KF = 1/2 \text{ (bit shift, egyszerű művelet)}$$

Mi ezzel a baj?

- Mi lehet egy jobb korrekciós faktor?

- ha nincs futásra kész (FK) taszk
→ a p_{cpu} elfelejthető
- ha kevés FK taszk van
→ gyorsan felejthetünk
- sok FK taszk van
→ fontos a löketidő becslése, lassan felejtsük a p_{cpu} -t

A fentiek alapján mitől függjön a KF?

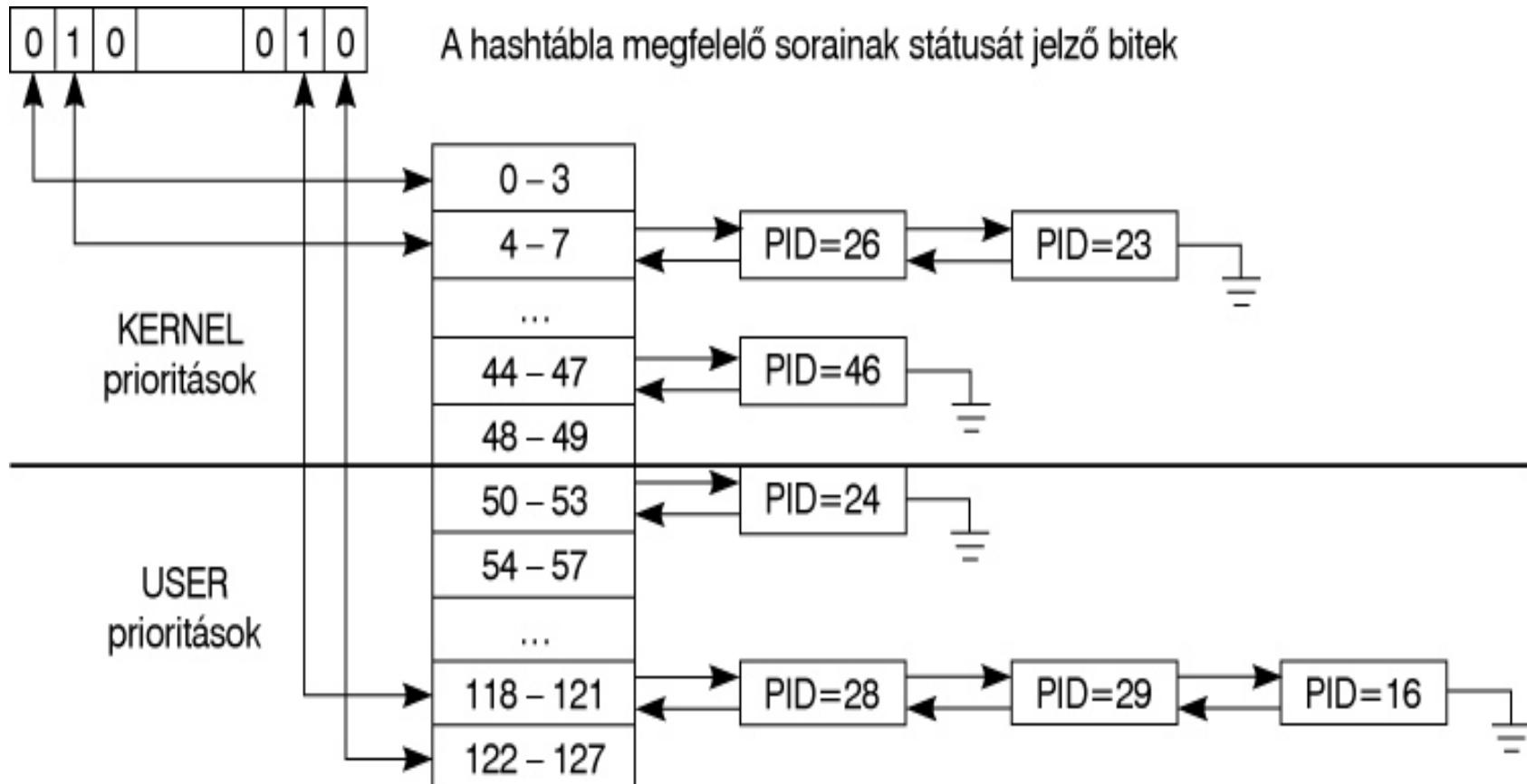
`load_avg`

$$KF = 2 * load_avg / (2 * load_avg + 1)$$

A felhasználói módú ütemezés

- Prioritásos ütemező, dinamikus prioritásokkal
 - a prioritás a löketidőt becsli és a felhasználónak is enged beleszólást
- Mi a helyzet az egyforma prioritású taszkokkal?
 - egyenletes kiszolgálást szeretnénk
→ RR ütemező
- Többszintű ütemező: **prioritásos + RR**
 - prioritástartományok → szintek
 - a szintek között a prioritásoknak megfelelően választunk
 - egy szinten belül időosztásos, körforgó ütemezőt használunk
- Az ütemező jellemzői
 - preemptív
 - jó várakozási és körülfordulási idők
 - jó válaszidő
 - alacsony rezsiköltség

Az ütemezőnk adatstruktúrája: FIFO + hash



Az ütemező működése

- Kernel módban eseményvezérelt
 - taszk esemény hatására felébred
 - kap egy statikus prioritást
 - fut, ameddig akar (kernel módban)
 - több FK taszk közül a prioritás dönt
- Felhasználói módban idővezérelt (óramegszakítás esemény)
 - minden óraciklusban
 - taszkváltás, ha magasabb prioritású taszk lett FK
 - p_cpu++ a futó taszkra
 - minden RR időszelet végén (10 óraciklus)
 - RR átütemezés, ha ugyanazon a prioritási szinten más taszk is van.
 - minden 100. óraciklus végén
 - p_cpu „öregítése” (KF, load_avg)
 - prioritások újraszámítása
 - FK sorok újrarendezése

(Számítási példa)

- Képletek

$$p_{pri} = P_{USER} + p_{cpu} / 4 + 2 * p_{nice}$$

$$P_{USER} = 50$$

$$p_{cpu} = p_{cpu} * KF$$

$$KF = 2 * load_avg / (2 * load_avg + 1)$$

- Algoritmus

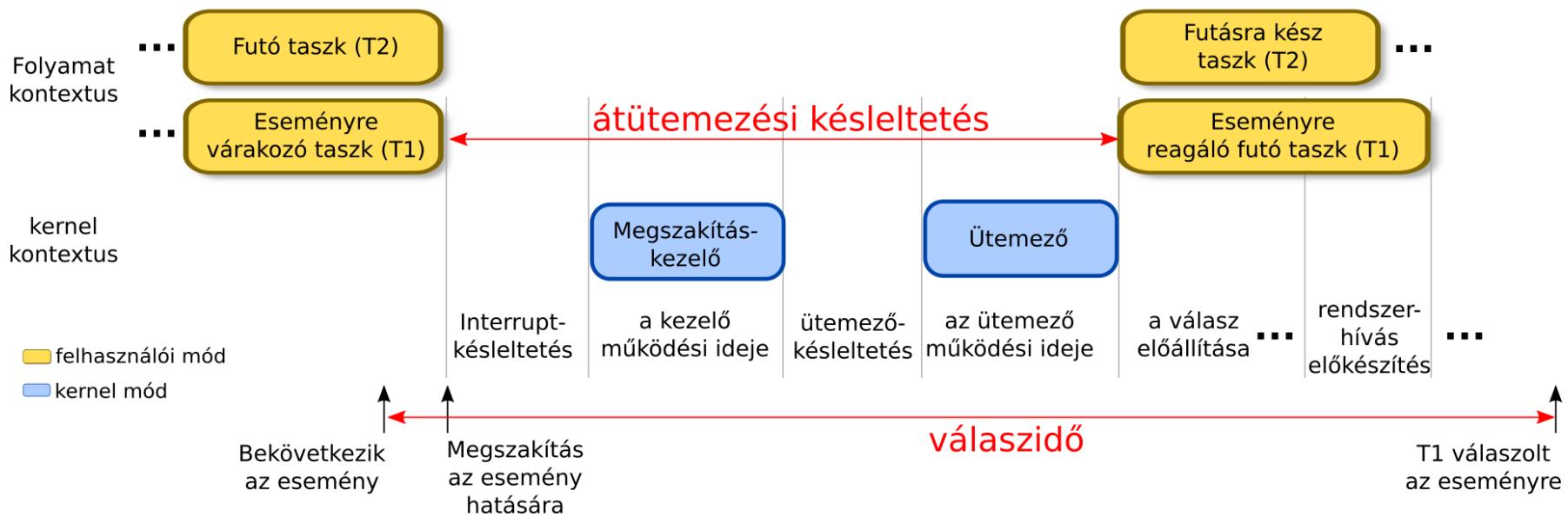
- minden óraciklusban
 - F és FK prioritások ellenőrzése
 - $p_{cpu}++$ a F taszkra
- minden 10. óraciklusban RR
- minden 100. ütemben újraszámítás

Az ütemezők neve és értékelése

- Klasszikus Unix ütemező
 - többszintű, prioritásos, időosztásos
 - System V R3, BSD 4.3, korai Linux stb.
- Erősségei
 - kötegelt és interaktív taszkok keverékére jól működik
 - jó válaszidőt biztosít az interaktív taszkok számára, miközben
 - nem engedi a háttérben futó kötegelt munkák kiheztetését
- Problémái
 - komplexitás?
 - pl. FK-sorok újrarendezése
 - késleltetés?
 - **valósidejű taszkok** Hogyan legyen kernel módban preemptív?
 - **prioritásinverzió** (priority inversion)
 - egyprocesszoros hardverre fejlesztették
 - a kernel belső konkurenciájával nem foglalkozik

Hogyan működjön többprocesszoros környezetben?

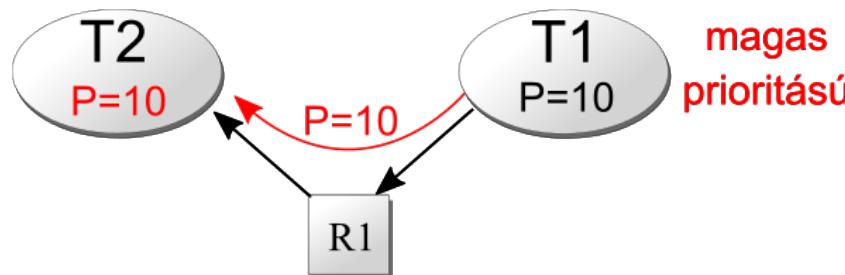
A válaszidő és a késleltetések



- Interrupt-késleltetés
 - pl. más megszakítások kiszolgálása
- Ütemező-késleltetés Ezen lehetne javítani!
 - más kernel tevékenységek zajlanak (pl. rendszerhívás)
- A válasz előállítása
 - további kernel-tevékenységeket igényelhet (rendszerhívás, memóriakezelés stb.)
- A válasz elküldése
 - rendszerhívás előkészítése – megszakítás – megszakításkezelés – ...

A prioritásinverzió és kezelése

- A prioritásinverzió



- Hogyan kezelhető?
 - prioritásöröklés (priority inheritance)
- A prioritásöröklés korlátai
 - bonyolult függőség
 - nem triviális az öröklés
 - egy taszk több másiktól is függhet
 - túl messzire gyűrűzik a prioritásnövelés
 - más is várhat az erőforrásra
 - nem a kívánt hatást érjük el
 - valósidejű működés esetén nem jó megoldás
 - az erőforrást foglaló taszk nem valósidejű, „bármeddig” futhat

Kernel preemptivitás

- A kooperatív ütemezésű kernel mód egyszerű, de
 - bonyolódó funkciók → egyre nagyobb késleltetés
 - több végrehajtó egységen túl korlátozó
- Miért jó a preemptív kernel?
 - jobb időkorlátok tarthatók → valósidejű működés
 - több taszk futhat egyszerre kernel módban
- Megoldási ötletek?
 - preemptív kernel módú ütemező
 - bonyolultabb ütemezési algoritmus
 - adatstruktúrák védelme a konkurens végrehajtás miatt
 - eljárások újrahívhatóságának biztosítása
 - jelentősen nő a rezsiköltség
 - részleges preemptivitás „átütemezési pontokon”
 - mérsékeltebb rezsiköltség-növekedés

Kernel preemptivitás átütemezési pontokon

- A magas késleltetésre hajlamos programágakat érdemes feldarabolni
- A kernel módú taszkváltáshoz...
 - biztosítani a kernel adatstruktúrák konzisztenciáját
- pl. Linux 2.2/2.4 „Low latency kernel patch” [Molnár Ingo](#) ~ 100 pont
 - zenei alkalmazások problémái [motiválták](#)
Magas diszk I/O esetén a taszkok 150-800ms késleltetést is elszennyezhetnek.

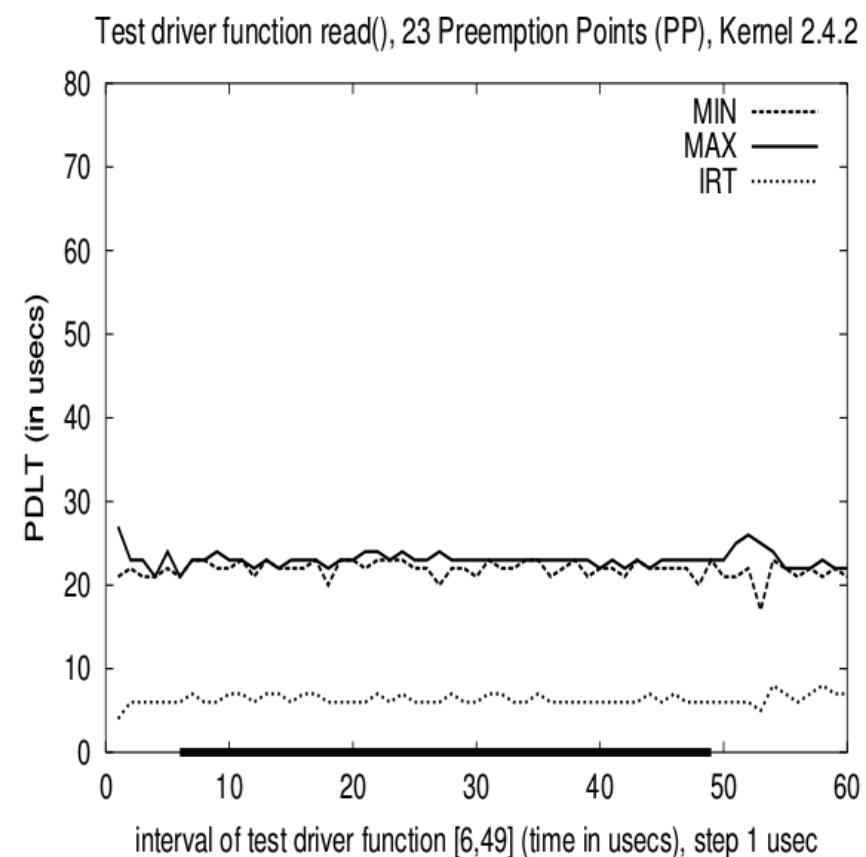
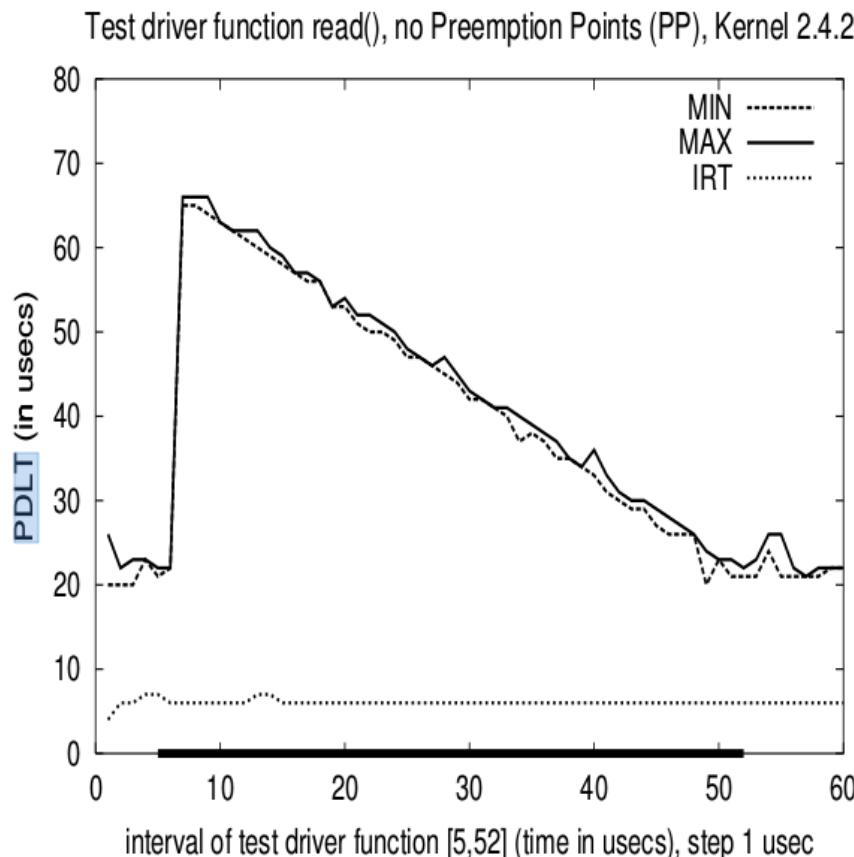
```
if (current->need_resched) {           // preemption point (PP)
    current->state = TASK_RUNNING;
    schedule();
}
```

- pl. System V R4 (SRV4) Unix ütemező
 - bevezette a valósidejű taszkok támogatását
 - amihez szükség átütemezési pontokra
 - az ütemező azt ellenőrzi, hogy van-e valósidejű taszk
 - ha igen, akkor az kapja meg a futás jogát

Az átütemezési pontok alkalmazásának hatása

Átütemezési késleltetés: PDLT – Process Dispatch Latency Time

A megszakítás és a kezelésére felébresztett folyamat első utasításának végrehajtása között eltelt idő.



Forrás: Arnd Christian Heursch PhD disszertációja, 2006

Teljesen preemptív kernel

- Több végrehajtó egység esetén
 - futhat több taszk egyszerre nem preemptív kernel esetén?
 - az átütemezési pontok nem elégségesek (miért?)
- Az adatstruktúrákat védeni kell
 - ha két kernel módú taszk ugyanazt módosítja, baj van
 - ha egyik taszk módosít miközben a másik olvas, szintén baj van
- Megoldás: kritikus régiók védelme zárakkal (lock)
 - zárak és kulcsok (részletesen lásd [Szinkronizáció](#))
 - akinél a kulcs van, az használhatja az adatokat (erőforrást)
 - akinél nincs, az vár, amíg megkapja
 - az átütemezés kritikus helyeken le is tiltható, ha nincs más megoldás (pl. hol?)
- A mai operációs rendszerekben elterjedt opción
 - Linux 2.6+
 - Windows az NT kernelektől kezdve

Linux 2.6 „kpreempt” patch

- A beágyazott rendszerekkel foglalkozó MontaVista megoldása
 - valósidejű rendszerekből származó ötletekből indultak el
 - nyílt forráskódú projekté válta „kpreempt” néven
 - majd integrálták a 2.6-os Linux kernelbe

```
$ grep PREEMPT /boot/config-`uname -r`  
CONFIG_PREEMPT_NOTIFIERS=y  
# CONFIG_PREEMPT_NONE is not set  
CONFIG_PREEMPT_VOLUNTARY=y  
# CONFIG_PREEMPT is not set
```

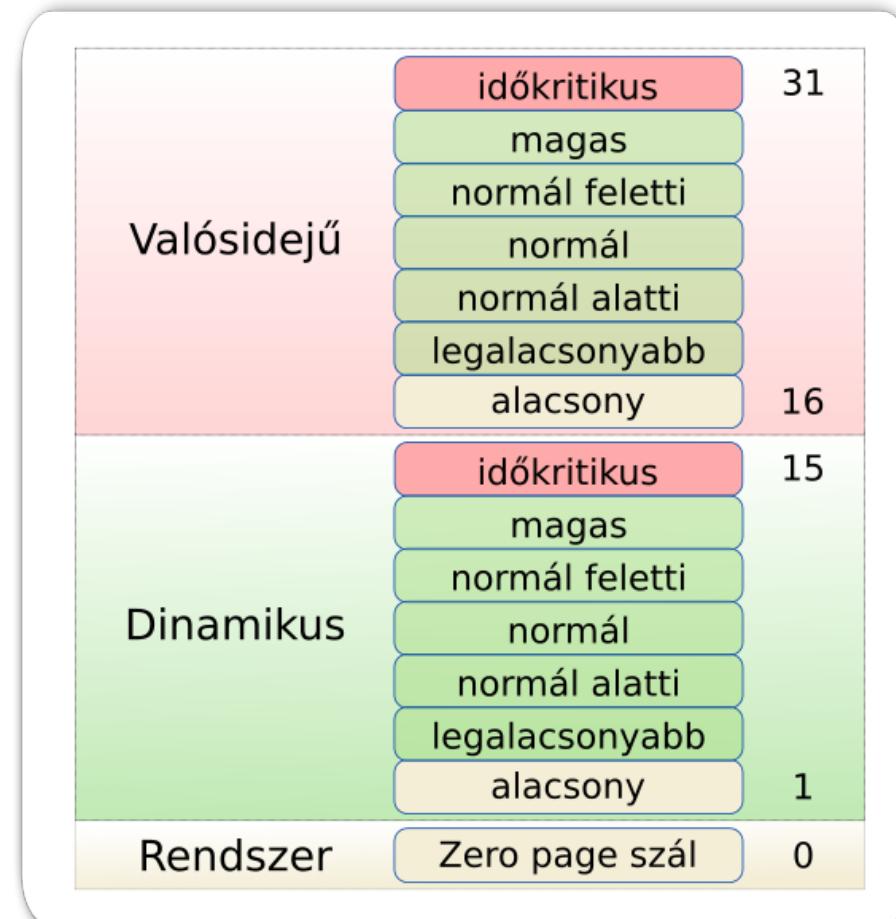
Kemény valósidejű ütemezés

- Hard real-time: 1 valószínűséggel határidőn belül válaszol
 - általános célú OS: **soft** real-time: igyekszik, de nem garantálja
 - a **biztos** válasz speciális algoritmusokat igényel
- Jellemzően periodikus taszkok
 - adott intervallumonként igénylik a CPU-t (p)
 - rögzített futásidejük van (t)
$$0 \leq t \leq d \leq p$$
 - előre ismert a határidő (d)
- „Legrövidebb periódusidejű előre” (Rate-Monotonic Scheduling, RMS)
 - statikus prioritás ($\sim 1/p$), preemptív
 - feltételezi, hogy a CPU-löket állandó (= jósolható)
 - **belépési feltétel:** becsült t , ismert p és d alapján
- „Legkorábbi határidejű előre” (Earliest Deadline First, EDF)
 - dinamikus prioritás ($\sim 1/d$), preemptív
 - nem csak periodikus taszkokra, nem igényli t állandóságát
 - a dinamikus prioritás miatt **költségesebb, nehezebben jósolható**
 - ha megoldható a feladat, akkor az EDF optimális

Ütemezők a gyakorlatban: Windows

Többszintű, prioritásos, időosztásos, preemptív, O(1) szál-ütemező

- Kernel prioritási szintek
- Windows API
 - folyamat prioritásosztályok
 - szál prioritásmódosítók
`SetThreadPriority()`
- Prioritásemelés (Boost)
 - a dinamikus tartományban
 - késleltetés csökkentése
 - I/O befejezés
 - UI esemény
 - éhezés elkerülése
 - prioritásinverzió kezelése



- Felhasználói prioritásemelés (pl. MultiMedia Class Scheduler Service)

Demo: éhezés és prioritásemelés Windows alatt

- Hozzávalók: Sysinternals cpustres, Teljesítményfigyelő, Feladatkezelő
- cpustres.exe: 1. thread: activity maximum, priority below normal
- Teljesítményfigyelő: számláló hozzáadása
 - végrehajtási szál – jelenlegi prioritás
 - objektum: CPUSTRES/1
- Feladatkezelő (adminként)
 - a Teljesítményfigyelő prioritását valósidejű szintre emelni (Részletek fül)
- Terhelésnövelés: újabb cpustres.exe (akár kettő is)
 - 1. thread: activity maximum
- A prioritásemelés „meghallgatása”
 - az első cpustress helyett egy audiolejátszó is használható
 - a MultiMedia Class Scheduler szolgáltatást ki kell kapcsolni

Ütemezők a gyakorlatban: Linux

(Az első változatok (v2 előtt) a tradicionális UNIX ütemezőre épültek)

- 2.4-es kernel előtt
 - real-time, nem-preemptív, normál
 - $O(N)$ ütemező
 - nem preemptív kernel
- kernel v2.6
 - **$O(1)$ ütemező**
 - prioritásos, visszacsatolt többszintű, preemptív, időosztásos
 - 140 szint, prioritások: 0-99 „valósidejű” (statikus), 100-139 időosztásos (dinamikus)
 - „active” (még van időszelete) és „expired” (lejárt időszeletű) sorok szintenként
 - az aktívból a lejártba mozgatás közben számolja újra a prioritást
 - ha az aktív kiürült, akkor megcseréli a lejárttal (pointer művelet)
 - a régóta várakozó folyamatok kapnak egy kis bónuszt a prioritásukhoz
- 2.6.23 kerneltől: CFS (Completely Fair Scheduler)
 - (következő fólia)
- [További részletek](#)

Linux CFS (Molnár Ingo)

- A korábbi $O(1)$ ütemezőt felváltó ütemező
 - szálakat ütemez
 - kernel átütemezési pontokat használ
 - teljesen preemptív kernel üzemmód is bekapcsolható
- Sor (lista) helyett *piros-fekete fa* <linux/rbtree.h>
 - egy virtuális futási idő (vruntime) szerint rendez a taszkokat
 - a kisebb értékek balra, a nagyobbak jobbra
 - $O(\log n)$ komplexitás
- Cél: egyenletes vruntime minden taszkra
 - akinek a legkisebb, az fut $\text{vruntime} += \text{idő}_\text{delta} * (\text{NICE}_0_\text{LOAD} / \text{curr}->\text{load.weight})$
 - ne ütemezzünk át túl gyakran:
 - target scheduling latency (TSL): maximális várakozás FK állapotban
 - pl. 2 egyforma taszk, 20ms TSL \rightarrow 10 ms időszelet
 - minimum granularity (MG): minimális garantált futásidő
 - pl. 10db egyforma taszk, 20ms TSL, 5ms MG $\rightarrow \max(2\text{ms}, 5\text{ms}) = 5\text{ms}$ időszelet
 - alacsony: jó késleltetés (desktop) magasabb: jó kötegelt működés (szerver)

Linux kísérletek

- Ismerkedés a parancsokkal

```
ps, kill, renice, nice, top, htop  
man renice
```

- A renice hatása

- 1. terminál: stress -v --cpu 2
- 2. terminál: top -u <uname> („b” futó taszkok kiemelése, „T” idő szerinti lista)
- Megfigyelni a TIME idő változását a két taszkra
- 3. terminál

```
sudo su -  
renice -20 <stressPID1>  
renice 19 <stressPID2>
```

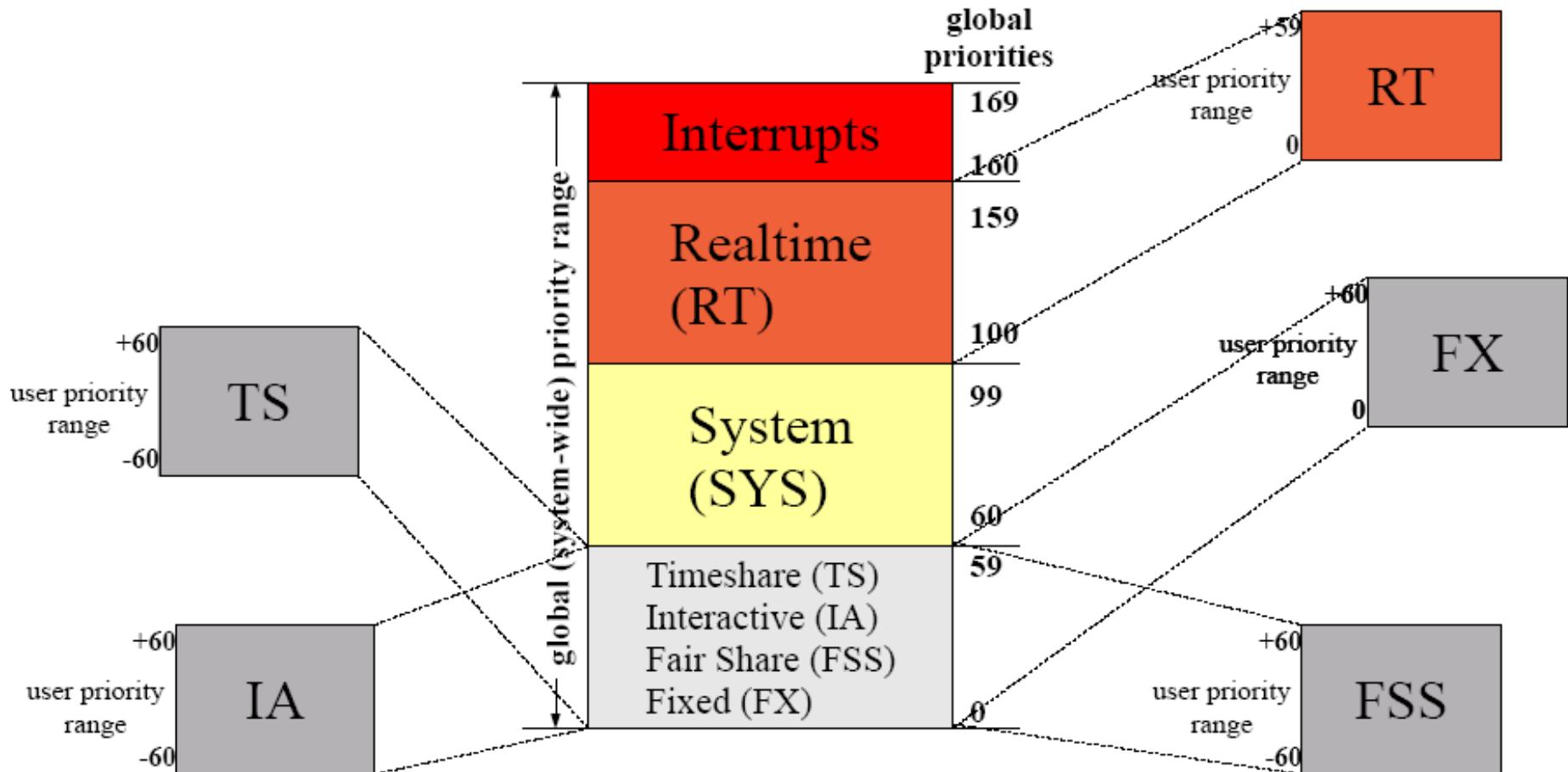
- (A terhelés további növelése: stress -q --cpu 2)
- Újra megfigyelni a CPU idő változását a két taszkra

- Az ütemező rombadöntése: fork() bomba

```
:() { :|:& };:
```

Ütemezők a gyakorlatban: Solaris

- Moduláris, szálalapú, teljesen preemptív, prioritásos, időosztásos

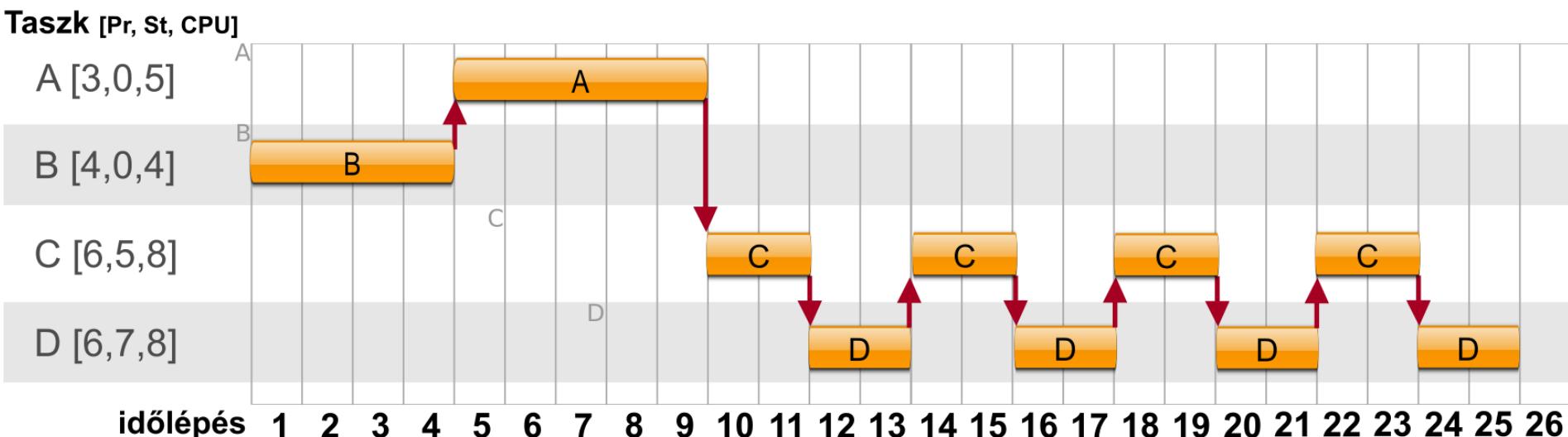


Ütemezők és jellemzőik (számítási példák)

- **Ütemezők**
 - FCFS: egyszerű FIFO (kooperatív)
 - RR: időszeletenként megszakítja a futó taszkot, és FIFO elv szerint átütmez
 - SJF: a legrövidebb löketidejű taszk fut legelőször (kooperatív)
 - SRTF: a legrövidebb hátralevő löketidejű taszk fut legelőször (preemptív SJF)
 - PRI: minden prioritású taszk fut (preemptív)
 - **Az ütemezők mérőszámai**
 - válaszidő** a taszk külső kérésre (pl. kezelői parancsra) adott első válaszáig eltelt idő
 - várakozási idő** a taszk összes nem futó állapotban eltöltött ideje
 - végrehajtási idő** a taszk futó állapotban eltöltött ideje
 - körülfordulási idő** a taszk belépéstől kilépési eltelt teljes idő
 - **Feladatok: ütemezők futtatása és mérőszámaik meghatározása**
 - Egyszerű ütemezők (FCFS, RR, SJF, SRTF, prioritásos)
 - Statikus többszintű ütemező (pl. FCFS+RR, SJF+RR, SRTF+RR) futtatása
 - prioritás (0 v. 1) szerint választ a sorok között
- T1 [0, 0, 6] T2 [0, 0, 5] T3 [0, 2, 6] T4 [0, 2, 2] T5 [0, 4, 8]
T6 [1, 1, 3] T7 [1, 3, 4] T8 [1, 4, 2] T9 [1, 5, 7]

Gyakorlópélda

- Az ütemező
 - 0 és 9 közötti **statikus** prioritású (0 a legmagasabb) taszkok ütemezése:
 - 1. szint (prioritás < 5) nem preemptív SJF ütemező
 - 2. szint (prioritás > 4) preemptív, RR ütemező, időszelet: 2
- A feladat: futási sorrend és átlagos várakozási idő számítása
- A megoldás:



- Átlagos várakozási idő: $(4+0+10+10) / 4 = 6$

Többprocesszoros ütemezés (haladó)

Az ütemezés megoldott kérdés ???

- 40-50 éve fejlesztik, bevált technikák és implementációk
- Pl. a Linux CFS 13+ éves

WHERE DO WE GO FROM HERE?

- Load balancing on a multicore machine usually considered a solved problem
- To recap, on Linux, load balancing works that way:
 - Hierarchical rebalancing uses a metric named *load*,
 - ↑ Fundamental issue here
 - to periodically balance threads between *scheduling domains*.
 - ↑ Fundamental issue here
 - In addition to this, threads balance load by *selecting core where to wake up*.
 - ↑ Fundamental issue here

Wait, does anything work at all? ☺

AMD Ryzen – Windows 7 vs. 10 ütemező

([demo video](#))

A többprocesszoros ütemezés alapkérdései

- Preemptivitás és újrahívhatóság (reentrancy)
- Lokalitás
 - $F \rightarrow FK \rightarrow F$ állapotátmenetek (pl. megszakítás) maradványadatai
 - CPU regiszterek, Lx cache, RAM
 - ne hagyjuk veszni ezeket
 - az ütemező feladata
- Erőforrás-allokáció (resource allocation)
 - garantált erőforrások taszkok számára
- Terheléselosztás (load balancing)
 - egyenletes terhelés
 - homogén végrehajtóegységek
 - képességeknek megfelelő terhelés
 - heterogén rendszerek
- Erősen hardverfüggő feladatok!

Hardver megoldások áttekintése

- Egyprocesszoros, „többszálú” rendszerek (SZGA 20.2)
 - „szál” = taszk
 - többszörözött erőforrások
 - utasításszámláló, tárolóregiszterek stb.
 - közös erőforrások
 - TLB, utasítás cache, elágazásbecslő stb.
 - adataikat azonosítókkal szálakhoz kötik
- Párhuzamosított végrehajtás egyprocesszoros rendszerekben
 - finom felbontású (fine-grained) időosztásos
 - minden órajelciklusban más taszkot hajt végre
 - jól kihasználja egy szál végrehajtásának kis szüneteit (pl. adatfüggőség miatt)
 - durva felbontású (coarse-grained) időosztásos
 - csak megakadás esetén vált taszkot (pl. cache hiba)
 - kis időre megakad a végrehajtás (pipeline feltöltés)
 - szimultán többszálú (**SMT**)
 - szuperskalár CPU: több műveleti egység → egyszerre több utasítás
 - a szabad műveleti egységekre más taszkok feladatait helyezi

Multiprocesszoros rendszerek

- Több teljes értékű processzor
- CPU-k közötti kommunikáció
 - közös memórián keresztül (monolitikus rendszer)
 - üzenetküldéssel (elosztott rendszer)
- Memóriaműveletek hatékonysága
 - **UMA** avagy **SMP**: azonos idővel használhatók
 - rosszul skálázódik
 - a cache memória elérésében van különbség
 - pl. többmagos CPU
 - **NUMA** (Non-Uniform Memory Access): nem mindegy, hova írunk
 - a memória egy részéhez direkt kapcsolat van
 - más részek **kommunikációs hálózaton** át érhetők el
 - nagyon fontos a lokalitás
 - pl. több-tokos (multi-socket) és több CPU-kártyás rendszerek

algorithm is distributed. Each thread monitors its own CPU usage and recomputes it when it awakens after blocking. The clock interrupt handler adjusts the usage factor of the current thread. To avoid starving low-priority threads that remain on queues without getting a chance to recompute their priorities, an internal kernel thread runs every two seconds, recomputing the priorities of all runnable threads.

The scheduled thread runs for a fixed quantum. At the end of the quantum, the thread is preempted by another thread of equal or higher priority. If the current thread has a lower priority than that of other runnable threads before the quantum ends, the scheduler does not cause context switches. This feature is called *quantum stealing* and is related to usage balancing. The current thread remains on the queue until it becomes runnable again, even though its quantum has not expired.

Mach provides a feature called *handoff*, which allows a thread to switch from one processor to another thread without searching for a free processor. The Mach interprocess communication (IPC) subsystem uses this technique for message passing. When a thread sends a message, the sending thread directly yields the processor to the receiving thread. This improves the performance of the IPC calls.

5.7.1 Multiprocessor Support

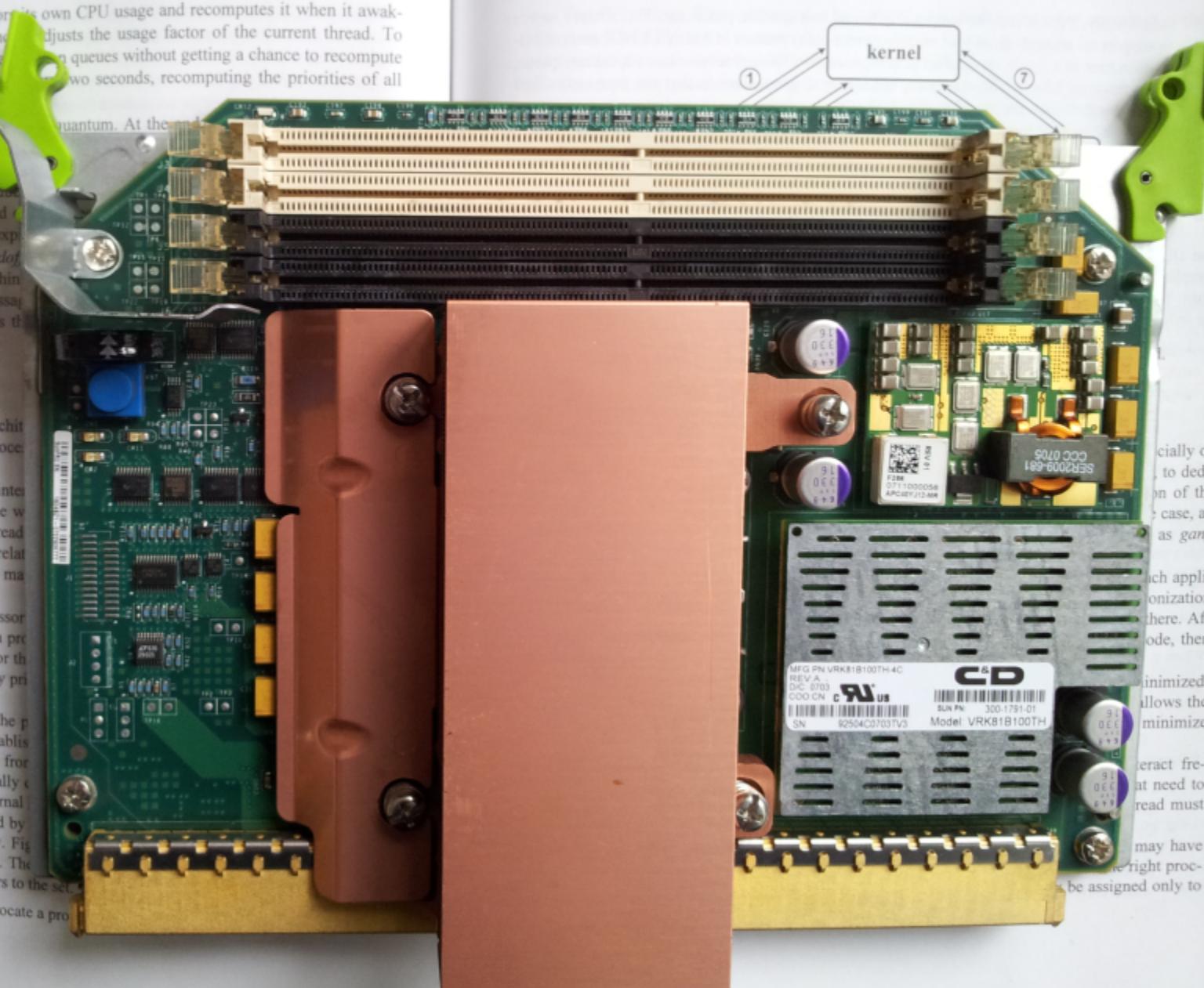
Mach supports a wide range of hardware architectures, ranging from single-chip machines comprising over a hundred processor cores to multiprocessor systems with client management of processors.

Mach does not use cross-processor interprocessor connections. Instead, a thread becomes runnable on one processor and then moves to another processor. This results in a thread becoming runnable while running on a different processor. The latter thread receives a clock interrupt or another scheduler-related event. The scheduler then decides whether to degrade time-sharing behavior, or to move the thread to a real-time application.

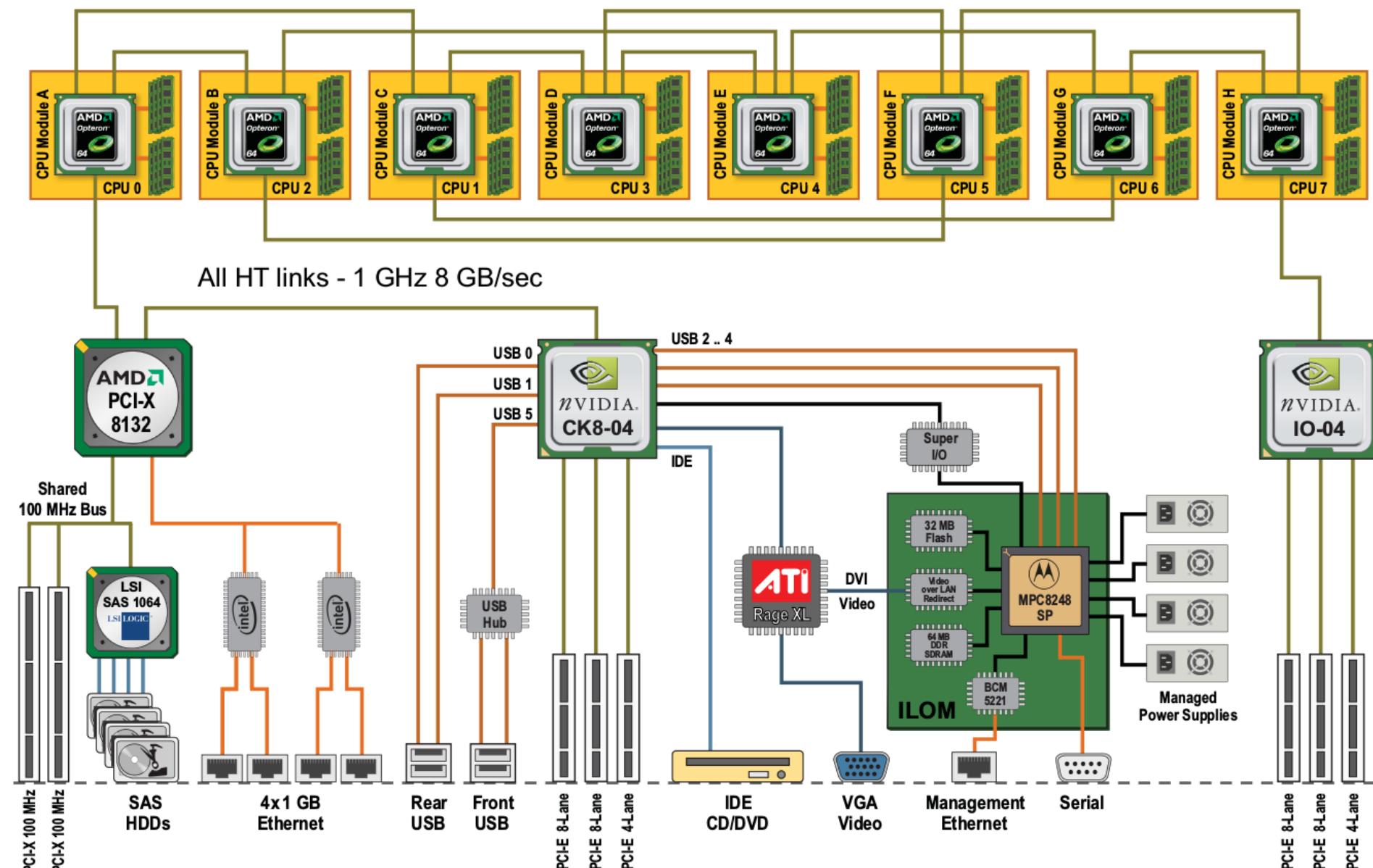
Mach allows users to control processor sets. A processor set is a collection of processors that may contain zero or more processors. Each processor can be moved from one set to another. Each task or thread can be assigned to a processor set. The assignment may be changed at any time. Only processes and threads belong to processor sets.

A thread may run only on one of the processors assigned to it. The assignment of the task to a processor set is established when the task is created. Tasks inherit the assignment from their parent task. The default processor set initially contains all processors. It must contain at least one processor, because internal tasks must run on a processor.

Processor allocation can be handled by a task that determines the allocation policy. Fig. 5.7 shows the application, the server, and the kernel. The server allocates processors to the application. The application asks the kernel to allocate a processor to it.



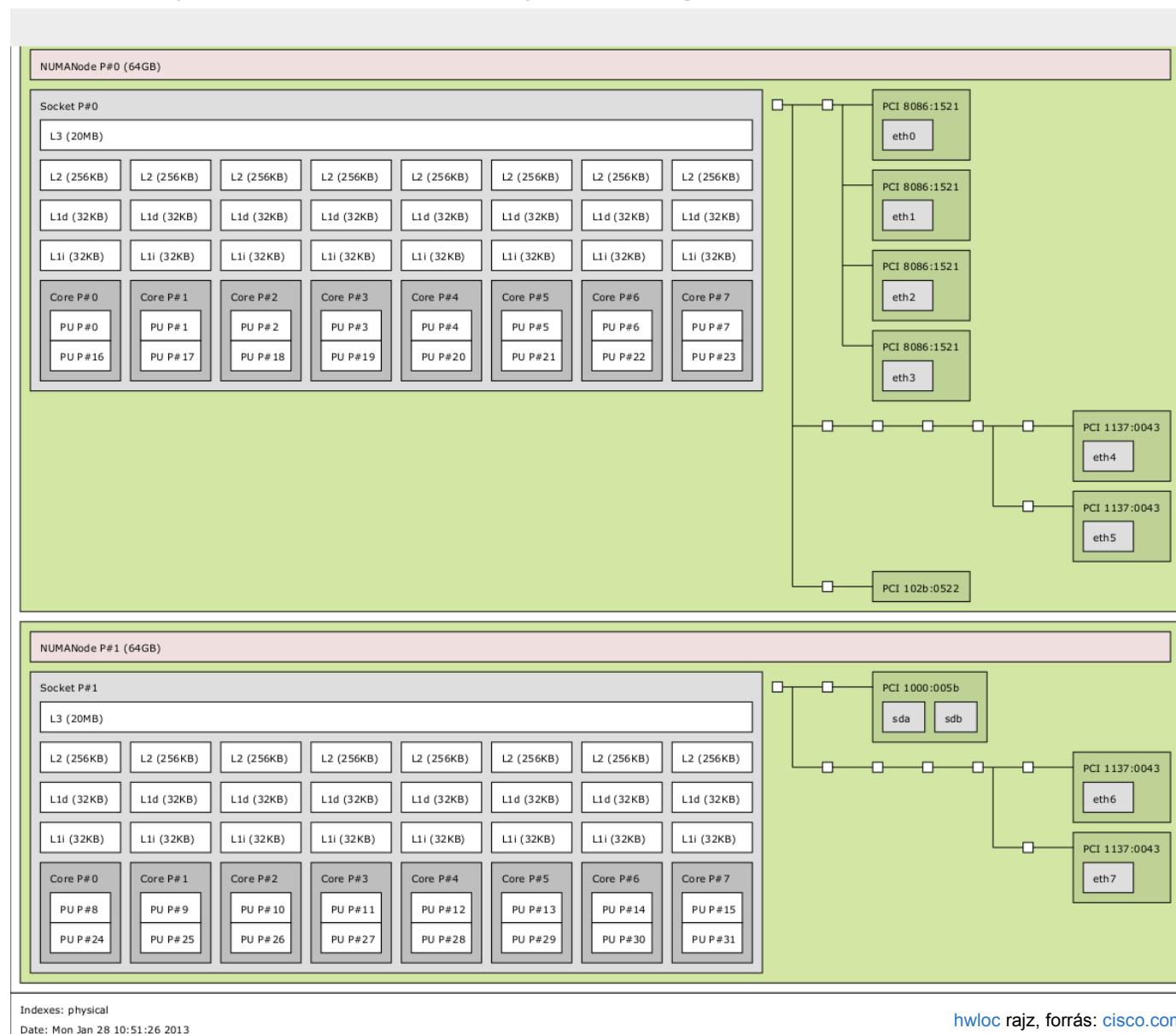
Sun Fire X4600M2 – AMD Opteron



Forrás: X4600 M2 Server Architecture. 2008.

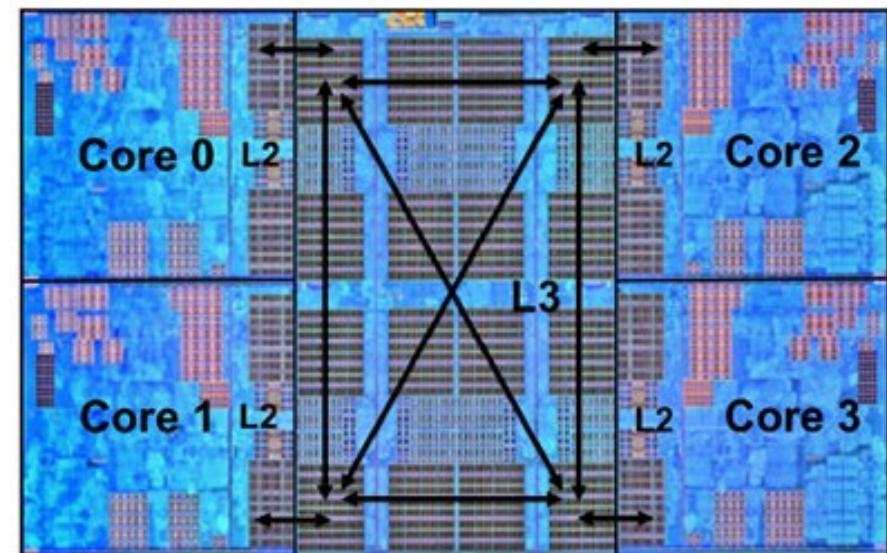
Esettanulmány: Intel “Sandy Bridge” szerver

- Magonként
 - L1
 - L2
- Tokonként
 - L3 (20 MB !)
 - RAM
 - hálózat
- Fájdalmas taszkmigrálás
 - Lx invalid
 - külső háló
 - NUMA háló
- Emiatt csökkenő teljesítmény



Esettanulmány: AMD Ryzen

- Egyetlen tokban 8 CPU mag
 - 2 „Core Complex” (CCX)
 - Infinity Fabric összekötőelem
 - L3 tekintetében ~NUMA
- CCX
 - 4 CPU mag (SMP)
 - 8MB L3 cache

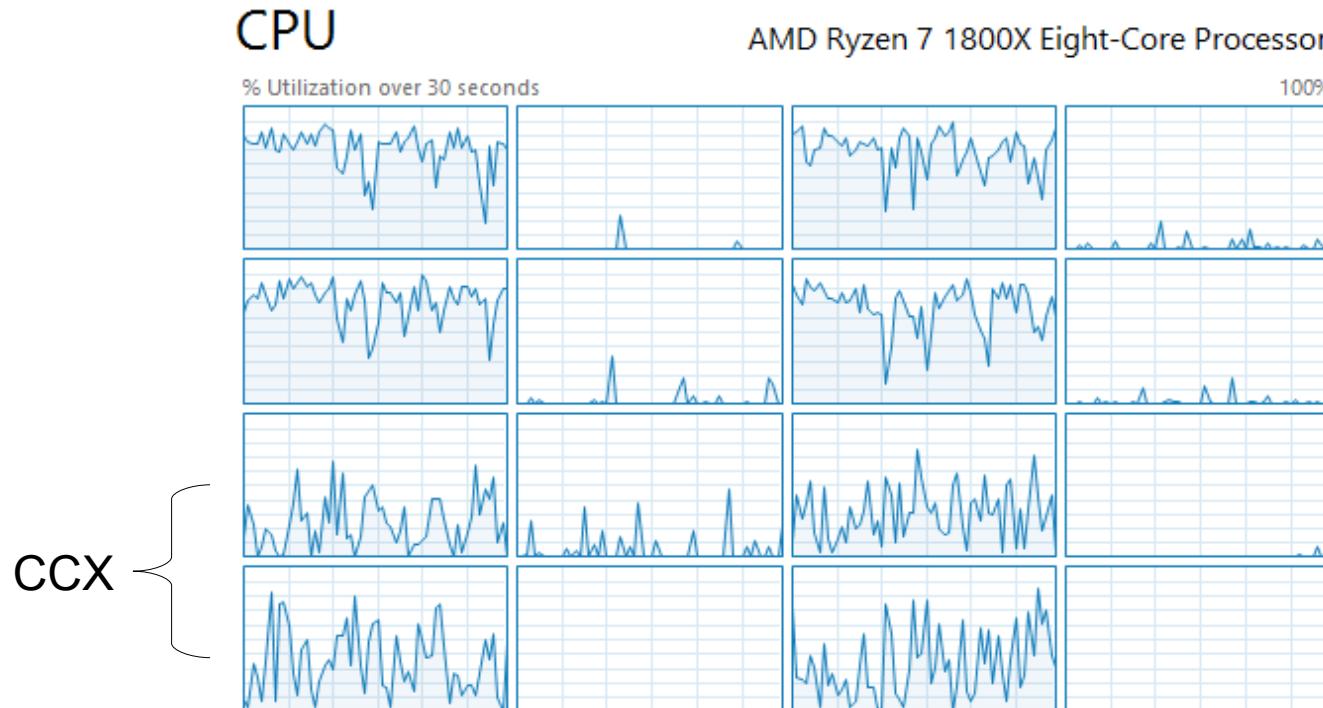


- CPU mag
 - 2 szál (SMT)
 - 512K L2 cache, 64K+32K L1 cache



Forrás: AMD

Hogyan viselkedik a Windows 10 ütemező?

forrás: pcper.com

SMT off körzegestehetőségi

Lokalitástudatos ütemezés

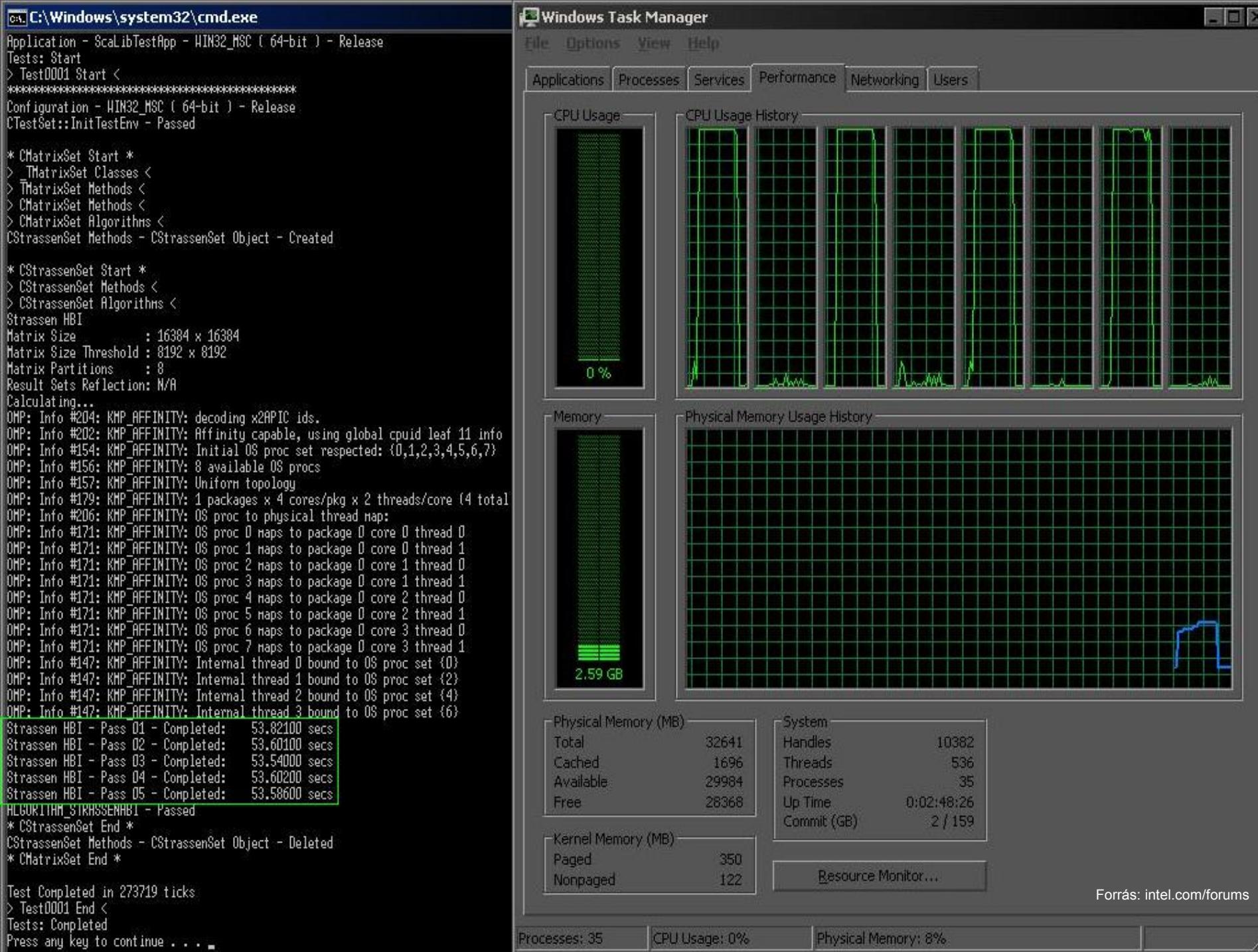
- **Lokalitástudatos programozás (SZGA 11.)**
 - a kernelfejlesztők számára is fontos ismeret
 - kiegészül a többszálú és többprocesszoros lokalitás kérdéseivel
- A taszkváltás elég nagy kárt tud okozni
 - kontextusváltást és lokalitás
- Ne rombolja le a processzor hatékonyságánövelő megoldásait
 - SMP/SMT esetén figyeljen a cache memóriákra
 - NUMA esetén az allokált memória helye is fontos
- Egyszerű ötlet: minden ugyanoda ütemezze a taszkokat
 - sok taszk? nagy/változó terhelés?
 - minden taszknál egyformán fontos?
- A feladatok jellege sem mindegy
 - lásd pl. HPC taszkok

Processzoraffinitás

a taszk és az azt végrehajtó processzor „kötődése”

- **laza affinitás** (soft affinity)
 - megróbálja, de nem garantált
 - jellemző a mai OS-ekben
- **kemény affinitás** (hard affinity)
 - garantált taszk – CPU párosítás
 - rendszerhívással és a felhasználói felületen állítható be
 - **processzorhalmaz affinitás**
 - taszk – CPU-halmaz párosítás

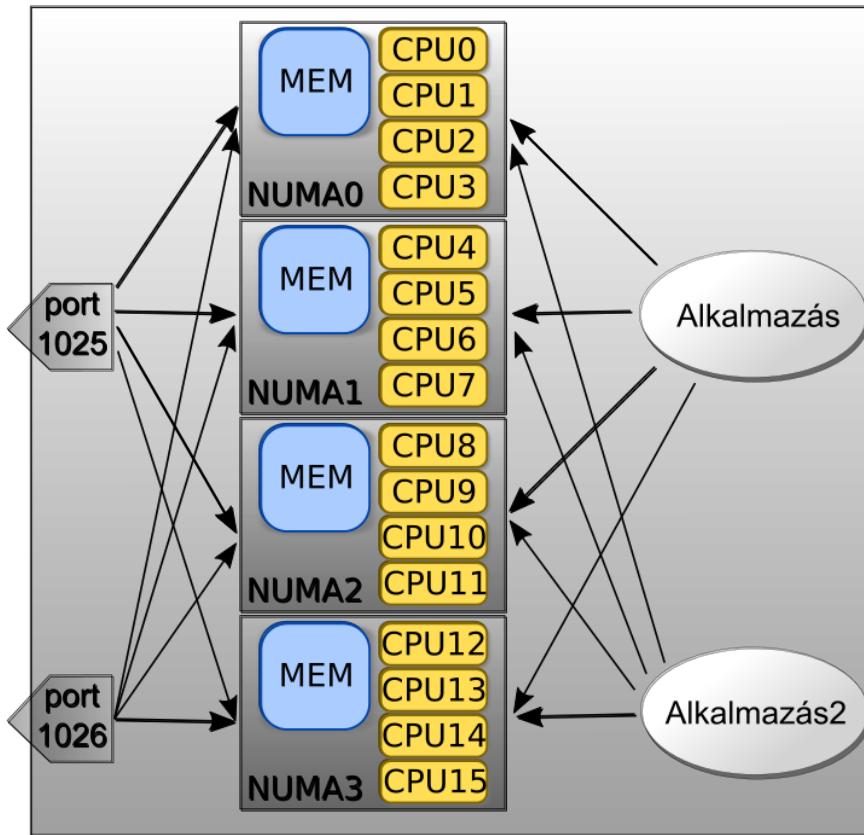
A programozó sokszor több információ birtokában jobban dönthet, mint a kernel.



Programozói szemmel...

- A kernel ütemezője általános szempontokkal dolgozik
 - egyenletes terhelés vagy energiahatékonyság
 - nem ismeri a feladatok jellegét
- A programozó jobban ismeri...
 - a feladat és a taszkok jellemzőit
 - a párhuzamosan futó taszkok viszonyát
- A feladatnak megfelelő affinitás beállítása
 - parancssor
 - taskset -c 0,2,7-11 világgyenlet-számító
 - grafikus felület
 - Feladatkezelő
 - programozói felület:
 - sched_getaffinity(...)
 - GetProcessAffinityMask(...)
 - GetThreadAffinityMask(...)
 - sched_setaffinity(...)
 - SetProcessAffinityMask(...)
 - SetThreadAffinityMask(...)

Példa: hálózati kiszolgálók és CPU-affinitás



Memóriaaffinitás

(a teljesség kedvéért)

- NUMA architektúra
 - CPU – memória elemek között hardver kapcsolat (affinitás) van
- Kernel: transzparens módon kezeli
 - az allokáció a végrehajtó CPU memóriatartományában történik
 - mikor?
 - igényléskor?
 - hozzáféréskor?
- Programozó
 - a CPU-affinitással együtt állítható

```
numactl --cpunodebind=0 --membind=0 világegyenlet-számító
```

Terheléselosztás

- Feladatok (taszkok) elosztása a végrehajtók között
- Milyen feladatokról beszélünk?
 - OS-feladatok
 - pl. az ütemezés
 - felhasználói feladatok
 - előre nem ismertek, de ...
 - lehetnek felhasználói elvárások (garantált erőforrások)
- Milyenek lehetnek a végrehajtóegységek?
 - egyformák
 - a feladatok tetszőlegesen szétoszthatók
 - képességeikben különbözők (heterogén ISA)
 - a feladatok egy része csak adott egységen oldható meg
 - teljesítményükben / fogyasztásukban különbözők
 - a taszkok elvileg bárhol végrehajthatók, de lehetnek preferenciák
 - a felhasználó feladatokra vonatkozó és globális elvárásai
 - a rendszer aktuális állapota

Menedzsmentfeladatok elosztása

- **Aszimmetrikus rendszerek**

- egy egység kapja az összes kernelfeladatot
- a többi egységen fut minden más (felhasználói) feladat
- egyprocesszoros rendszerből kiindulva egyszerűen megvalósítható
- egyszerűen programozható a kernel
- a kernel végrehajtója kihasználatlan lesz

Ritkán használt, de heterogén CPU-architektúrákon ésszerű megoldás lehet.

- **Szimmetrikus rendszerek**

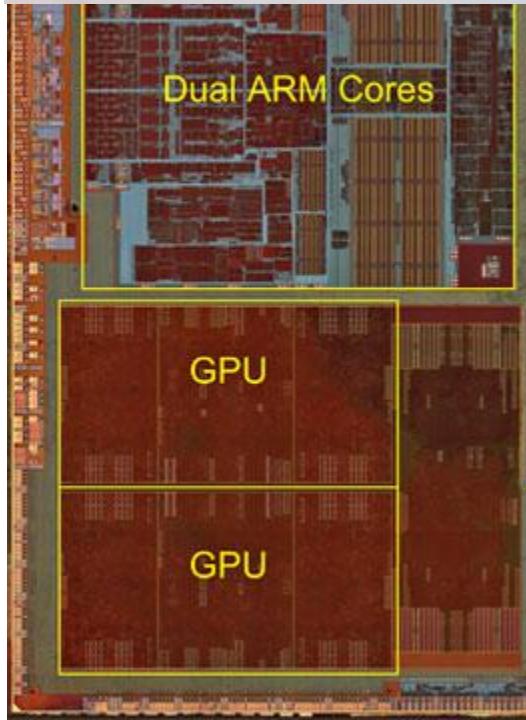
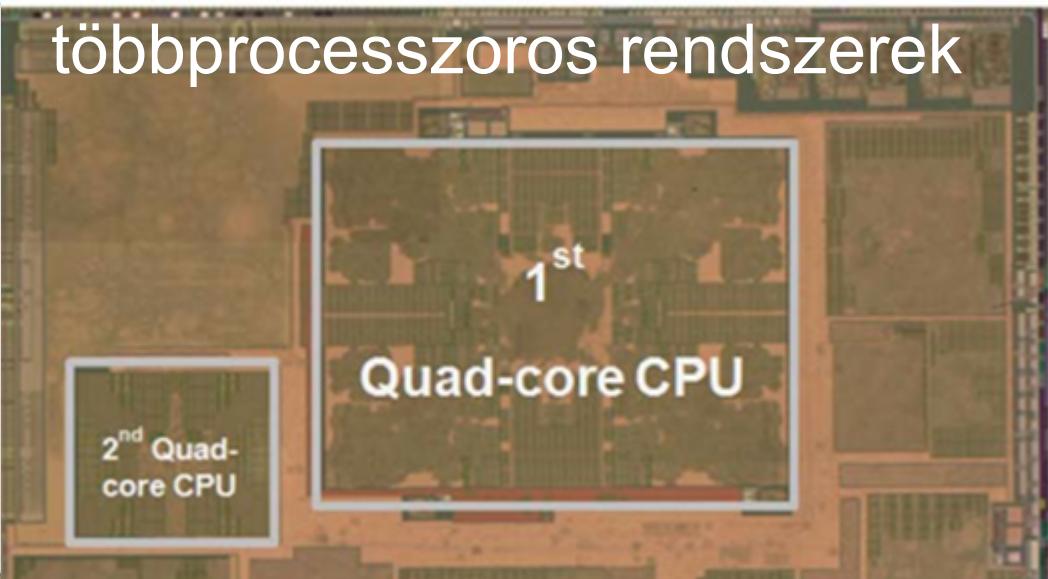
- minden végrehajtóegység ellátja saját ütemezését
- a futásra kész taszkok lehetnek
 - egy közös sorban
 - egységenként külön sorokban
- jobb CPU kihasználtság
- gondosabb programozást igényel

A mai operációs rendszerek ezt a megoldást használják.

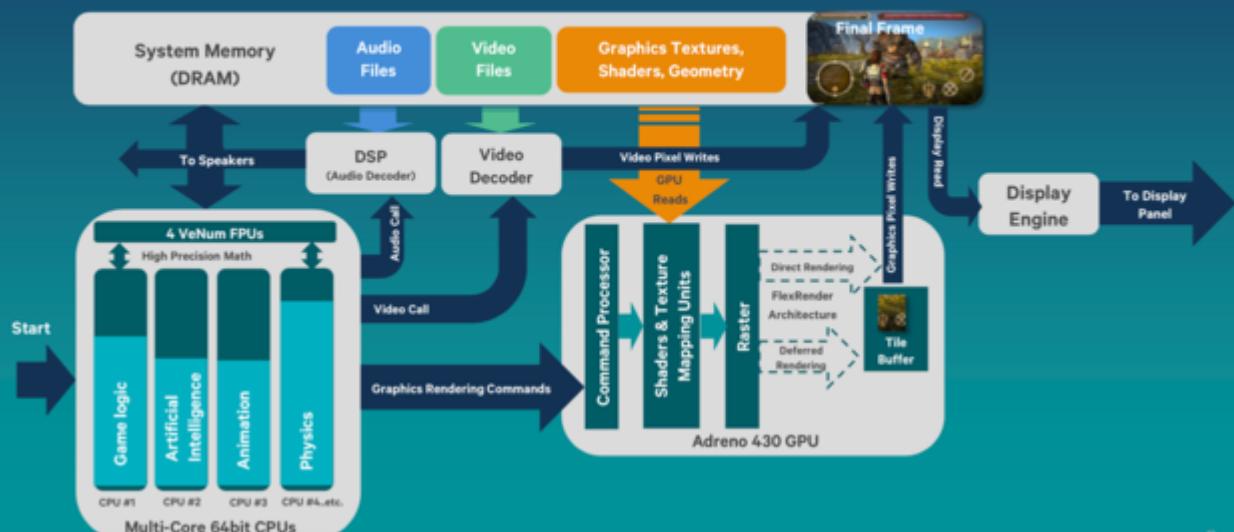
Futásra kész taszkok nyilvántartása

- Globális FK adatstruktúra
 - egyszerű, hiszen globálisan dönthetünk
 - nehéz kezelní a processzoraffinitást
- Lokális ütemezési sorok (ez a jellemző)
 - lokálisan egyszerű
 - a processzoraffinitás működik
 - a terhelés aszimmetrikussá válhat
 - mozgatni kell a taszkokat a különböző FK halmazok között
 - mérlegelni kell az affinitás sérüléséből fakadó károkat
 - Ki mozgatja a taszkokat?
 - push: egy kernel taszk
 - pull: az „unatkozó” processzor lokális ütemezője
- Összetartozó / együttműködő taszkok
 - gang scheduler: taszkok egy csoportja végrehajtó egységek egy csoportjára kerül
 - virtualizációs rendszereknél is érdekes kérdés, pl. VMware ESXi co-scheduling

Heterogén többprocesszoros rendszerek

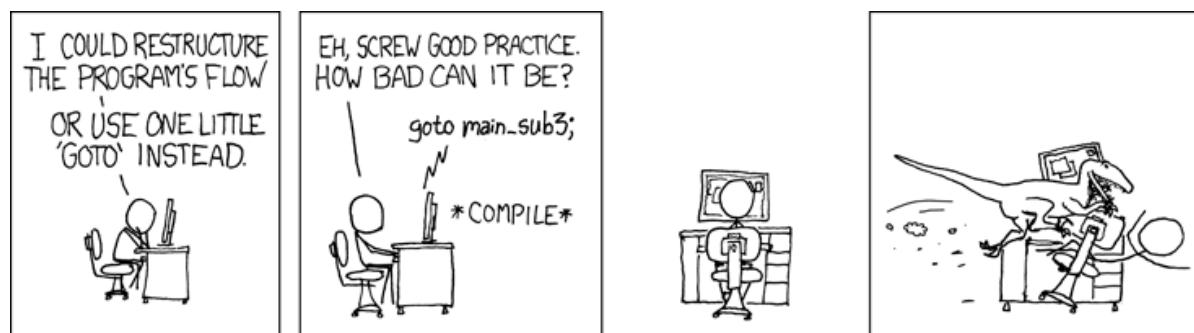


Advantages of heterogeneous architecture for gaming use cases
Heterogeneous hardware blocks and data flow



Heterogén többprocesszoros rendszerek

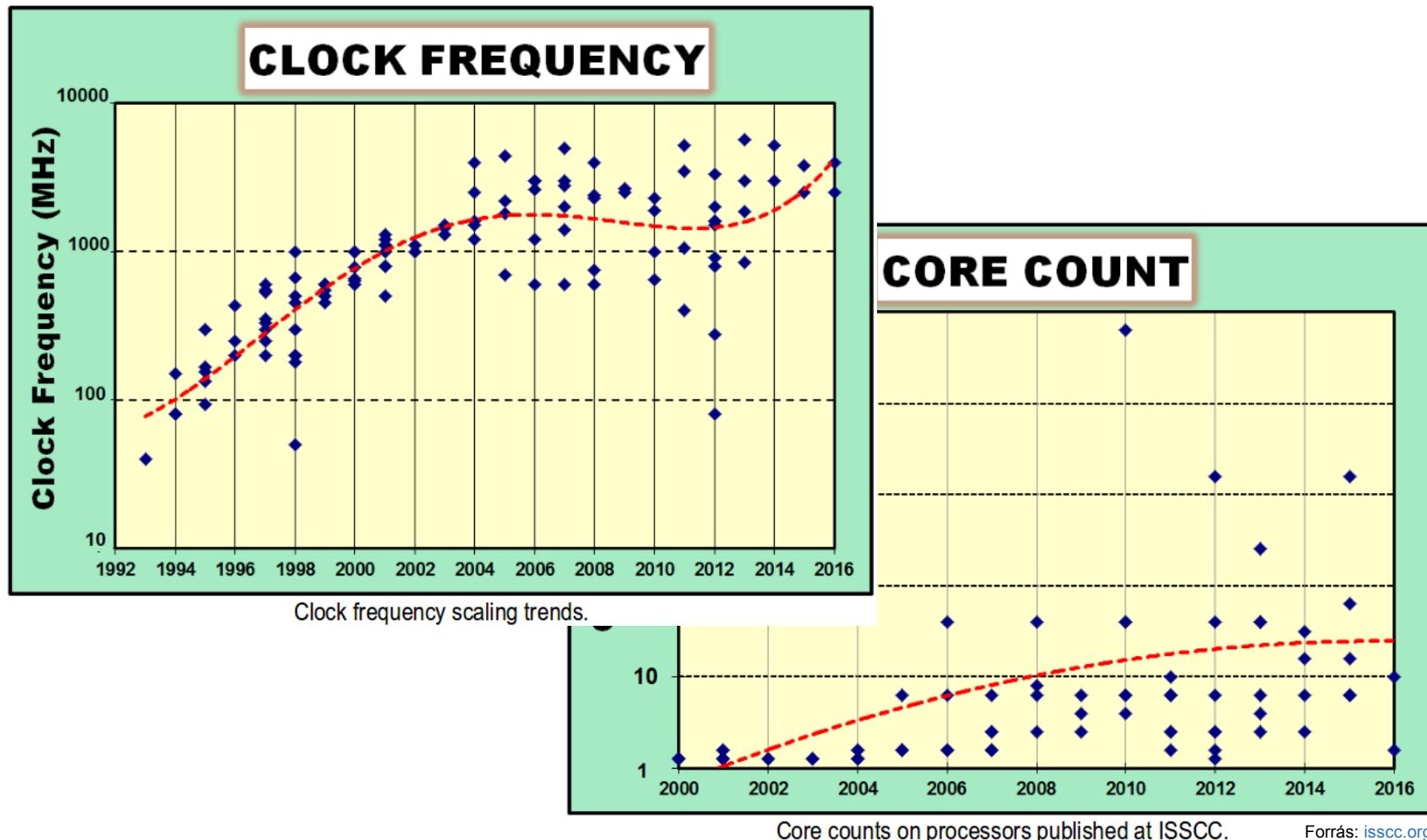
- Mi ez?
 - többféle végrehajtóegység egy rendszerbe integrálva
- Miért? Hol?
 - a feladatok is heterogének (algoritmikusan, adataikban stb.)
 - energiahatékonyság (telefon / tablet)
 - miniatürizálás (többféle funkció integrálása egy csipben)
 - nagy számításigényű (HPC) alkalmazások hardvergyorsítói
 - játékkonzolok CPU + GPU rendszerei
- Miért nem?



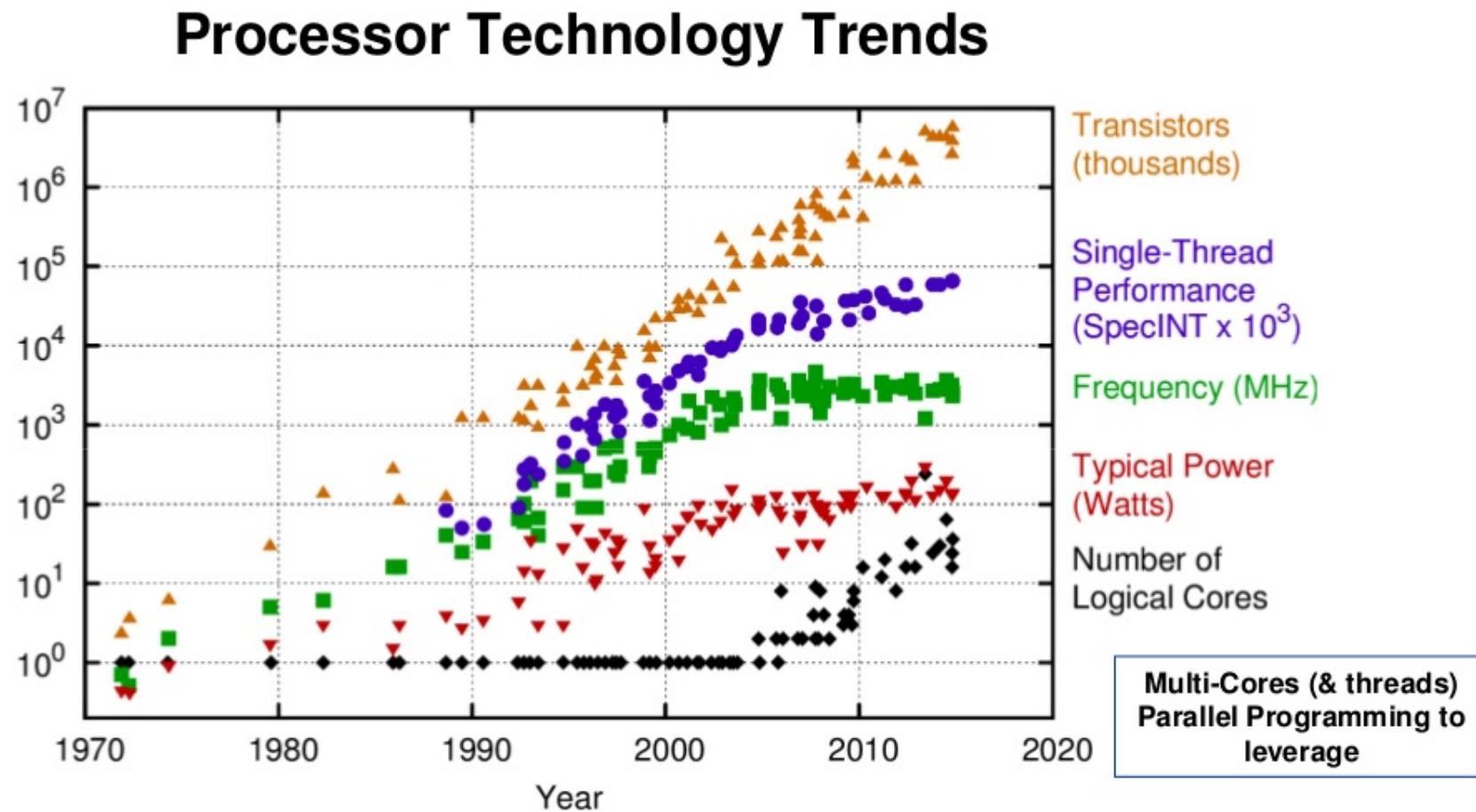
Forrás: [xkcd](#)

Miért fontosak a heterogén rendszerek?

- Meddig nő a teljesítmény és csökken az energiafelhasználás?

Forrás: isscc.org

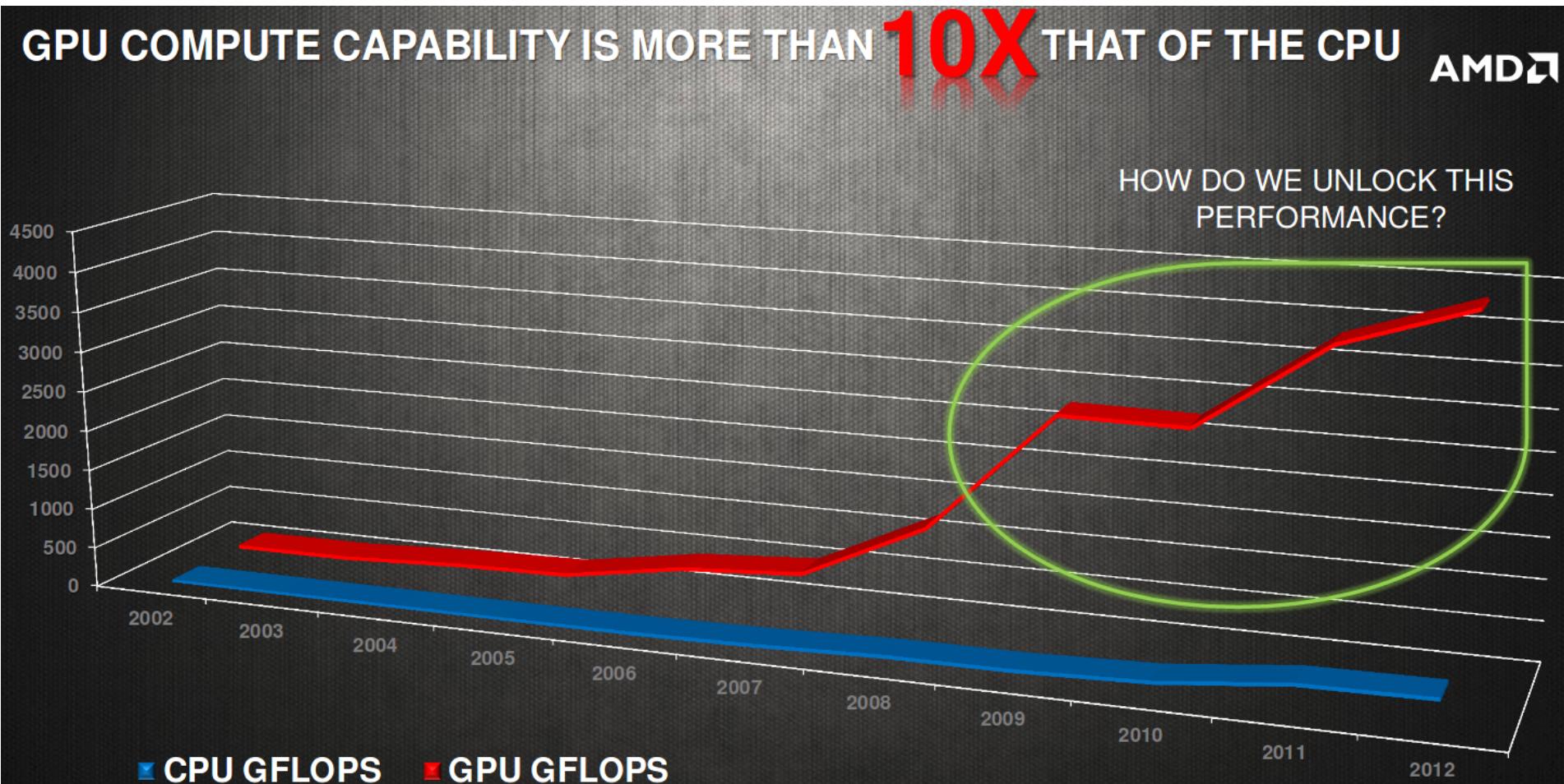
Miért fontosak a heterogén rendszerek?



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Miért fontosak a heterogén rendszerek?

- Kihasználható a CPU-n kívüli teljesítmény?

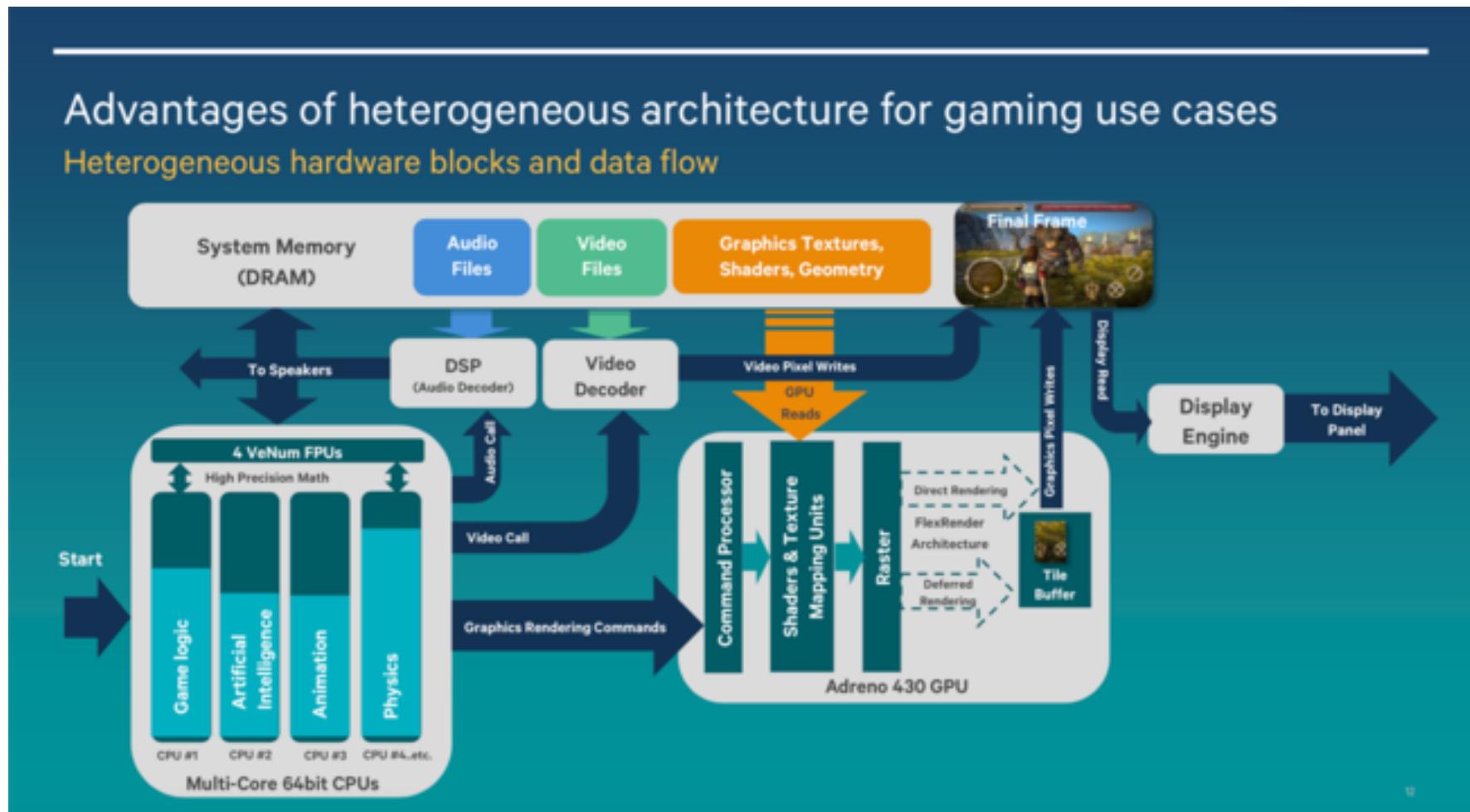


Forrás: [AMD](#)

Miért fontos a programozó szemszögéből?

- Teljesítménynövelés (homogén ISA)
 - szekvenciális → aszinkron, több taszkból álló konkurens rendszerek
 - egyre több végrehajtó egység
- Feladat – taszk – végrehajtó egység összerendelés
 - heterogén ISA: nagyon eltérő hatékonyság érhető el
 - vannak nehéz (NP) és adat-intenzív feladatok
 - egyes algoritmusokra remek hardvereket terveztek (lásd pl. [AI-gyorsítók](#))
- Heterogén rendszerek
 - sokféle képességű végrehajtó és API közül választhatunk
 - OpenMP
 - OpenCL
 - CUDA
 - OpenACC
 - PYNQ
 - ...

Feladatok – taszkok – végrehajtó egységek



Példák: ARM big.LITTLE

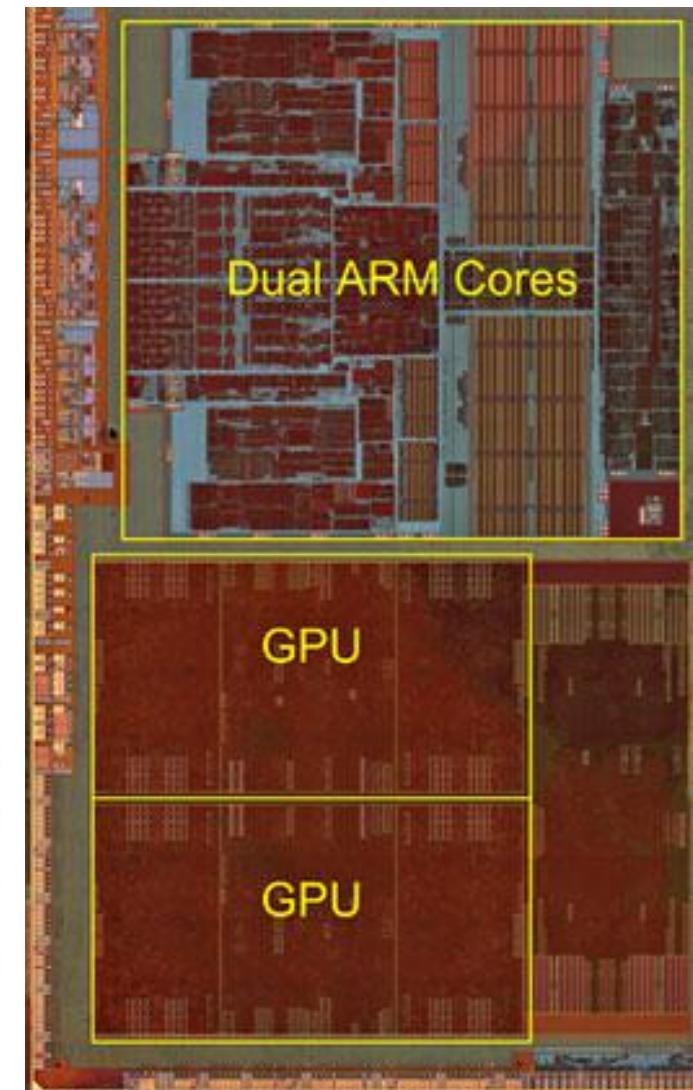
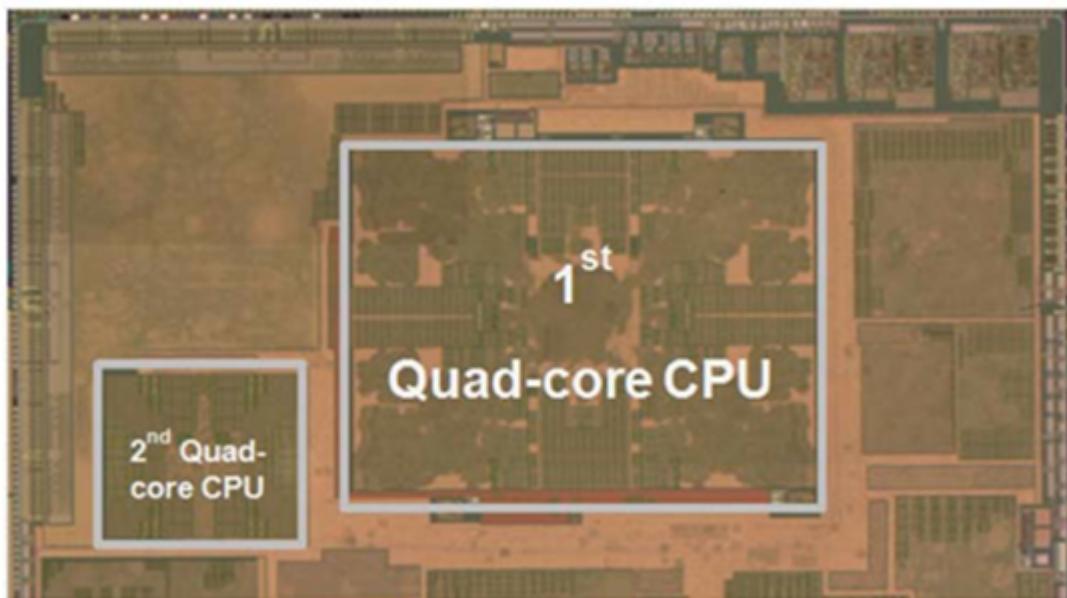
- Egy erős + egy alacsony fogyasztású CPU

pl.: Snapdragon 810, Nvidia Tegra X1,
MediaTek MT6595, Exynos 5410 stb.

„hatékony energiafelhasználás”

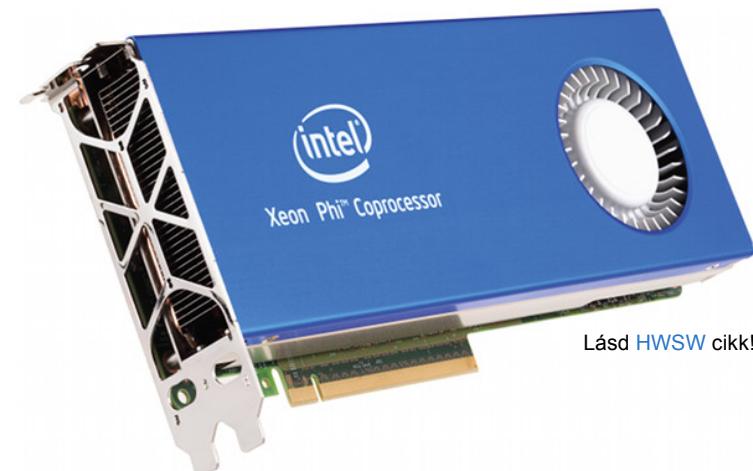
Samsung Exynos 5 vs. 4:

70%-al erősebb (big)
20%-al kevesebbet fogyaszt (LITTLE)



További példák heterogén rendszerekre

- AMD HSA + hUMA
 - HSA: Heterogeneous System Architecture
 - hUMA: heterogeneous Uniform Memory Access
 - integrált CPU+GPU közösen használt memóriával
 - pl.: PlayStation 4, AMD „Kaveri” architektúra (A10-7850K)
- Intel HD graphics
 - a Broadwell már lép a koherens memória-használat felé
- CPU + hardvergyorsító
 - Intel MIC (Many Integrated Core) architektúra
 - pl.: Xeon Phi (60+ CPU mag) →
 - FPGA-alapú
 - pl.: IBM CAPI koherens memóriakezelésű
 - IBM Power8 CPU + Altera FPGA
 - (2015 végén megvette az Intel)
 - egyedi feladatra tervezett rendszerek
 - pl. Google Tensor Processing Unit (mély tanulás)
 - ide sorolhatók a GPGPU rendszerek is jól párhuzamosítható feladatokra



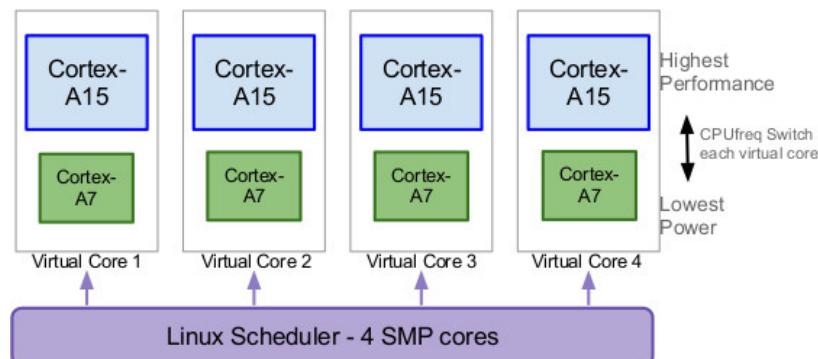
Lásd [HWSW](#) cikk!

Milyen egy mai számítógép?

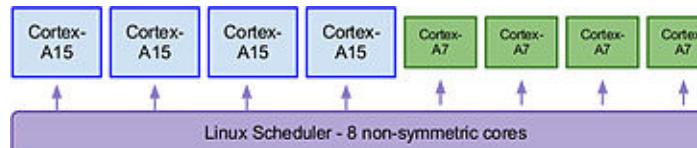
- A mai számítógepek ~~homogén architektúrájúak.~~
- Az elterjedt processzorok önmagukban heterogén rendszerek
 - más- és más órajellel működnek a magok (energiahatékonyság)
 - integrált GPU-t tartalmaznak
- A tárolórendszerek is heterogének
 - CPU regiszterek, L_x cache
 - DRAM („hitting the wall”), SRAM
 - HDD, NAND/SSD
 - MRAM, STTRAM, PRAM, FeRAM, ReRAM, stb.
- A hálózat (interconnect) is heterogén
 - ezerféle (elektromos, optikai)

Ki osztja el a feladatokat?

- CPU-alapú
 - automatikusan migrálja a taszkokat a végrehajtó egységek között
 - az OS nem tud semmiről (legfeljebb órajelet skáláz)
- OS kernel (ütemező)
 - ha tudomással bír a végrehajtók különböző képességeiről
 - taszk-migrálás (pl. a big.LITTLE végrehajtók között)



- heterogén többprocesszoros ütemezés



- Programozó
 - a legjobban ismeri a feladat jellegét és a futtatórendszer képességeit

A feladatkezelés kihívásai

- Utasításkészlet (ISA)
 - homogén (Single-ISA)
 - heterogén – nehezebb ügy
 - létezik legalább valamilyen szintű bináris kompatibilitás?
 - teljesen eltérő architektúrák?
 - a kernelnek nincs esélye kezelní
- Memóriakezelés
 - koherens memória
 - AMD hUMA (heterogén UMA)
 - IBM Power 8: a PCIe felületen át oldja meg a koherens memória-kezelést
 - elosztott
 - Intel Xeon Phi: a kártya saját memóriát használ, hálózaton át érhető el
 - GPGPU: a grafikus kártya jellemzően dedikált memóriát használ

Összefoglalás

- Az ütemezés feladata az elkövetkezőkben futó taszkok kiválasztása
 - időskálái: rövid (erről beszélünk részletesen), közép- és hosszú távú
 - alapkérdései:
 - adatstruktúra (hogyan tartja nyilván a taszkokat)
 - felhasznált taszk jellemzők (mi alapján dönt)
 - algoritmus (hogyan választ)
 - komplexitás és rezsiköltség
- Egyszerű ütemezési megoldások
 - FIFO: egyszerű, de nagyon rossz lehet (konvoj hatás)
 - RR: széles körben alkalmazott, jó válaszidővel, mérsékelt rezsiköltséggel
 - SJF és SRTF: a taszk löketideje alapján választ, optimális várakozási idővel
 - PRIO: számmal kifejezett fontosság, külső és belső, kiéheztetés előfordulhat
- Összetett ütemezési megoldások
 - többszintű statikus: rögzített prioritásokkal többféle ütemezési algoritmust ötvöz
 - MFQ: becsült futásidő szerint dinamikusan több szinten osztja el a taszkokat
- Többprocesszoros ütemezés
 - fontos a processzor affinitás kezelése
 - jellemzően szimmetrikus, CPU-nként lokális sorokkal, köztük pull-push átvitellel
 - a heterogén rendszerek már itt vannak – *mit kezdünk velük?*

Az operációs rendszerek belső működése

Memóriakezelés

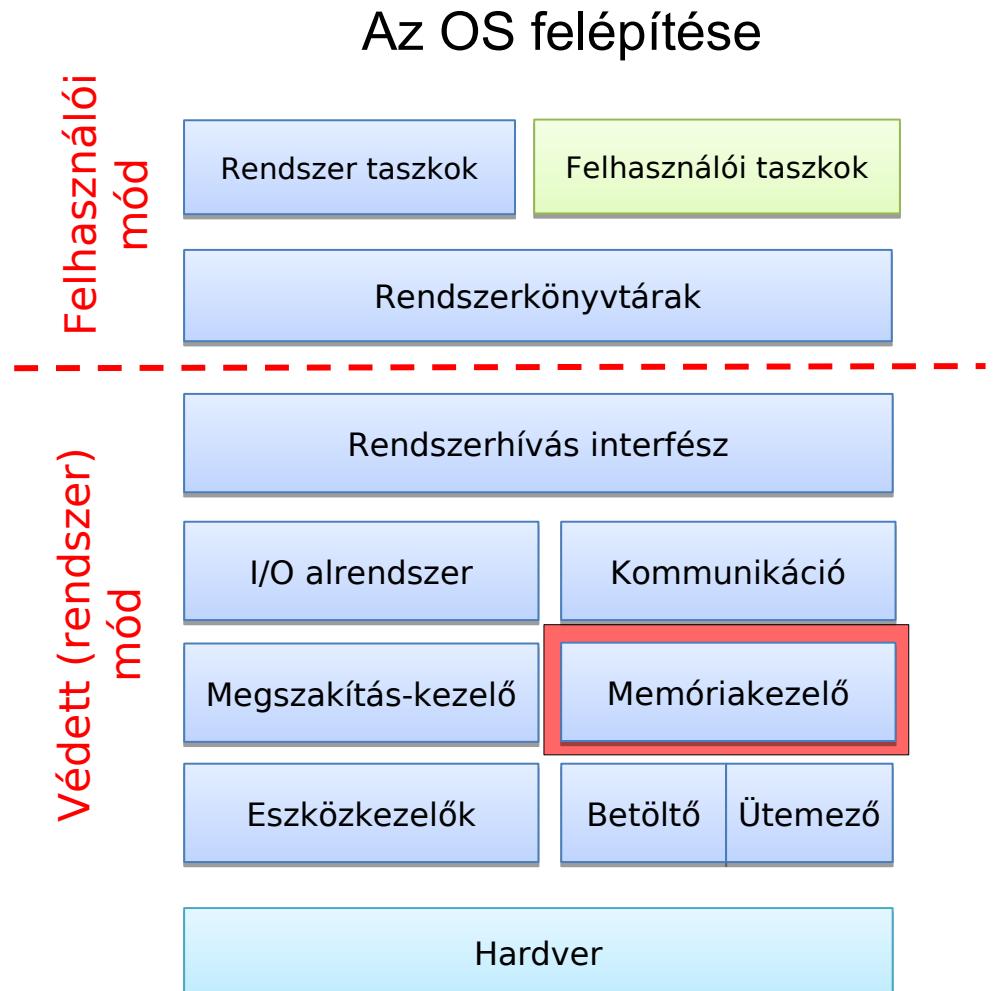
Mészáros Tamás
<http://www.mit.bme.hu/~meszaros/>

Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

Az előadásfóliák legfrissebb változata a tantárgy [honlapján](#) érhető el.
Az előadásanyagok BME-n kívüli felhasználása és más rendszerekben történő közzététele előzetes engedélyhez kötött.

Az eddigiekben történt...

- Az operációs rendszer
 - feladatok végrehajtása
 - vezérlőprogram
 - **erőforrás-alkotátor**
 - Erőforrások
 - absztrakt virtuális gép (CPU, mem)
 - sok taszk osztozik az erőforrásokon
- multiprogramozott rendszer



A taszkok adatai (ismétlés)

- Saját
 - programkód
 - statikusan allokált adatok
 - verem, átmeneti adattár pl. függvényhívások számára
 - halom, a futásidőben, dinamikusan allokált adattár
- Adminisztratív (kernel)
 - taszk- (folyamat-, szál-) leíró
 - egyedi azonosító (PID, TID)
 - állapot (l. később)
 - a taszk kontextusa
 - CPU regiszterek (pl. PC)
 - ütemezési információk
 - **memóriakezelési adatok**
 - tulajdonos és jogosultságok
 - I/O állapotinformációk
 - ...



Az absztrakt virtuális gép koncepció (ismétlés)

- Ideális esetben minden taszk teljesen önállóan fut
- A valóságban osztoznak az erőforrásokon
 - processzor, memória stb.
- Az OS elszeparálja egymástól a taszkokat

absztrakt virtuális gép

virtuális CPU (ezt volt) + **virtuális memória (ez jön)**

A memóriakezelés felhasználói szemmel

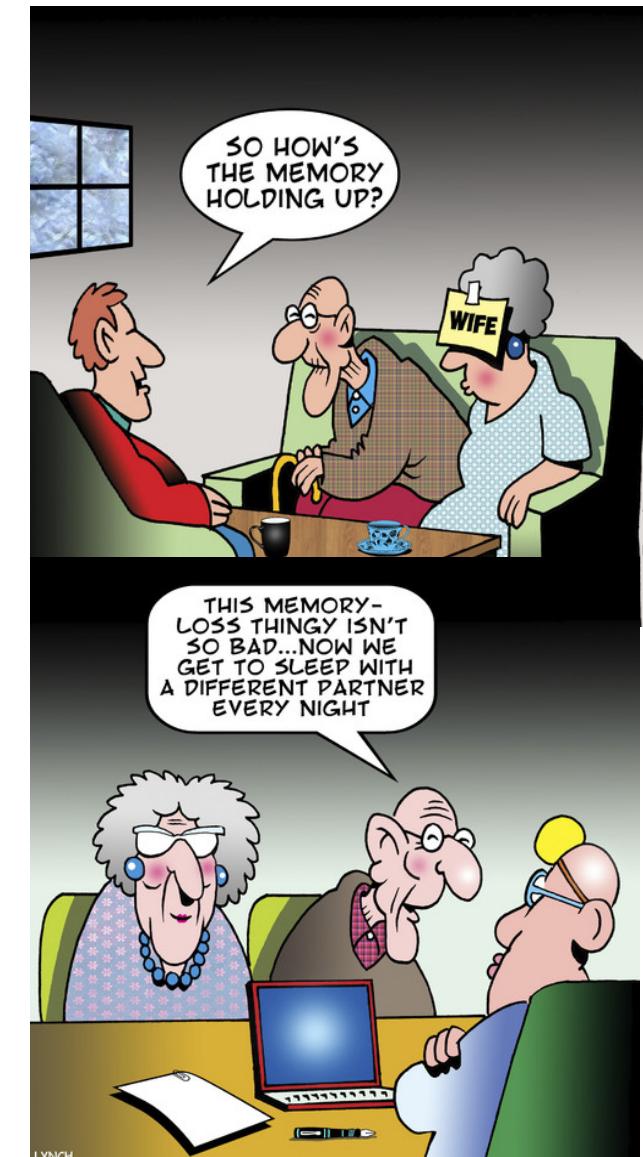
- Felhasználó („end user”)
 - alapvetően nem érdekli
 - van-e elég egy feladat megoldásához (program futtatásához)?
- Adminisztrátor
 - mennyi foglalt, mennyi szabad, hogyan növelhető
 - top, free, mkswap, Erőforrás-figyelő
- Programozó
 - hogyan használható (foglalható, olvasható, írható)
Hozzáférlek a programomból a RAM-hoz?
 - mennyi használható
4GB RAM van a gépen, használhatok 5GB memóriát?
 - milyen részei vannak a memóriának, és azokban mennyi hely foglalható le
Mindegy, hogy lokális vagy globális változókat használok? (demo)
 - milyen hibák fordulhatnak elő a használata során
Mi történik, ha elfogy? ... ha hibás címet használok?

```
char a[30000000];  
int main () {  
}
```

```
int main () {  
    char a[30000000];  
}
```

Mivel foglalkozik a memóriakezelés?

- Kiosztja az erőforrást
 - erőforrás: fizikai memória
 - igénylők: taszkok és kernel
- Elhelyezi a taszkok adatait
 - programkód + statikus adatok
 - dinamikusan allokált
- Elhelyezi a kernel adatait
 - programkód
 - adminisztratív adatok
- Biztosítja a védelmet
 - szeparáció
 - hibák
- Támogatja a kommunikációt
 - adatcsere taszkok között



A memóriakezelés kihívásai

- Nem elég az erőforrás
 - sok taszk → sok memória
 - memória-intenzív taszkok
- Hatékonyság
 - minden CPU művelet érint
- Biztonság
 - sok incidens forrása



Forrás: toonpool

Hogyan valósítsuk meg?

Megfigyelések a taszkok memóriahasználatáról

- Neumann-architektúra (lásd szga)
- Induláskor nincs szükségük a teljes programra és adatkészletre
- Működésük során dinamikusan foglalnak memóriát
 - a rendelkezésre álló fizikai memória méretével nem törődnek
 - az allokált memória „szellőz”
- Megfigyelhetők lokalitási jellemzők (szga)
 - időbeli, térbeli és algoritmikus
- Vannak sosem használt memóriarészeik
 - sokféle lehetséges lefutásból csak egy következik be
 - hibakezelés, ritkán használt funkciók
- Vannak közösen használt memóriaterületek
 - pl. dinamikus rendszerkönyvtárak, folyamatklónozás (`fork()`)

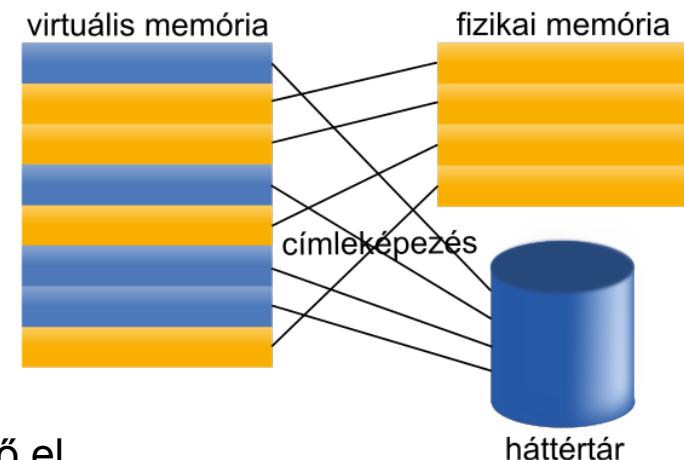
A virtuális tárkezelés

Összefüggő virtuális memóriatartomány a taszkok számára.
Részekre bontjuk, és csak a használatban levő részeit tároljuk.

- A feladat
 - a virtuális és fizikai címek megfeleltetése
→ **címleképezés (address translation)**
 - a taszkok memóriatartományának részekre bontása
→ **lapozás (paging)**
 - a (gyors) fizikai memória kapacitásának kiterjesztése
→ **cserehely (swap)**
- Elvárásaink
 - minél több taszk működjön párhuzamosan
 - feleslegesen ne foglaljon erőforrást
 - a fizikai memóriát meghaladó igények kiszolgálása
 - szeparáció és együttműködés
 - **alacsony rezsiköltség**

Címleképezés és lapszervezés (szga)

- MMU (Memory Management Unit)
 - virtuális és fizikai címek összerendelése
- Virtuális és fizikai címek
 - a taszkok a CPU teljes címtartományát látják
 - ez a **virtuális címtartomány**
pl. x86-64 esetében 2^{48} byte = 256 terabyte
 - a fizikai memória a **fizikai címtartománnyal** érhető el
jellemzően a ... gigabyte tartományban (néhány száz megabyte / gigabyte)
 - ami a fizikai memóriában nem fér el, azt a háttértáron tároljuk
- Lapszervezésű virtuális memória-kezelés
 - virtuális címtartomány ← **lapok** (page)
 - fizikai memória ← **keretek** (frame)
 - háttértár ← **blokkok**
 - **laptábla:** lapok ↔ keretek
 - Translation Lookaside Buffer (TLB): címfordító gyorsítótár
(van szegmens+lapszervezésű is, pl. x86 valós mód)

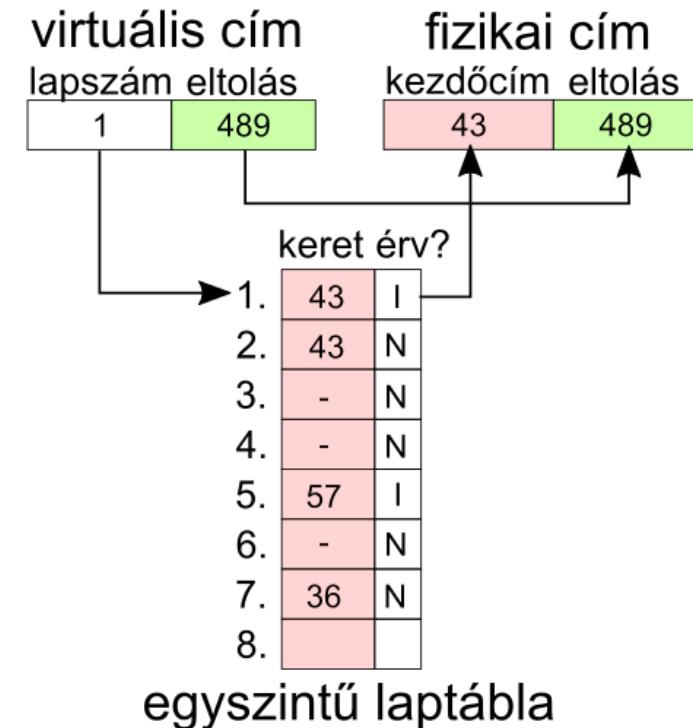


A címléképezés és a laptábla

- A címléképezés lépései
 - a virtuális cím kettébontása
 - lapszám index
 - eltolás
 - index → fizikai keret
 - fizikai cím előállítása
 - fizikai keret kezdőcím
 - eltolás

lásd szga jegyzet és [x86 példa](#)

- Címtér-elkülönítés (szeparáció)
 - taszkonkénti laptábla
a kontextus része
 - futó taszk esetén
az MMU támaszkodik a tartalmára

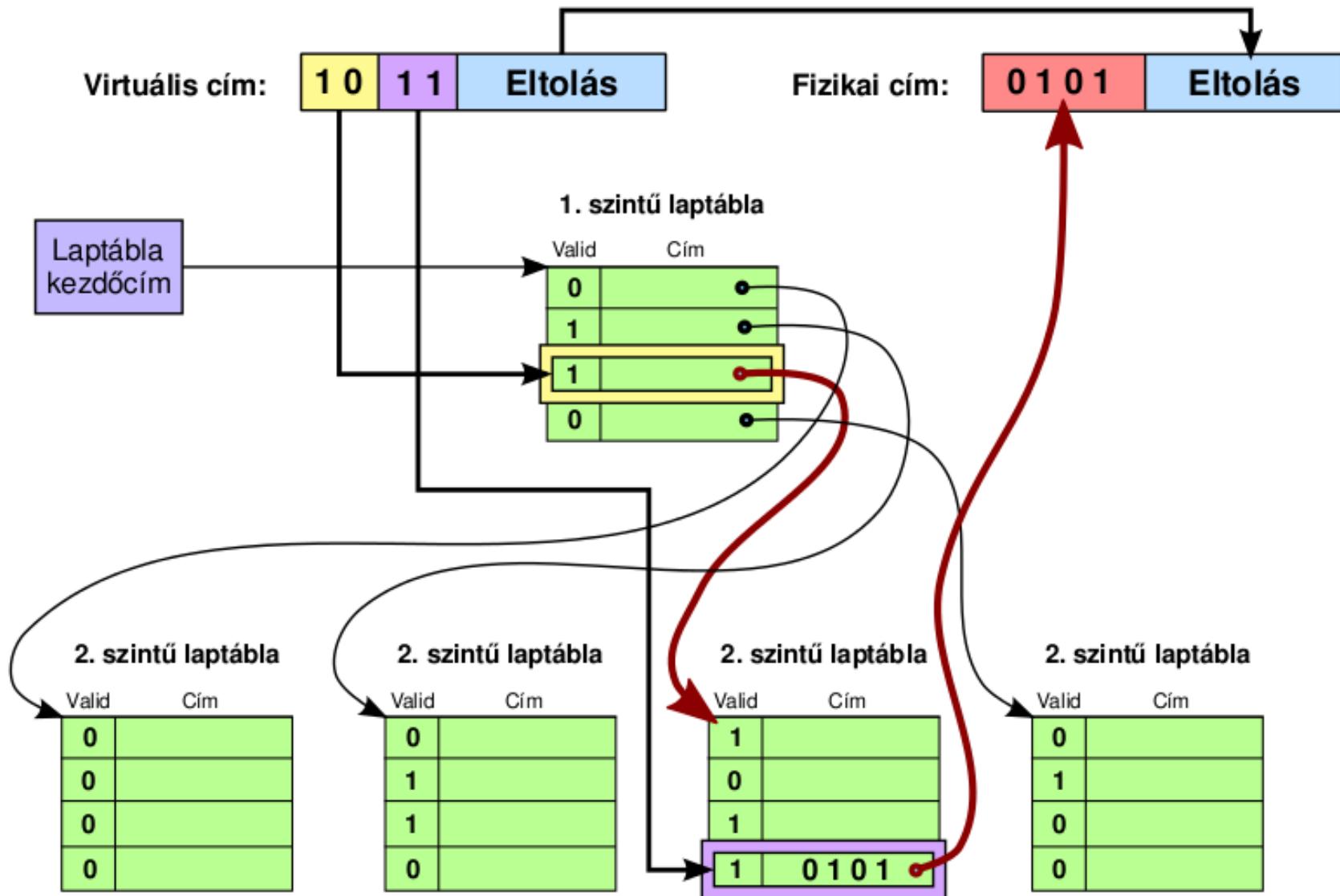


Előfordulhat-e a rendszerben két egyforma virtuális cím különböző tartalommal?

Létezhet-e két egyforma fizikai cím különböző tartalommal?

Mi a helyzet a túl nagy méretű laptáblákkal?

Többszintű laptáblák (szga)



Forrás: szga jegyzet

Cserehely (swap) avagy lapozófájl (page file)

- Ami nem fér a központi memóriába...
vagy felesleges ott tárolni...
- Nagy kapacitású, de LASSÚ tároló
 - részekre (blokkokra) bontott
 - a lapokat a blokkokban helyezzük el
 - **a CPU közvetlen módon nem fér hozzá**
- A cserehely használata
 - amikor a cserehelyen tárolt adatokra van szükség
 - be kell tölteni azokat a fizikai memóriába
 - amikor a fizikai memóriában szabad helyre van szükség
 - gondoskodni kell a memóriában tárolt adatok cserehelyre mentéséről
 - mindenek a memóriakezelés feladatkörébe tartoznak
- Demo: cserehely létrehozása

Tárcsere (swapping)

- Taszkok teljes memóriatartományának háttértárra írása
 - a lapozás előtti időkben fejlesztették ki
- A memória és a cserehely egyre nagyobb **töredezettségét** okozta
 - változó méretű „lyukak” jelennek meg
 - nehéz jól kitölteni a szabad helyeket
- A töredezettség csökkentése
 - kezelés:
taszkok áthelyezése (töredezettség-mentesítés)
 - megelőzés:
ügyesebb elhelyezési algoritmusok
- A lapozás is megoldja a töredezettség problémáját.
- A teljes tárcsere a lapozás mellett is működhet
 - túlterhelés esetén a felfüggesztett taszkok összes lapját kiírhatjuk a cserehelyre

Memóriakezelés programozói szemmel (demó)

- Mit csinál? – C pointerkezelés

```
static int ns = 5;
printf("%d @ %p\n", ns, &ns);
int *pd = (int *) malloc(sizeof(int));
*pd = 10;
printf("%d @ %p\n", *pd, pd);
```

Kimenet:

```
5 @ 0x601048
10 @ 0x1430010
```



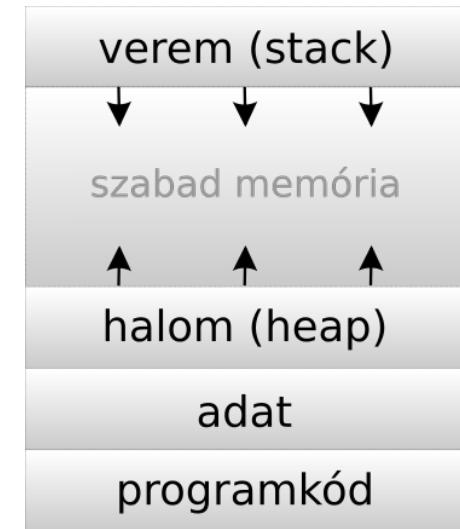
Forrás: [xkcd](https://xkcd.com/117/)

- Figyeljük meg (memprobe.c):

- a memóriaszerkezetet
- a statikus, lokális és globális változók elhelyezését
- a dinamikusan allokációt
- a memória foglalást allokáció előtt és után
- a foglalás változását az adatok módosítása közben

- malloc() bomba

```
int Mb = 0;
while ( malloc(1<<20) ) ++Mb;
```



Hogyan működik a virtuális tárkezelés?

- A taszkok memóriaterületét lapokra bontja
 - a használatban levő lapokat elhelyezi a memóriában és a háttértáron (cserehely)
 - beállítja az MMU-t a kialakított elrendezésnek megfelelően:
 - hardveres címleképezés és védelmi funkciók (taszkok szeparációja)
 - kezeli az MMU által generált megszakításokat
- A taszkok futása alatt
 - a TLB és az MMU végzi a címfordítást
 - a hardver betartatja a védelmi korlátokat
 - az MMU megszakításokat generál, amennyiben hibát észlel
- A hardver által generált megszakítások kezelése
 - **védelmi hiba**
hibás címzés (érvénytelen cím, hozzáférési hiba)
 - **laphiba**
a hivatkozott lap nincs a fizikai memóriában

Emlékeztető: a modern OS eseményvezérelt

Laphiba kezelése: szoftveres címleképezés

- A lap nincs a memóriában → laphiba (megszakítás)
 - a laptábla megfelelő bejegyzése nem érvényes jelzésű (valid bit = 0)
- Elindul a kernel megszakításkezelője
 - észleli a laphibát → aktiválja a memóriakezelőt (lap behozása)
 - létezik a lap a cserehelyen?
 - betölti egy szabad keretbe
 - igény szerint kitöltendő? (fill-on-demand: zero-fill, fill-from-text)
 - kitölt egy szabad keretet
 - a taszk laptáblájában beállítja az új lap-keret összerendelést (valid = 1)
 - frissíti a laptáblát az MMU számára
 - visszatér a megszakításból
 - az CPU újra végrehajtja a műveletet, ezúttal sikeresen
- Gondok?
 - Van szabad keret?
Ha nincs, fel kell szabadítani egyet. → **lapcsere**
 - A lap betöltése a diszkről lassú
Célszerű addig más taszkot futtatni.

A memóriakezelő további feladatai

- Szabad kereteket biztosítása
 - célszerű nem laphiba alatt foglalkozni vele
- Lapok kiírása a háttértárra
 - a nem használt lapokat célszerű a háttértárra írni, és
 - az általuk foglalt kereteket felszabadítani
- Nyilvántartás
 - taszkok lapjai → **laptábla** (page table)
 - keretek → **kerettábla** (page frame data)
 - cserehely → **diszk blokk leíró** (disk block descriptor)
swap térkép (swap map)
- További feladatok
 - az MMU is módosíthatja a laptáblát (pl. hivatkozásszámláló)
 - a kernel kiolvassa és tárolja a módosításokat
 - szükség esetén tárcsere
 - túlterhelés esetén teljes taszkok kiírása

A virtuális memóriakezelés adatstruktúrái

- **kerettábla** (pfd: page frame data) (kernel kontextus)

- a keret sorszámával indexelt
- **állapot** (szabad, foglalt), módosult (dirty), DMA alatt áll stb.
- **hivatkozásszámláló** (acc): hány taszk használja a keretet

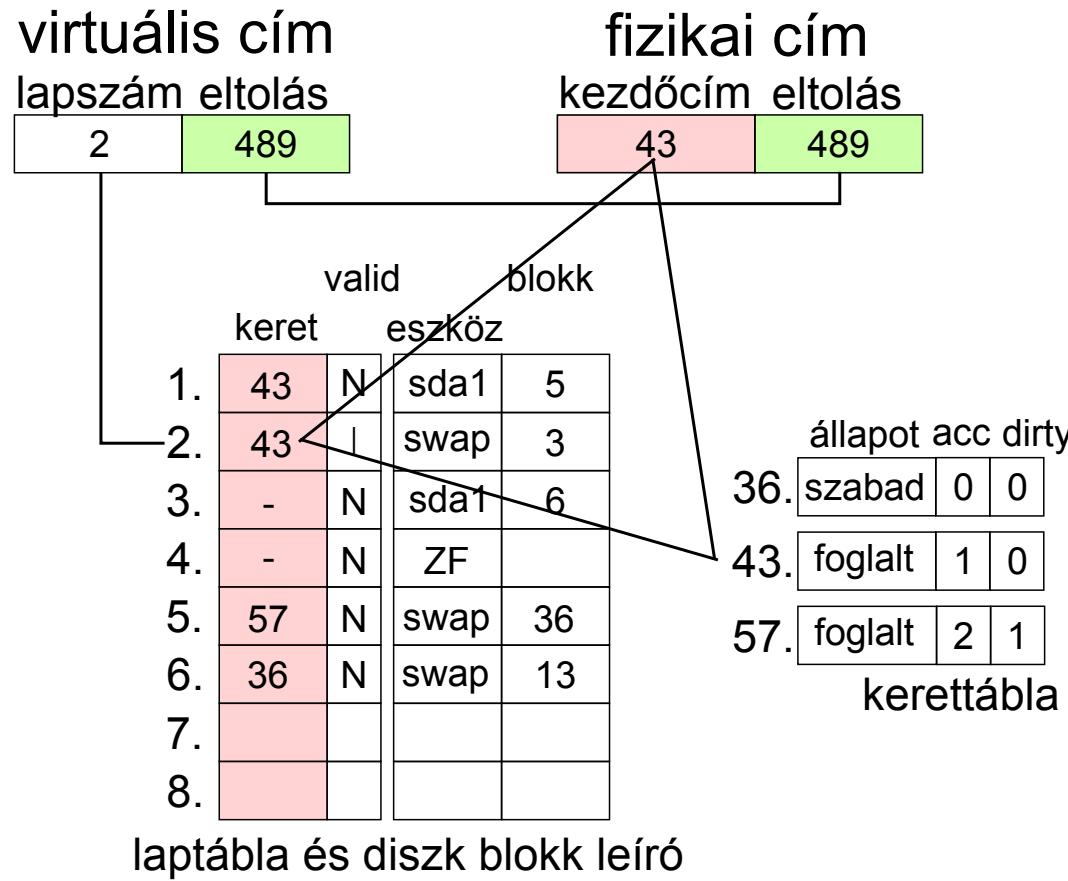
- **laptábla** (page table) (taszk kontextus)

- lap sorszám
 - keret sorszám
 - jelzőbitek:
„valid”, „dirty” (módosult), „accessed” (használt), „read-only”
 - állapot: memoriában / háttértáron / igény szerint kitöltendő
 - a taszk azonosítója
 - másolás-írás-esetén (COW) jelzőbit,
 - jogosultságok stb.
- 

- **diszk blokk leíró** (kernel kontextus)

- háttértár eszközazonosító
- blokk sorszám
- típus: swap (a háttértáron van), zero-fill, fill-from-text stb. (OS-függő)

Példa az adatszerkezetekre



Értelmezzük a 2., 4. és 6. lap adatait!
 Mi történik, amikor a 6. lapra hivatkozik a program?
 Hol található az 57. keret lapja?

Teljesítménynövelő technikák: fill-on-demand

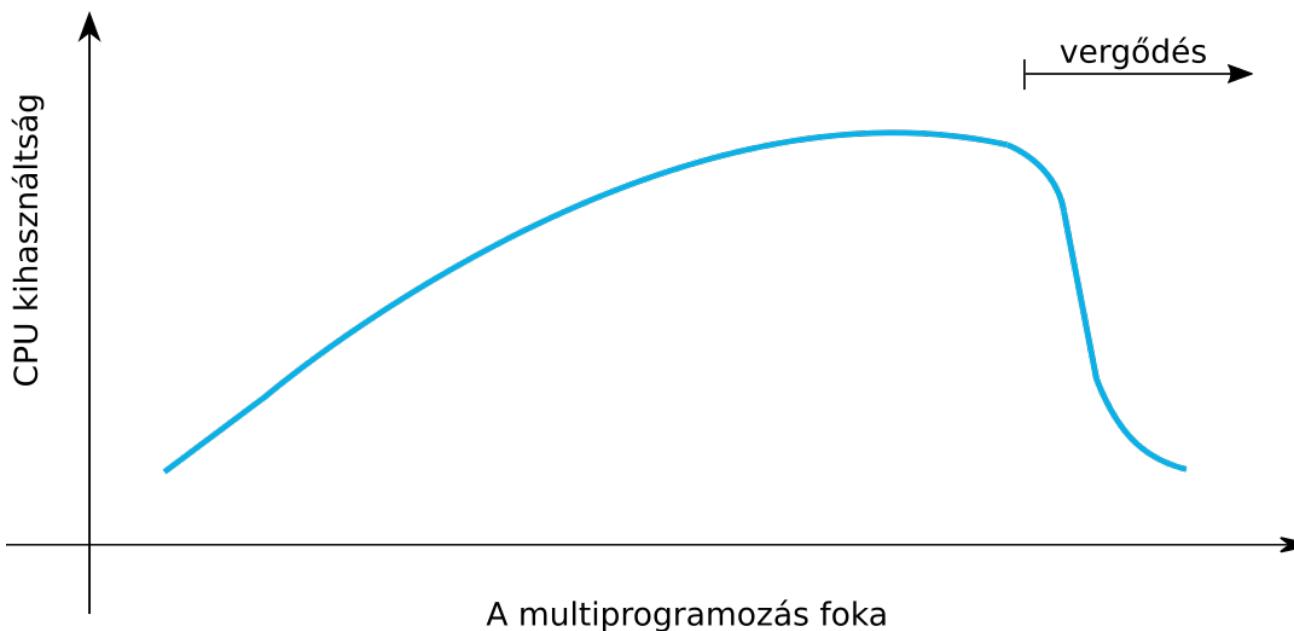
- Mi történik a `malloc()` kiadásakor?
 - a memória tartalma nem definiált, ezért nem foglal neki keretet vagy cserehelyet
 - a laptábla megfelelő elemei fill-on-demand (pl. zero-fill) bejegyzést kap(hat)nak
- Amikor először hivatkoznak az allokált lapra
 - az MMU megszakítást generál
 - elindul a szoftveres címleképezés
 - allokál és kitölt egy keretet ← **csak ekkor történik memóriafoglalás!**
- Eredmény?
 - allokáció során csak egy laptábla bejegyzést kell kitölteni (gyors)
 - csak akkor foglal tényleges memóriát, ha azt használják
 - jól kezeli a lefoglalt memória „szellősséget”
- hasonló módon működik a programkód betöltése is
 - fill-on-demand: fill-from-text
 - a laptáblában a kódot tartalmazó diszkblokkokat állítja be

Teljesítménynövelő technikák: COW

- Emlékeztető: a fork() működése
 - több taszk ugyanazt a programkódot futtatja
- A lapok taszkok közötti megosztása
 - egy keret – több lap (több taszk)
 - olvasás: nem gond, írás: probléma
- A fork() és a copy-on-write (COW) technika
 - duplikálja a laptáblát (kereteket nem)
 - növeli a hivatkozásszámlálókat
 - beállítja a read-only (RO) és a copy-on-write (COW) jelzőbitet
 - írás esetén
 - a read-only bit miatt megszakítás
 - a kernel megszakításkezelője látja a COW bitet
 - duplikálja a lapot ← **csak itt allokál új memóriát**
 - törli a lapok RO és a COW bitjeit
 - visszatér a megszakításból
 - az MMU megismétli az írás műveletet

Mely lapok legyenek a fizikai memóriában?

- Mennyi keretet rendeljünk egy taszkhoz?
 - túl sok – jó neki, de másoknál lesz laphiba
 - túl kevés – sok taszk futhat, de mindenkinél laphibák lesznek
- A magas **laphiba-gyakoriság** (page fault frequency, PFF) hátrányos



Laphibák

- A magas **laphiba gyakoriság** (page fault frequency, PFF) hátrányos
 - lassítja a taszkok működését
 - futása megszakad, újra kell ütemezni
 - emlékeztető: **memória-intenzív taszk** → **I/O-intenzív**
 - nő a rezsiköltség és a terhelés → újabb laphiba keletkezhet

Vergődés (trashing): gyakori laphibák miatt a teljesítmény jelentősen romlik
A taszkok számának kordában tartásával (középtávú ütemezéssel) kezelhető.

- Jobb, ha nem kezeljük, hanem elkerüljük
 - a lapok kiírása és behozása során legyünk körültekintők

Hogyan?

- lapok behozása során → jó **lapozási stratégiával**
- keretek felszabadítása során → megfelelő **lapcsere algoritmussal**

Lapozási stratégiák

- Mely lapokat töltök be a fizikai memóriába?
- Ideális algoritmus: amelyikre szükség lesz
 - ha a jövőbe látna az OS, akkor pontosan tudná
- A valóságban...
 - megpróbálhat jóslani:

előretekintő lapozás (anticipatory paging)

- inkább csak a jelenlegi igényekkel foglalkozik:

igény szerinti lapozás (demand paging)

Igény szerinti lapozás (demand paging)

- Működés
 - csak laphiba esetén fut
 - csak a szükséges lapot hozza be
- Értékelés
 - egyszerű
 - korábban nem használt lapokra való hivatkozás **mindig** laphibát generál
 - ez lassítja a taszk futását
- Példa: műveletek nagy adatstruktúrán
 - laponként laphiba (gyakori)
 - laphiba → I/O-ra vár a taszk → átütemezik
 - CPU-intenzív helyett I/O-intenzív
 - sok megszakítás, kontextusváltás
 - nő a rezsiköltség

Előretekintő lapozás (anticipatory paging)

- Működés
 - lapcsere során több lapot hoz be
 - „előre dolgozik”
 - megpróbálja kitalálni, mely lapokra lesz szükség:
 - lokalitási jellemzők
 - laphibák a múltból
- Értékelés
 - jó becslés → kevesebb laphiba
 - korábban nem hivatkozott lapok is a fizikai memóriában vannak
 - kevesebb megszakítás, I/O és átütemezés
 - rossz jóslás → több laphiba
 - nem a megfelelő lapokat töltötte be, nem csökken a laphibák száma
 - sok felesleges lapot tart a fizikai memóriában
 - kevesebb a szabad keret

Keretek felszabadítása

- Feladat
 - melyik keretet?
 - mikor? hogyan?
- Működés (lapcsere)
 - laphiba-megszakítás
 - felszabadítandó keret kiválasztása **Hogyan? Lapcsere algoritmus**
 - a keret tartalmának mentése a cserehelyre (ha szükséges)
 - a kerethez tartozó laptábla-bejegyzés módosítása és a keret felszabadítása

6.

36	N	swap	13	36.	szabad	0	0
----	---	------	----	-----	--------	---	---

- a keret új tartalmának betöltése a cserehelyről (vagy előállítása)

36.

foglalt	1	0
---------	---	---

- új laptábla-bejegyzés készítése

9.

36	I	swap	43
----	---	------	----

- MMU beállítása
- visszatérés a megszakításból

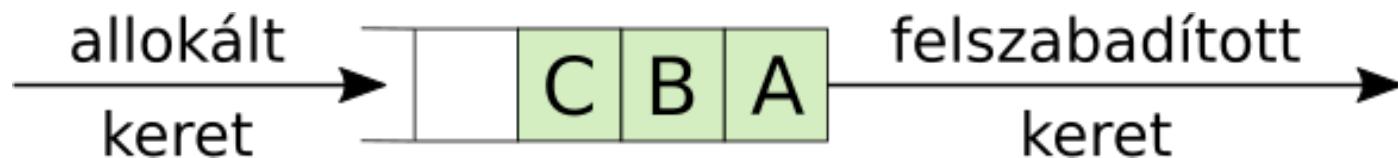
Lapcsere algoritmusok

- **Melyik keret tartalmát írjuk ki a cserehelyre?**
- Ideális megoldás: amelyik lapra legkésőbb lesz szükség
 - ismét a jövőbelátás...
- A valóságban...
 - ... valamilyen módon közelítjük a jövőbelátást:
 - FIFO: amelyiket legrégebben hoztuk be
 - Újabb esély (SC): legrégebben behozott és nem hivatkozott lap
 - Legrégebben nem használt (LRU)
 - Legkevésbé használt (LFU)
 - Utóbbi időben nem használt (NRU): nem hivatkozott és nem módosított lap

Mire támaszkodhat egy lapcsere algoritmus?

- mikor allokáltak a laphoz keretet
 - milyen sorrendben
- hivatkozás jelzőbit:
 - használták-e az adott lapot „mostanában”
- mikor használták
 - milyen sorrendben
- módosított jelzőbit
 - módosult-e a keret tartalma
 - ha igen, hosszabb lehet a felszabadítása
- a választás jellege
 - az aktuális taszk címteréből: **lokális**
 - az összes lap közül: **globális**

A FIFO lapcsere



Feladatmegoldás: FIFO lapcsere 3 kerettel

Lapkérés:	1	2	3	4	1	2	5	1	2	3	4	5
Keretek	A	1			4			5				5
	B		2			1			1		3	
	C			3			2			2		4
Foglalás:	A1	B2	C3	A4	B1	C2	A5	-	-	B3	C4	-
FIFO	A	A	A	B	C	A	B	B	B	C	A	A
	B	B	C	A	B	C	C	C	C	A	B	B
	C	A	B	C	A	A	A	A	A	B	C	C
Ütem:	1	2	3	4	5	6	7	8	9	10	11	12

Feladatmegoldás: FIFO lapcsere 4 kerettel

Lapkérés:	1	2	3	4	1	2	5	1	2	3	4	5
Keretek	A	1			1		5				4	
	B		2			2		1				5
	C			3					2			
	D				4					3		
Foglalás:	A1	B2	C3	D4	-	-	A5	B1	C2	D3	A4	B5
	A	A	A	A	A	A	B	C	D	A	B	C
FIFO		B	B	B	B	B	C	D	A	B	C	D
		C	C	C	C	C	D	A	B	C	D	A
			D	D	D	A	B	C	D	A	B	
Ütem	1	2	3	4	5	6	7	8	9	10	11	12

A FIFO lapcsere értékelése

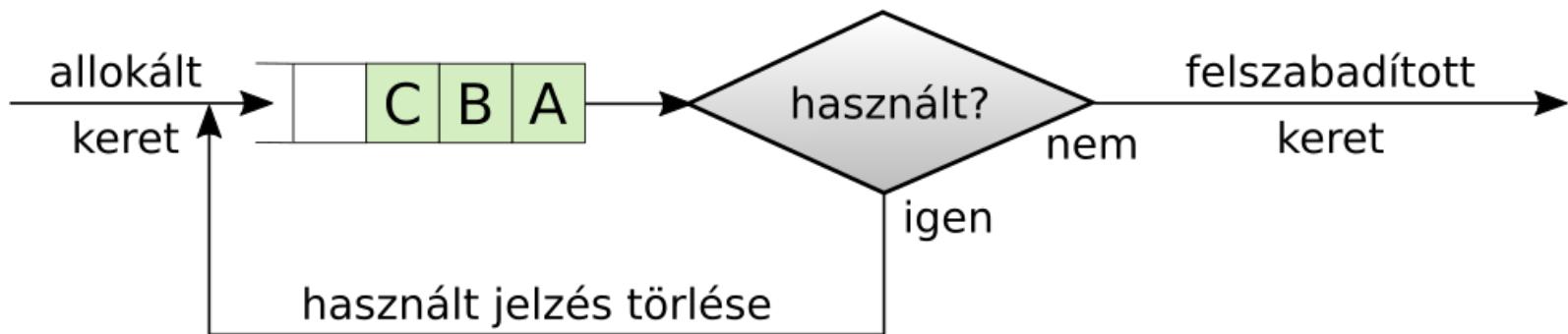
- Tulajdonságai
 - egyszerű algoritmus és adatstruktúra (FIFO)
 - előrenéző (nem érdekli a múlt)
- Komplexitás
 - $O(1)$
- Rezsiköltség
 - minimális
- Előnyök, problémák
 - könnyű megvalósítani
 - a lapok jövőbeli használatát (optimális algoritmus) gyengén becsli
 - nem figyeli a módosítást (a kiírás sokáig tarthat)

Mi történik, ha növeljük a rendelkezésre álló keretek számát?

- csökken a laphibák száma
- **időnként** nem, sőt, nő!!

Bélády-féle anomália (Bélády László, IBM)

Az újabb esély (second chance, SC) lapcsere

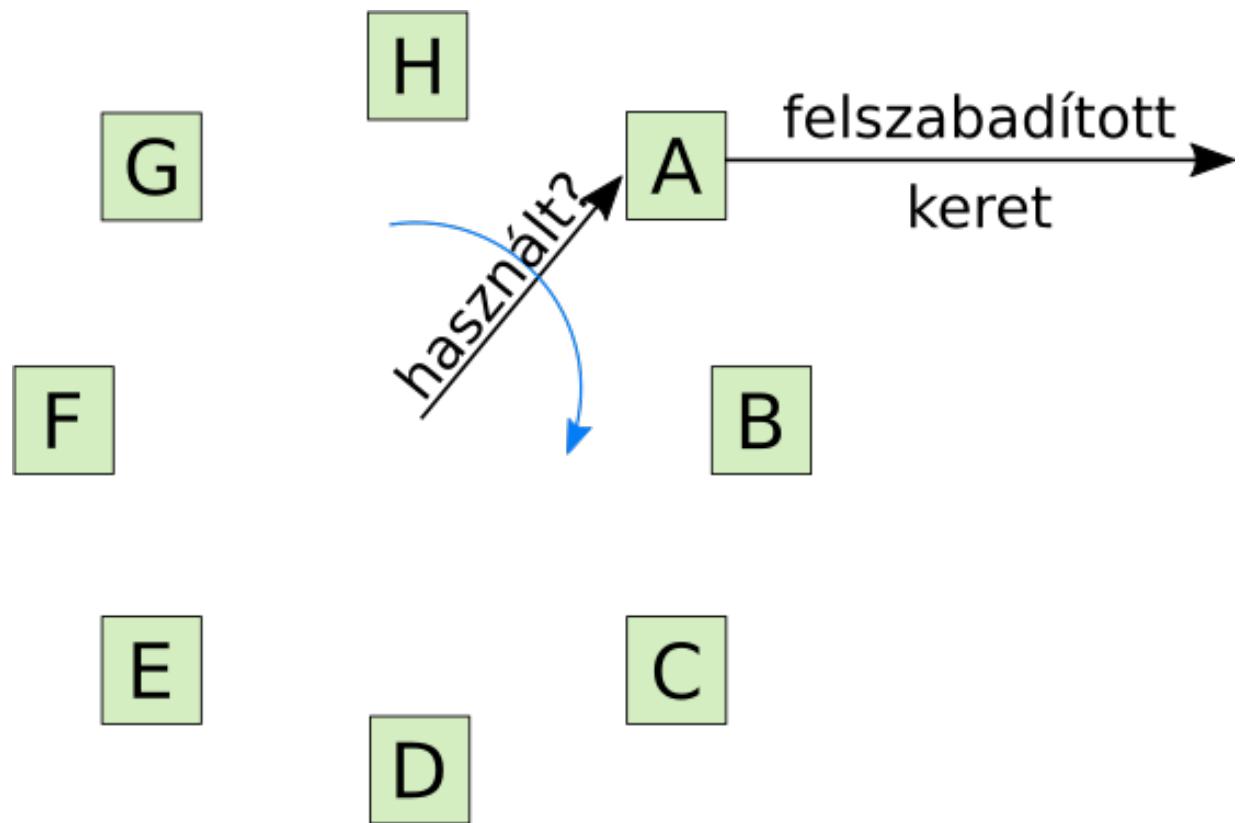


A „használt” avagy „hivatkozott” (referenced) jelzést az MMU állítja be.

Az „újabb esély” lapcsere értékelése

- Tulajdonságai
 - egyszerű algoritmus és adatstruktúra (FIFO)
 - hátranéző
 - a használt lapokat nem szabadítja fel
- Komplexitás
 - $O(1)$
- Rezsiköltség
 - minimális
- Előnyök, problémák
 - könnyű megvalósítani
 - jobban becsüli a jövőbeli használatot a FIFO-nál
 - nem figyeli a módosítást (a kiírás sokáig tarthat)
 - állandóan mozgatja az adatokat a FIFO-ban

Van jobb adatstruktúránk az „újabb esély” számára?



Óra (clock) lapcsere

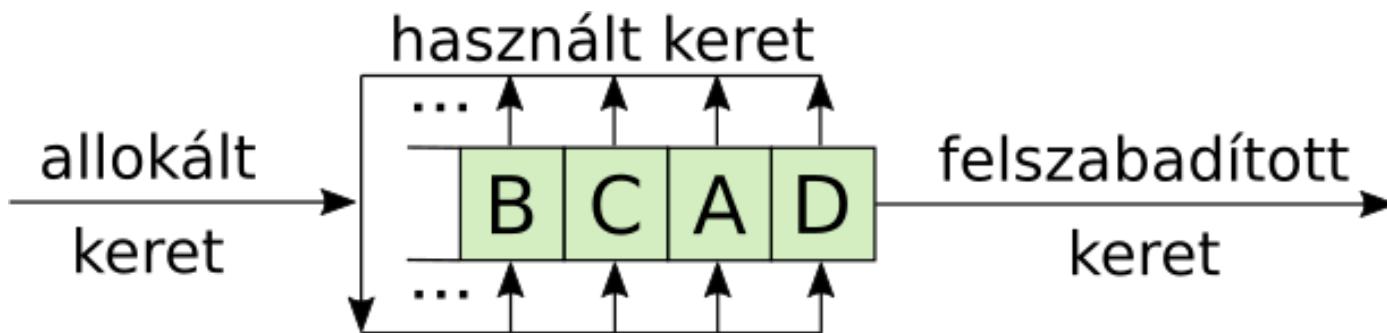
Az óra lapcsere-algoritmus értékelése

- Tulajdonságai
 - nem túl bonyolult algoritmus és adatstruktúra (lista + óramutató)
 - hátranéző
 - a használt lapokat nem szabadítja fel
- Komplexitás
 - $O(1)$
- Rezsiköltség
 - minimális
- Előnyök, problémák
 - ugyanolyan jó, mint az újabb esély
 - könnyű megvalósítani
 - jobban becsüli a jövőbeli használatot a FIFO-nál
 - nem figyeli a módosítást (a kiírás sokáig tarthat)
 - de megspórolja az adatok mozgatását a FIFO-ban

Lehet-e még jobban jósolni?

- Az eddigi algoritmusok...
 - a FIFO nem nagyon foglalkozik a múlttal (minél „öregebb”, annál jobb)
 - az SC és Clock a múltat 1 bitbe zsúfolták...
- Nincs több információink a keretekről?
- Van...
 - módosítás
 - hányszor használták
 - mikor használták
 - hány taszk használta

Legrégebben nem használt (least recently used, LRU)



A lapokat a használati gyakoriságuk szerint rendez sorba.

többféle megvalósítás, amelyek ötvözik a

- a használati idő szerinti
- hivatkozások száma szerinti

hardverfüggő

rendezést

Az LRU értékelése

- Tulajdonságai
 - bonyolultabb algoritmus és adatstruktúra (láncolt lista)
 - hátranéző
 - a használt lapokat nem szabadítja fel
- Komplexitás
 - jellemzően $O(N)$
- Rezsiköltség
 - hardvertámogatással kicsi, anélkül nagy
- Előnyök, problémák
 - a legjobban becsli a lapok jövőbeli használatát (az optimális algoritmust)
 - nyilvántartja a lapok múltbeli használatát, ami egy jó becslő
 - **a frissen behozott lapot nagy eséllyel lecseréli (miért?), ami nem jó**
 - csak hardvertámogatással érdemes megvalósítani
 - nem figyeli a módosítást (a kiírás sokáig tarthat)

A lapok tárba fagyasztása (page locking)

- Frissen behozott lapoknak nincs múltja
 - a jövőjük nem becsülhető
 - könnyen kiírásra választhatja őket a lapcsere
- I/O művelet alatt álló lapot ne szabadítsunk fel
 - az I/O műveletek fizikai címeket használnak
 - akár a CPU-t megkerülve, DMA vezérlő segítségével is módosíthatják a memóriát
 - a laptípusok erre is figyelnie kell
- Megoldás: **lapok tárba fagyasztása**
 - page lock bit jelzi a zárolt (fagyasztott) állapotot
 - az ilyen lapok nem lehetnek a lapcsere „áldozatai”
 - I/O művelet esetében annak végéig tart a zárolás
 - az első hivatkozás feloldja a zárolást

A laplopó taszk

- Sokféle név alatt létezik:
 - Page daemon, kswapd, Working Set Manager
- Feladata: üres keretek biztosítása
 - rendszeres időközönként felébred vagy a kernel felébreszti
 - a szabad keretek számát igyekszik két határérték között tartani
 - ha egy minimum szint alá esik, elkezd kereteket „lopni”
 - a maximum szint elérésekor alvó állapotba lép
- A laplopó végezheti az összes lapcserét
 - ha nem volt elég „ügyes”, és mégis elfogytak a szabad memóriakeretek
 - kiválaszt egy lapot és levezényli a lapcsere folyamatát
- Egyéb feladatai lehetnek:
 - referenced bit törlése
 - használati számlálók öregítése

Legkevésbé használt (least frequently used, LFU)

- Not frequently used (NFU) néven is ismert.
- Az LRU egyszerűsített változata (közelítése)
 - az OS időnként növel egy használati számlálók a referenced = 1 lapokra
 - a számláló alapján választja ki a felszabadítandó keretet
- Az algoritmus értékelése
 - közepes rezsiköltségű, periodikusan igényli a számlálók növelését
 - a laptopó taszkkal jól kombinálható
 - viszonylag jól becsli a lapok jövőbeli használatát (az optimális algoritmust)
 - közelítőleg nyilvántartja a lapok múltbeli használatát, ami egy jó becslő
 - a frissen behozott lapot nagy eséllyel lecseréli, ami nem jó
 - a számláló túlcordulhat (a probléma **öregítéssel** kezelhető)
 - nem tud különbséget tenni a módosított és változatlan lapok között, ezért a lapcsere diszk művelettel is járhat, ami jelentősen lassítja a működést

Mostanában nem használt (not recently used, NRU)

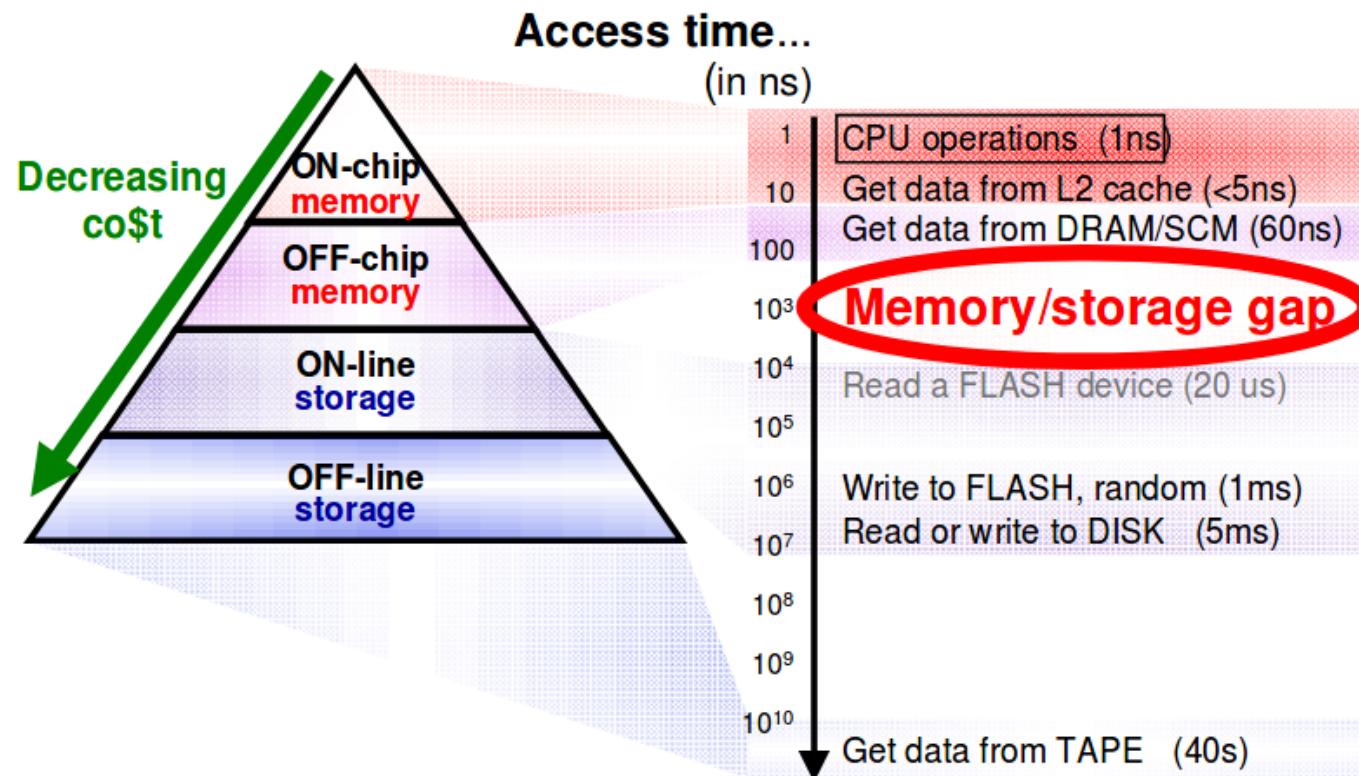
- A második esély (SC) algoritmus finomított változata
 - a referenced jelzőbit mellett a dirty (módosult) bitet is figyeli
 - a két bit segítségével egy „prioritást” rendel a lapokhoz:
 - ref=0 dirty=0 → a prioritás 0, nem hivatkozott, nem módosított
 - ref=0 dirty=1 → a prioritás 1, nem hivatkozott, módosított
 - ref=1 dirty=0 → a prioritás 2, hivatkozott, nem módosított
 - ref=1 dirty=1 → a prioritás 3, hivatkozott, módosított
- Amikor egy szabad keretre van szükség
 - véletlenszerűen választ a legkisebb prioritású lapok közül
- Az algoritmus értékelése
 - kis rezsiköltségű, jól kihasználja a hardvertámogatást
 - a SC-nél jobban becsli a lapok jövőbeli használatát (az optimális algoritmust)
 - a hivatkozások mellett a módosításokat is figyeli
 - különbséget tesz a módosított és változatlan lapok között

Érdekességek

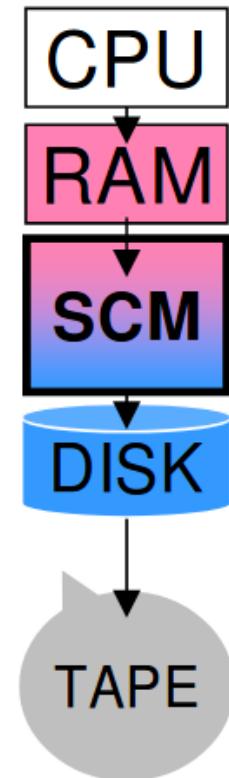
- Miért történik hiba, ha a 0x00000000 címre hivatkozik egy program?
 - tipikus hiba („invalid / NULL pointer”) inicializálatlan mutatóval
 - miért okoz hibát?
 - kapcsolódó: miért nem kezdődik a 0x00000000 címen a programkód?

[Tipp](#)
- Hogyan akadályozható meg, hogy a verem túl nagyra nőjön?
 - nem érdemes statikusan lefoglalni a teljes lehetséges méretét
 - ha viszont dinamikusan nőhet, hogyan állítható meg a növekedése?
 - akár bajt is okozhat: lásd [Stack Clash bug](#) (helyi root jogot ad)

[Tipp](#)
- Storage Class Memory (SCM)
 - **nem felejtő**, gyors (az I/O már nem lassú), „olcsó” / GB, nagy (TB)
 - koncepcióváltás: számítás-orientált → adat-centrikus (lásd gépi tanulás)
 - többprocesszoros, heterogén és elosztott rendszerek közös adattárolással
 - [IBM](#), [Gen-Z](#), [ACM](#)
(lásd következő fóliák)



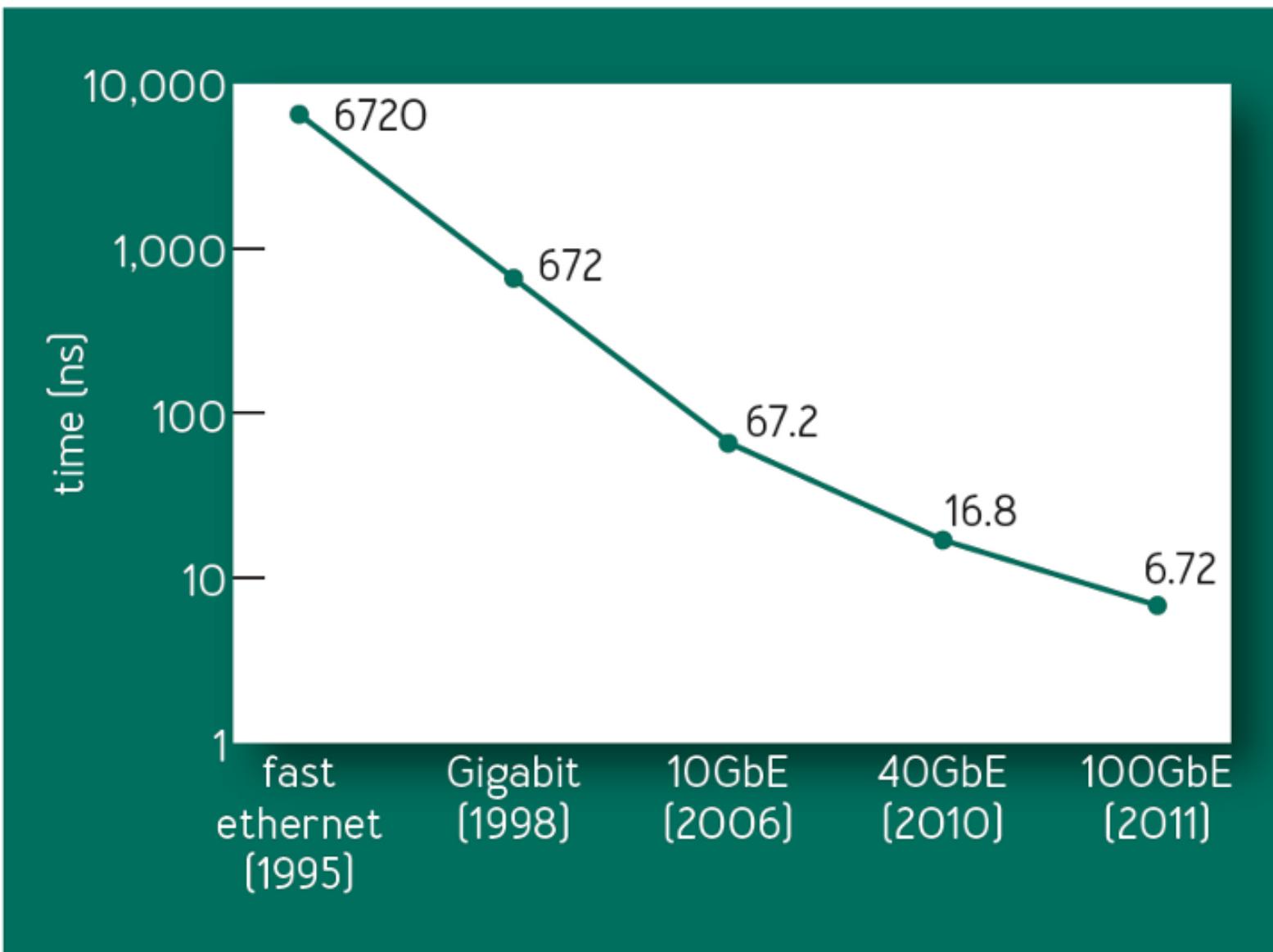
Near-future



Research into new solid-state non-volatile memory candidates
– originally motivated by finding a “successor” for NAND Flash –
has opened up several interesting ways to change the memory/storage hierarchy...

Forrás: IBM

FIGURE 1: PER-PACKET PROCESSING TIME WITH FASTER NETWORK ADAPTERS



Forrás: [ACM](#)

Összefoglalás

- Absztrakt virtuális gép + lapkezelés
 - a taszkok egy lapokra bontott, virtuális címtartományt használnak
 - a kernel kereteket rendel a lapokhoz (címtábla) – **erőforrás-alkotátor**
- A memóriakezelés
 - alapvetően az MMU végzi a logikai – fizikai címleképezést
 - ha egy lapot nem ér el (**laphiba**) a kernel szoftveres címleképezése segíti
 - a háttértárral bővíti a fizikai memóriát
 - gondoskodik a taszkok szeparációjáról és védelméről
- Lapok betöltése:
 - igény szerint jellemző, korlátozottan előretekintő módon
- Szabad keretek biztosítása (laplopó taszk)
 - lapcsere algoritmusok
 - az ideális megoldás nem érhető el, csak közelíthető

Operációs rendszerek: taszkok kommunikációja

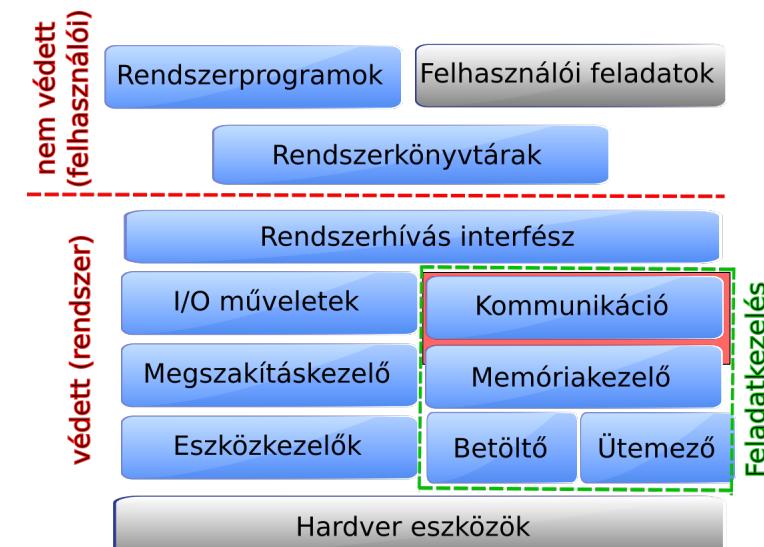
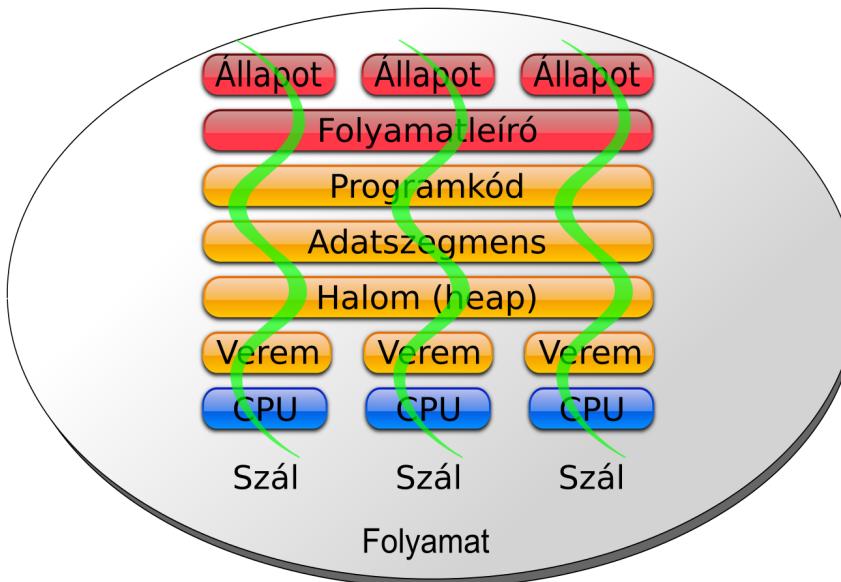
Mészáros Tamás
<http://www.mit.bme.hu/~meszaros/>

Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

Az előadásfóliák legfrissebb változata a tantárgy [honlapján](#) érhető el.
Az előadásanyagok BME-n kívüli felhasználása és más rendszerekben történő közzététele előzetes engedélyhez kötött.

Az eddigiekben történt...

- A feladatokat taszkok végzik el
 - szétoszthatnak részfeladatokat
 - egyesíthetnek részeredményeket
- Absztrakt virtuális gép
 - szeparálja a taszkokat
 - gátolja az együttműködést



- Taszkok megvalósítása
 - folyamat
 - szál
- Szál
 - szekvenciális működésű taszk
 - egy folyamaton belüli más szálakkal közös memóriát használ

A kommunikáció alapvető sémái

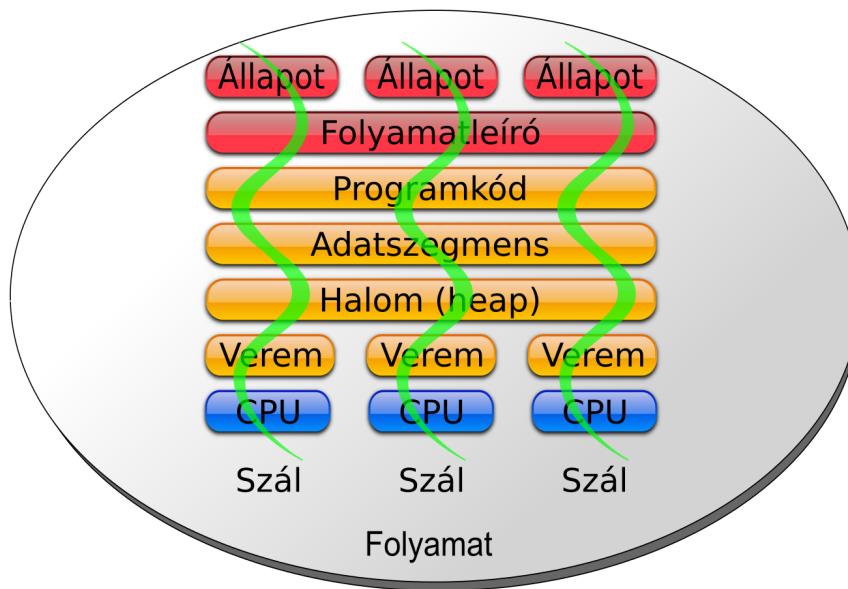
- Közös memórián keresztül
 - PRAM modell
 - versenyhelyzetek
 - megvalósítási példák
 - folyamaton belüli szálak
 - POSIX osztott memória
- Üzenetváltás segítségével
 - adatátviteli rendszer (nagyon sokféle)
 - alapvető műveletei:
 - Küld(címzett, adatcím[, adatméret])
 - Fogad(címzett, puffercím[, adatméret])
 - megvalósítási példák
 - hálózati kommunikáció
 - távoli eljáráshívás
 - elosztott rendszerek
 - mikrokernel

A PRAM (pipelined RAM) modell

- A taszkok párhuzamosan használnak egy közös memóriaterületet
 - a taszkok műveletei egymástól függetlenek (random access)
 - ütközhetnek is
- Az ütközésfeloldás szabályai
 - olvasás–olvasás mindkettő a memória tartalmát adja vissza
 - olvasás–írás az olvasás vagy a régi, vagy az új értékét adja vissza
 - írás–írás a két érték valamelyike kerül a memóriába
- A szabályok hatása
 - a műveletek nem hatnak egymásra (nem keverednek)
(pipelined: sorba rendezett, nem kevert)
 - a párhuzamos kérések sorrendje nem definiált
bizonytalan, hogy melyik hajtódik előbb végre (de nem keverednek)
 - a párhuzamos kéréseket össze kell hangolni
→ **szinkronizáció**

PRAM: folyamaton belüli szálak

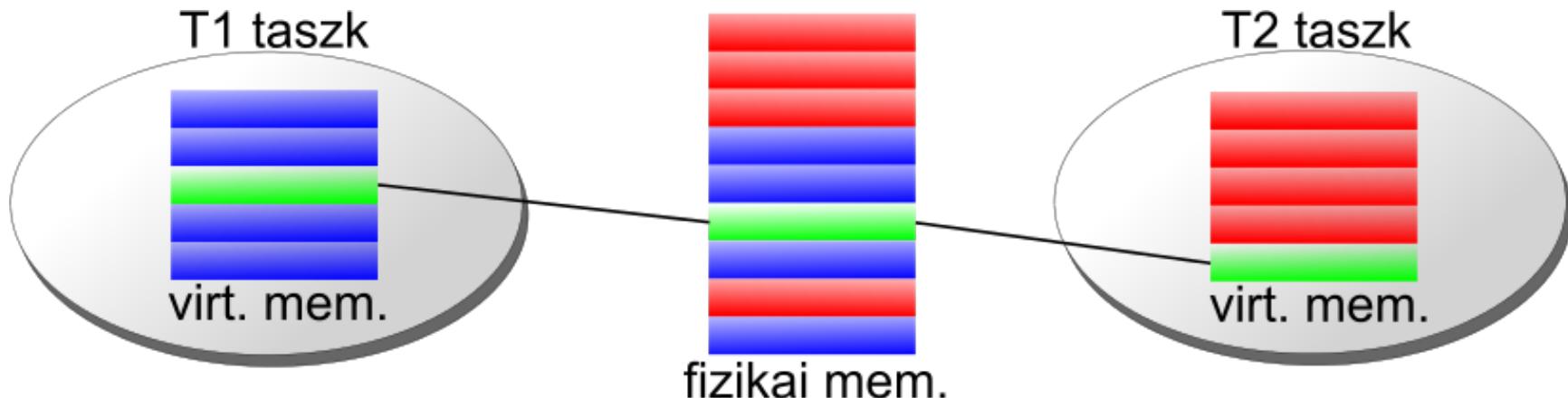
- Adatcsere globális változókkal
 - az OS egy folyamaton belül közös memóriát biztosít a szálaknak
 - a kommunikáció nem az OS fennhatósága alatt történik
 - a programozó alakítja ki a működést



Példa: munkamegosztás szálak között

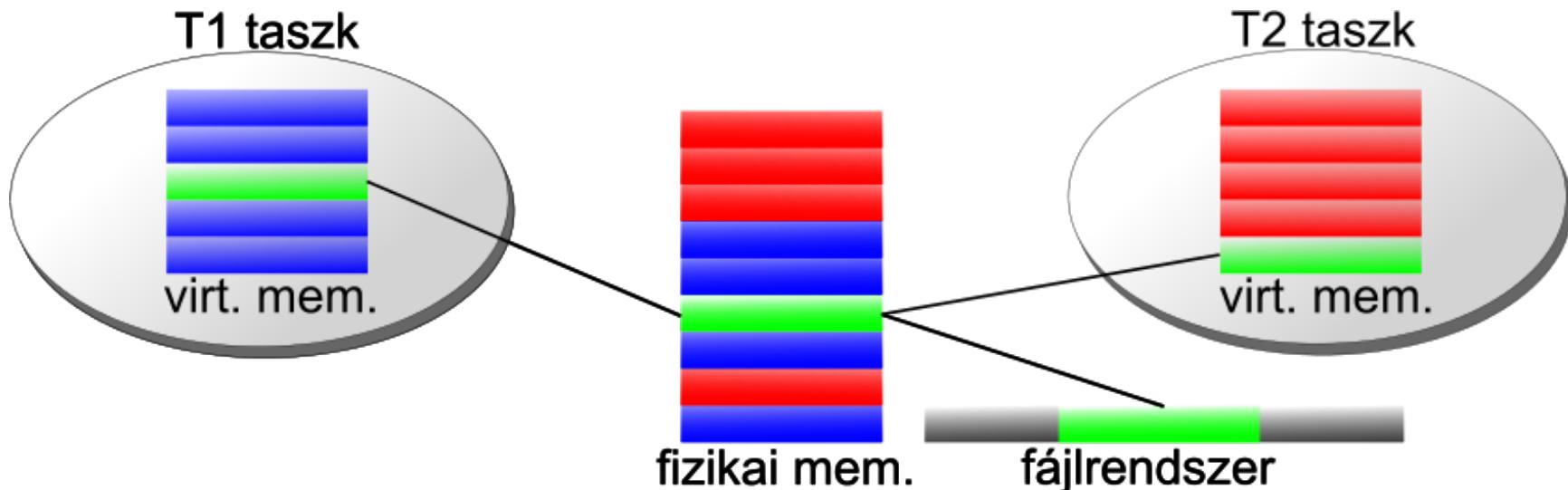
```
struct data_type data[N];  
  
compute (void *part) {  
    ... műveletek a data[] *part részén ...  
}  
  
pthread_t *tid;  
  
for (i=0; i < N; ++i) {  
    // szálak indítása  
    pthread_create(&tid[i],NULL,compute,&data[i]);  
}  
  
for (i=0; i < N; ++i) {  
    // megvárjuk, míg elkészülnek  
    pthread_join(tid[i], NULL);  
}
```

PRAM: osztott memória (shared memory, SHM)



- Értékelése
 - nincs rendszerhívás, nulla rezsiköltség
 - rendkívül gyors kommunikációt biztosít (zero-copy)
 - korlátos kapacitással rendelkezik
- Megvalósítása
 - szabvány: POSIX Shared Memory
 - Unix, Windows (a kernel része, a memóriakezelés támogatja)
 - felhasználói címtérben, pl. [C++ könyvtárként](#)

PRAM: memóriába ágyazott fájlelérés



- Értékelése
 - lehet ilyen az SHM, ahol az OS nem támogatja
 - a klasszikus fájlrendszer interfész helyett is jó
- Megvalósítása
 - széles körben elérhető (CreateFileMapping(), mmap())
 - az OS virtuális memóriakezelője végzi a leképezést
 - sokféle programozási környezetben elérhető

Üzenetváltásos kommunikáció



Az üzenetváltásos kommunikáció kérdései

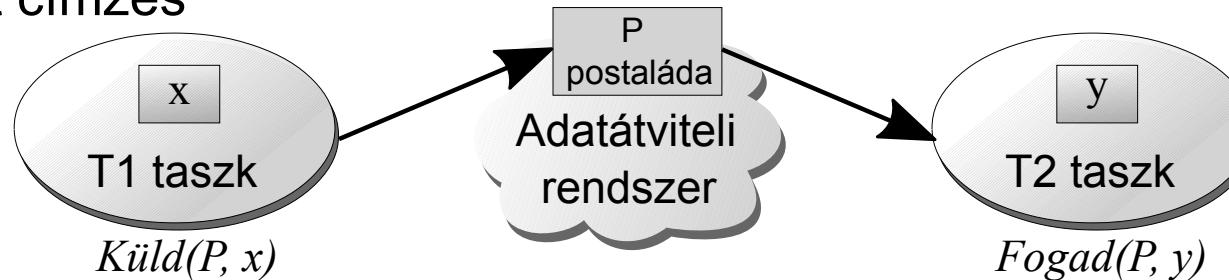
- Címzés
 - direkt vagy közvetítővel?
 - egy vagy több címzett?
 - a címzett szűrheti a feladókat?
- Szinkronitás
 - hol az adat a Küld() művelet visszatérésekor?
 - hogyan értesül a feladó a kézbesítésről?
 - mi történik, ha kiadjuk a Fogad műveletet, de az adat még nem érkezett meg?
 - ha fogadtunk az adatokat, kell visszaigazolást küldenünk?
- Az adatátvitel szemantikája
 - az adat a küldőnél is megmarad, vagy a fogadónál lesz, esetleg mindkettőnél?
- **Teljesítmény, megbízhatóság**
 - milyen adatátviteli sebesség érhető el és mekkora az üzenetek késleltetése?
 - hány és milyen méretű üzenet küldése lehetséges?
 - ki veszi észre és mi történik adatátviteli hiba esetén?

Alapvető címzési módszerek (lásd még Komháló 1.)

- Direkt címzés



- Indirekt címzés



- Aszimmetrikus (küldő oldalon direkt) címzés



- Többszörös (multicast, broadcast)

Szinkronitás

- Szinkron adatátvitel
 - A Küld() és a Fogad() blokkoló művelet
 - a taszk várakozó állapotba kerül
 - egyszerűen programozható
 - az eredmény és az esetleges mellékhatások is beérkeznek
 - megszakad a taszk futása, átütemezés
 - Denial-of-Service (DoS) támadások...
 - időtúllépés...
- Aszinkron adatátvitel
 - a műveletek nem blokkolnak
 - a taszk tovább futhat, de a műveletek eredménye még nem érhető el
 - az esetleges mellékhatások, hibák sem jelentkeznek
 - a műveletek eredményeit ellenőrizni kell
 - **a még nem kézbesített üzeneteket átmenetileg tárolni kell (pl. a kernelben)**
 - hasznos, ha van más csinálnia a taszknak
 - nem hasznos, ha ciklusban ellenőrzi a küldést/fogadást

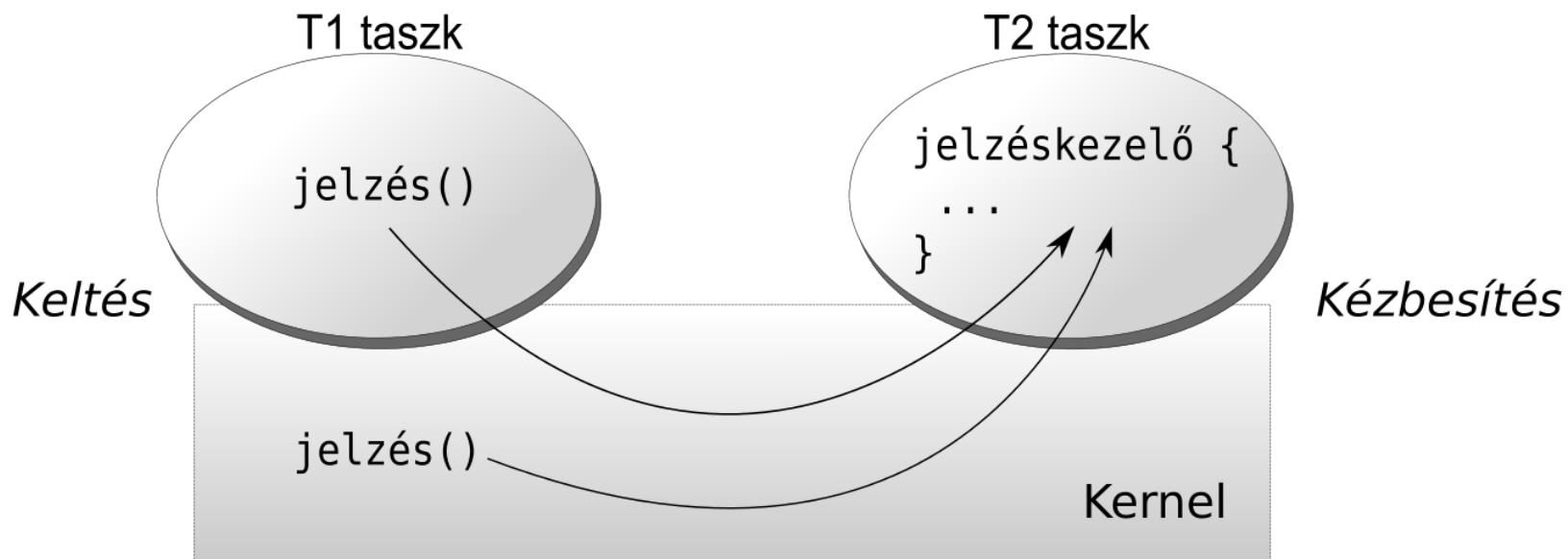
Adatátviteli szemantika, adatbirtoklás

- A kommunikáció résztvevői meddig férnek hozzá az adatokhoz?
- Kizárolagosan vagy megosztva birtokolják azokat?
 - **Másolat** (copy semantics)
 - a küldő és a fogadó is saját példánnyal rendelkezik
 - a módosítások hatása lokális
 - **Megosztás** (share semantics)
 - ugyanazt használják, jogosultság lehet különböző
 - a módosítás hatása globális (szinkronizáció!!!)
 - csökkenheti az adatmozgatást (azonos rendszeren belül)
 - **Mozgatás** (move semantics)
 - a küldő elveszíti a hozzáférését az adatokhoz, amikor elküldi őket
 - megvalósítható megosztással és a jogosultságok elvételével is
- Az „**adatbirtoklás**” fontos kérdés a párhuzamos programozásban
 - szinkronizáció
 - munkamegosztás

Direkt és aszimmetrikus kommunikációs megoldások

- Jelzés (signal)
 - értesítés eseményekről: taszk → taszk, kernel → taszk
- Hálózati kommunikáció (socket communication)
 - adatátvitel akár rendszerek között is
 - aszimmetrikus kommunikáció kliens-szerver modell szerint
 - sokféle protokoll és címzés
- Távoli eljáráshívás (remote procedure call)
 - egy másik taszk programjában levő eljárás meghívása
 - aszimmetrikus kommunikáció kliens-szerver modell szerint
 - hálózati kommunikációra épül
 - adatkonverziót is végez

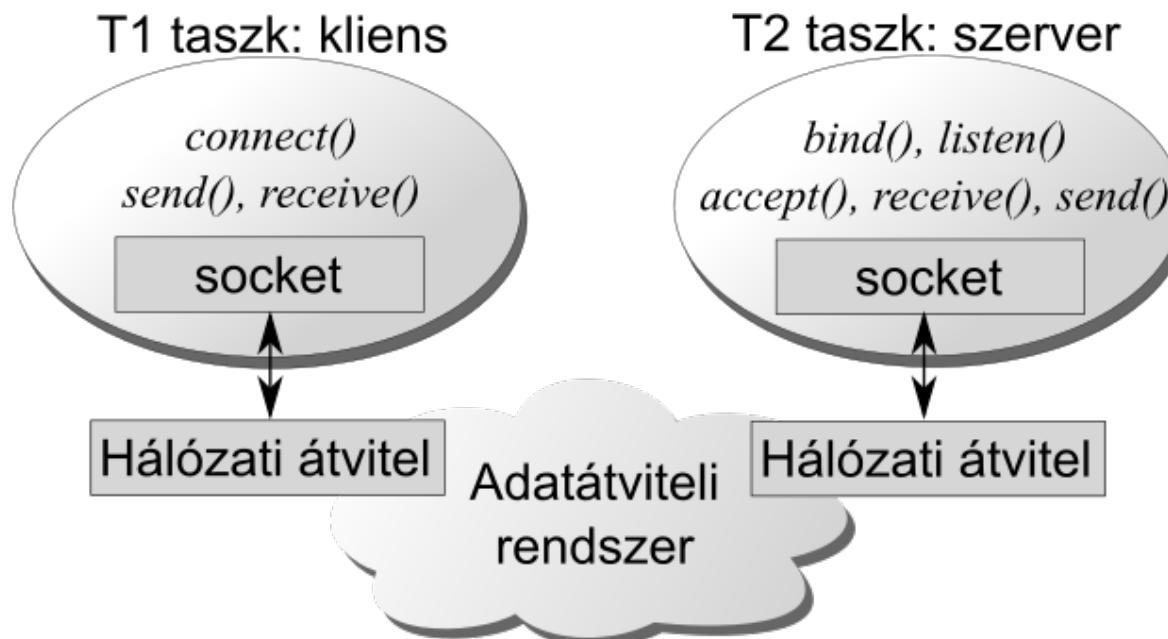
Jelzések



- Célja
 - értesítés eseményekről
- Kétfázisú működés
 - keltés
 - kézbesítés (kezelés)
- Típus
 - felhasználói
 - rendszer
 - kivételek, értesítések, riasztások
- Kezelés
 - kezelőfüggvény
 - figyelmen kívül hagyás

Hálózati kommunikáció (ismétlés, Komháló 1.)

- Címzés és hálózati protokollok
 - gépen belül (localhost, 127.0.0.1 ill. ::1) és gépek között
 - többféle címzés (*cast) és protokoll (pl. IP, TCP és UDP)
- A kommunikációs csatorna leírása: **hálózati csatoló (socket)**
 - a kommunikációs végpont (logikai) azonosítója a taszkokban



A távoli eljáráshívás (remote procedure call, RPC)



Példa [Open Network Computing](#) (korábban Sun) RPC

Indirekt kommunikációs megoldások

- Postaláda (mailbox)
 - véges számú (sokszor csak egyetlen) üzenet
 - az üzenetek mérete korlátos
 - a postaláda címezhető, nem a fogadó (indirekt)
- Üzenetküldés (message queue / message passing)
 - korlátos méretű adathalmaz átküldése
 - sokféle cél és implementáció (helyi OS, elosztott rendszerek, HPC stb.)
 - jellemzően indirekt (csatornán kereszttüli)
 - működhet rendszerek közötti hálózaton is
 - sokféle implementáció [message oriented middleware](#), [RabbitMQ](#), Java MS, MSMQ
 - többféle szabvány: POSIX üzenetsorok, [AMQP](#), [MQTT](#), HPC [MPI](#) stb.
- Csővezeték (pipe)
 - végtelen (gyakorlatilag korlátos) adatmennyiség továbbítására
 - folytonos adatküldésre és fogadásra alkalmas (nincs üzenethatár)
 - a csővezeték címezhető (nem a fogadó)
 - egyszerre több vevő is lehet

Kommunikációs megoldások teljesítménye

- A PRAM közvetlenül a virtuális memóriakezelésre támaszkodik
 - nincs felesleges kernel réteg
 - nincs adatmozgatás
 - nincs extra rezsiköltségnincs +késleltetés, nagy adatátviteli sebesség (zero copy)
- Az üzenetváltásos modell kommunikációs infrastruktúrára épít
 - mozgatja az adatokat (többször is)
 - esetenként konvertálja is
 - átmenetileg tárolhatja iskésleltetés okoz, az adatátviteli sebesség jelentősen csökken
- Hol okoz ez igazán gondot?
 - pl. a mikrokernelekben
 - pl. a nagy teljesítményű (HPC) rendszerekben

Mi okozza a késleltetést és a lassulást?

- Rendszerhívások (Küld() / Fogad())
 - megszakítás
 - kontextusváltás
 - esetenként átütemezés
- Adatmásolás
 - taszk1 címtér → kernel → taszk2 címtér
- Kontextusváltás
 - T1 és T2 taszkok párbeszéde
 - T2 Fogad() → blokkolódik (kontextusváltás)
 - T1 Küld() → blokkolódik (kontextusváltás)
 - T2 felébred → átütemezés → Fogad() Küld() → blokkolódik
 - T1 felébred → átütemezés → Fogad() Küld() → blokkolódik
 - ...
 - sok átütemezés és kontextusváltás (a TLB állandóan kiürül)

A OS védelmi mechanizmusai miatt csökken a hatékonyság.

Teljesítménynövelés modern mikrokernelekben

- A mikrokernelekben kritikus a problémák megoldása minél kevesebb
 - adatmásolás
 - kontextusváltás
 - átütemezésa kernel üzenetalapú kommunikációja miatt
- A kontextusváltás és az ütemezés rezsiköltségének **csökkentése**
 - **direkt kontextusváltás**
 - nem az ütemező dönt a következő futtatandó taszkról
 - a Küld() – Fogad() séma határozza meg az átütemezést
 - nem fut az ütemező (nincs rezsiköltsége)
 - kicsi lesz a késleltetés a küldés és vétel között
 - **lazy queueing**
 - Küld() – Fogad() párbeszédek kezelése
 - átmenetileg felfüggeszti a taszkok állapotváltozásának adminisztrációját
 - a taszkok nem mozognak a Fut, FK és Vár sorok között
 - ha véget ér a párbeszéd, helyreállítja a sorok tartalmát

Az adatátvitel gyorsítása

- Hogyan gyorsítható?
 - az adatmennyiségtől függ...
 - mekkora mennyiségről van szó?
 - mikrokernel belső működése
 - függvényhívások
- Nagyon kevés adat (< ~ 16 byte)
 - a processzor regisztereiben
 - nincs sok erre a célra alkalmas regiszter
 - a teljes kommunikáció gyorsulása: ARM11: 10%, CortexA9 (újabb) 4%
x86-on akár ronthat is – [Vajon miért csökken vagy negatív a hatása?](#)
- Közepes méretű adathalmaz (kb. 16-64 byte)
 - **virtuális regiszter tároló**
 - elférhet a regiszterekben
 - erre a célra allokált memóriaterületen PRAM modell szerint
 - hardverfüggő módon implementálja a kernel (lásd pl. [ARM](#))

„Nagyobb” adathalmazok átvitele lokális gépen

- Osztott memóriás (SHM) átmeneti tárolással
 - Küldő → SHM → Fogadó
 - korlátozott a mérete, két másolás (Copy-in / Copy-out)
- Egy másolással (single-copy)
 - Taszk-taszk memóriamásolás (pl. [KMEM](#))
 - Küldő → Fogadó direkt másolás rendszerhívással
- Másolás nélkül (zero-copy)
 - Lapmegosztás (pl. [XPMEM](#))
 - egy taszk megoszthat memóriatartományt másokkal (kernel-támogatással)
 - Távoli memóriaelérés (pl. [CMA](#))
 - egy taszk elérheti egy másik memóriatartományát (kernel-támogatással)
 - ötlet: /proc/<PID>/mem olvashatóvá tétele (teljes?)
- Hardvertámogatással (kernel [DMA Engine](#), [RDMA](#))
 - számítási csomópontok között is működhet
 - pl. HPC, SMP környezetben

A kommunikáció alapvető sémái (összefoglalás)

- **PRAM modell**
 - közösen használható memória
 - az egyidejű műveleteket nem keveri valamilyen sorrendbe állítja őket
- Szinkron adatátviteli műveletek
- Beépített címzés nincs
 - közvetett módon kialakítható
- Alkalmazási példák:
 - folyamaton belüli szálak
 - **osztott memória**
- Előnyök
 - nagyon gyors adatcsere
 - egyszerűen használható
 - beállítás után nincs rezsiköltség
- Hátrányok, kockázatok
 - R-W és W-W konfliktusok
→ **szinkronizáció** szükséges
 - korlátos méretű
- **Üzenetalapú rendszerek**
 - adatátviteli rendszerrel működik
 - Küldés és Fogadás műveletek
 - az egyidejű műveleteket nem keveri valamilyen sorrendbe állítja őket
- Adatátvitel: szinkron / aszinkron
- Címzés
 - direkt, indirekt, többes (*cast)
- Alkalmazási példák:
 - postaláda
 - csővezeték
 - üzenetsor
 - **hálózati kommunikáció**
 - **távoli eljáráshívás**
- Előnyök
 - széleskörű elérhetőség (hálózat is)
- Hátrányok, kockázatok
 - kezelendő kommunikációs hibák

Taszkok közötti kommunikáció a gyakorlatban

Unix jelzések keltése és kezelése

- **Jelzések keltése**

```
#include <signal.h>          /* kill() */  
kill(pid, SIGUSR1);          /* jelzés küldése */
```

- **Jelzések kezelése**

```
signal(SIGALRM, alarm);      /* kezelőfüggvény beállítása */
```

- **beépített jelzskezelők**

- Core: core dump és leállítás (`exit()`)
- Term: leállítás (`exit()`)
- Ign: figyelmen kívül hagyás
- Stop: felfügesztés
- Cont: visszatérés a felfügesztett állapotból (vagy ignore)

- **saját kezelőfüggvény**

```
signal(SIGALRM, alarm);      /* kezelőfüggvény beállítása */  
void alarm(int signum) { ... } /* a kezelőfüggvény */
```

Nem minden írható felül az alapértelmezett jelzskezelő (pl. a SIGKILL nem)

man -s 7 signal (részlet)

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

Példák jelzések használatára (demó)

```
kill(pid, SIGSTOP);          /* STOP jelzés küldése */
kill -STOP <PID>

signal(SIGCLD, SIG_IGN);    /* nem foglalkozunk a gyerekekkel */

signal(SIGINT, SIG_IGN);    /* nem foglalkozunk a ctrl+c jelzéssel */

signal(SIGINT, SIG_DFL);    /* alapértelmezett kezelő beállítása */

signal(SIGALRM, myalarm);  /* jelzéskezelő függvény beállítása */
void myalarm(int signum) { ... } /* az eljárás */
alarm(30);                  /* alkalmazás: ALARM jelzés 30mp múlva */
```

Unix csővezetékek: pipe()

- Cél: folyamatok közötti adatátvitel (`ls -la | more`)
- Jellemzők
 - csak szülő-gyerek (leszármazott, testvér) viszonylatban
 - adatfolyam (nincs üzenethatár)
 - nincs adatszerkezet, adattípus
 - egyirányú adatfolyam (író → olvasó) (több író és olvasó is lehet!)
 - limitált kapacitás: pl. 4k (Linux < 2.6.11), 65k (Linux >= 2.6.11)
- Alkalmazás (demó)

```
int pipefd[2];
pipe(pipefd);
```

- író

```
close(pipefd[0]);
write(pipefd[1], string,
      strlen(string)+1);
close(pipefd[1]);
```

Miért?

Miért kell lezárni? Ötlet

- olvasó

```
close(pipefd[1]);
read(pipefd[0], buffer,
      sizeof(buffer));
close(pipefd[0]);
```

Hogyan működik?
Megoldás

Elnevezett csővezetékek (named pipe, FIFO)

- Cél: folyamatok közötti adatátvitel
- Jellemzők
 - független folyamatok között is működő csővezeték
 - fájlrendszeri bejegyzéssel azonosítható (`mkfifo`, `mknod`)
 - kétirányú kommunikáció (megnyitás olvasásra és írásra)
- Alkalmazás + demó

```
int pipefd[2];
char *fifo_fname="/path/to/fifo_filename";
mkfifo(fifo_fname, 0600);
```

- író
 - pipefd[1] =
open(fifo_fname, O_WRONLY);
 - write(pipefd[1], string,
 strlen(string)+1);
 - close(pipefd[1]);

```
unlink(fifo_fname);
```

- olvasó
 - pipefd[0] =
open(fifo_fname, O_RDONLY);
 - read(pipefd[0], buffer,
 sizeof(buffer))
 - close(pipefd[0]);

Ki és mikor szünteti meg?

Unix System V IPC / POSIX IPC

- Cél: folyamatok közötti egységes kommunikáció
- Erőforrások
 - adatátvitel: üzenetsor, osztott memória
 - szinkronizáció: szemafor
- Közös alapok
 - kulcs: azonosító az erőforrás eléréséhez (egy 32 bites szám)
 - közös kezelőfüggvények: `*ctl()`, `*get(... kulcs ...)`
 - jogosultsági rendszer (szereplők és hozzáférési szabályok)
 - bővebb infó: `man svipc ipc ipcs`
- POSIX IPC
 - kulcs helyett szöveges azonosítók
 - más (egyszerűbb) kezelőfüggvények
 - **példákkal illusztrált különbségek**

SysV szemaforok

- Cél: folyamatok közötti szinkronizáció
 - P() és V() operátorok
 - szemaforcsoportok kezelése
- Alkalmazás
 - szemaforok létrehozása (vagy elérése)

```
sem_id = semget(kulcs, szám, opciók);
```

- az ops struktúrában leírt műveletek végrehajtása (részletek: man semop):

```
status = semop(sem_id, ops, ops_méret);
```

- egyszerre több művelet, több szemaforon is végrehajtható
- blokkoló és nem blokkoló P() operáció is lehetséges
- egyszerű tranzakciókezelésre is van lehetőség

Unix System V IPC: üzenetsorok

- Cél: folyamatok közötti adatátvitel
 - diszkrét, tipizált üzenetek
 - nincs címzés, üzenetszórás
- Alkalmazás
 - üzenetsor létrehozása (vagy elérése)

```
msgq_id = msgget(kulcs, opciók);
```

- üzenetküldés és -fogadás

POSIX MQ demó

```
msgsnd(msgq_id, msg, méret, opciók);
```

az msg tartalmaz egy típust is

```
msgrcv(msgq_id, msg, méret, típus, opciók);
```

a típus (egész szám) beállításával szűrést valósíthatunk meg

= 0 a következő üzenet (tetszőleges típusú)

> 0 a következő adott típusú üzenet

< 0 a következő üzenet, amelynek a típusa kisebb vagy egyenlő

Unix System V IPC: osztott memória

- Cél: folyamatok közötti egyszerű és gyors adatátvitel
 - PRAM modell szerint működik
- Alkalmazás + demó
 - osztott memória létrehozása (vagy elérése)

```
shm_id = shmget(kulcs, méret, opciók);
```

- hozzárendelés saját virtuális címtartományhoz

```
változó = (típus) shmat(...);
```

az adott változót hozzákötjük a visszakapott címhez

- lecsatolás

```
shmdt(cím);
```

- a kölcsönös kizárást meg kell valósítani (pl. szemaforokkal)

Érdekesség: Make Dragonfly BSD great again!

2017-03-23

```
$ uname -s
DragonFly
$ id
uid=1001(shm) gid=1001(shm) groups=1001(shm)
$ ./shellcode3
$ uname -s
FreeBSD
$ ipcs
Message Queues:
T      ID      KEY          MODE        OWNER        GROUP
Shared Memory:
T      ID      KEY          MODE        OWNER        GROUP
Semaphores:
T      ID      KEY          MODE        OWNER        GROUP
s  65536    4196472  --rw-----  shm        shm
$ ipcrm -S 4196472
$ id
uid=1001(shm) gid=1001(shm) groups=1001(shm)
$ ./final
# id
uid=0(root) gid=0(wheel) egid=1001(shm) groups=1001(shm)
# █
```

Hálózati (socket) kommunikáció

- Cél: címzéssel és protokollokkal támogatott adatátvitel
 - kliens – szerver architektúra
 - sokféle célra (egy gépen belül / gépek között)
 - megjelenés: BSD UNIX (Berkeley sockets)
 - később Windows Winsock, POSIX socket
- Fogalmak
 - hálózati csatoló avagy azonosító (socket): a kommunikáció végpontja
 - IP cím és portszám (l. hálózatok)
- Alkalmazás `sfid = socket(domén, típus, protokoll);`
`szerver: bind(sfid, cím, ...);`
`kliens: connect(sfid, cím, ...);`
`szerver: listen(sfid, sor_ajánlott_mérete);`
`szerver: accept(sfid, cím, ...);`
`send(sfid, üzenet, ...);`
`recv(sfid, üzenet, ...);`
`shutdown(sfd);`

Hálózati (socket) kommunikáció alkalmazása

Kliens program

socket()

connect()

send()

recv()

close()

Szerver program

sfd1 = socket()

bind(sfd1)

listen(sfd1)

while

sfd2 = accept(sfd1)

fork()

szülő: vissza a ciklusba

gyerek:

recv(sfd2)

send(sfd2)

close(sfd2)

exit()

Készítsünk egy egyszerű webszervert!

```
sfd1 = socket()  
bind(sfd1, ...)  
listen(sfd1, 10)  
  
while  
  
    sfd2 = accept(sfd1)  
    fork()  
  
    szülő:      close(sfd2)  
  
    gyerek:     recv(sfd2)  
                ...  
                send(sfd2)  
                close(sfd2)  
                exit()
```

- Létrehozza a csatolót
- A 8080-as porthoz köti
- Beállítja a várakozási sort
- Beérkező kérésre vár
- Elindít hozzá egy kiszolgálót
 - fork()
 - pthread_create()
- Fogadja a kérést
 - (A szülő már új kérésre vár.)
- Elküldi a választ
- Lezárja a kliens kapcsolatot
- A kiszolgáló kilép
 - (A szülő még fut.)

A szerver (túl)terhelésének kezelése

- A TCP/IP **háromfázisú kézfogást** használ a kapcsolat felépítésekor
 - a kialakuló kapcsolat először egy SYN állapotba kerül a szerveren (1.)
 - majd szerver (2.) és kliens (3.) nyugtázs után lép ESTABLISHED állapotba
- A párhuzamos kapcsolatok nyilvántartására két lehetőség van
 - egy LISTEN sor minden kapcsolat tárolására (az eredeti BSD megvalósítás)
 - külön-külön SYN és ACCEPT sorok a fenti két állapot számára (pl. [Linux](#))
 - ebben az esetben a `listen()` rendszerhívás az ACCEPT sorra vonatkozik
 - ha betelik az ACCEPT sor (azaz a SYN állapotból nem lehet oda átlépni)
 - a szerver nem nyugtázza a kliens kapcsolatát
 - ezért a kliens később (exponenciálisan növekvő várakozással) újra próbálkozni fog
 - ha közben sikerült helyet felszabadítani az ACCEPT sorban, akkor rendben
 - ha nem, akkor a szerver előbb-utóbb zárni fogja a kapcsolatot
- Az időegység alatt kiszolgálható kérések száma a fő kérdés
 - a `listen()` által beállított várakozási sor kezelheti a **rövid**, átmeneti túlterhelést
 - ha tartósan gyorsabban érkezhetnek be, mint ahogy kiszolgáljuk őket baj van
 - ha a baj a mi folyamatunkban van, akkor növelhetjük a kiszolgálási kapacitást
 - pl. új folyamatokat, vagy szálakat indítunk
 - előfordulhat, hogy kernel szinten telnek be a sorok (rendszerszintű gond)
 - ekkor a kernel kiszolgálás nélkül zárni fog a beérkező kapcsolatokat (nem jó)
 - a szerverkapacitás növelésével és terheléselosztással kezelhető a helyzet

Távoli eljáráshívás (RPC) a gyakorlatban



RPC a gyakorlatban

- Kommunikációs infrastruktúra
 - elosztott rendszer
 - hálózati kommunikáció
 - címzés: gép, taszk, eljárás
 - adatátvitel: paraméterek, eredmények
 - implementációfüggetlen adatátviteli formátum
 - képes az implementációs különbségek áthidalására
 - portmapper: a programazonosítók és a hálózati portok összerendelése
- RPC nyelv
 - a hívható eljárások és típusai (interfész) leírása
 - megnevezés, azonosító, paraméterek és visszatérési értékek
 - adattípusok és strukturált adatszerkezetek deklarációja
 - implementációfüggetlen
- Programozói támogatás
 - rpcgen: az interfészleírásból programkódot generál
 - kliens csonk
 - szerver csontváz

RPC interfészleírás és programgenerátor

- RPC nyelv (példa: date.x)

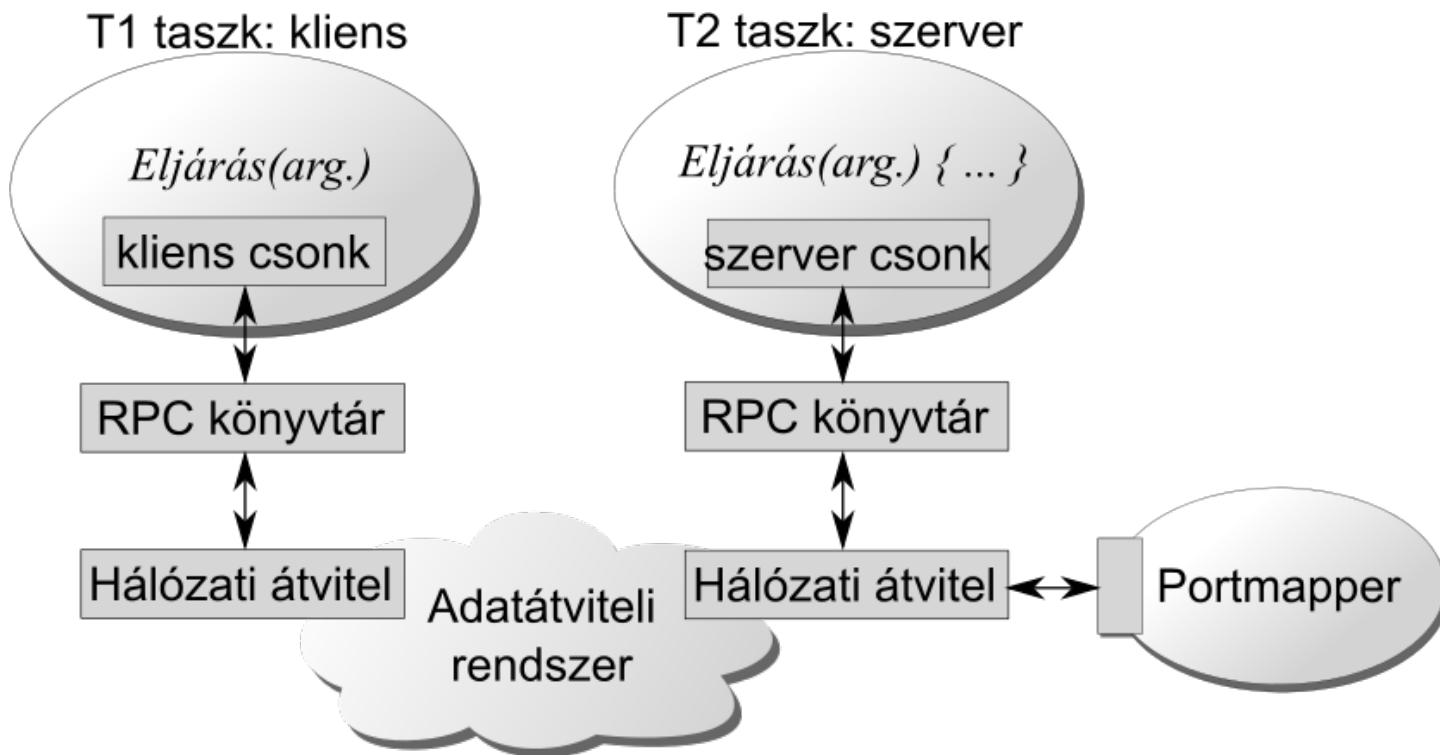
```
program DATE_PROG {
    version DATE_VERS {
        long BIN_DATE(void) = 1;      /* eljárás azon. = 1 */
        string STR_DATE(long) = 2;    /* eljárás azon. = 2 */
    } = 1;                            /* verziószám = 1 */
} = 0x31234567;                   /* program azon. = 0x31234567 */
```

- Kódgenerátor: rpcgen

rpcgen date.x eredményei

- date.h: **adattípusok deklarációja**
- date_clnt.c: a kliens kódjában felhasználható date_... függvények
- date_srv.c: a szerver date implementációját meghívó függvények
- date_xdr.c: adatkezelő eljárások architektúrafüggetlen XDR formátumhoz

A ONC (Sun) RPC részletes felépítése és működése



Az RPC-alapú és ihletésű megoldások

- OS szolgáltatások
 - pl. hálózati fájlrendszerek
NFS (Network File System)
- Elosztott alkalmazásfejlesztés (kitekintés)
 - pl. **CORBA**, DCOM, Java RMI, .NET Remoting, D-Bus, XML-RPC, SOAP stb.
 - interfészleíró nyelv (interface description/definition language, IDL)
 - implementációs nyelvtől független
de a nyelvi kötés definiált
 - széles körben használt
OMG IDL, WSDL, Microsoft MIDL, Facebook/Apache Thrift, LibreOffice UNO
 - szabványokban is
lásd W3C szabványok formális leírása, pl. [DOM IDL](#)
 - programkód generálása interfész leírásból
 - IDL leírás → forráskód
 - pl. CORBA IDL fordítók, IDL2Java, MS IDL fordítók stb.

Választás a kommunikációs lehetőségek között

- (Implementációs kötöttségek: programozási nyelv, környezet, stb.)
- A kommunikáció végpontjai szerint
 - számítógépen belül: mindegyik, RPC esetleg nem (nincs lokális portmapper)
 - számítógépek között: socket, RPC, hálózati diszkeken fájlon keresztül is
- A kommunikáció jellege
 - értesítés eseményekről: jelzések (SIGUSR1)
 - szinkronizálás: szemaforok
 - adatfolyam (pl.: csővezeték, socket) vs. diszkrét üzenetek (üzenetsorok)
 - üzenettípusok, szűrés? (üzenetsorok)
 - adatmennyiség (osztott memória: kicsi, csővezeték: közepes, socket: nagy)
- Teljesítmény
 - gyorsak: osztott memória, csővezeték, socket (PF_UNIX)
 - korlátos méret: osztott memória
- Kényelem
 - RPC, osztott memória (szemaforok?)

Programozási példák: <http://beej.us/guide/bgipc/>

A kommunikáció alapvető formái (összefoglalás)

- közös memórián keresztül
 - PRAM modell: sorrendezi (nem keveri) a kéréseket
 - hatékony
 - szinkronizáció szükséges
 - megvalósítás: szálak, SHMEM, mmap()
- üzenetváltás segítségével
 - Küld() és Fogad() műveletek
 - többféle címzés
 - szinkron és aszinkron működés
 - többféle adatátviteli szemantika
 - közvetítő komponens
 - késleltet, lassít
 - sokféle megvalósítás
 - jelzések: értesítés eseményekről
 - csővezetékek, üzenetsorok, hálózati kommunikáció: adatátvitel
 - távoli eljáráshívás: egy távoli taszk eljárásainak egyszerű meghívása

Operációs rendszerek: taszkok szinkronizációja

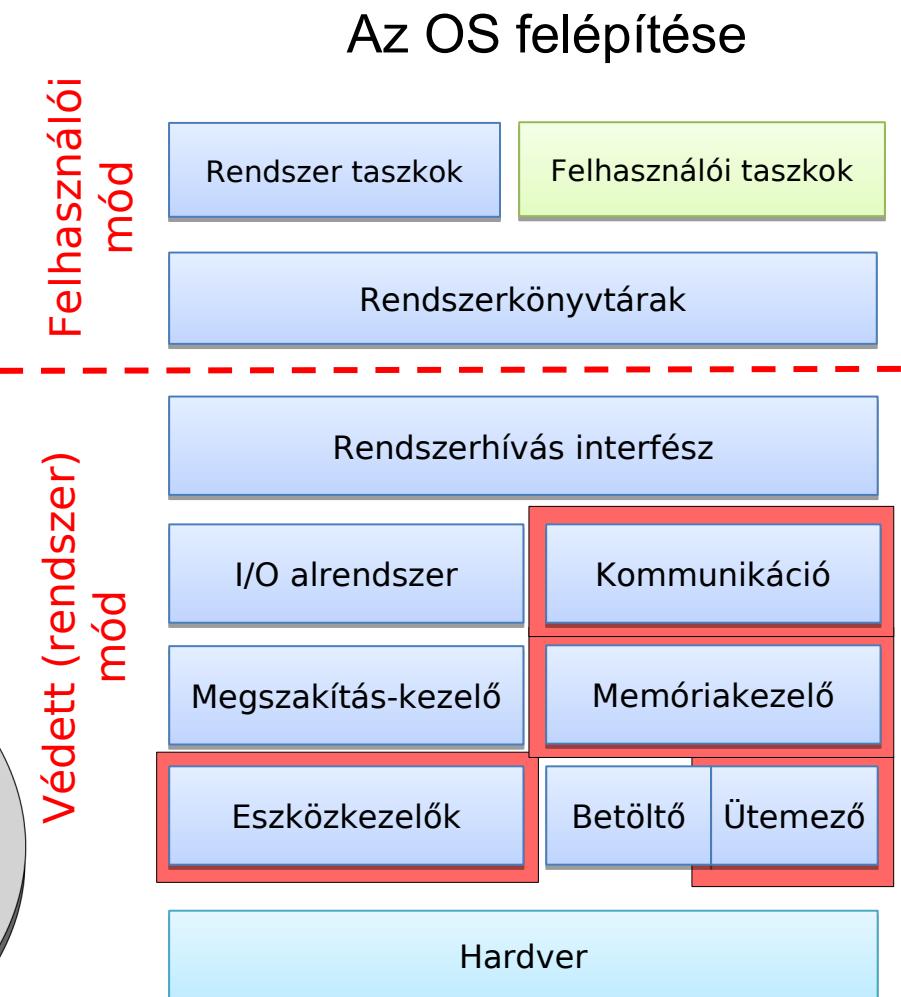
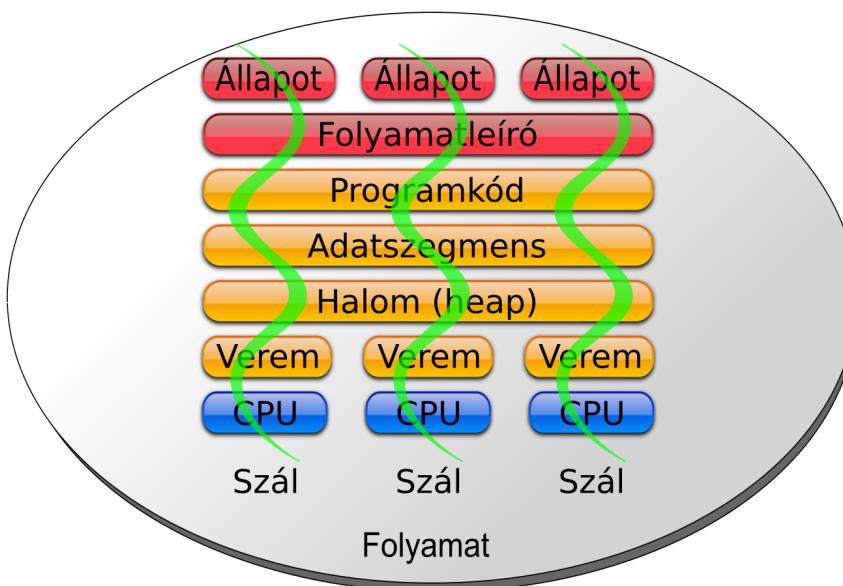
Mészáros Tamás
<http://www.mit.bme.hu/~meszaros/>

Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

Az előadásfóliák legfrissebb változata a tantárgy honlapján érhető el.
Az előadásanyagok BME-n kívüli felhasználása és más rendszerekben történő közzététele előzetes engedélyhez kötött.

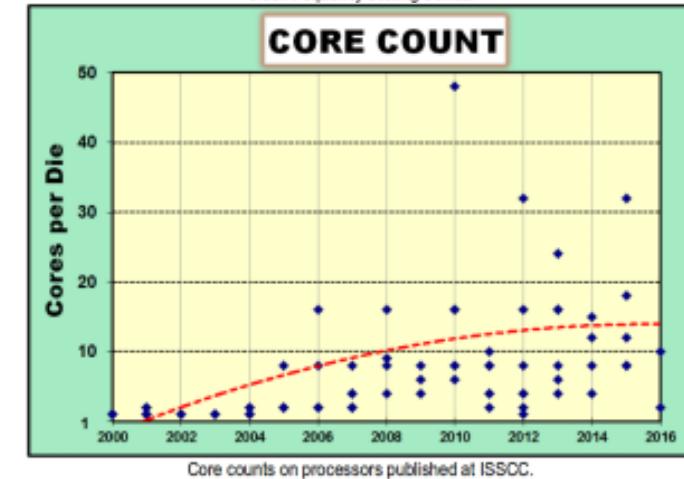
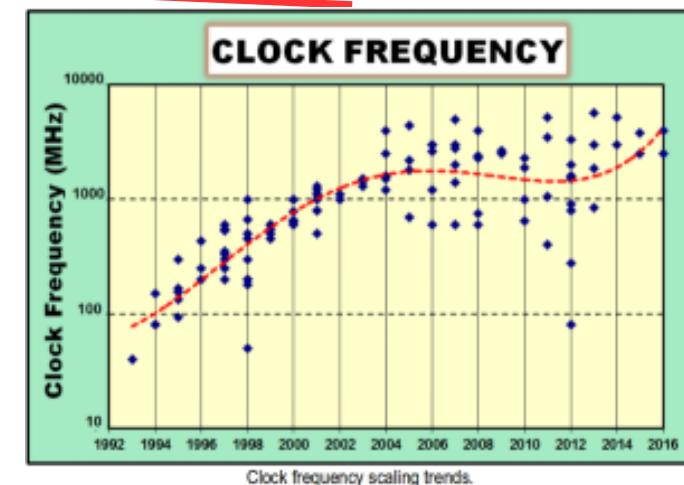
Az eddigiekben történt...

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



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



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

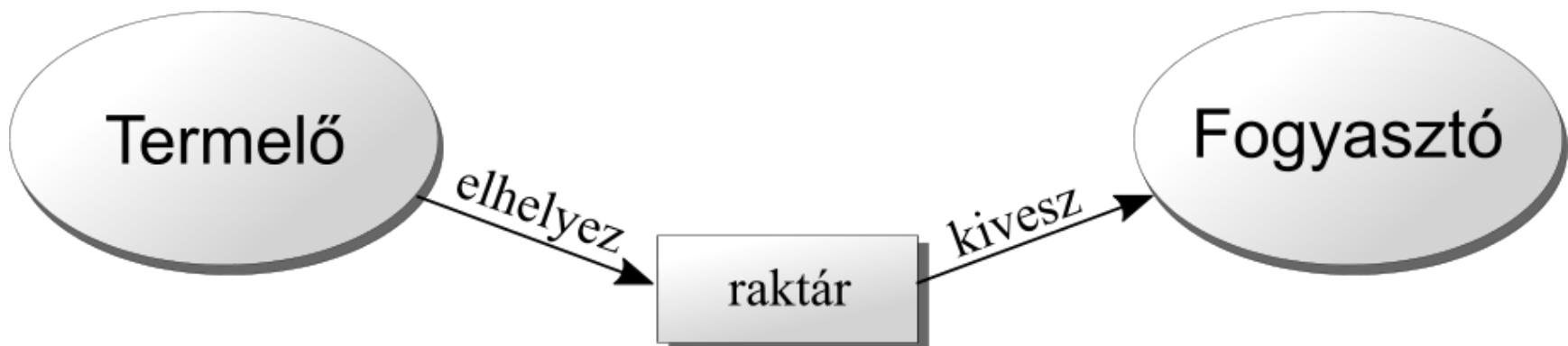
Taszkok együttműködése és versengése

- Együttműködő taszkok
 - részekre bontott feladat
 - a taszkok **kooperálnak**
 - adatcsere, végrehajtási függőségek
 - versenyhelyzetek alakulhatnak ki
 - az OS védelmi mechanizmusai **nehezítik** a megvalósítást → kommunikáció
- Független taszkok
 - (nincs szükségük egymásra)
 - aszinkron végrehajtás
 - de **multiprogramozott rendszert** alkotnak
 - 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.

A programozó valósítja meg.

Az OS valósítja meg.

Egyszerű példa: a termelő-fogyasztó probléma



- Párhuzamosan működnek
 - különböző ütemben
 - eltérő sebességgel
 - Megoldandó feladatok
 - a raktár konzisztenciája (PRAM modell)
 - a Fogyasztó blokkolása üres raktár esetén
 - a Termelő blokkolása tele raktár esetén
- konkurens programozás
- 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

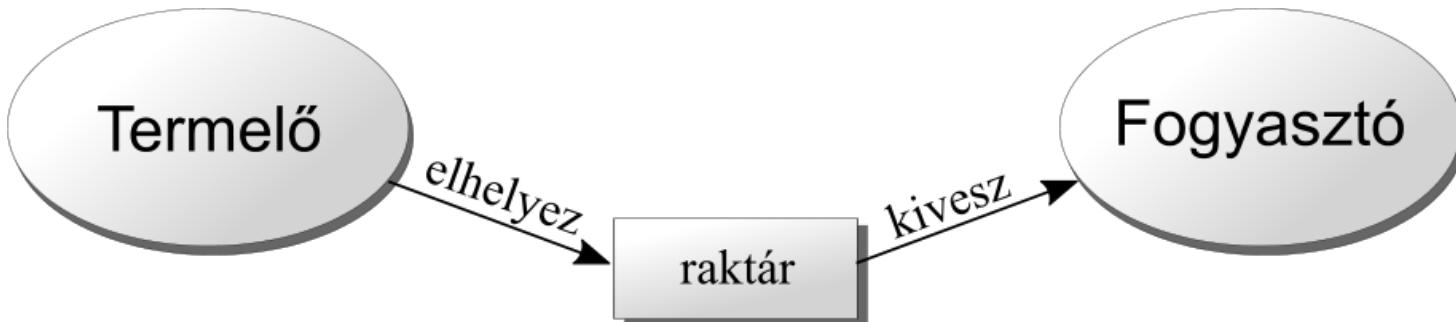
- alulterv
 - he
- felülter
 - fel
 - roi

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

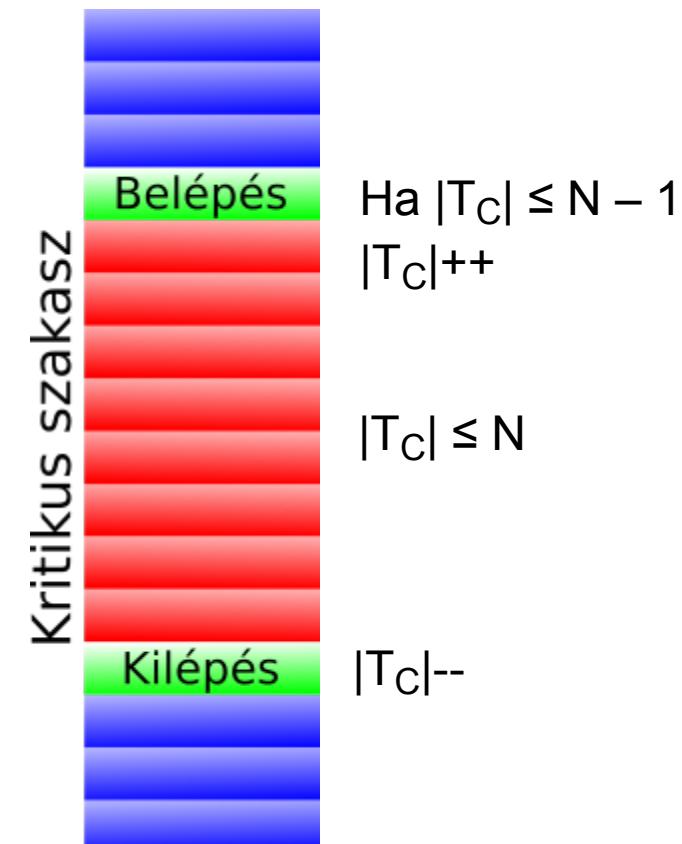
A szinkronizáció megvalósítása



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 belépéssben
 - 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 átütémezés, 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 értékét IGAZ-ra és visszaadja a régit
 - alkalmazása:
 $\text{while } (\text{TSL(zár}) \{ \}$ 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:
 $\text{while } (\text{CAS(zár, a, b) } != \text{ a}) \{ \}$ Hogyan működik?

„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
```

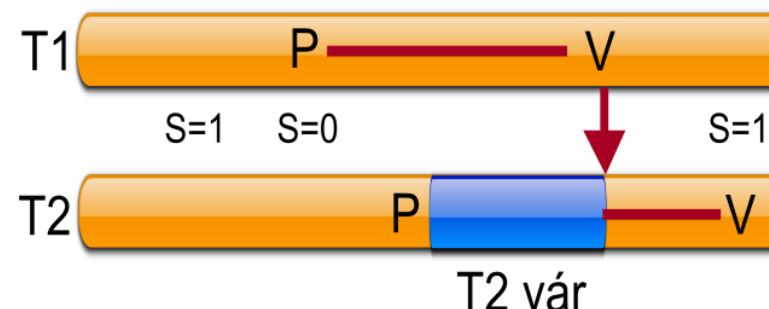
- Mi a gond ezekkel?
 - a „spinning”: folyamatosan fut a taszk („busy waiting”)
 - jobb lenne egy blokkoló művelet, hogy a taszk várakozó állapotba kerüljön.
- Mikor / miért jó mégis?
 - a hardver támogatja (x86: **XCHG** és **CMPXCHG**)
 - ha tudjuk, hogy nem kell sokat várni (pl. a kernelben)

Zárolási eszközök áttekintése

- Lock bit
 - egy bites, 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
```

```
...
```

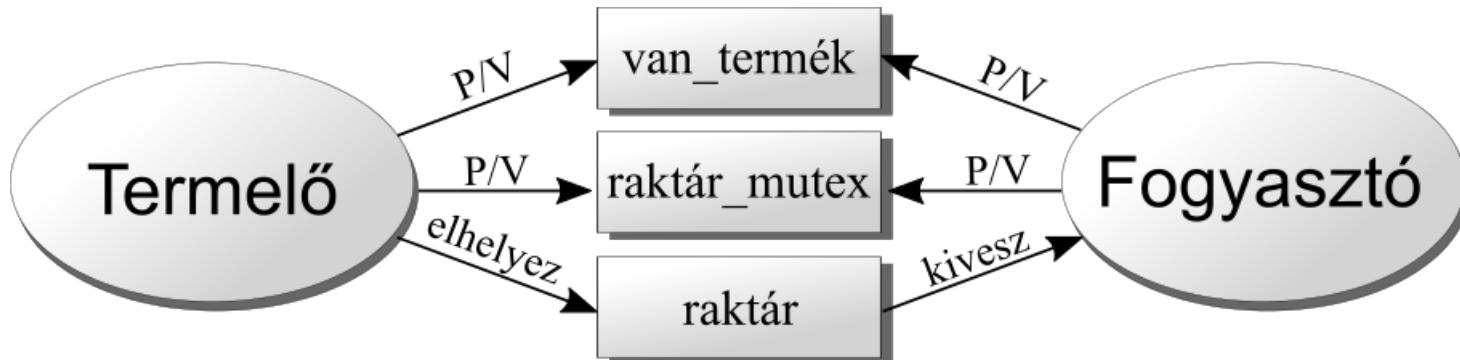
- feloldás

```
// 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);
}
```

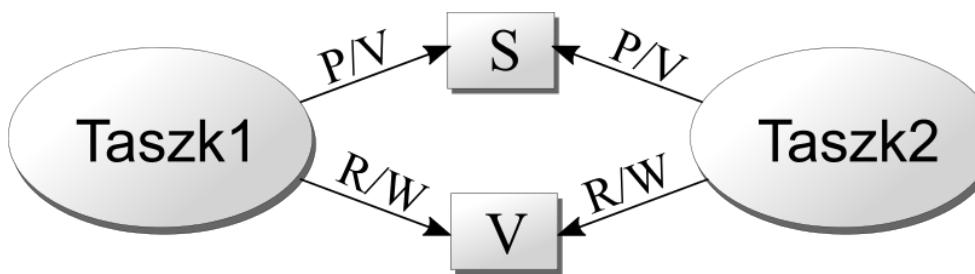
Fogyasztó

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

Otthoni gyakorlás: szemaforral hogyan kezelhető a raktár túlcordulásvédelme?

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);

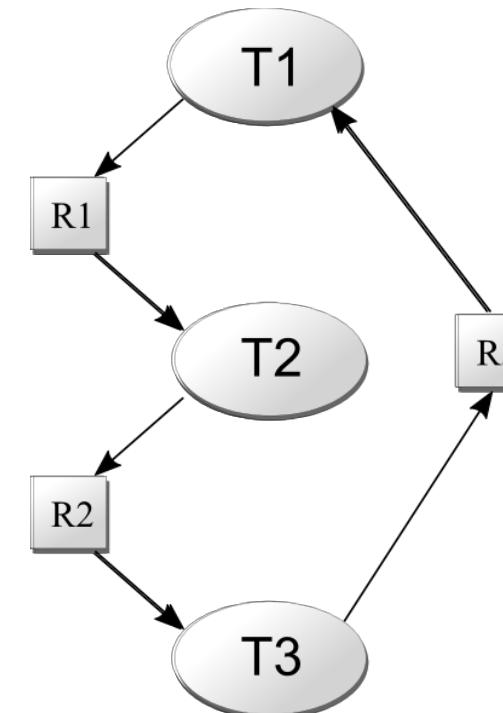
T2

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?
Holppont 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?

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

T1

P (R3) ;

P (R1) ;

V (R1) ;

V (R3) ;

T2

P (R1) ;

P (R2) ;

V (R2) ;

V (R1) ;

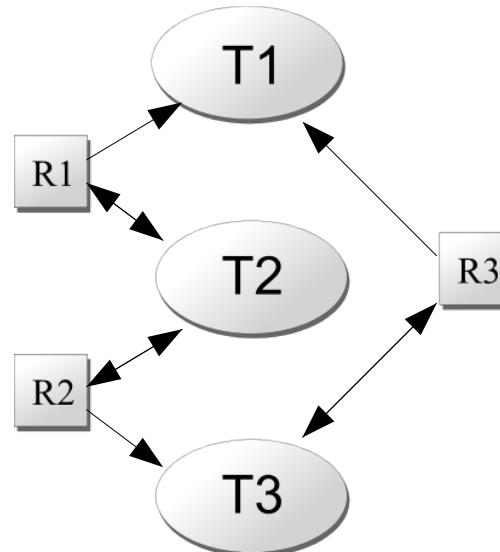
T3

P (R2) ;

P (R3) ;

V (R3) ;

V (R2) ;



Holtpont kialakulásának feltételei

1. kölcsönös kizárási feltétel

legyenek kizárolagosan 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 kezelní
 - detektáljuk a holtpontot
 - feloldjuk
 - csak erőszakosan megy: leállítunk taszko(ka)t, elveszünk erőforrás(ok)a)t
 - ki? hogyan? mit eredményez?
 - a programozó talán tudja...
- Védekezünk ellene
 - **holtpontmentesre tervezzük**
 - valamelyik feltételt kizártuk (melyiket lehet?)
 - futásidőben ellenőrizzük a foglalásokat
 - még kialakulása előtt detektáljuk
 - **biztonságos állapot**: holtpont nélkül erőforrást allokálhatunk

← Ez a helyes út.

Holtpont kialakulásának detektálása / megelőzése

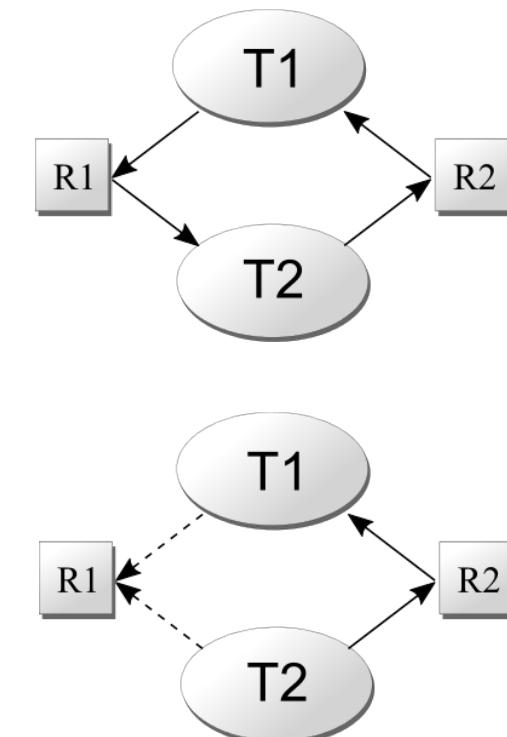
- A rendszer kiindulási állapota biztonságos
 - nincs erőforrás-foglalás, nincs holtpont
 - az előrejelzéshez bekérjük a jövőbeli foglalásokat is

- 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. $R1 \rightarrow T2$ 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

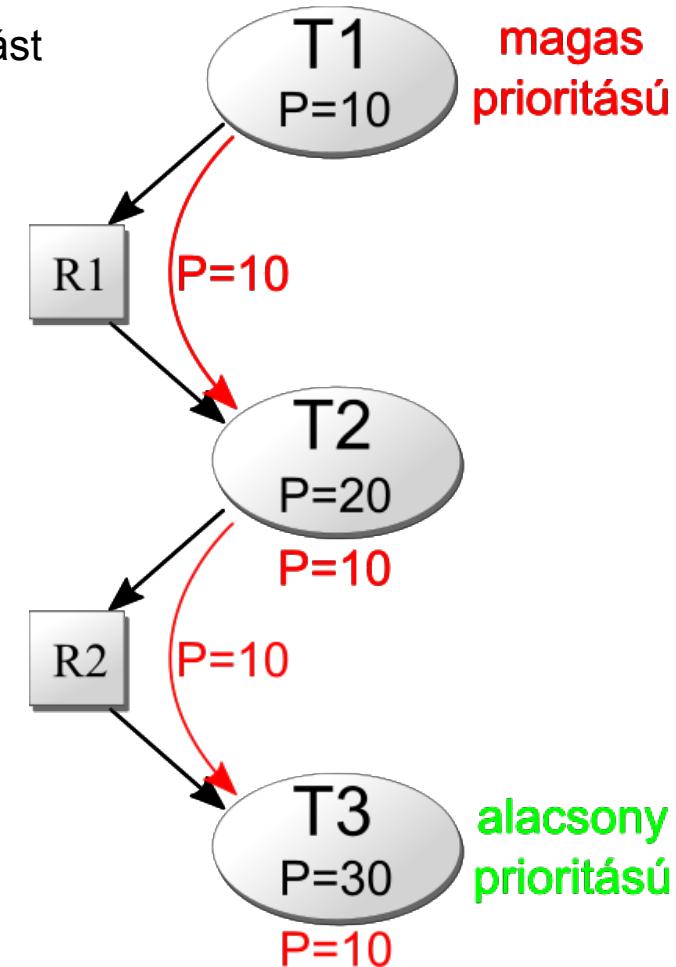
- egy taszk folyamatosan foglal egy erőforrást
- az erőforrásra várók blokkolódnak
- feloldása: a hibás működés javítása

• 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:** minden véd
- **Optimista zárolás:** nem zárol, de detektálja és korrigálja a hibát
- Tranzakció-alapú megvalósítása
 - BEGIN: feljegyzi a kiinduló állapotot
 - MODIFY: végrehoztatja 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.) minden művelet minden művelet véges időn belül végrehajtható
 - **garantálja az erőforrás teljes kihasználtságát**, nincs teljesítményvesztés
 - **Várakozásmentes** (wait-free) erőforrás-használat
 - zárolásmentes + minden művelet minden művelet véges időn belül végrehajtható
 - sokkal nehezebb megvalósítani
- Kizáják 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

Feladatok együttműködésének ellenőrzése

dr. Vörös András

<https://inf.mit.bme.hu/members/vorosa>

dr. Micskei Zoltán

<https://inf.mit.bme.hu/members/micskeiz>



Méréstechnika és
Információs Rendszerek
Tanszék

- 2003. augusztus 14:
áramszünet Észak-Amerikában
 - **8 állam**, kb. **45 millió** ember maradt áram nélkül
 - a teljes áramellátó rendszer percek alatt összeomlott
 - a helyreállítás több mint **egy hétag** tartott
 - 265 erőmű esett ki, 61,8 GW összteljesítményben
(≈125 paksi reaktorblokk teljesítménye)
 - leállt a közlekedés, az ivóvíz-ellátás, az olajfinomítók, a gyárak, fennakadások voltak a kommunikációban
 - felbecsülhetetlen anyagi kár, számos halálos áldozat



Forrás: http://en.wikipedia.org/wiki/Northeast_Blackout_of_2003

A hiba oka

- kiesett az Eastlake erőmű egyik blokkja és három távvezeték,
- túlterhelődtek a távvezetékek,
- a First Energy operátorai nem tudták megfelelően kezelní a helyzetet ...
- ... mert nem működött az informatikai rendszerük órákon át (és erről nem is tudtak) ...
- ... mert **versenyhelyzet** volt a GE Unix-alapú rendszerében (több millió kódsor)
- *az észak-amerikai villamosenergia-rendszer összértéke kb. 1 millió USD
(≈273 millió HUF akkori árfolyamon, 100 évnyi magyar GDP)*

Kölcsönös kizárási példa

kliens _i	szerver
<ol style="list-style-type: none">1 ... other activity ...2 $r_i := \text{TRUE}$3 wait until $g_i = \text{TRUE}$4 critical section5 $r_i := \text{FALSE}$6 go to 1	<ol style="list-style-type: none">1 wait until at least one r_i is <i>TRUE</i>2 let c be such that $r_c = \text{TRUE}$3 $g_c := \text{TRUE}$4 wait until $r_c = \text{FALSE}$5 $g_c := \text{FALSE}$6 go to 1

Algoritmusok helyességének ellenőrzése

- Hogyan döntsük el, hogy jó?
- Erősen nézzük, és próbálunk rájönni :)
- Végigpróbálunk néhány lefutást
 - Ha hibázik: javítjuk a kódot

Kölcsönös kizárási példa

kliens _i	szerver
<ol style="list-style-type: none">1 ... other activity ...2 $r_i := \text{TRUE}$3 wait until $g_i = \text{TRUE}$4 critical section5 $r_i := \text{FALSE}$6 go to 1	<ol style="list-style-type: none">1 wait until at least one r_i is <i>TRUE</i>2 let c be such that $r_c = \text{TRUE}$3 $g_c := \text{TRUE}$4 wait until $r_c = \text{FALSE}$5 $g_c := \text{FALSE}$6 go to 1

Kölcsönös kizárást példa

kliens_i

1 ... other activity ...

2 $r_i := \text{TRUE}$

3 ~~wait until $g_i = \text{TRUE}$~~

4 critical section

5 $r_i := \text{FALSE}$

6 go to 1

```
public void run() {  
    while (true) {  
        other_activity();  
        RaceCond.set_request(id);  
        while (RaceCond.get_grant(id)  
              == false)  
            ;  
        critical_section();  
        RaceCond.release_request(id);  
        other_activity();  
    }  
}
```

Kölcsönös kizárást példa

```
int chosen_id = -1;  
for (int i = 0; i < client_number;  
i++) {  
if (RaceCond.get_request(i) ==  
true) {  
chosen_id = i;  
break;}}  
if (chosen_id > -1) {  
RaceCond.set_grant(chosen_id);  
while (RaceCond.  
        get_request(chosen_id)  
        == true);  
RaceCond.  
        release_grant(chosen_id);  
}
```

szerver

- 1 **wait until** at least one r_i is *TRUE*
- 2 let c be such that $r_c = \text{TRUE}$
- 3 $g_c := \text{TRUE}$
- 4 **wait until** $r_c = \text{FALSE}$
- 5 $g_c := \text{FALSE}$
- 6 **go to 1**

DEMO

- Kérdés: helyes-e az algoritmus?
 - Biztosítja-e a kölcsönös kizárást?

Algoritmusok helyességének ellenőrzése

- Hogyan döntsük el, hogy jó?
- Erősen nézzük, és próbálunk rájönni :)
- Végigpróbálunk néhány lefutást
 - Ha hibázik: javítjuk a kódot
 - **Ha nem találunk hibát: ??**
- Szisztematikus megoldás kell:
 - „formális módszerek”

Algoritmusok helyességének ellenőrzése

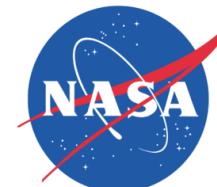
■ Milyen jó lenne egy eszköz:

- Algoritmusaink egyszerű **leírására**
- Rendszer működésének **szimulálására**
- Összetett követelmények **megfogalmazására**
- Követelmények **ellenőrzésére** gombnyomásra



■ Jó hír: vannak ilyen eszközök ☺

- **Modelellenőrzők** (model checkers)
- 30+ év kutatás eredménye
- Valós ipari eredmények HW és SW rendszereknél



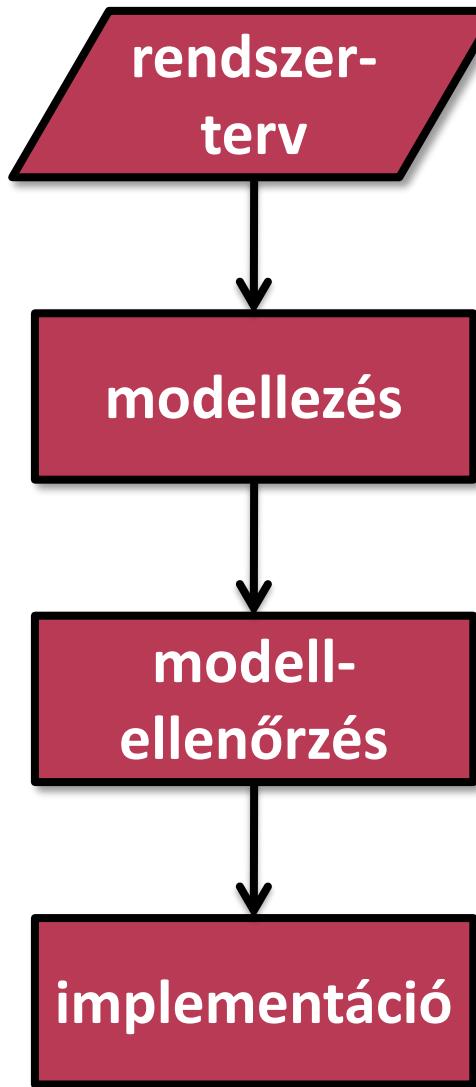
Turing award

- Edmund M. Clarke,
- E. Allen Emerson,
- Joseph Sifakis

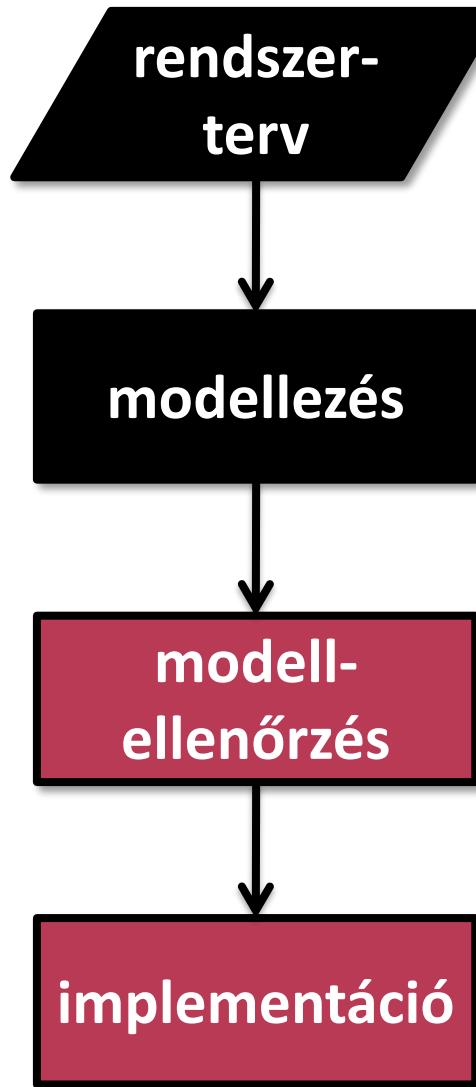


- „For their role in developing Model-Checking into a highly effective verification technology that is widely adopted in the hardware and software industries.”

Modellvezérelt fejlesztés



Modellvezérelt fejlesztés



Rendszterterv

■ Pszeudo kód

kliens;	szerver
<ol style="list-style-type: none">1 ... other activity ...2 $r_i := \text{TRUE}$3 wait until $g_i = \text{TRUE}$4 critical section5 $r_i := \text{FALSE}$6 go to 1	<ol style="list-style-type: none">1 wait until at least one r_i is <i>TRUE</i>2 let c be such that $r_c = \text{TRUE}$3 $g_c := \text{TRUE}$4 wait until $r_c = \text{FALSE}$5 $g_c := \text{FALSE}$6 go to 1

Rendszterterv

```

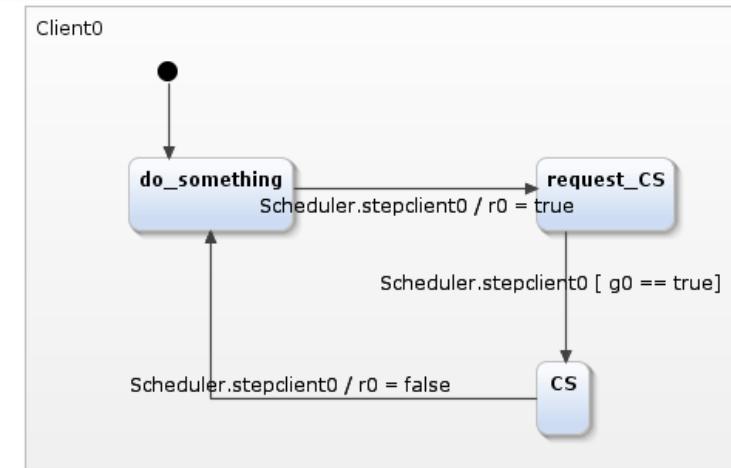
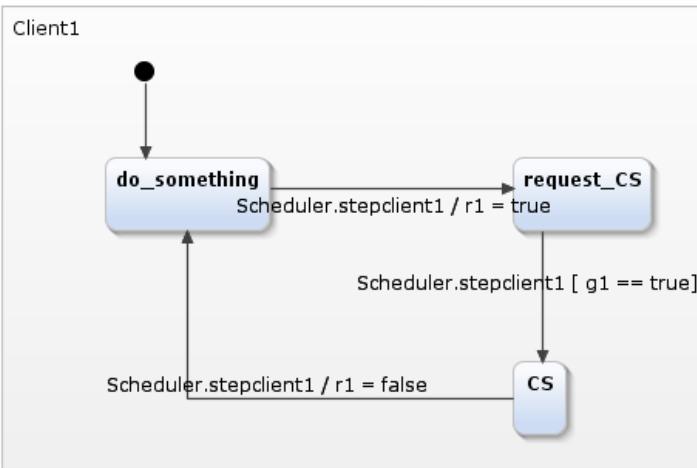
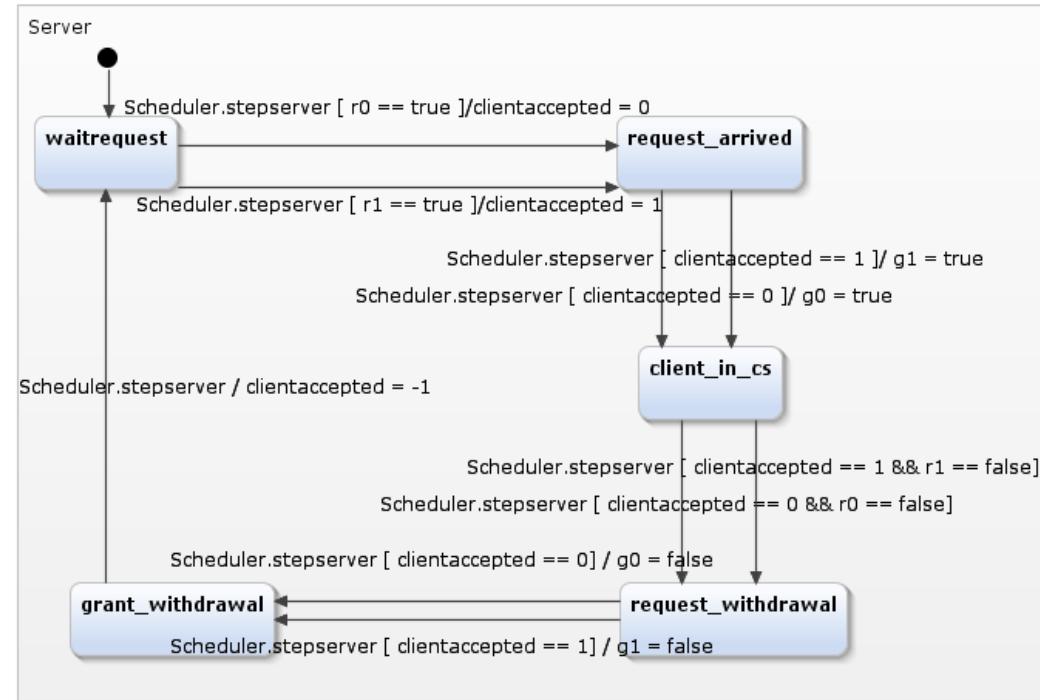
simplemutex
*****
Scheduler interface
*****
interface Scheduler:
in event stepserver
in event stepclient0
in event stepclient1

internal:
*****
internal variables:
*****
//variable c in pseudo code
var clientaccepted:integer = -1

//request variables
var r0:boolean = false
var r1:boolean = false

//grant variables
var g0:boolean = false
var g1:boolean = false

```



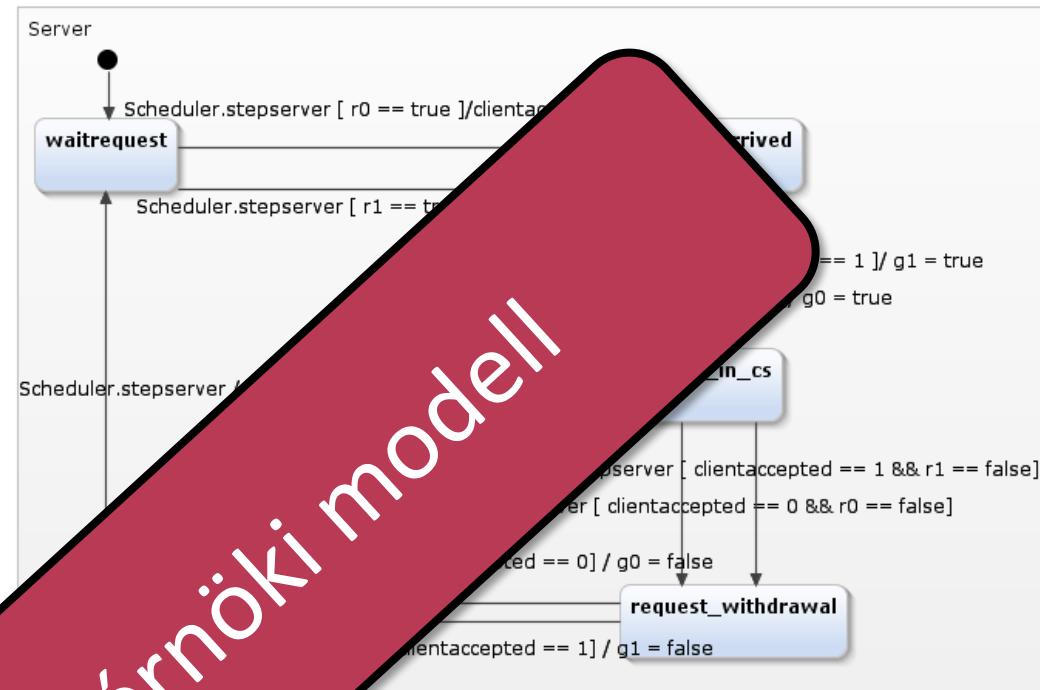
Rendszterterv

```
simplemutex
*****
Scheduler interface
*****
interface Scheduler:
in event stepserver
in event stepclient0
in event stepclient1

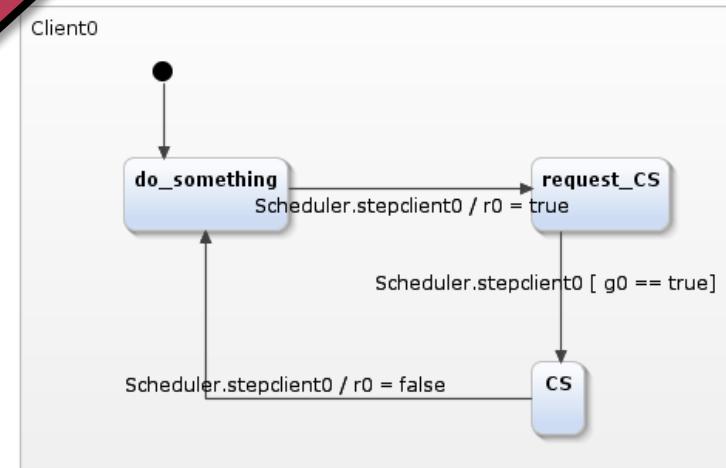
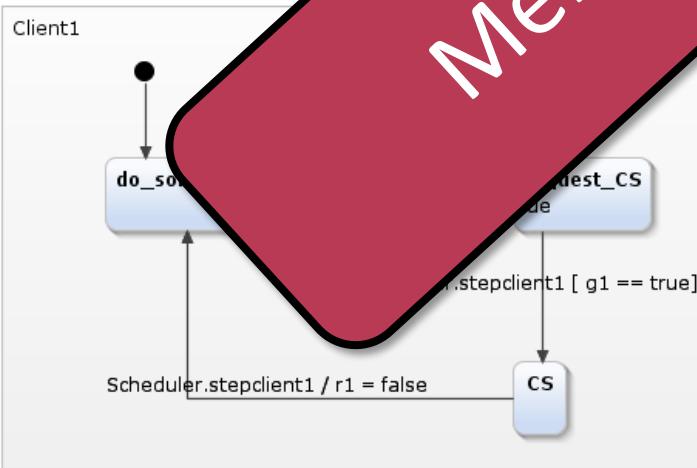
internal:
*****
internal variables:
*****
//variable c in pseudo code
var clientaccepted:integer = -1

//request variables
var r0:boolean = false
var r1:boolean = false

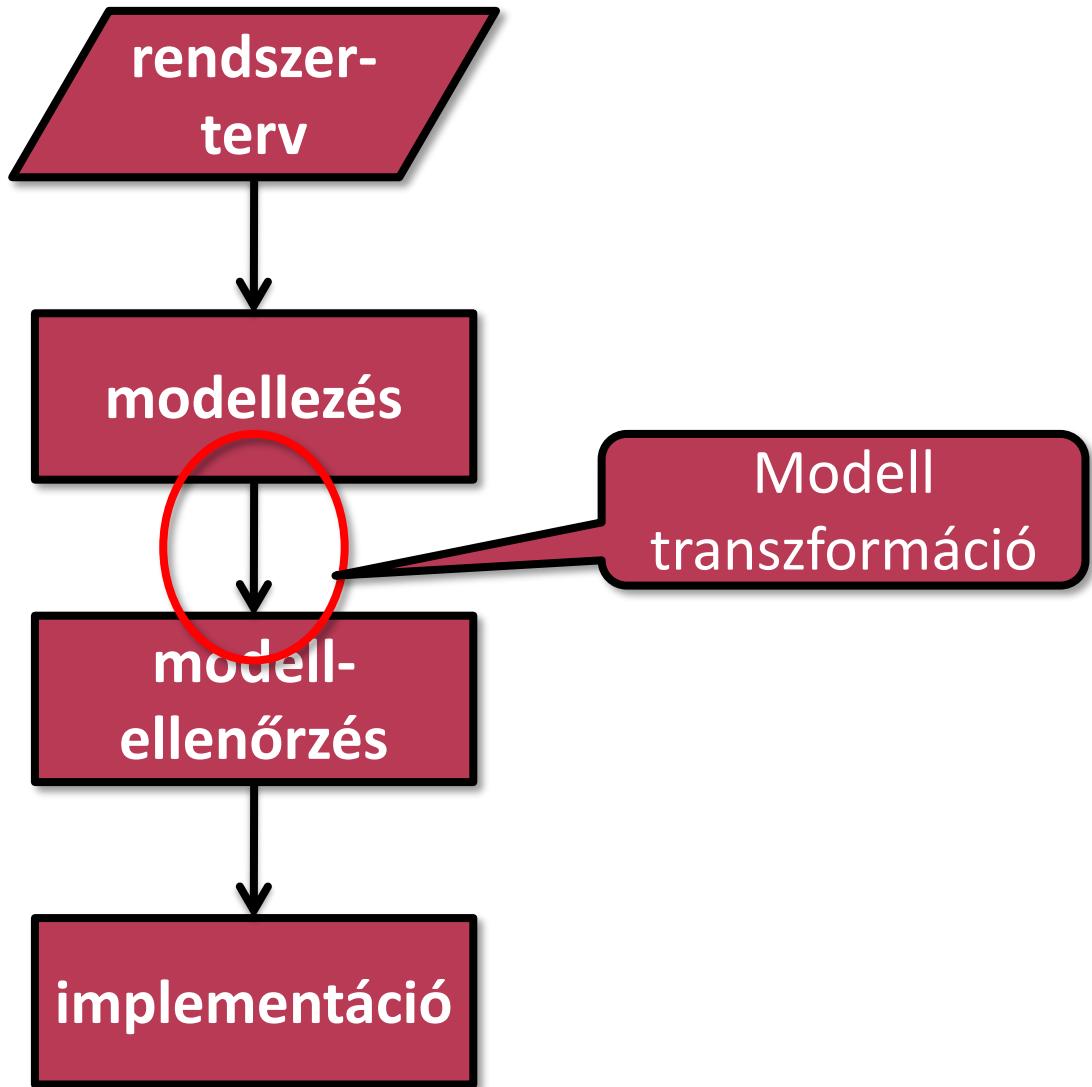
//grant variables
var g0:boolean = false
var g1:boolean = false
```



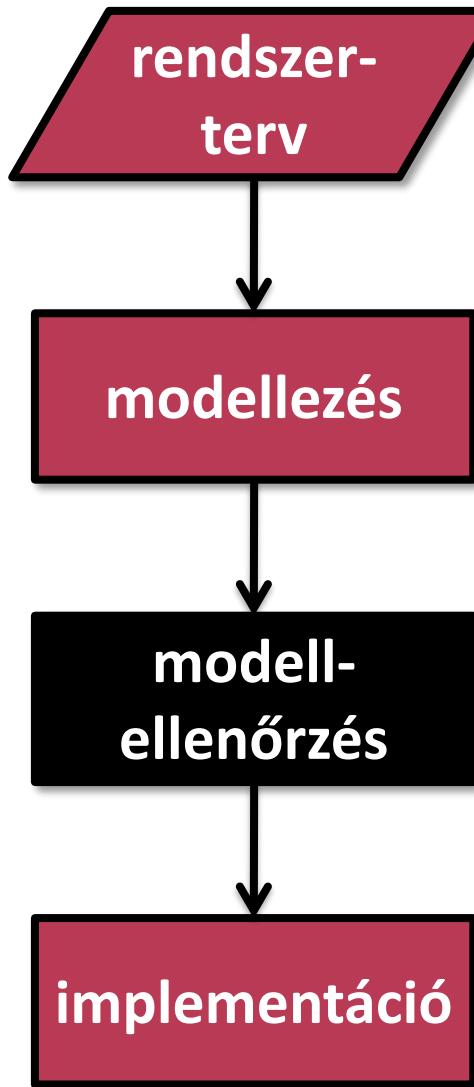
Mérnöki modell



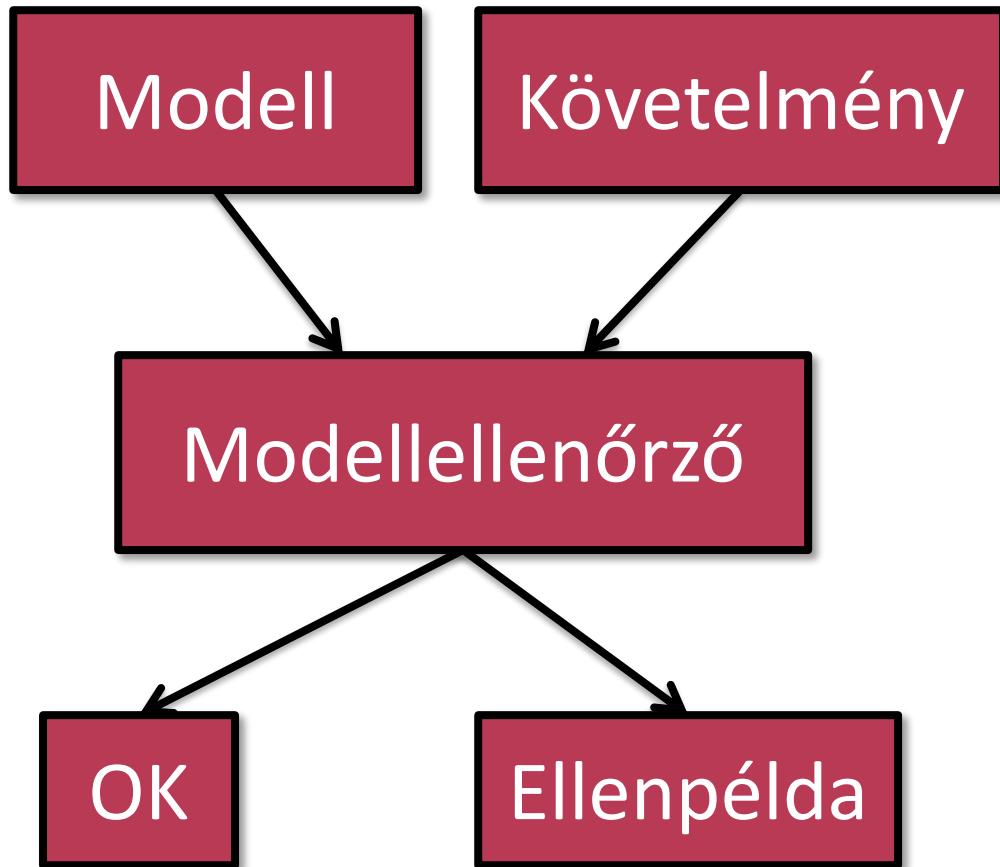
Modellvezérelt fejlesztés



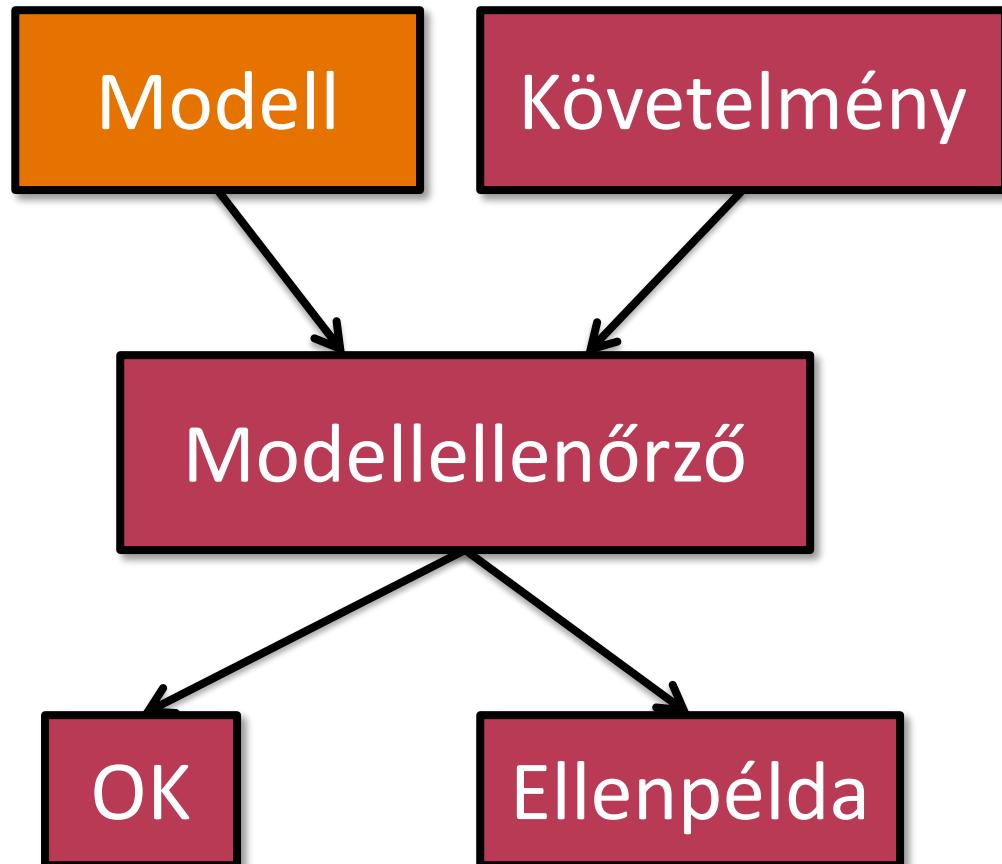
Modellvezérelt fejlesztés



Modellellenőrzők



Modellellenőrzők

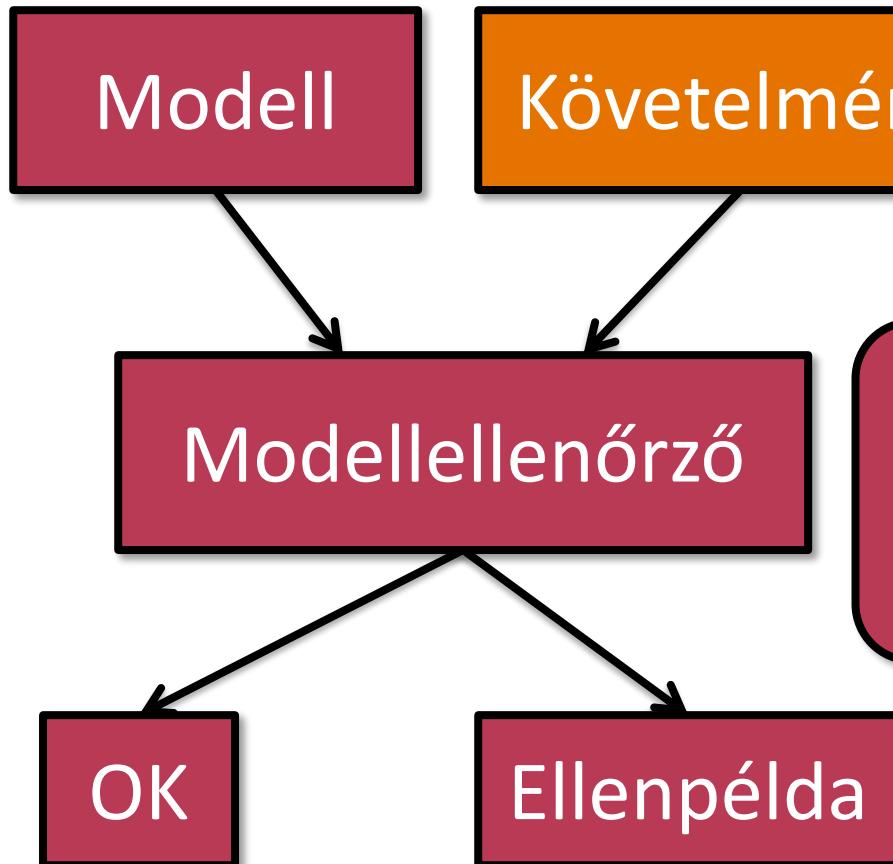


- Rendszer működésének leírása
- Formális modell
- Tipikusan állapotgépszerű

De: lehet program is!

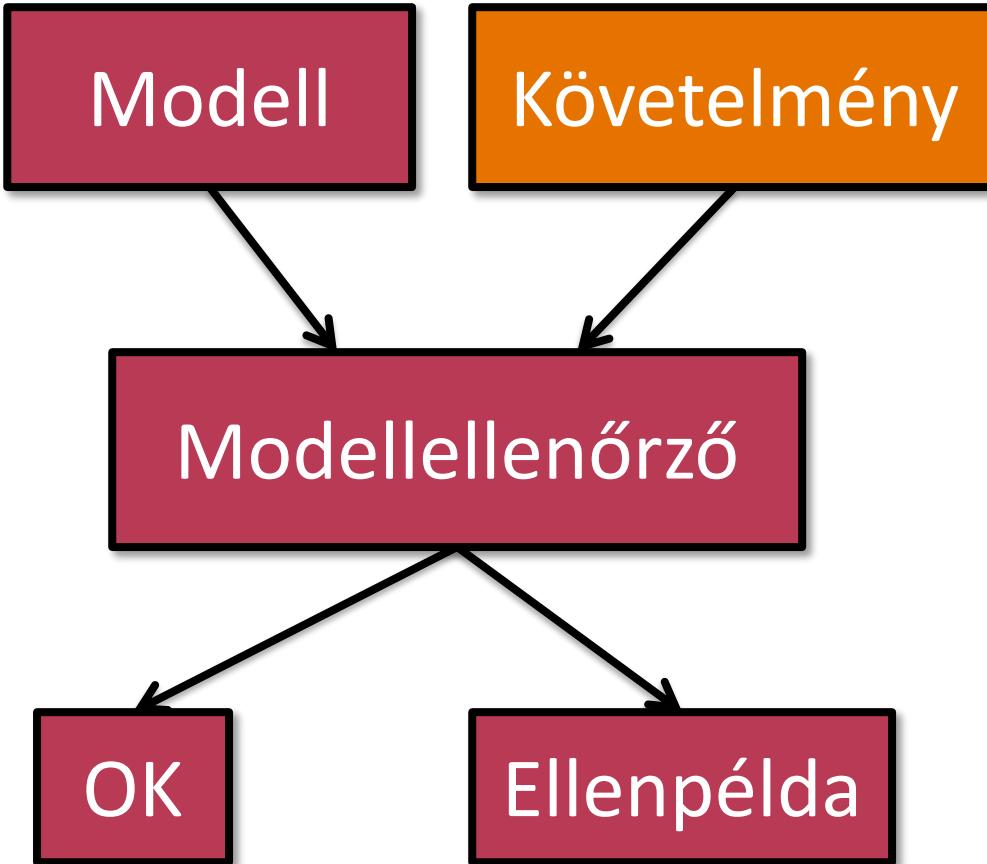


Modellellenőrzők



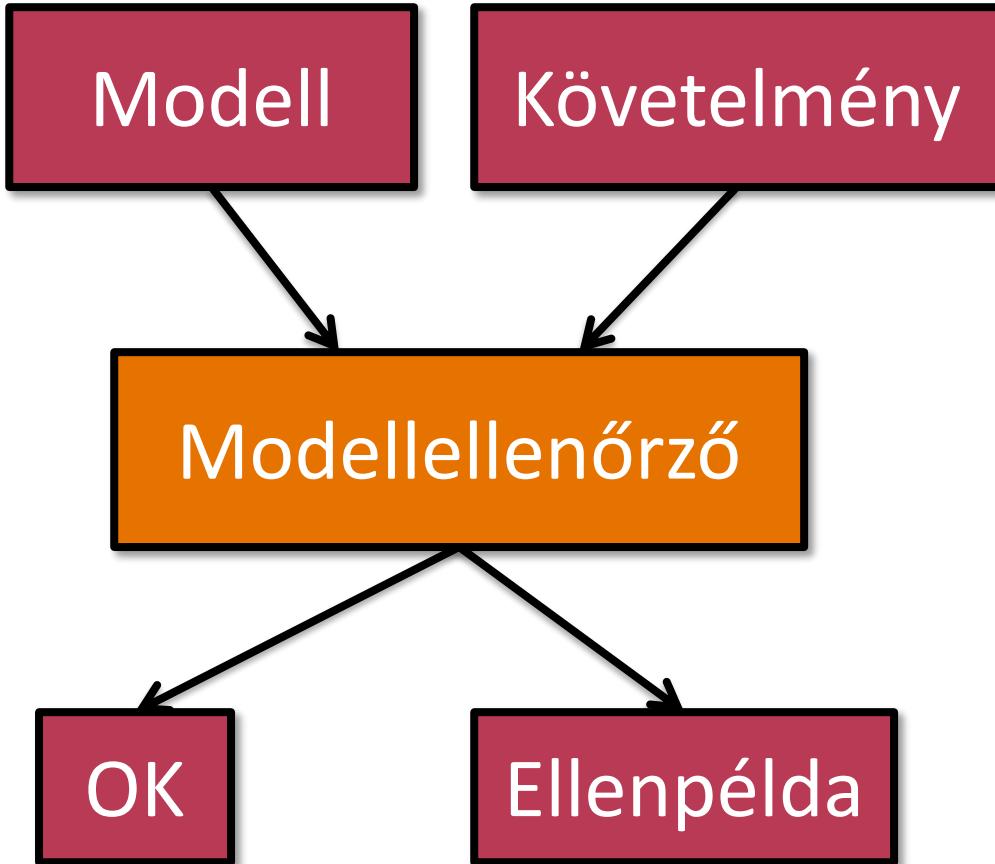
“A design without specification
cannot be right or wrong, it
can only be surprising!”
– Young et al., 1985

Modellellenőrzők



- Mit akarunk ellenőrizni
 - Kölcsönös kizárási
 - Holtpont mentesség
 - ...
- Logikai kifejezés:
 - Pl.:
 $! (A_var \text{ AND } B_var)$

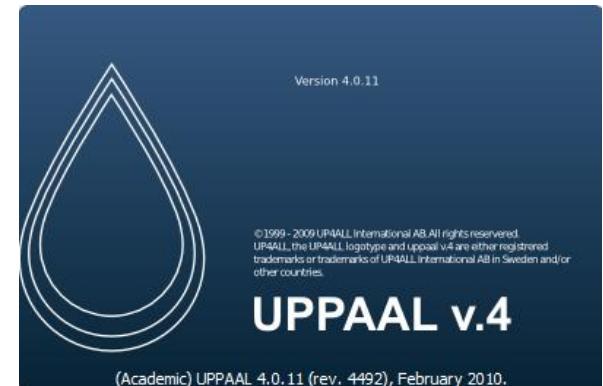
Modellellenőrzők



- „Fekete doboz”
- Automatikus
- Eredmény:
 - Kovenantmény igaz
 - Kovenantmény nem teljesül + ellenpélda

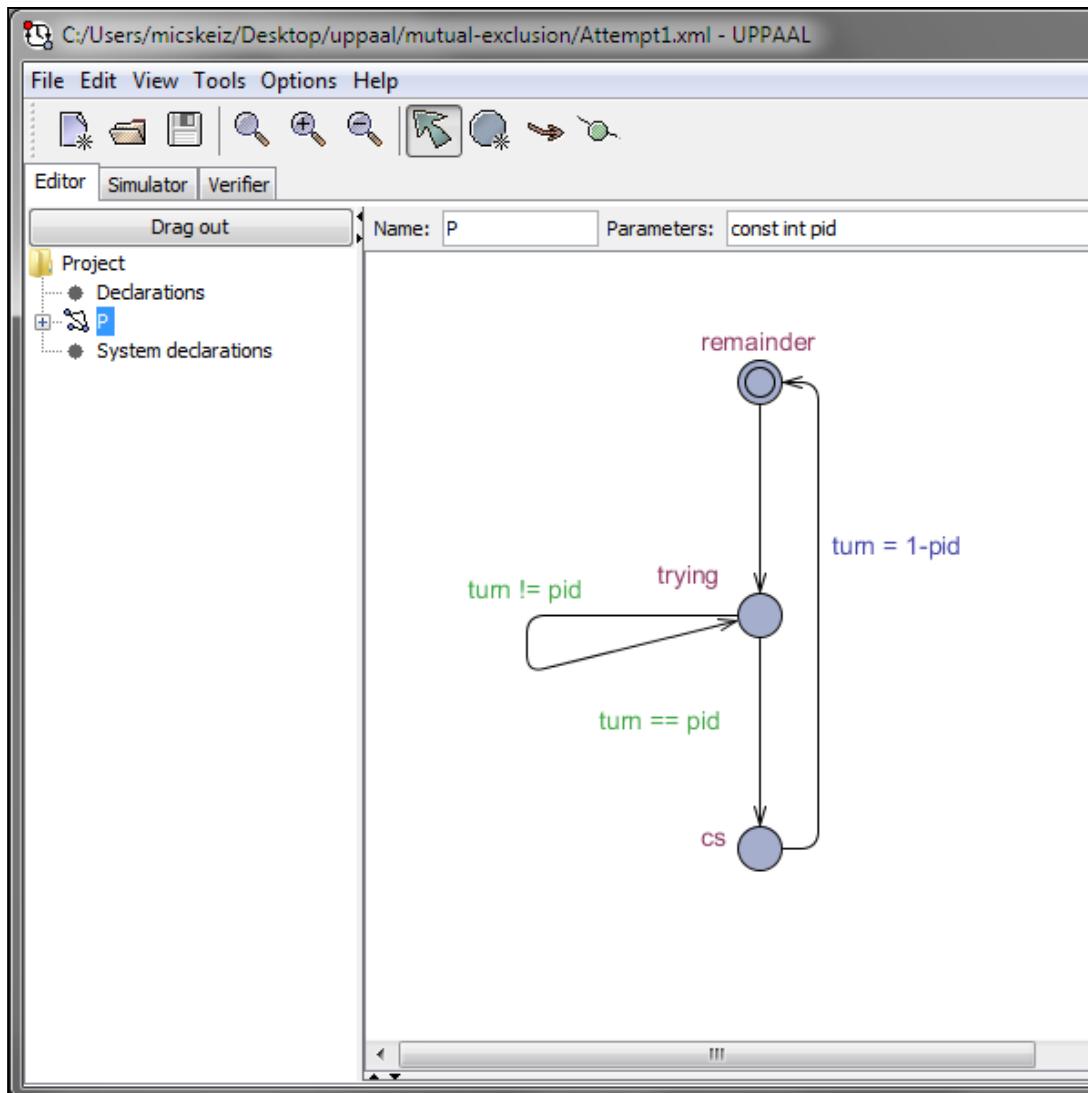
UPPAAL

- Időzítést is támogató modellellenőrző
- Uppsala & Aalborg egyetemek, 15+ éve fejlesztik
- Cél: hatékonyság, könnyű használhatóság
- <http://www.uppaal.com/>
 - Akadémiai célra ingyenesen letölthető
 - Leírások
 - Részletes súgó
 - Esettanulmányok
 - Sok kiegészítés (tesztgenerálás)



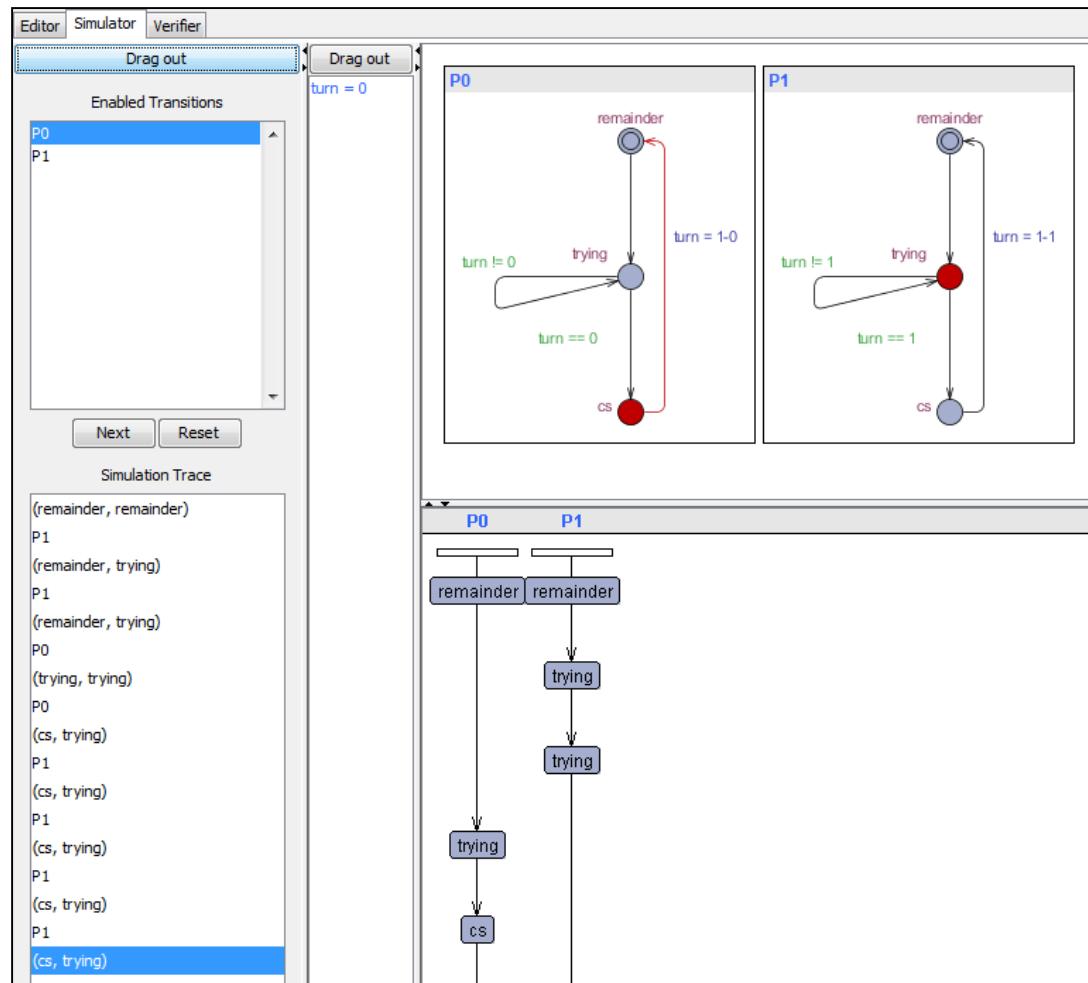
- Példa modell megnyitása
 - OPRE-hoz kapcsolódó modellek:
<http://www.mbsd.cs.ru.nl/publications/papers/fvaan/MCinEdu/>
- Deklarációk megnézése
- Szimulátor:
 - Modell „animálása”
 - Végrehajtás visszajátszása
 - Véletlenszerű végrehajtás

Az UPPAAL felülete: modell szerkesztő



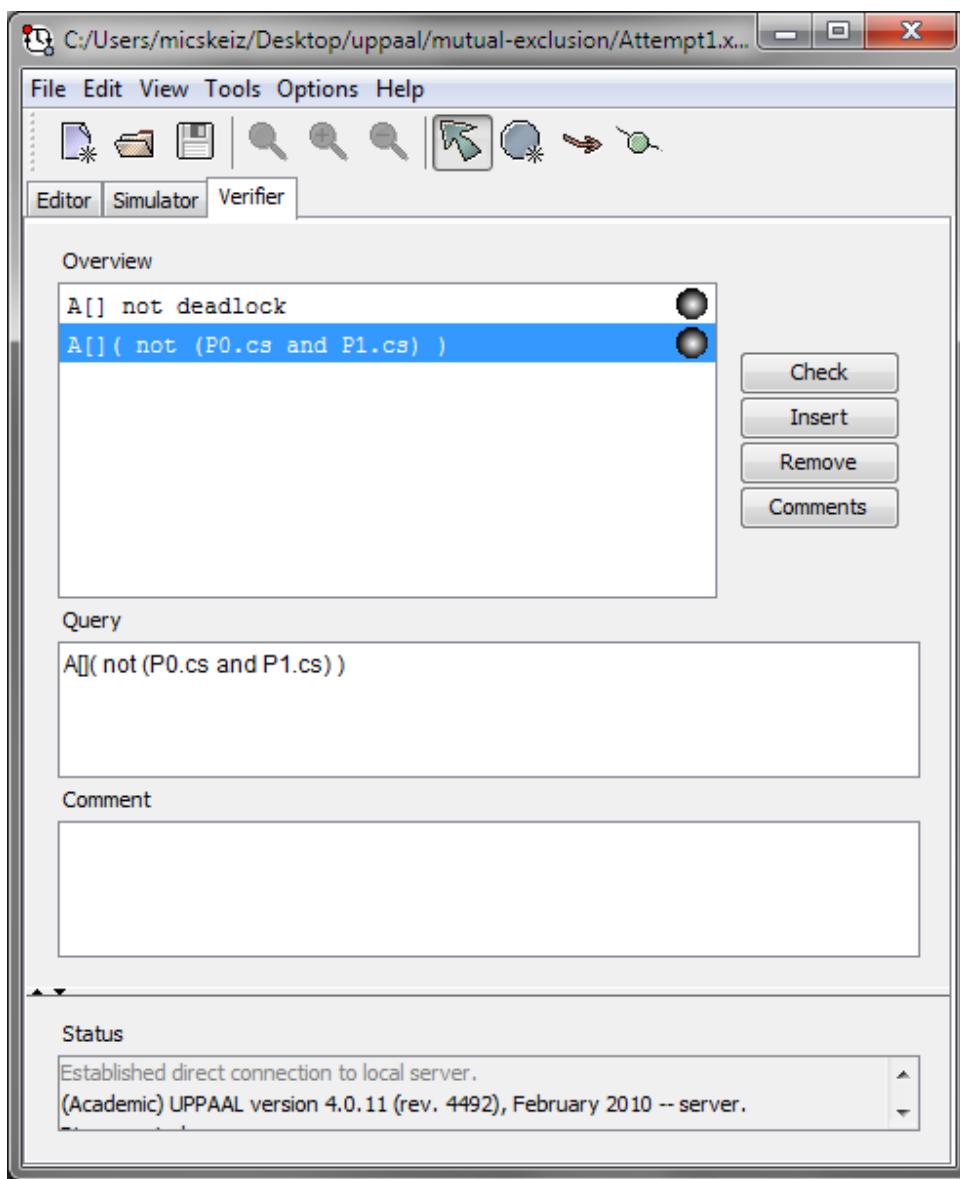
- Globális változók
- Automata
 - Állapot
 - Átmenet
 - Őrfeltétel
 - Akció
 - Órák
- Rendszer:
 - Automata példányok

Az UPPAAL felülete: szimulátor



- Átmenet kiválasztása
- Változók állapota
- Automaták képe
- Trace:
 - Szöveges
 - Grafikus: *Message Sequence Charts*

Az UPPAAL felülete: ellenőrzés



- Követelmény:
 - Logikai formula
- Elemei:
 - Állapotra hivatkozás
 - NOT, AND, OR
- További operátorok:
 - A: minden úton
 - E: legalább egy úton
 - []: minden időben
 - <>: valamikor a jövőben

Vissza a saját algoritmusunkhoz

```

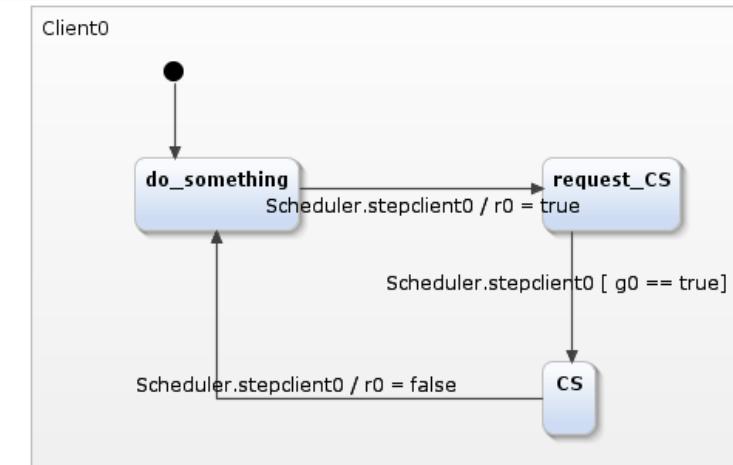
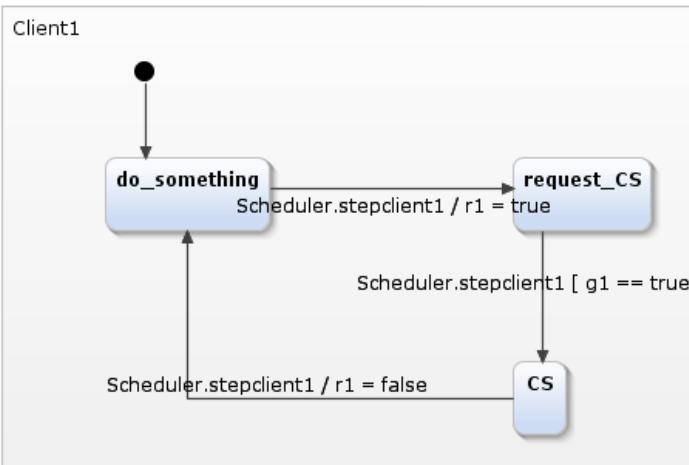
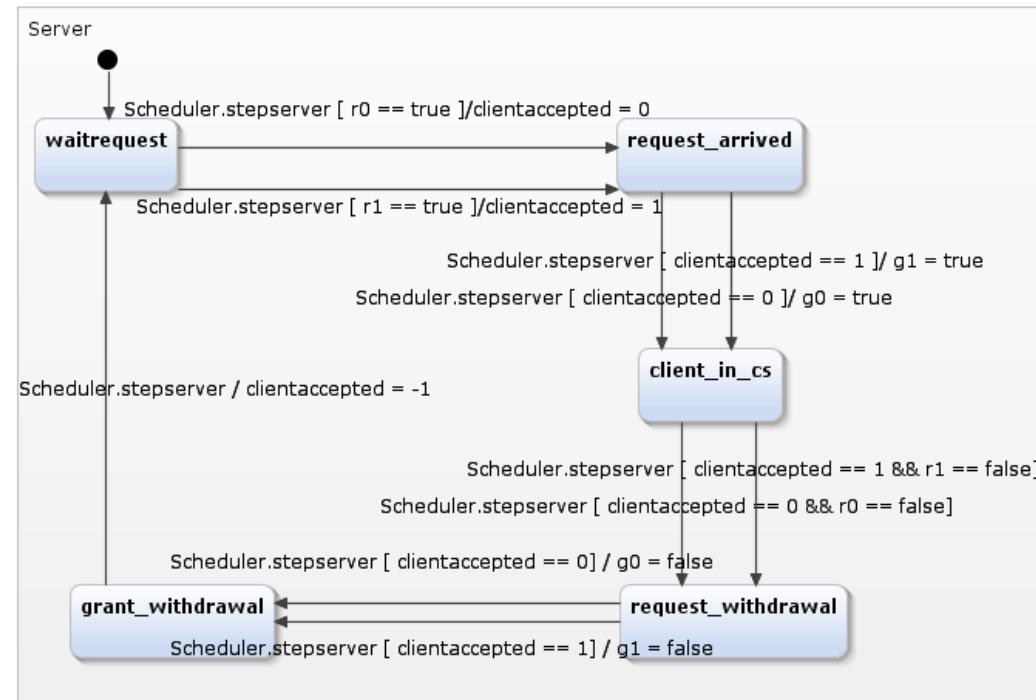
simplemutex
*****
Scheduler interface
*****
interface Scheduler:
in event stepserver
in event stepclient0
in event stepclient1

internal:
*****
internal variables:
*****
//variable c in pseudo code
var clientaccepted:integer = -1

//request variables
var r0:boolean = false
var r1:boolean = false

//grant variables
var g0:boolean = false
var g1:boolean = false

```



Kölcsönös kizárás példa - Kliens

1 ... other activity ...

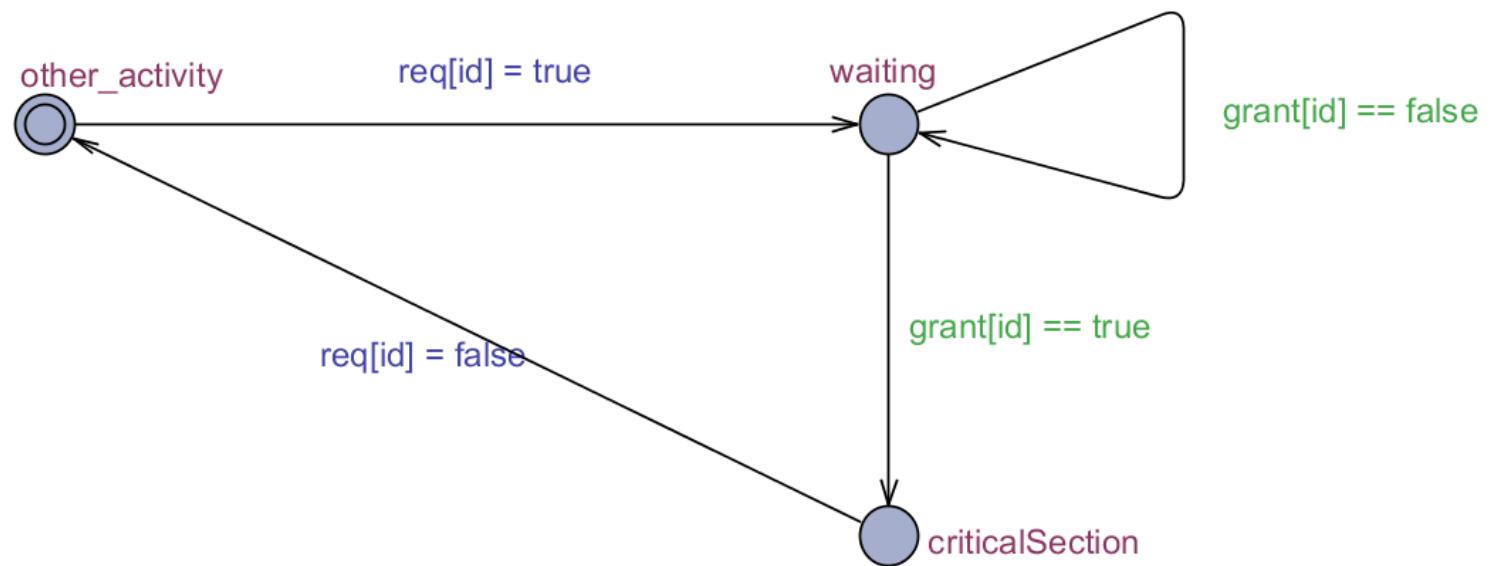
2 $r_i := \text{TRUE}$

3 wait until $g_i = \text{TRUE}$

4 critical section

5 $r_i := \text{FALSE}$

6 go to 1



Kölcsönös kizárási példa

1 wait until at least one r_i is TRUE

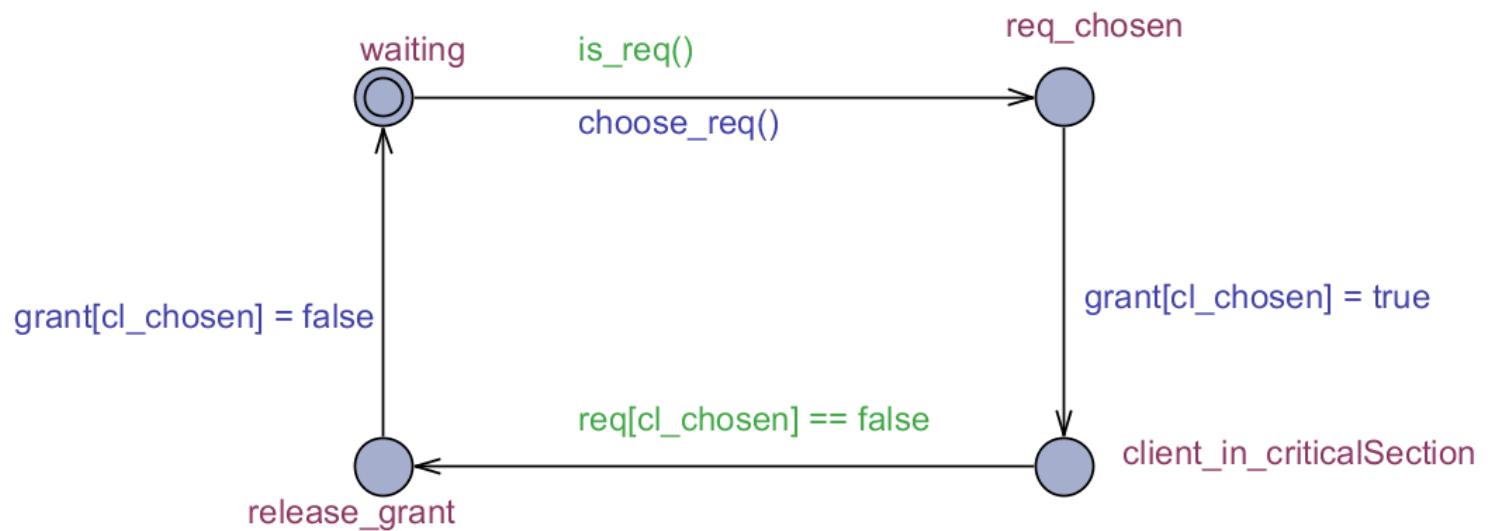
2 let c be such that $r_c = \text{TRUE}$

3 $g_c := \text{TRUE}$

4 wait until $r_c = \text{FALSE}$

5 $g_c := \text{FALSE}$

6 go to 1



- Algoritmusokat leíró modellek vizsgálata
- Szimuláció
- Követelmények ellenőrzése:
 - Egyszerre csak egy példány lehet a kritikus szakaszban:
 - **A[] not (CL0.criticalSection and CL1.criticalSection)**
- Ellenpélda generálása:
 - *Options / Diagnostic Trace / Shortest*

```
E<> CL0.criticalSection
Property is satisfied.
E[] not CL0.criticalSection
Property is satisfied.
A<> CL0.criticalSection
Property is not satisfied.
A[] not (CL0.criticalSection and CL1.criticalSection)
Property is not satisfied.
A[] not deadlock
Property is satisfied.
```

Motiváció

- Történelem során problémák az algoritmusokkal:
 - Harris Hyman, Comments on a problem in concurrent programming control, Communications of the ACM, v.9 n.1, p.45, Jan. 1966

Hyman algoritmusa

```
turn=0, flag[0]=flag[1]=false;
```

```
Protocol (int id) {
    do {
        flag[id] = true ;
        while (turn != id) {
            while (flag[1-id]) /* do nothing */ ;
            turn = id;
        }
       CriticalSection(id);
        flag[id] = false;
    } while (true) ;
}
```

Lehetnek-e ketten egyszerre a kritikus szakaszban?

Peterson algoritmusa

```
turn=0, flag[0]=flag[1]=false;
```

```
Protocol (int pid) {
```

```
    while (true) {
```

```
        flag[pid]=true;
```

```
        turn=1-pid;
```

```
        while (flag[1-pid]&&turn==1-pid) /**/;
```

```
       CriticalSection(id);
```

```
        flag[pid]=false;
```

```
}
```

```
}
```

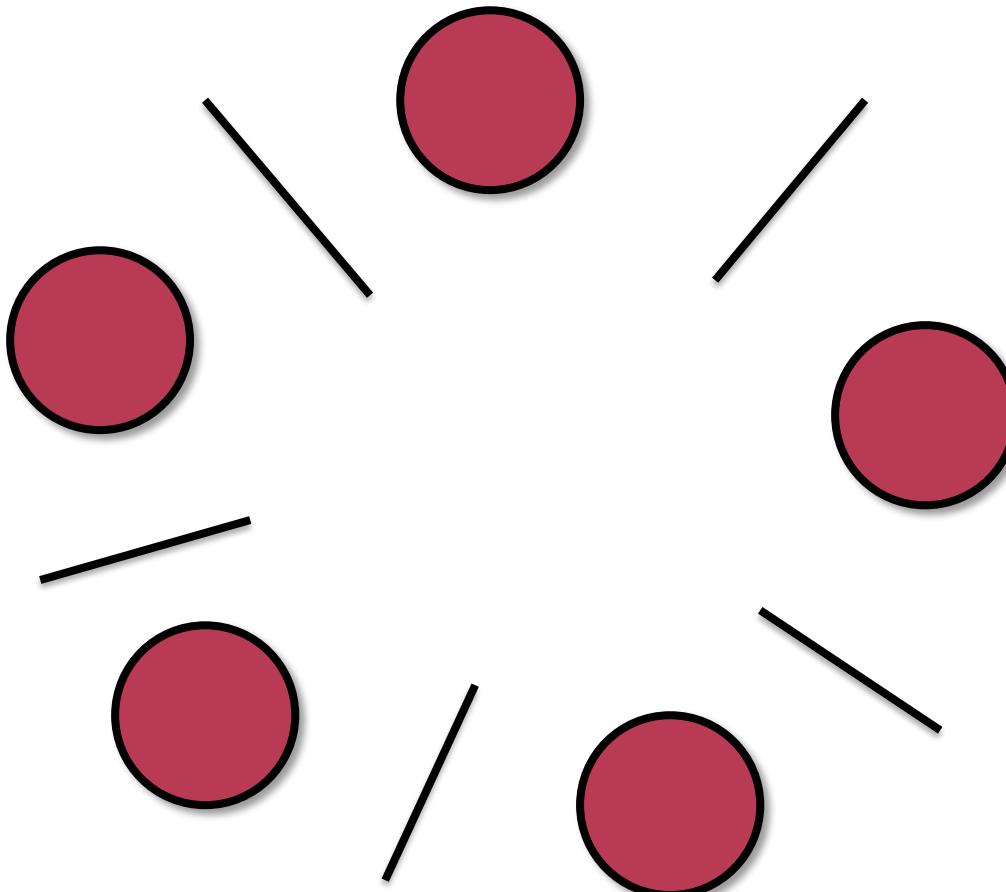
Lehetnek-e ketten egyszerre a kritikus szakaszban?

Étkező filozófusok



Kép: en.wikipedia.org

Étkező filozófusok



- **petridotnet** from BME-MIT - Tanszéki fejlesztés

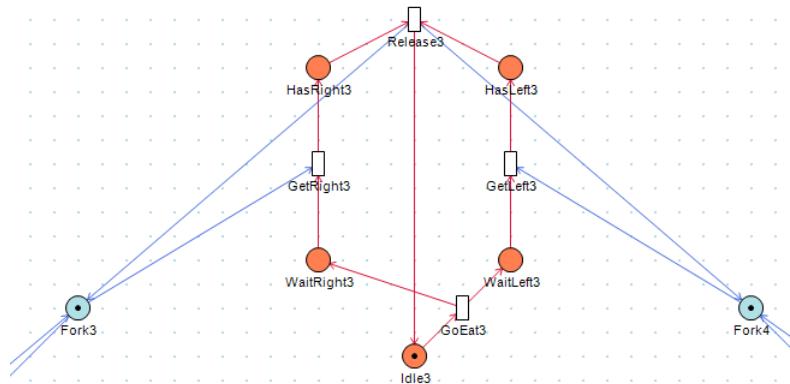
- TDK díjak az elmúlt években:

- 6 első helyezés,
 - 3 második

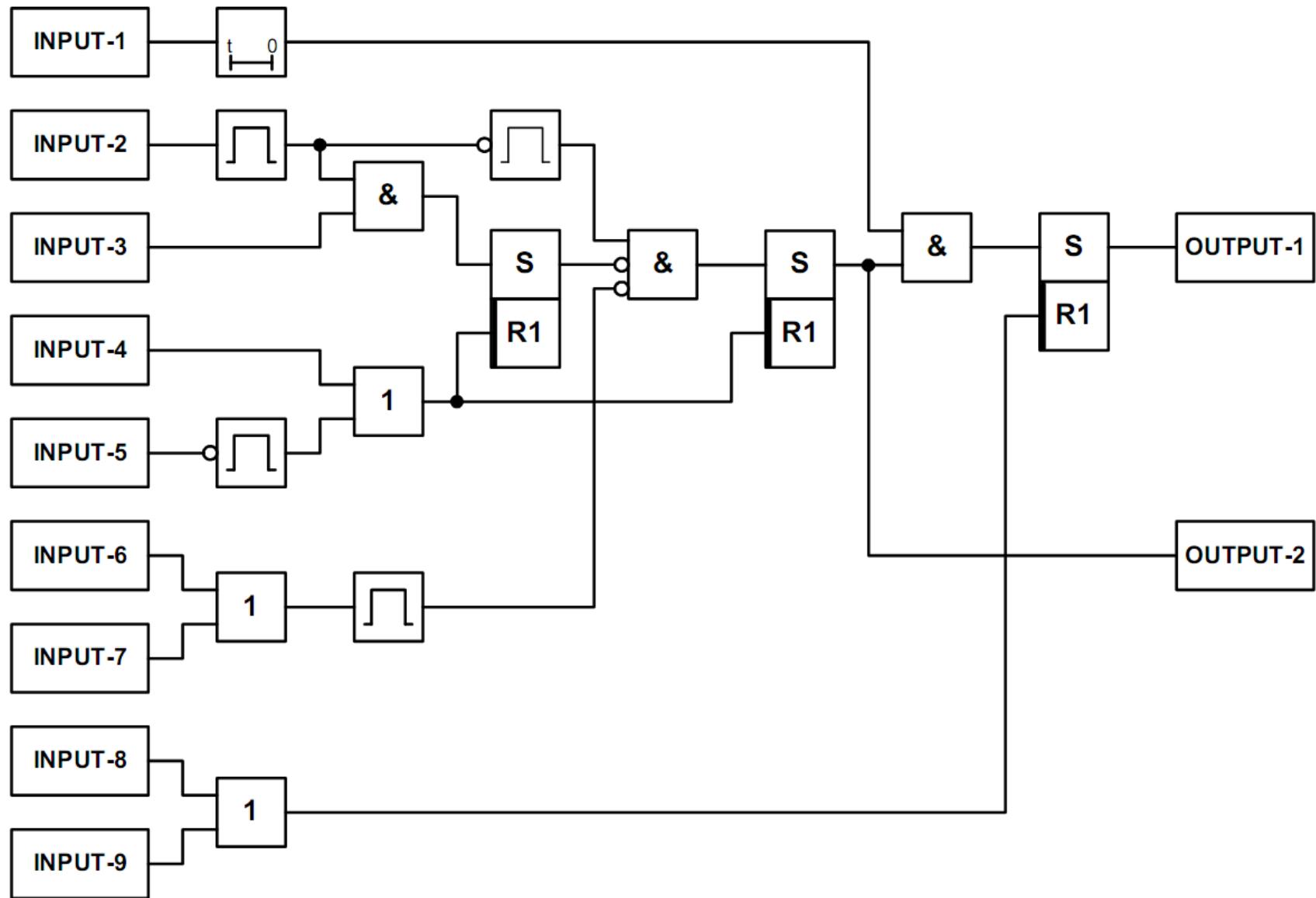
- OTDK:



- **petridotnet** - Tanszéki fejlesztés
- Modellek:
 - Petri-háló és Színezett Petri-háló
- Sokféle analízis lehetőség
 - Temporális logikai specifikációk
 - Végtelen állapotterű rendszerek vizsgálata



Paksi atomerőmű biztonsági logikája



Állapotfelderítés jellemzői

- Futási idő: 950 s
- Memóriafoglalás: 2,53 GB
- Globális állapotok száma: $4,8 \cdot 10^{12}$

Szimbolikus kódolás hatékonysága:

- Állapottér MDD csomópontszáma: kb. 1500

Errors in Smart Contracts

A Hacking of More Than \$50 Million Dashes Hopes in the World of Virtual Currency

By Nathaniel Popper

June 17, 2016

A hacker on Friday siphoned more than \$50 million of dig

ETHEREUM, TECHNOLOGY

Batch
Ether
Depos

Sam To



Someone ‘Accidentally’ Locked Away \$150M Worth of Other People’s Ethereum Funds

And a hard fork is on the table.

Parity Multisig Hacked. Again

Yesterday, Parity Multisig Wallet was hacked again:

[blog/security-alert.html](#)

National Vulnerability Database (NVD)
400+ vulnerability records for blockchain
95%+ are programming errors in contracts

] multi-

wallets.

<https://nvd.nist.gov/vuln/>

Formal verification of Smart Contracts

SRI approach: automated formal verification

- Specification annotations
- Automatic translation to verification language
- Modular reasoning with logic solvers
- Automated, user friendly, expressive
- Correct and precise

Érdeklődők számára: TDK nyertes dolgozatok

- Dobos-Kovács Mihály: Formális verifikációval támogatott tesztgenerálás autóipari környezetben (I. helyezett **Rektori különdíj**)
- Bajczi Levente: Konkurens programok HW-SW együttes verifikációja (I. helyezett **Rektori különdíj**)
- Klenik Attila, Marussy Kristóf: Configurable stochastic analysis framework for asynchronous systems (I. helyezett **Rektori különdíj**)
- Sallai Gyula: Fordító optimalizációk szoftver verifikációhoz
- Élő Dániel, Soltész Adrián: Symbolic model checking and trace generation by guided search
- Molnár Vince, Segesdi Dániel: Múlt és jövő: Új algoritmusok lineáris temporális tulajdonságok szaturáció-alapú modellellenőrzésére.
- Hajdu János, László Ákos, Nagy László, Szabó Péter, Varga Péter, Viselkey Zoltán, Vida Péter, Zámbó Gábor: viselkedési modellökkel támogatott algoritmusok fejlesztése a hibakeresésben (II. helyezett **Rektori különdíj**)

<https://inf.mit.bme.hu/edu/results/tdk>

További eszközök

- Java Pathfinder
 - Modelellenőrző Java byte kódhoz
 - NASA fejlesztés, 2005 óta nyílt forrású
- CHESS
 - .NET-es kódokhoz
 - Párhuzamosságból fakadó hibák keresése
- jchord
 - Java kód statikus analízise
 - Versenyhelyzet, holtpont detektálás
- Facebook: Infer
 - Statikus analízis
 - C, Java, Objective-C

További eszközök

- Static Driver Verifier (SDV, korábban SLAM)
 - Windows eszközmeghajtók ellenőrzése
- Linux Driver Verification
 - Linux eszközmeghajtók ellenőrzése
- Linux Deductive Verification
 - Linux kernel kód ellenőrzése

Alkalmazási példák, pár siker

- Futurebus+ Cache Coherence Protokoll
 - IEEE szabványban hibát találtak
- Microsoft Hyper-V Hypervisor ellenőrzése
 - Virtualizációs technológia
- FlexRay (for Avionics) protokoll verifikációja
 - UPPAAL segítségével
- Airbus repülőgépek kernel moduljának verifikációja
- The seL4 Microkernel
- Amazon: elosztott rendszerek tervezése

Amazon

Amazon Simple
Storage Service

high-performance
"no SQL" data store

System	Components	Line Count (Excluding Comments)	Benefit
S3	Fault-tolerant, low-level	804 PlusCal	Found two bugs, then others in proposed optimizations
DynamoDB	Replication and group-membership system	645 PlusCal	Found one bug, then another in the first proposed fix
EBS	Volume management	939 TLA+	Found three bugs requiring traces of up to 35 steps
distributed lock manager	Lock-free data structure	102 PlusCal	Found three bugs
	Fault-tolerant replication-and-reconfiguration algorithm	223 PlusCal	Improved confidence though failed to find a liveness bug, as liveness not checked
		318 TLA+	Found one bug and verified an aggressive optimization

Elastic Block Store

Összefoglalás

- Feladatok együttműködésénél sok hibalehetőség
- Versenyhelyzet, holtpont...
- DE: léteznek eszközök a vizsgálathoz
- Modelellenőrzők, tételbizonyítók, statikus ellenőrzők...

További információ

- R. Hamberg and F. Vaandrager. [Using Model Checkers in an Introductory Course on Operating Systems](#). OSR 42(6):101-111.
- [Formális módszerek](#) MSc tantárgy (VIMIM100)

További információ

- UPPAAL modelellenőrző
- PetriDotNet modelellenőrző
 - <http://petridotnet.inf.mit.bme.hu/publications/>
- Theta szoftverellenőrző keretrendszer
 - <https://github.com/FTSRG/theta>
 - CERN
- Gamma Statechart Composition Framework
 - <https://inf.mit.bme.hu/en/gamma>
 - Rendszerek modell-alapú tervezése és ellenőrzése



Specializáció

- BSc: Szoftverfejlesztés, Rendszertervezés
- MSc: Kritikus rendszerek, Intelligens rendszerek

- Szoftver és rendszerfejlesztés/ellenőrzés,
- formális módszerek és algoritmusok,
- adat és mesterséges intelligencia alapú rendszerek;

Május 8. 13:00-14:00

Csatlakozás a Teams csoporthoz: **9xmh636**
(lépj be most és kérdezz a specializációkról!)

Az operációs rendszerek működése: fájl- és tárolórendszerek

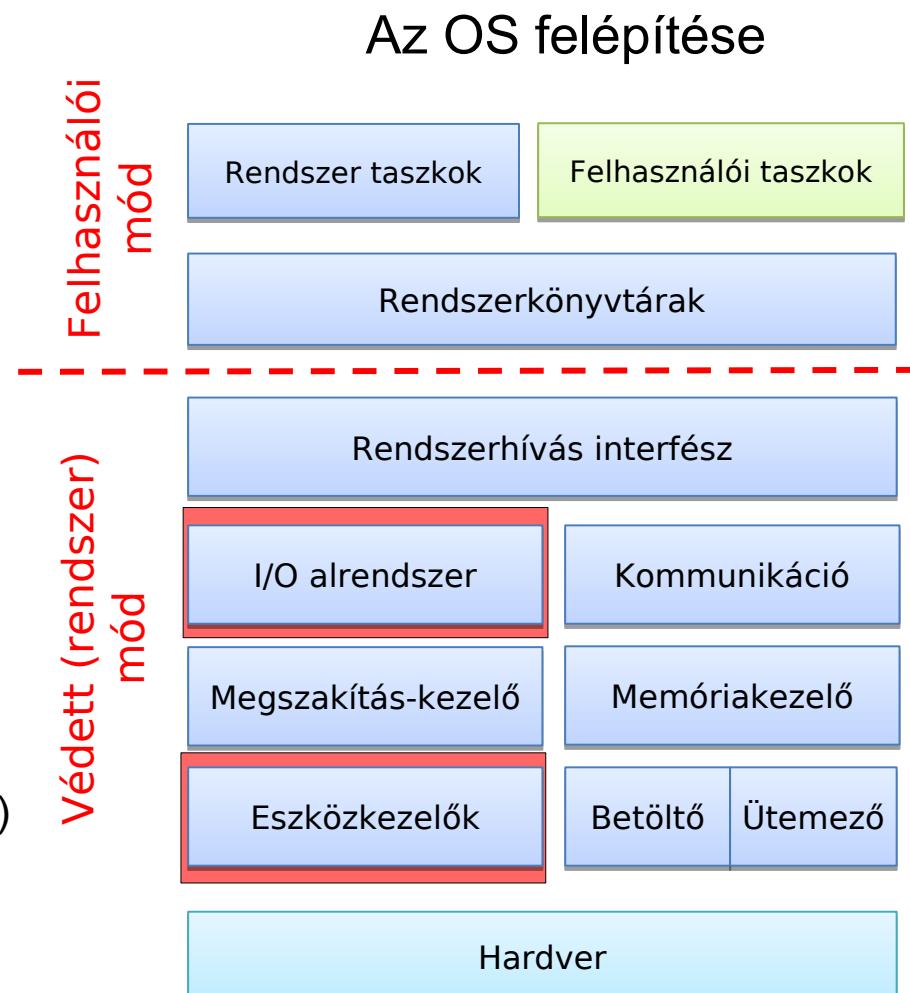
Mészáros Tamás
<http://www.mit.bme.hu/~meszaros/>

Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

Az előadásfóliák legfrissebb változata a tantárgy honlapján érhető el.
Az előadásanyagok BME-n kívüli felhasználása és más rendszerekben történő közzététele előzetes engedélyhez kötött.

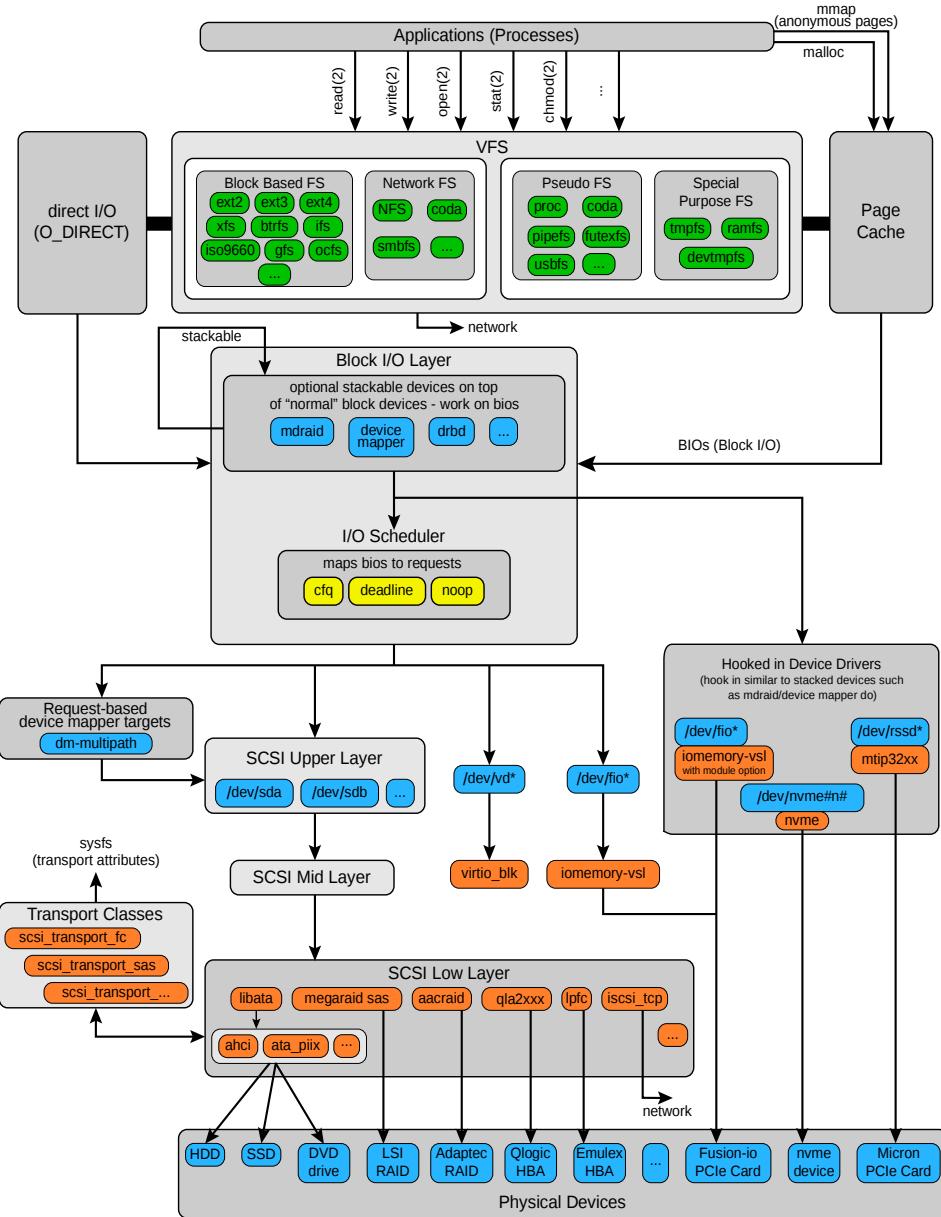
Az eddigiekben történt...

- A taszkok...
 - jellemzően I/O-intenzívek
 - sok fájlműveletet végeznek
 - programkódjuk a fájlrendszerben
- Memóriakezelés...
 - a háttértárral bővül (cserehely)
- Kommunikáció
 - kommunikáció fájlon keresztül (mmap)
- Laborok
 - Linux: hálózati fájlrendszer (Samba)
 - Windows: fájlleírók, jogosultságok, megosztás, hálózati meghajtók



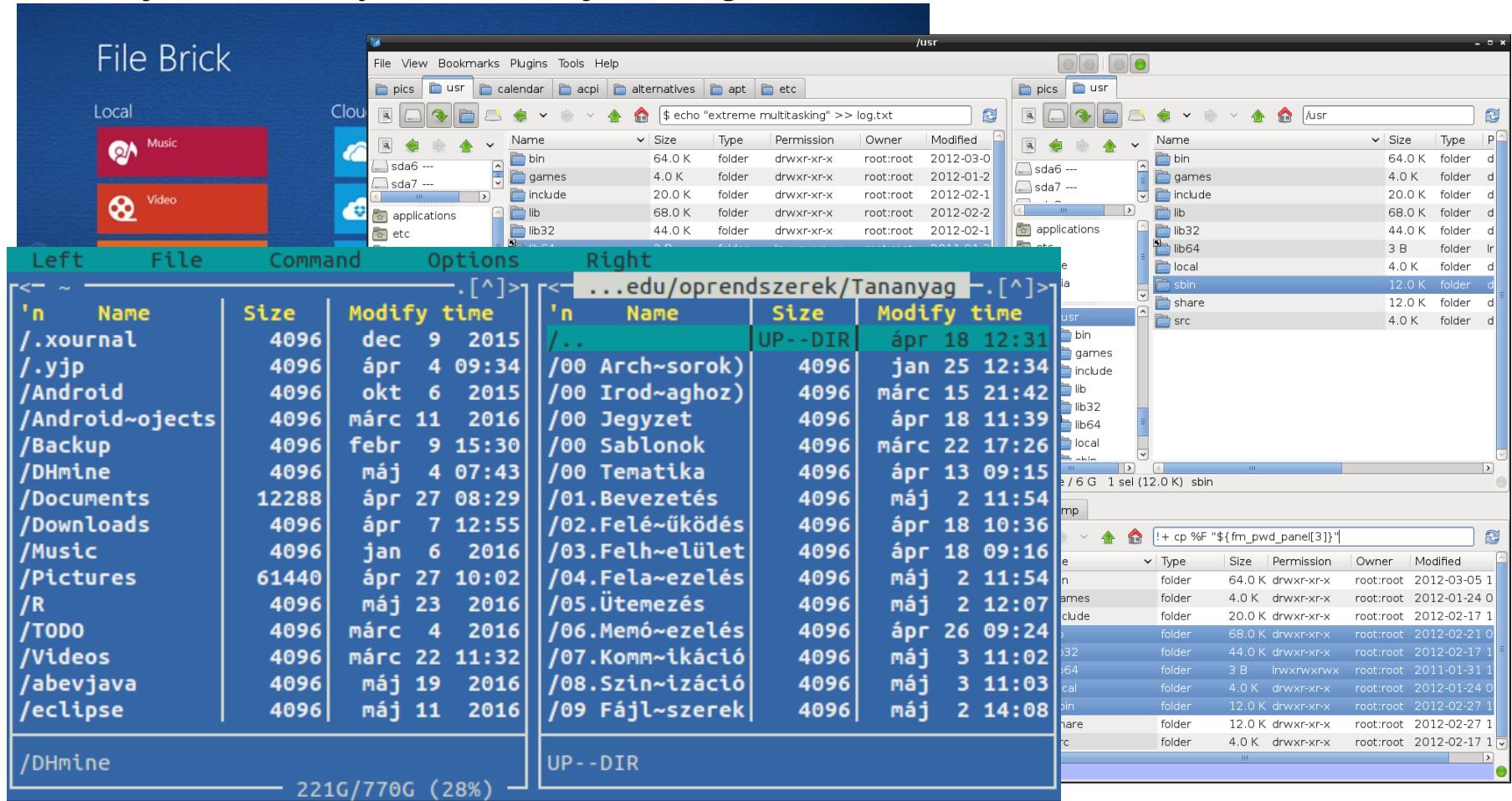
Áttekintés

- Felhasználói szemmel...
 - végfelhasználó
 - adminisztrátor
 - programozó
- Belső működés
 - fájlrendszer interfések
 - kernel adatstruktúrák
 - a háttértár szervezése
 - virtuális fájlrendszerek
- Adattárolás
 - fizikai tárolók (HDD, SSD)
 - I/O ütemezés
 - tárolórendszer-virtualizáció:
 - helyi (RAID, LVM)
 - hálózati (SAN, NAS)
 - elosztott fájl- és tárolórendszerek



A fájlrendszer felhasználói szemmel

- Parancssori és grafikus fájlkezelők
- A tárolási rendszer logikai felépítése (helyek)
- Fájlok és könyvtárak tulajdonságai



A fájlrendszer felhasználói szemmel (folyt.)

- Adminisztrátor
 - létrehozás, ellenőrzés, megszüntetés
 - helyi és távoli fájlrendszerök használatba vétele (csatolás)
 - teljesítményhangolás
 - helyfoglalás ellenőrzése és felszabadítása
 - biztonsági másolatok készítése
- Programozó (alkalmazásfejlesztő)
 - programozói interfések
 - rendszerhívások
 - rendszerkönyvtárak
 - fájlleírók és fájlműveletek
 - megnyitás, létrehozás, írás, olvasás, pozicionálás, bezárás, törlés
 - fájlok zárolása kizárolagos használatra

Alapfogalmak

- **Fájl (file)**, állomány
 - az adattárolás logikai egysége
 - név (+ esetenként kiterjesztés)

- **Könyvtár (directory)**
 - a szervezés logikai egysége
 - fájlok és könyvtárak halmaza

- **Kötet (volume), meghajtó**
 - fájlok és könyvtárak tárolásának logikai egysége
 - fizikai tárolási egységhez (pl. partíció) rendelhető

Logikai

-
- **Fájlrendszer (file system)**
 - fájlok és könyvtárak fizikai tárolása és szervezése

Fizikai

- **Partíció (partition)**
 - a háttértár szervezési egysége
 - fájlrendszer tárolására képes

Fájlrendszer logikai szervezése

- Irányított fával reprezentálható
 - csomópontok: könyvtár, fájl, (tárolt adat)
 - élek: tartalmazás reláció
 - gyökér csomópont: a kötethez (meghajtóhoz) rendelt elem
- Elérési út (path)
 - egy csomópont elérési helye
 - **abszolút**: a fa gyökerétől kezdve
 - **relatív**: egy másik csomóponttól (pl. munkakönyvtár)
- A fa bővítése irányított gráffá
 - (rögzített) **link** (hard link)
több fájl ugyanarra az adatra hivatkozik
 - **szimbolikus link** (symbolic link, symlink, soft link)
egy másik fájlrendszeri elemre (fájl, könyvtár) mutat

A link és az adat melyik esetben mikor és hogyan törölhető?

Mit okoz egy irányított kör a gráfban?

Példa: Windows 10

- Fizikai tárolók logikai meghajtókhoz rendelve
 - könyvtárakhoz rendelt kötetek is léteznek, de ritkák
- A boot meghajtó (jellemzően C:) a kiinduló pont (`dir c:\`)
 - \Program Files a telepített alkalmazások (x86: minden egyik, x64: 64 bites)
 - \Program Files (x86) a telepített 32-bites alkalmazások x86 esetben
 - \ProgramData az alkalmazások felhasználófüggetlen adatai
 - \Users felhasználói könyvtárak (adataik, fájljaik, programok egyedi adatai)
 - \Windows az operációs rendszer saját fájljai, könyvtárai
- További meghajtók (D: E: stb.)
 - CD/DVD/USB fizikai tárolóeszközök
 - további partíciók a diszkeken
 - hálózati fájlrendszerek

Példa: Unix / Linux

- Könyvtárakhoz rendelt fizikai tárolók
 - egyetlen összefüggő gráfot alkot
- Gyökér: / avagy ROOT (`ls /`)

/bin	a rendszer működéséhez szükséges alapvető bináris állományok
/sbin	hasonló, de alapvetően a rendszergazda által futtatható programok
/dev	hardver eszközök
/etc	a rendszer konfigurációs beállításait tároló fájlok
/home	a felhasználók saját könyvtárai (jellemzően külön fizikai tárolóval)
/lib	alapvető (megosztott, shared) rendszerkönyvtárak
/mnt	alkalmilag felcsatolt partíciók helye (mount)
/tmp	átmeneti fájlok (programok és felhasználók számára)
/usr	felhasználói programok, programkönyvtárak, dokumentáció, stb.
/var	a rendszerműködés „dinamikus” fájljai, naplófájlok, adatbázisok

részletesebben lásd `man hier`

- Szabványok, változások
 - jelentős eltérések lehetnek a részletekben
 - FHS (Filesystem Hierarchy Standard): inkább csak ajánlás
 - UsrMove: a /bin, /sbin, ... átkerül a /usr alatti helyére (Solaris11, Fedora)

Példa: Android

- Unix-szerű, de eltérő könyvtárak
 - nem triviális megnézni a teljes gráfot (demo)

- Gyökér: / avagy ROOT (ls /)

/cache gyorsítótár az alkalmazások számára

/data felhasználói **programok és adatok**

/data/app a felhasználó által telepített alkalmazások

/data/data az alkalmazások adatfájljai

/data/anr app-not-responding: alkalmazáshibák adatai

/data/tombstones hibával (pl. SIGSEGV) leállított alkalmazások memóriaképei

/data/dalvik-cache az alkalmazások optimalizált bináris állományai

/data/misc felhasználói konfigurációs fájlok (pl. wifi, bluetooth, vpn beállítások)

/data/local átmeneti fájlok

/mnt v. /storage további csatolt fájlrendszerek (pl. SD kártya) elérhetőségei

/mnt/asec az SD kártyára írt alkalmazások futásidejű (titkosítatlan) változatai
titkosítva az .android_secure könyvtárban vannak

/system előtelepített alkalmazások, rendszerkönyvtárak, konfigurációk

Fájlok tulajdonságai (Unix példákkal)

- Listázzuk ki fájlrendszeri bejegyzések adatait! `ls -la <fájlnév>`

```
-rw-r--r--  1 root root   2290 júl  5 2014 /etc/passwd
-rwxr-xr-x  1 root root  616920 nov 17 2015 /bin/bash
srwxr-xr-x  1 clamilt clamilt 0 ápr 22 10:16 clamav.sock
crw-rw----  1 root tty      4, 0 ápr 20 2007 /dev/tty0
---s--x--x. 1 root root  123832 Aug 13 2015 /usr/bin/sudo
```

- Mit látunk a listában?

- a bejegyzés típusa: (- d p l b c s)
- POSIX jogosultságok (lásd következő fólia)
- linkek (hard) száma
- tulajdonos és csoport
- méret
- időbélyeg (ctime: metaadatok változása, mtime: adatmódosítás, atime: olvasás)
- a bejegyzés neve

- Amit fent nem látunk, de az OS tárolja (lásd később)

- egyedi azonosító (belő használatra)
- elhelyezkedés (hol vannak a fájl adatai)

Unix hozzáférési jogosultságok

- **POSIX** jogosultságok (*alap*)

- 3 x 3 bit: { tulajdonos, csoport, mások } x { olvasás, írás, futtatás }
- könyvtárak használatához olvasás és „futtatás” is kell
- beállítás: **chmod** <jogosultság> <fájl v. könyvtár>

pl.: `chmod 750 /home/me`

`chmod u+rwx,g+rx,o-rwx /home/me`

- Speciális jogosultságok: SETUID, SETGID, StickyBit

- SETUID/GID: futási tulajdonos/csoport beállítása

`chmod u+s setuid_file` `chmod g+s setgid_file`

KOCKÁZATOS !

- StickyBit: csak a tulajdonos törölhet

`drwxrwxrwt 44 root root 12288 máj 9 15:25 /var/tmp`

- **POSIX ACL (access control list) (*kiterjesztett*)**

- rugalmasabb, többféle jogosultság egyidőben

pl.: `setfacl -m u:student:r file`

- lásd `ls` parancs kimenetén + jel

Adminisztrátori alapfeladatok

- Fájlrendszer létrehozása (formázás)
 - típus (l. köv. fólia)
 - jellemzők (alapértelmezett jó + esetleg titkosítás)
 - név (emberi), azonosító (gépi)
 - tárolási hely
- Csatlakoztatás (mount)
 - fizikai → logikai tárolási hely
 - **csatlakoztatási pont** (mount point)
 - **elfedés**
- Ellenőrzés, hangolás
 - állapotellenőrzés és hibajavítás (offline)
 - a méret megváltoztatása (online) a tárolórendszerrel összhangban
 - teljesítmény: tárolóhoz igazítás (alignment), tömörítés stb.
- Biztonsági mentés

Széles körben elterjedt fájlrendszerek [áttekintése](#)

- FAT32
 - kompatibilis
 - eredetileg 8+3 karakteres fájlnév 255-re bővítve, 4GiB maximális fájlméret (!)
- NTFS
 - a Windows alapértelmezett fájlrendszere ([továbbiak áttekintése](#))
- UFS avagy Berkeley FFS (lásd KK. tankönyv)
 - tradicionális BSD Unix fájlrendszer
- ext2,3,4 (UFS-alapokra épült)
 - Linux
- XFS
 - eredetileg SGI, újabban pl. RedHat Linux 7
- HFS+, újabban APFS (iOS 10.3)
 - Apple
- Integrált fájl + tárolórendszerek (lásd még később)
 - **ZFS**: Solaris, később nyílt forrású, BSD-körökben is népszerű
 - Linux **btrfs**: újabb, aktív fejlesztés alatt álló Linux fájlrendszer
- [Ezernyi más](#) fájlrendszer, pl.:
 - CD/DVD (ISO 9660 és kiterjesztései)

Demók (otthonra is!)

- Alapvető fájl- és könyvtárműveletek cp mv cd pwd mkdir
Hogyan lehet átnevezni egy fájlt?
- Fájlok attribútumai: ls -la ls -laz setfacl
- Fájlrendszer kezelése: mount umount df mkfs fsck
mount (/proc?) df umount /boot mount /boot (honnan?)
mount -bind ...

Hozzunk létre egy új fájlrendszer egy fájlban (sudo su – kiadása után)!

```
dd if=/dev/zero of=filesystem.img bs=1k count=1000
losetup /dev/loop0 filesystem.img
mke2fs /dev/loop0
mount /dev/loop0 /mnt
```

Az egyik tipikus, bosszantó hibajelzés

```
umount: /mnt: device is busy
```

Miért nem sikerül? Valaki foglal (nyitva tart) fájlt, könyvtárat.

Mit tehetünk? Megnézzük, ki mit tart fogva: lsof /mnt (esetleg remount,ro?)

- Mi történik a fájlrendszerben? iotop sar dstat vmstat ...
 - sudo sysctl vm.block_dump=1
 - tail -f /var/log/kern.log
- Tegyük tökkre a fájlrendszert, és próbáljuk helyreállítani!

Fájlrendszerk hangolása (demók)

- Szabad hely növelése
 - diszkhelyszámítás elemző: du xdu baobab kdiskstat filelight
 - töltük tele a korábban létrehozott fájlrendszer-a-fájlban eszközöt!
cp -r /bin /mnt (ne root-ként futtassuk!)
 - miért 0 a szabad hely, miközben nem foglalt minden blokk?
súgó: man tune2fs
 - futásidőjű tömörítés bekapcsolása (tárolórendszerrel integrált fájlrendszerben)
pl.: btrfs mount opció: compress = { zlib | lzo | snappy }
(A már létrehozott fájlrendszerre utólag is bekapcsolható.)
- Teljesítménynövelés – Lassú diszk I/O: mi az oka, mi az elvárt IOPS?
 - a noatime opció hatása a teljesítményre
A /etc/fstab fájlban módosítsuk az attribútumokat (lásd man mount)
 - fájlrendszerszintű tömörítés
Lényegesen kisebb adatmozgatás, CPU terhelés kismértékű növelése
Lásd pl.: www.phoronix.com/scan.php?page=article&item=btrfs_lzo_2638
 - nr_requests, read_ahead_kb, fájlrendszer naplázás és blokkméret
 - fizikai és logikai blokkok összehangolása: Partition Alignment
 - prioritás növelése (ionice) a kiemelt folyamatokra

Biztonsági mentés és visszaállítása

- Adatvesztés oka
 - nem javítható meghibásodás
 - fizikai hiba
 - inkonzisztencia
 - felhasználó
 - kártevők
- Jellege
 - korlátozott
 - teljes (SSD „hirtelen halál”)
- Mentés (backup)
 - hogy: automatizált / kézi
 - mit: rész / teljes
 - hova: szalag, diszk, net
- Visszaállítás (restore)
 - „bare metal” / reinstall + restore

Használat közben mi konzisztens?

Fájlok a programozó szemszögéből ...

Programozói interfész

- Megnyitás (és létrehozás)

`open()`

- fájlleíró + nyitott fájl objektum (kernel)

- Írás, olvasás, pozicionálás

`read()` `write()` `fseek()`

- **soros elérés** (sequential access)

az adatokat tárolási sorrendben olvassuk illetve írjuk

- **közvetlen elérés** (direct access)

az adatok rögzített méretű részei tetszőleges sorrendben elérhetők

- Fájlok lezárása

`close()`

- Könyvtárak kezelése:

`opendir()` `readdir()` `rewinddir()` `closedir()`

Mi történik egy fájl megnyitásakor?

- open()...
 - a cél lokalizálása (hol a fájl?)
 - metaadatok beolvasása
 - létrejön a **nyitott fájl objektum** (metaadatok a kernelben)
 - megnyitási mód
 - **fájlmutató (file pointer)**
 - fájl metaadatok
 - lehetséges műveletek
 - ezen objektum azonosítója a **fájlleírót (file descriptor)**
- A további műveletek során... (read(), write() stb.)
 - a fájlleíró azonosítja az objektumot
- Amikor lezárjuk (close())
 - a kernel megszünteti a létrehozott adatstruktúrákat
- Miben más az fread() és a fwrite() pufferelt I/O? Hatékonyabb?
 - Otthoni gyakorlat: az fread() vagy a read() gyorsabb különféle terhelésekre?

Fájlok zárolása

- Fájlok zárolása (= kölcsönös kizárási mechanizmus)

 - fájl = erőforrás *konzisztencia?*
 - szemaforokkal is lehetne,
de a fájlműveletekkel egyszerűbb
 - **holtpont** itt is kialakulhat

- Ajánlott zárolás (advisory locking)
 - az OS csak eszközöket biztosít (rendszerkönyvtárakban), **nem kényszeríti ki**
 - a taszkok számára opcionális
 - példák: Java [FileLock\(\)](#), Unix `flock()`
- Kötelező zárolás (mandatory locking)
 - kernel mechanizmusok biztosítják (pl. fájlrendszer csatolásakor megadható)
 - a rendszerhívások **kikényszerítik** a betartását
 - példák: Windows általában, Unix / POSIX `fcntl()` `lockf()`
- Fájlok részleges (tartományi) zárolása
 - pl. Windows [LockFileEx\(\)](#), Unix `fcntl()`

Fájlok megosztott elérése memórián keresztül (mmap)

- Egyszerűbb, mint a `read()`, `write()` és `fseek()`

- UNIX mmap (Windows: [CreateFileMapping](#))

`mmap(addr, size, prot, flags, fd, offset)`

- `addr`: ezt a címet rendeljük hozzá a fájl tartalmához (0: a kernel választ)
- `size`: az elért adatmennyiség mérete
- `prot`: a hozzáférés típusa (R, W, X), egyezik az `open()`-nél megadottal
- `flags`: saját vagy megosztott fájl, stb.
- `fd`: az `open()` rendszerhívás által visszaadott fájlleíró
- `offset`: ettől a pozíciótól kezdődik a hozzárendelés
 - Visszatérési érték: az adatokhoz rendelt virtuális memóriacím (változóhoz köthető)
- A hozzárendelés megszüntetése: `munmap(addr, len)`
- Többszörös hozzáférés, konzisztencia és kölcsönös kizárási mechanizmus
 - a programozó dolga...
- **Fájlműveletek helyett is jó, ha sok direkt elérésű olvasást végzünk.**

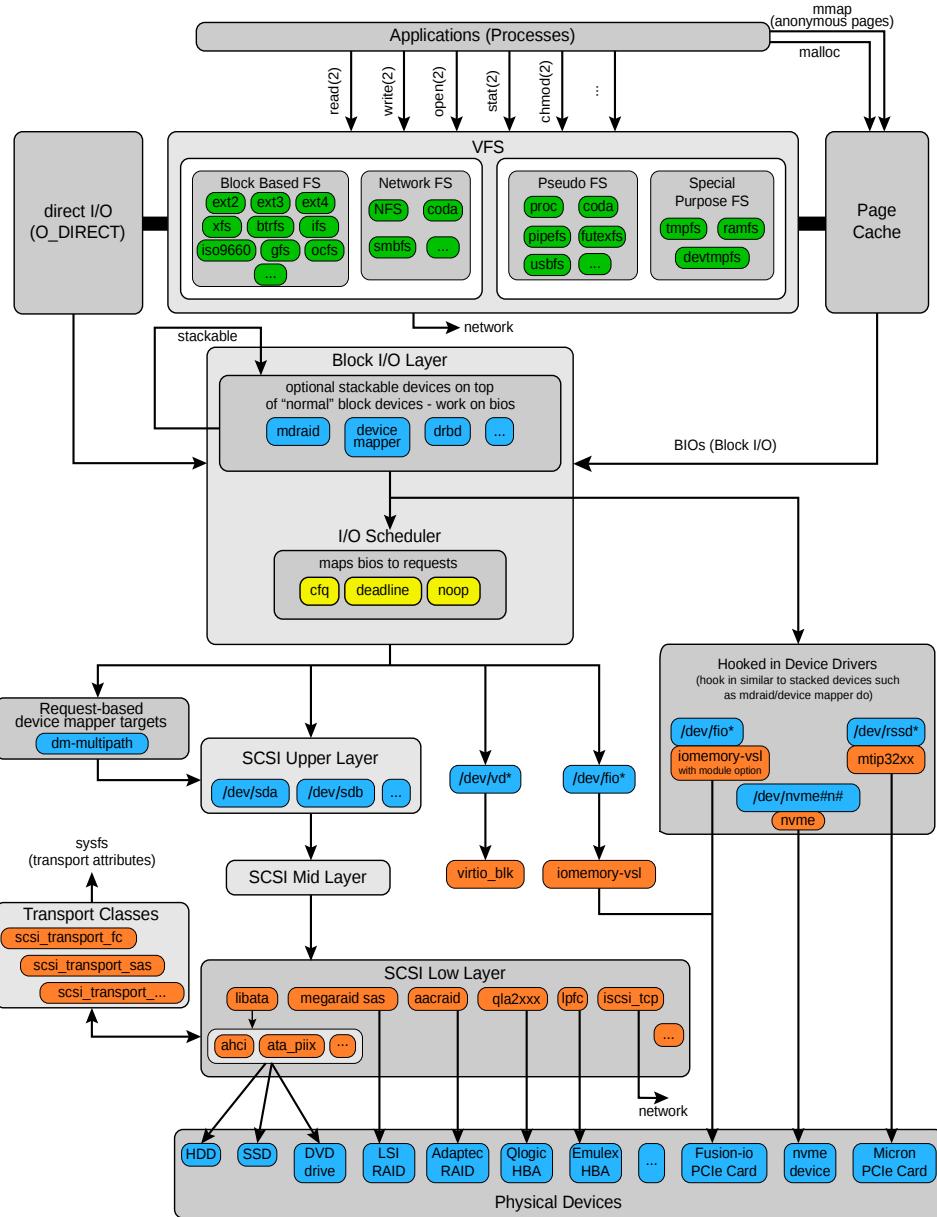
Várakozásmentes I/O: nem blokkoló és aszinkron

- Emlékeztető...
 - **ha van teendőnk, miért várakozzunk?**
 - lassú I/O lassú → nagyon sok várakozás
- Nem blokkoló I/O műveletek
 - a rendszerhívás egyből visszatér
 - ha van adat, azzal
 - ha nincs adat, hibával
 - a programozó dolga időnként ellenőrizni
- Aszinkron I/O műveletek
 - beállítjuk az I/O műveletet és az adattároló puffert
 - elküldjük az aszinkron I/O kérést
 - a háttérben elindul az I/O művelet kiszolgálása
 - a rendszerhívás azonnal visszatér
 - a taszkunk tovább fut
 - amikor az I/O elkészült → esemény (jelzés)
 - az eseménykezelő kezeli az adatokat

Lásd pl. [POSIX aio](#), [Windows I/O Completion Ports](#)

Áttekintés

- Felhasználói szemmel...
 - végfelhasználó
 - adminisztrátor
 - programozó
- Belső működés
 - fájlrendszer interfések
 - kernel adatstruktúrák
 - a háttértár szervezése
 - virtuális fájlrendszerek
- Adattárolás
 - fizikai tárolók (HDD, SSD)
 - I/O ütemezés
 - tárolórendszer-virtualizáció:
 - helyi (RAID, LVM)
 - hálózati (SAN, NAS)
 - elosztott fájl- és tárolórendszerek



Fájlrendszerek megvalósítása (áttekintés)

- Felhasználói felület (rendszerhívások)
 - fájlok könyvtárak elhelyezése és kezelése
 - formázás, csatolás, lecsatolás stb.
 - ellenőrzés, javítás, paraméterek módosítása stb.
- Tárolás
 - logikai egységek → fizikai tárolók
 - **blokkos adattárolás**
 - adatok + metaadatok
 - szabad helyek nyilvántartása
- Belső működés
 - fájlrendszerek leírói (csatlakoztatott fájlrendszerek metaadatai)
 - csatlakoztatás nyilvántartása (elfedéssel)
 - fájlok leírói (metaadatok) a memóriában
 - kapcsolat a nyitott fájl objektumokhoz
 - beolvasott adatok elhelyezése a memóriában, pufferelés

Tárolás: mit és hol?

- Metaadatok
 - partíciók típusai és elhelyezkedése
 - fájlrendszerek leírói (típus, méret, szabad helyek stb.)
 - fájlok (könyvtárbejegyzések) leírói (név, adatok elhelyezkedése stb.)
- Adatok
 - különféle rendszerindító programok (fájlrendszerekben és azokon kívül)
 - fájlok (és könyvtárbejegyzések) adatai (ezek tárolása a végső cél)
- Partíció
 - a tárolás legnagyobb fizikai egysége
 - fájlrendszer tárolására képes

Tárolás a fájlrendszerben

- A fájlrendszer (FS) felépítése

- FS metaadatok (szuperblokk, master file table, partition control block)
- (rendszerindulási adatok, ha ez egy boot partíció, boot control block)
- fájl metaadatok (inode, file control block, Windows: a master file table része)
- tárolt adatok

szuperblokk	fájl metaadatok	adatblokkok
-------------	-----------------	-------------

- A fájlrendszer metaadatai

A háttértáron

- típus és méret
- szabad blokkok jegyzéke
- fájl metaadatok elhelyezkedése
- állapot
- módosítás információk
- ...

A memóriában

- ami a háttértáron van
- csatlakozási információk
- „**dirty**” jelzőbit
- zárolási információk
- ...

- A fájlrendszer érzékeny a metaadatok elvesztésére (pl. blokkhiba)

- ezért másolatok készülnek, lásd `dumpe2fs /dev/sda1 | grep -i uperblo`
- demo: töröljük és állítsuk helyre a metaadatokat

Fájlok metaadatainak elhelyezése

- **Diszken**

- hitelesítési információk (UID, GID)
- típus
- hozzáférési jogosultságok
- időbélyegek
- méret
- adatblokkok elhelyezkedése (lásd később)

Példa: UNIX inode (index node), Windows Master File Table bejegyzések

- **Memóriában** (nyitott fájl objektum) továbbá

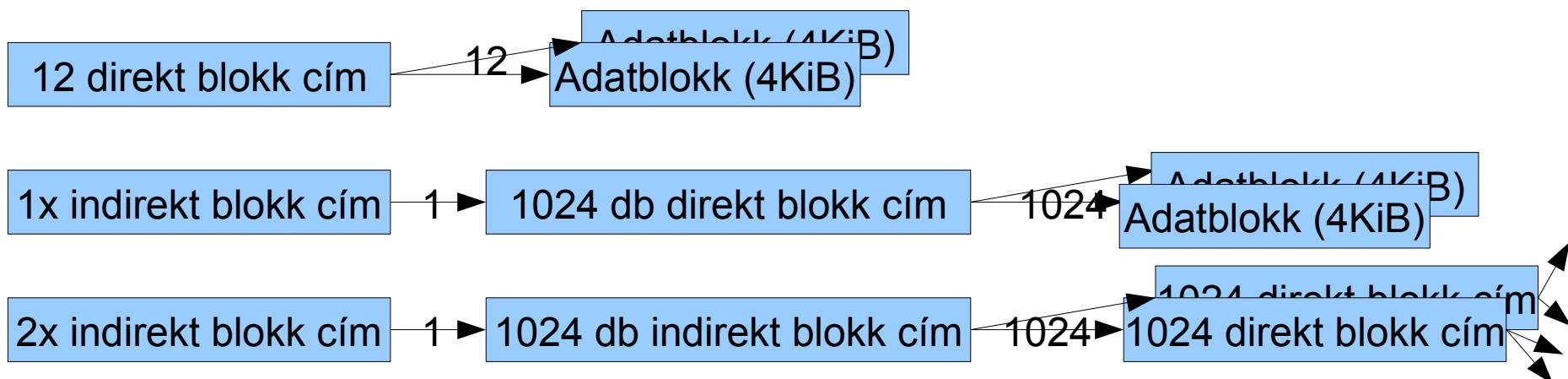
- státusz (zárt, módosított, stb.)
- háttértár eszköz (fájlrendszer) azonosítója
- hivatkozás számláló (fájlleírók)
- csatlakoztatási pont leírója (elfedés)

Adatblokkok allokációja

- Folytonos tárolás
 - törléssel egyre változatosabb méretű üres helyek keletkeznek
- Láncolt listás (soros hozzáférésű)
 - blokkokra bontott tartalom + hivatkozás további adatrészekre
 - pl. egyszeres láncolt lista
 - lassú a sokadik rész elérése
 - soros elérésre hatékony
 - érzékeny a hibákra (láncszakadás)
 - más variációk is léteznek, pl. a **FAT**, amely egy táblában épít listát blokkszámokból
- Indexelt (direkt elérésű)
 - blokkokra bontott tartalom + elhelyezkedési térkép (index)
 - ügyes elhelyezés: szekvenciális
 - szekvenciálisan (gyorsan) olvasható
 - az index segítségével direkt elérésű
 - gond lehet az index mérete
 - a túl nagy index nem fér el egy blokkban
 - láncolt listában tárolhatjuk

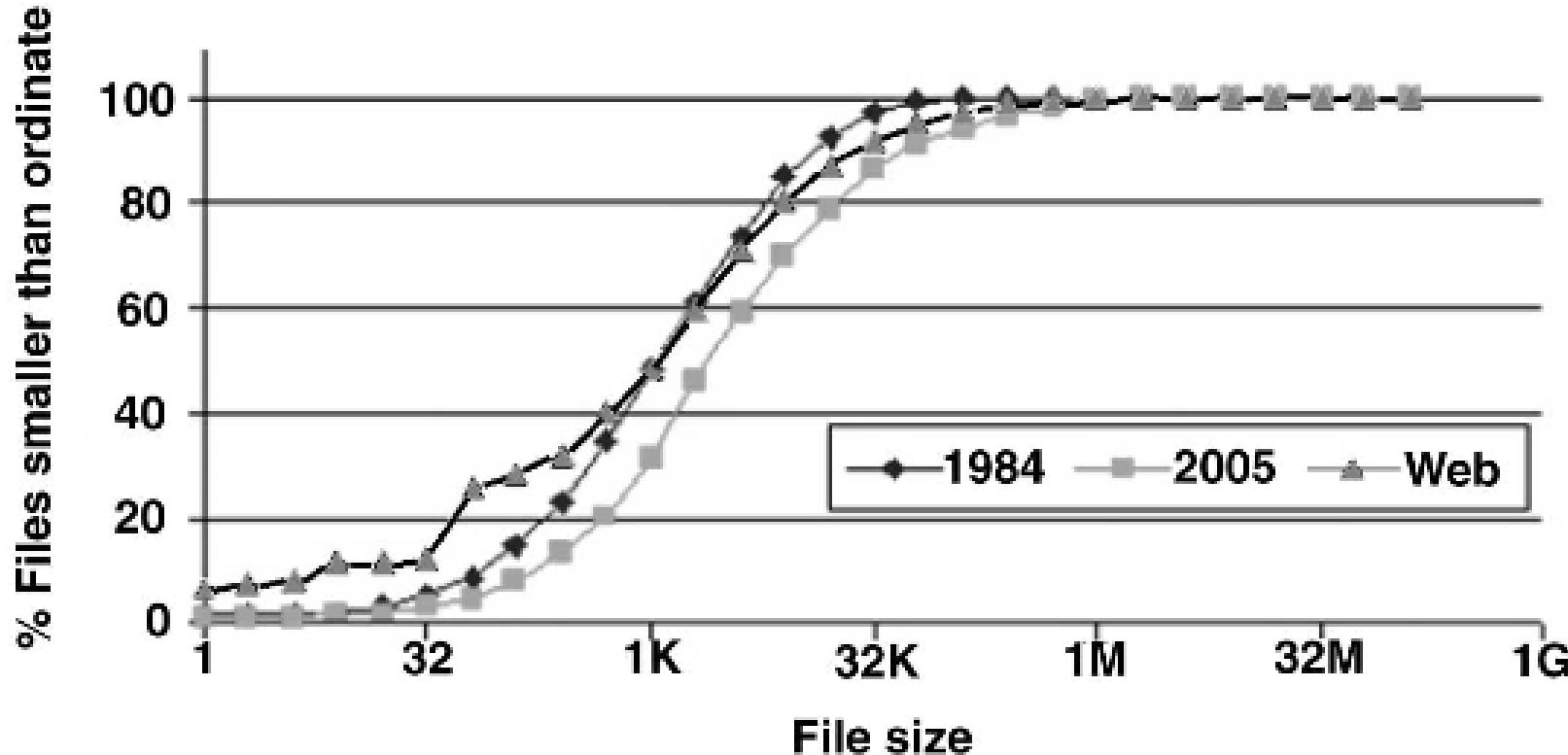
Példa: többszörösen indirekt adatblokk címtábla

- A címtábla és az adattárolás jellemzői (példa)
 - 4 byte-os címek
 - 12 db direkt blokkcím
 - 1x és 2x indirekt blokkcímek
 - 4KiB-os blokkméret ($4\text{KiB} / 4 = 1024$ cím tárolására képes)



„Mekkora a maximális fájlméret?”

Hogyan válasszuk meg az adatblokkok méretét?



Fprrás: Andrew S. Tanenbaum, Jorrit N. Herder, Herbert Bos
File size distribution on UNIX systems: then and now. Operating Systems Review 40(1): 100-104 (2006)

Üres helyek menedzselése

- Bittérképes, bitvektoros
 - egy blokk egy bit (1 = szabad, 0 = foglalt)
 - egyszerű, és könnyű szabad blokkot találni
 - akár a memóriában is elférhet
 - egy CPU utasítás elég lehet
 - nagy fájlrendszereknél egyre kevésbé hatékony
- Láncolt listás
 - minden üres blokk egy következőre mutat
 - csak az első szabad blokk címét kell megjegyezni
 - egyszerű, bár nem a leghatékonyabb módszer (diszkműveletek)
 - összeolvasztható a láncolt listás (pl. FAT) adatblokk nyilvántartással
- Hierarchikus módszerek
 - üres helyek csoportjait kezelik (hasonlítanak a többszörös indexelt címtáblára)
 - csoportok létrehozhatók pl. a fájlrendszer mérete alapján és globálisan kezelhetők
 - csoporton belül egyszerű belső struktúra használható (pl. mind szabad, térkép stb.)

Adatblokkok gyorstárazása

- Diszkpufferelés (disk buffering)
 - a memóriát használja gyorsítótárként: **blokkgyorsítótár (buffer cache)**
 - növeli a hatékonyságot, írásnál csökkenti a megbízhatóságot
- A blokkgyorsítótár szervezése
 - ismétlés: a SZGA „cache szervezés”
 - használhatjuk a virtuális tárkezelés mechanizmusait
egységes blokkgyorsítótár (unified buffer cache) (Linux: page cache)
 - olvasás gyorsítása
előreolvasás (readahead), beállítható, lásd [posix_fadvise](#)
 - nem használt blokkok törlése: pl. LRU
 - írásműveletek kezelése (mikor írjuk ki a háttértárra)
 - **írásáteresztő gyorsítótár (write through cache)**
azonnal háttértárra ír
lassú, de megbízható
 - **pufferelt gyorsítótár**
csak időnként írjuk ki (flush, sync)
nagyobb teljesítményű, de kevésbé megbízható

Metaadatok konzisztenciája

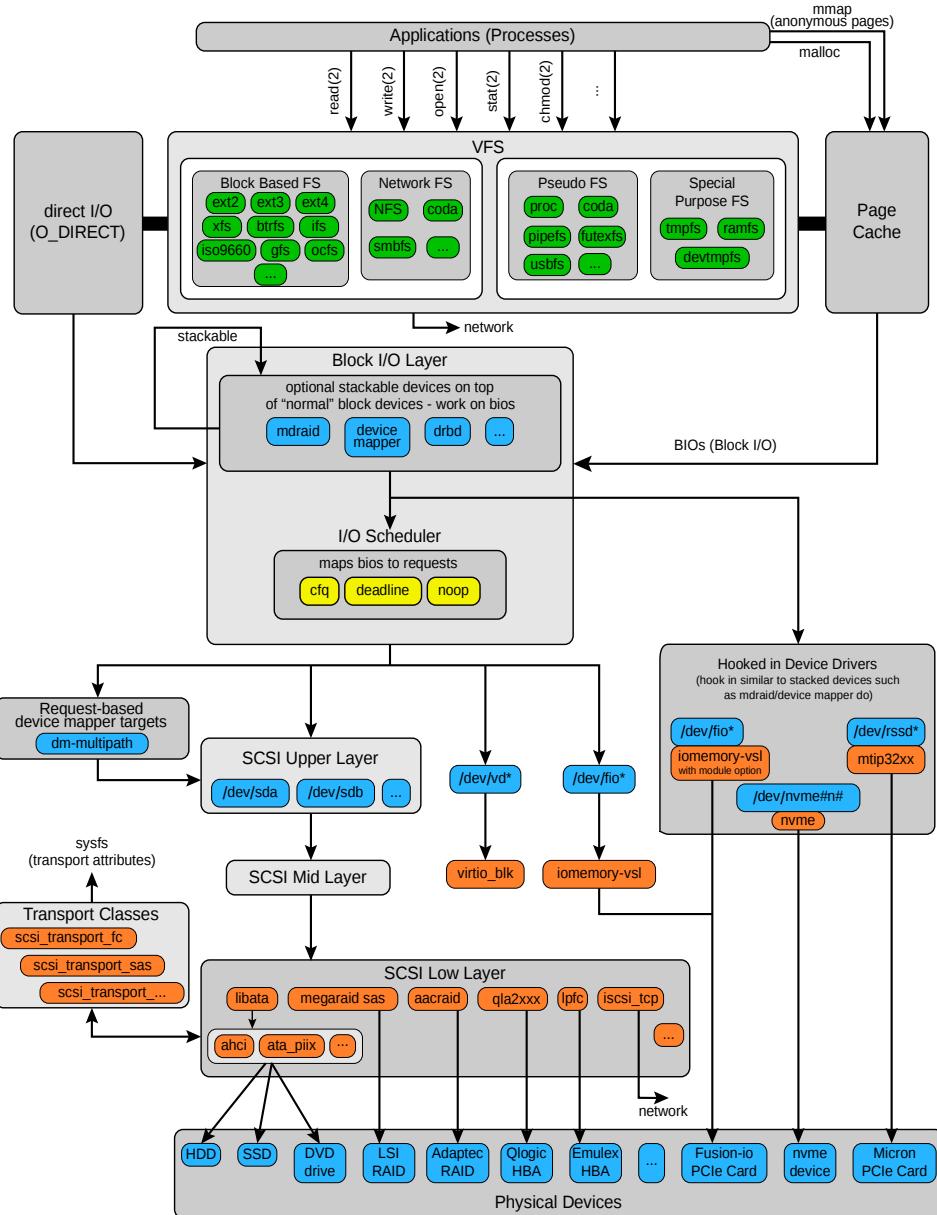
- Konzisztencia-problémák
 - módosított adatok gyorstárazása
 - metaadatok változásai (fájlok, könyvtárak, fájlrendszerek) és a gyorstárazás
- Metaadatok sérülése
 - komolyabb kiterjedésű lehet a hatás
 - pl. tárhelyelszivárgás (storage leak): inode törlés, összeomlás (adattörlés előtt)
 - akár a teljes fájlrendszer összeomlásához is vezethet
- Megoldási ötletek
 - adatok esetén: írásáteresztő gyorsítótár
 - lassítja a működést, de ésszerű, ahol nagy a kockázat
 - metaadatokra is működik?
 - önmagában nem, hiszen hosszabb tranzakciókról van szó

Naplózó fájlrendszer (journaling)

- **Napló (journal)**
 - szekvenciálisan írható körpuffer a **háttértáron**
 - az elvégzendő **műveleteket tartalmazza** (metaadat [+ adat])
 - pl. [NTFS LFS](#), Linux ext3/4 stb.
- Megvalósítás: tranzakcióalapú működés
 - a tranzakció akkor zárul, amikor kiírt minden műveletet a naplóba
 - a naplóba írt tranzakciókat dolgozza fel és hajtja végre a fájlrendszeren
- Mi történik, ha összeomlik a rendszer?
 - Induláskor feldolgozza a naplót.
- **Log-structured fájlrendszer:** a napló a fájlrendszer ([cikk](#), [példák](#))
 - gyors szekvenciális írás (pufferelt), olvasás térképpel, szemétgyűjtés
- Más megoldás is lehetséges: **Copy-on-write fájlrendszer**
 - az írásműveleteket másolt adatokon hajtja végre, majd átírja a metaadatokat
 - pl. [ZFS](#), [btrfs](#)

Áttekintés

- Felhasználói szemmel...
 - végfelhasználó
 - adminisztrátor
 - programozó
- Belső működés
 - fájlrendszer interfések
 - kernel adatstruktúrák
 - a háttértár szervezése
 - virtuális fájlrendszerök
- Adattárolás
 - fizikai tárolók (HDD, SSD)
 - I/O ütemezés
 - tárolórendszer-virtualizáció:
 - helyi (RAID, LVM)
 - hálózati (SAN, NAS)
 - elosztott fájl- és tárolórendszerek



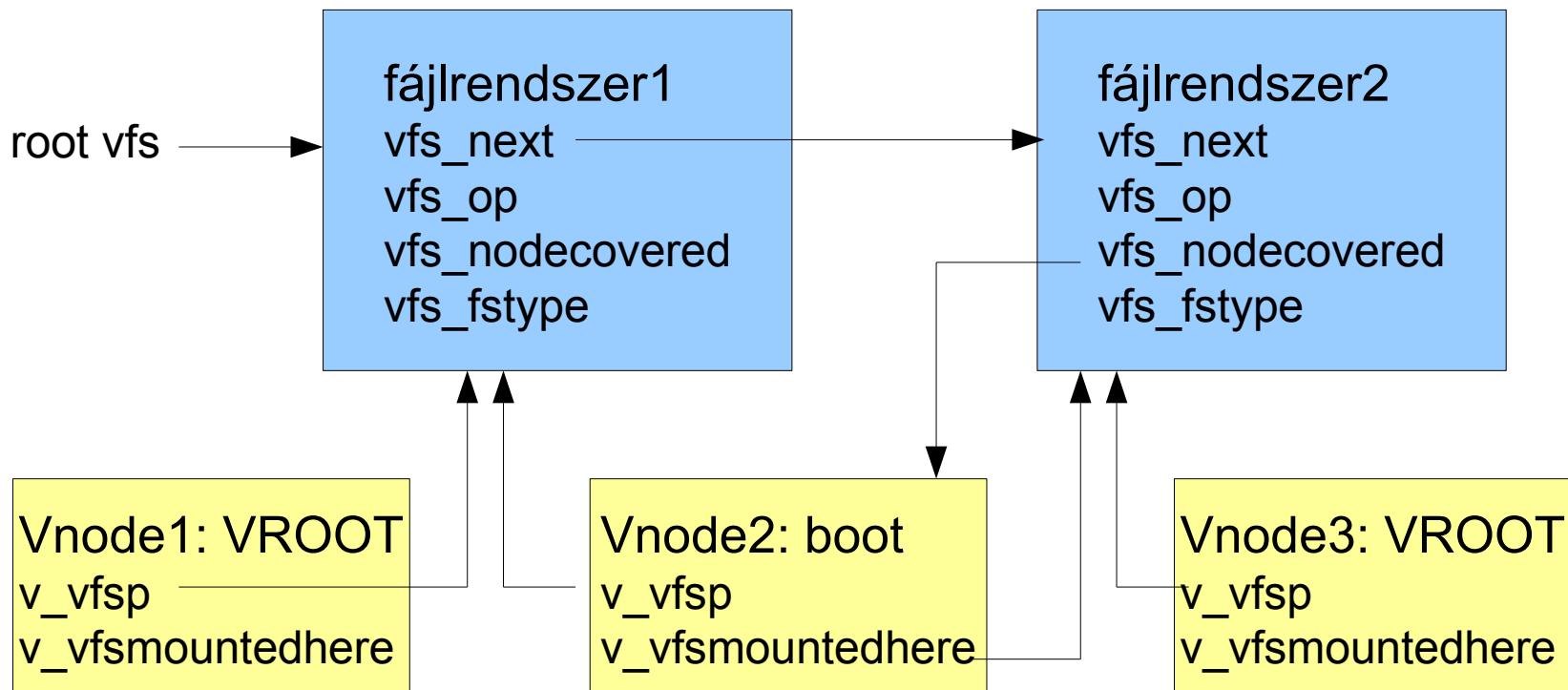
A virtuális fájlrendszer ([VFS](#))

- Nagyon sokféle fájlrendszer létezik
 - főleg UNIX alatt jellemző, hogy egy időben is sokfélét használ
 - a programozóktól (és kernelfejlesztőktől) nem várható el azok különálló kezelése
- A VFS egy implementáció-független fájlrendszer absztrakció
 - a modern UNIX fájlrendszerek alapja
- Célok:
 - többféle fájlrendszer egységes egyidejű támogatása
 - egységes kezelés a csatlakoztatás után (programozó IF)
 - speciális fájlrendszerek uniform megvalósítása (hálózati, /proc stb.)
 - modulárisan bővíthető rendszer
- Az absztrakció lényege
 - fs (fájlrendszer metaadatok) → vfs
 - inode (fájl metaadatok) → vnode

A vnode és a vfs

- **vnode adatmezők**
 - közös adatok (típus, csatlakoztatás, hivatkozás száml.)
 - `v_data`: állományrendszerrel függő adatok (`inode`)
 - `v_op`: az állományrendszer metódusainak táblája
- **vfs adatmezők**
 - közös adatok (fájlrendszer típus, csatlakoztatás, hivatkozás, `vfs_next`)
 - `vfs_data`: állományrendszerrel függő adatok
 - `vfs_op`: az állományrendszer metódusainak táblája
- **virtuális függvények**
 - `vnode`: `vop_open()`, `vop_read()` ,...
 - `vfs`: `vfs_mount` `vfs_umount` `vfs_sync`
 - az állományrendszernek megfelelő hívásokra képződnek le
- **segédrutinok, makrók**

A vfs és a vnode kapcsolata



Speciális VFS fájlrendszerek (példák, demók)

- Milyen fájlrendszereket támogat a Linux?

```
cat /proc/filesystems
```

- devtmpfs és devfs
 - a hardvereszközök elérése fájlrendszeri interfészen keresztül
- procfs
 - taszkok adatainak és kernel adatstruktúrák elérésére
- sysfs
 - kernel alrendszerek elérése fájlműveletekkel
- cgroup, cpuset
 - folyamatcsoportok erőforrás-allokációinak beállítása

```
mount | egrep "cgroup|cpuset"
```

Saját fájlrendszer készítése VFS alapon

- Otthoni gyakorlat (nem nehéz...)
- Dokumentáció

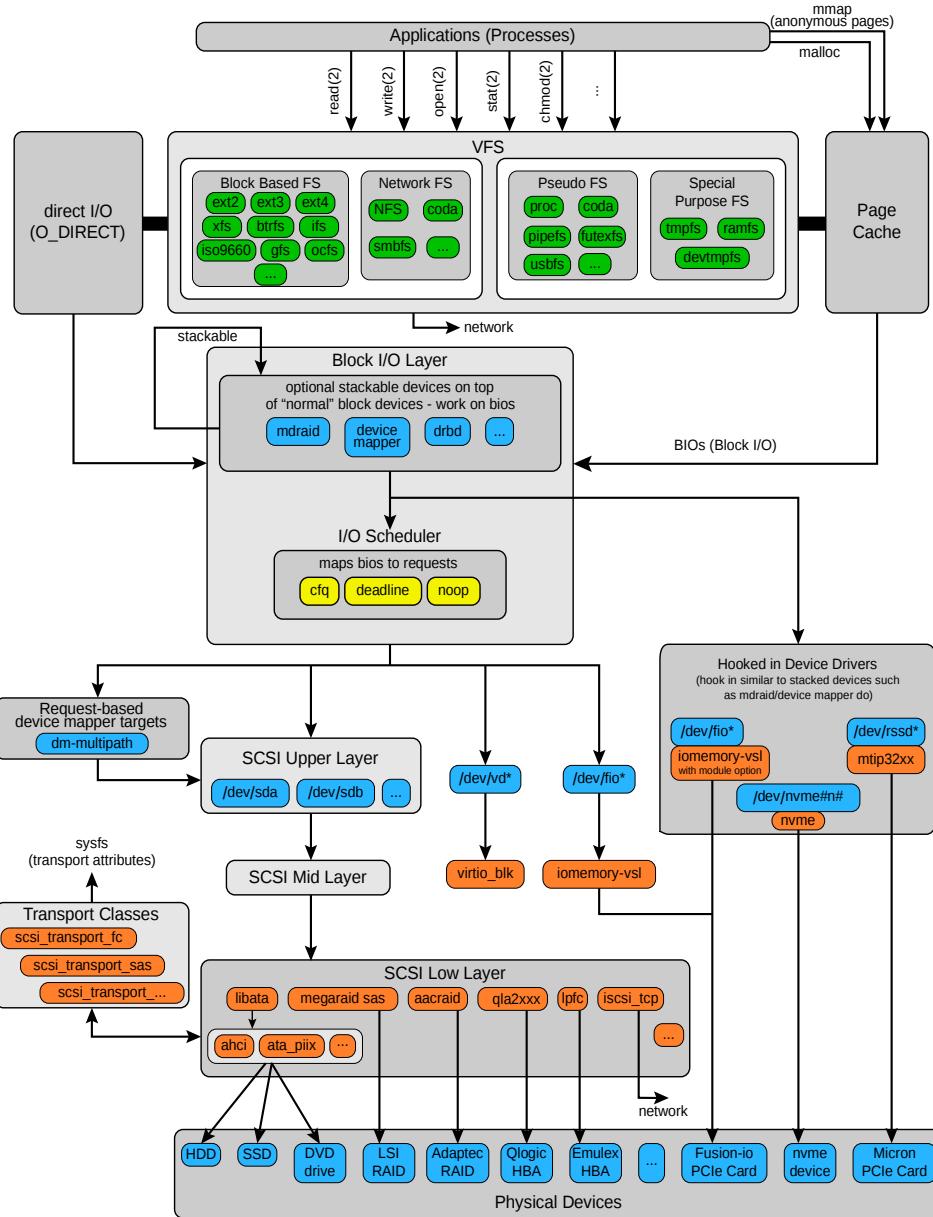
Ravi Kiran, „[Writing a Simple File System](#)”

Steve French, „Linux Filesystems 45 minutes” [ODP](#) [PDF](#)

„A Step by Step Introduction to Writing (or Understanding) a Linux Filesystem”
Az IBM mérnöke, aki SAMBA fejlesztéssel foglalkozik

Áttekintés

- Felhasználói szemmel...
 - végfelhasználó
 - adminisztrátor
 - programozó
 - Belső működés
 - fájlrendszer interfések
 - kernel adatstruktúrák
 - a háttértár szervezése
 - virtuális fájlrendszerek
 - Adattárolás
 - fizikai tárolók (HDD, SSD)
 - I/O ütemezés
 - tárolórendszer-virtualizáció:
 - helyi (RAID, LVM)
 - hálózati (SAN, NAS)
 - elosztott fájl- és tárolórendszerök



Tárolási megoldások a fájlrendszerek mögött

- Fizikai tárolóeszközök
 - mágneses elven működő
 - pl.: HDD és szalagos egységek (tape)
 - optikai elven működő
 - pl.: CD / DVD / Blu-ray
 - nemfelejtő memóriaalapú eszközök
 - pl.: SSD, USB flash diszk, SD kártya stb.
- Virtualizált tárolórendszerek
 - a fizikai tárolóeszközökre építenek egy (vagy több) további szolgáltatási réteget
 - összeolvasztanak tárolóeszközöket
 - megbízhatóság- és kapacitásnövelés érdekében
 - pl. RAID, LVM
 - hálózati elérést tesznek lehetővé
 - fájl vagy blokkszintű adatátvitellel
 - pl. NAS, SAN
 - elosztott tárolási rendszert valósítanak meg
 - megbízható és jól skálázható tárolórendszerek építéséhez
 - pl. Ceph, GlusterFS

Fizikai tárolórendszerek legfontosabb jellemzői

- Teljesítmény

- kapacitás: 32/64 bit → TB
- sebesség: 10 MiB/s → 200 GiB/s
- késleltetés: 0,5 ns → perc

- Megbízhatóság

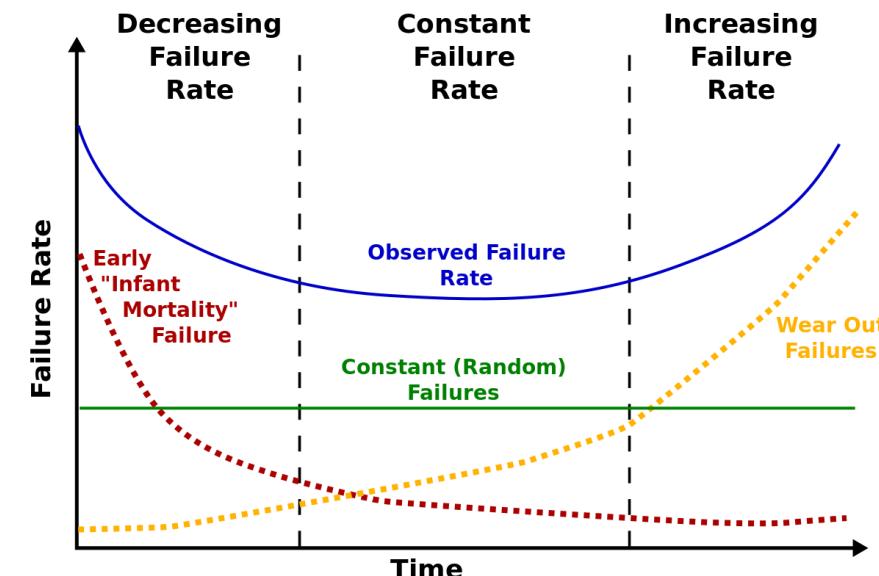
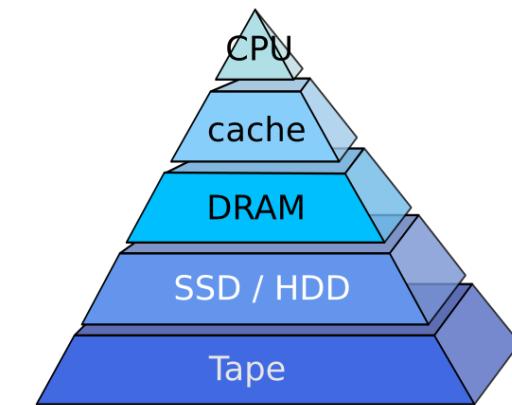
- gyártói és tapasztalati adatok (lásd még [SMART](#))
- **éves hibaarány (annualized failure rate, AFR)**
 - jellemzően 2-4%, esetenként >10% tönkremegy

- **a meghibásodásig eltelt átlagos idő (mean time to failure, MTTF)**

- gyártók: >100 év (eszközhalmazra)
[fürdőkád-görbe](#)
MTTF of 1,000,000 hours?

- **teljes írható adatmennyisége (SSD, tot)**

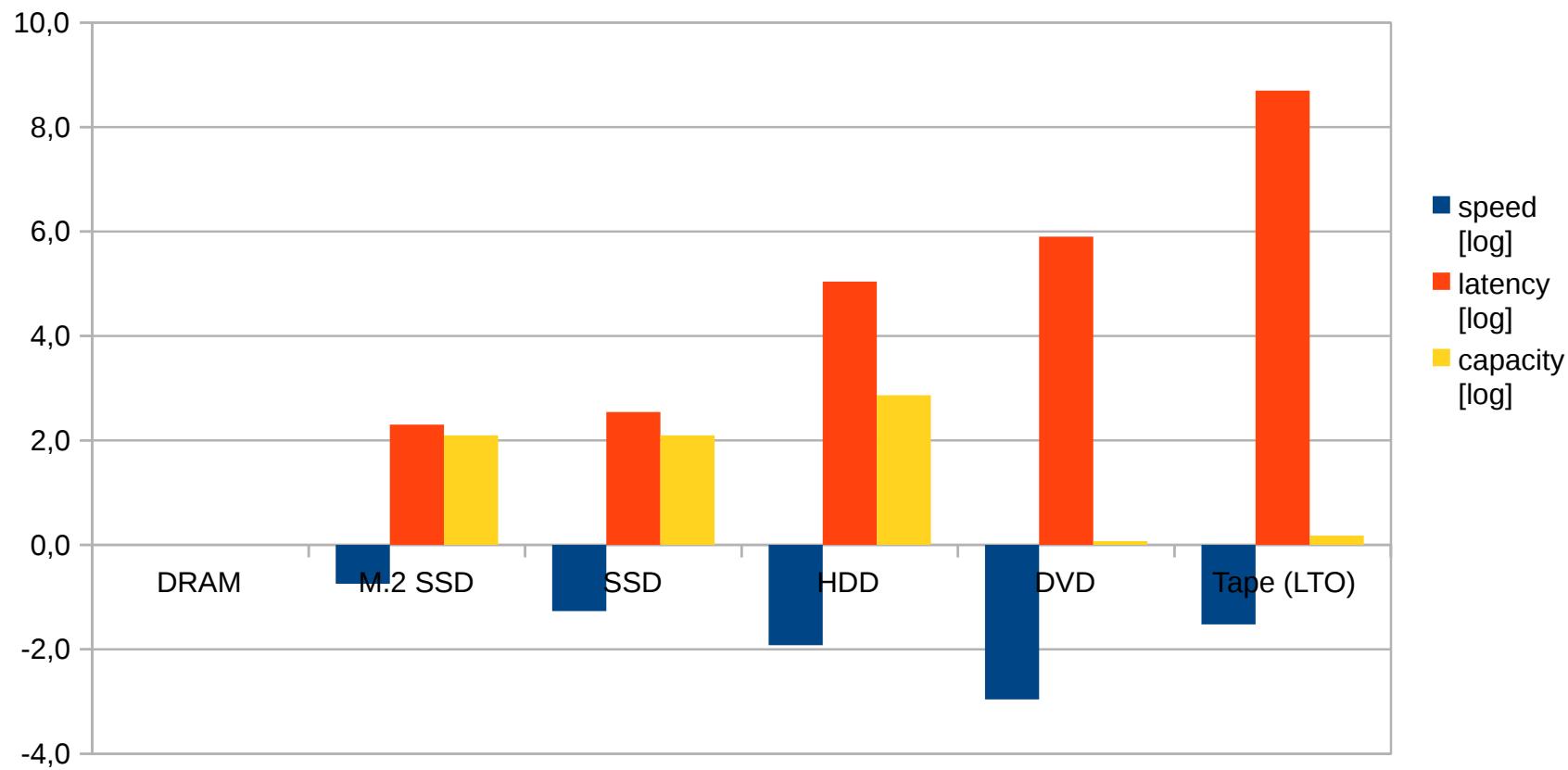
- a memóriacellák korlátos számmal írhatók
 - napi 50GB esetén is [évtizedekben](#) mérhető



Tárolóeszközök teljesítménye

Fizikai tárolók teljesítménye a DRAM-hoz képest

A sebesség, a késleltetés és a tárolókapacitás logaritmikus összehasonlítása

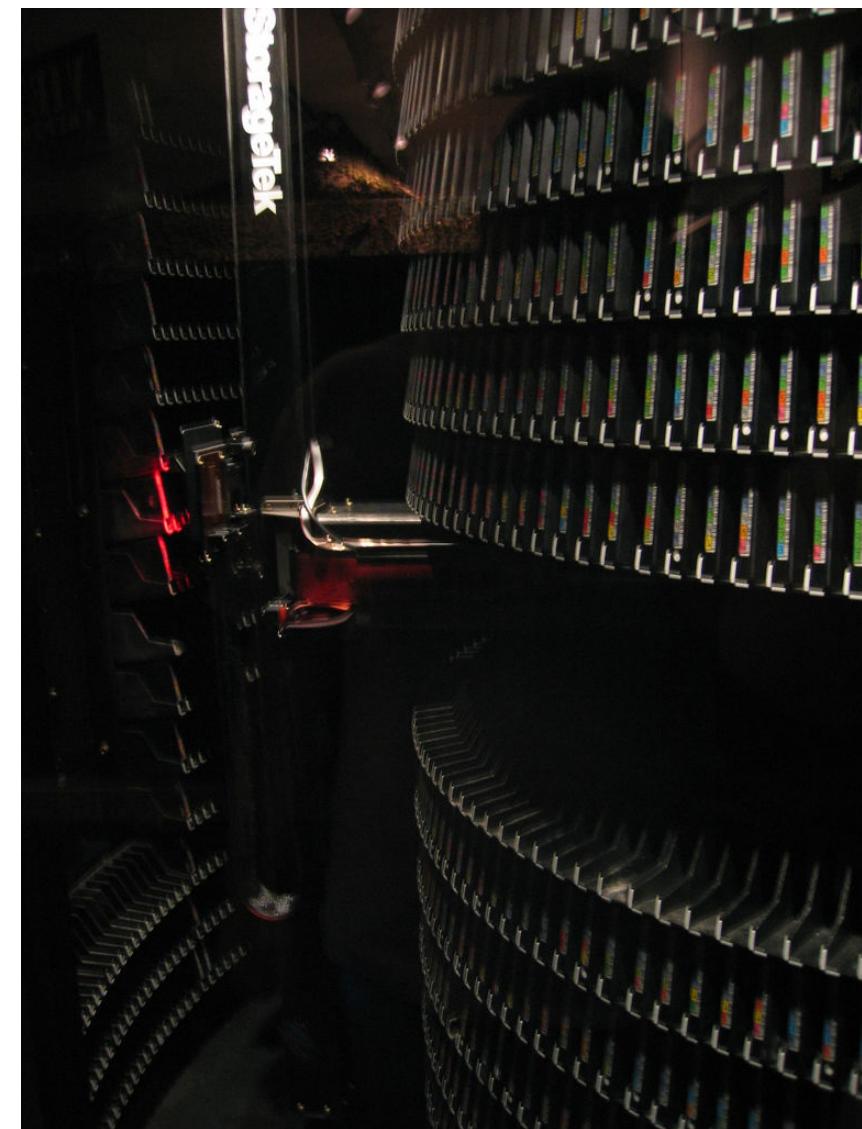


Trendek a fizikai tárolórendszerekben

- „A lassú I/O” évtizedei
 - a processzorok teljesítménye dinamikusan nőtt
 - adatelérési késleltetésük jelentősen csökkent
 - a háttértárak a kapacitásra koncentráltak
- Változó teljesítményviszonyok
 - sok RAM → nagy puffer gyorsítótár
 - gyors CPU → I/O teljesítménynövelés
 - futásidéjű adattömörítés (pl. btrfs, zfs)
 - deduplikáció
 - memóriaalapú háttértárak
 - növekvő sebesség, minimális késleltetés
 - „**storage class memory**”: DRAM-hoz közelítő tárolórendszer
- Következmények
 - Eltűnik az elsődleges – másodlagos határ.
 - Az I/O-ra vár állapot időtartama csökken.

Szalagos tárolók (tape drive)

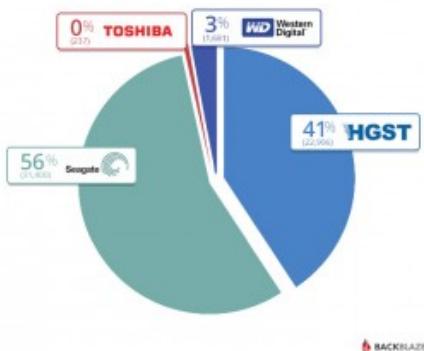
- Biztonsági mentésre
 - nagy kapacitás
 - hosszú élettartam
 - lassú
 - drága automatizálás
- Trendek
 - szekvenciális olvasási sebesség:
 - Tape: 300 MB/s
 - SSD: 500 MB/s
 - HDD → SSD / Tape?
 - nagy puffer gyorsítótárak
 - a taszkok onnan dolgoznak
 - szekvenciális olvasással töltik
 - log-strukturált fájlrendszerek
 - szekvenciálisan ír/olvasadattárházakban **újra feltűnhetnek**



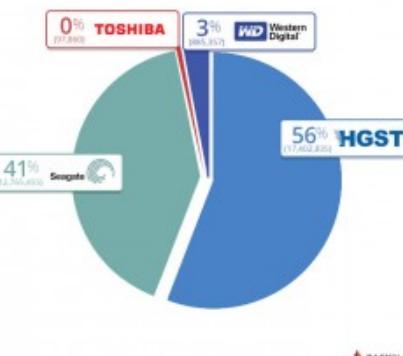
Merevlemezes meghajtók megbízhatósága

A Backblaze adatcenter kb. 56 ezer diszkjének statisztikái

Backblaze Datacenter Drive Count
by Manufacturer
as of 12/31/2015

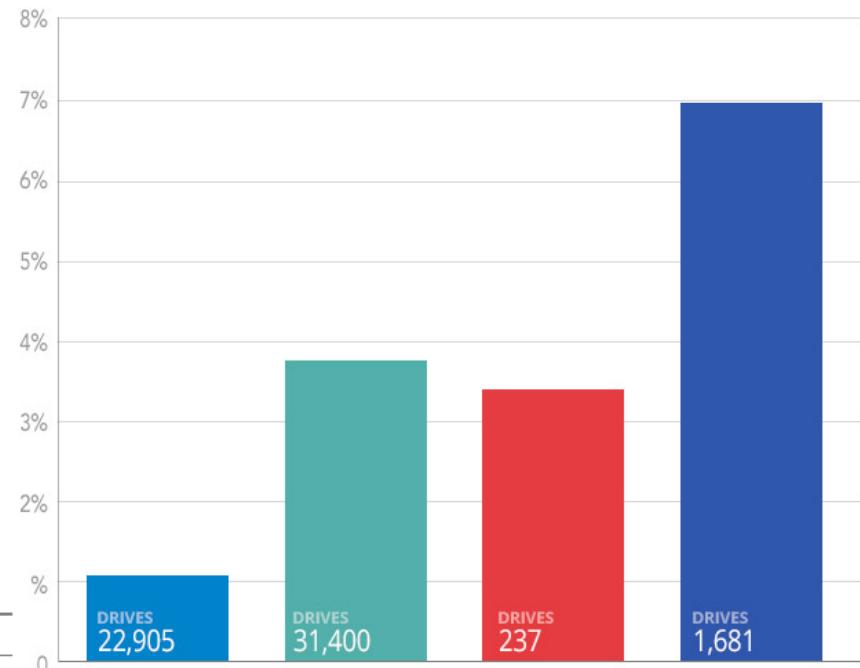


Backblaze Datacenter Drive Days
in Service by Manufacturer
as of 12/31/2015



Failure Rate by Manufacturer

Cumulative from 4/2013 to 12/2015



Meglepő adatok is akadnak:

Cumulative Failure Rate through the Period Ending

MFG	Model #	Highest QTY	12/31/13	12/31/14	12/31/15
HGST	HDS5C3030ALA630	4,596	0.9%	0.7%	0.8%
HGST	HDS723030ALA640	1,022	0.9%	1.8%	1.8%
Seagate	ST3000DM001	4,074	9.8%	28.3%	28.3%
Seagate	ST33000651AS	325	7.3%	5.6%	5.1%
Toshiba	DT01ACA300	58	-	4.8%	3.8%
WDC	WD30EFRX	1,105	3.2%	6.5%	7.3%



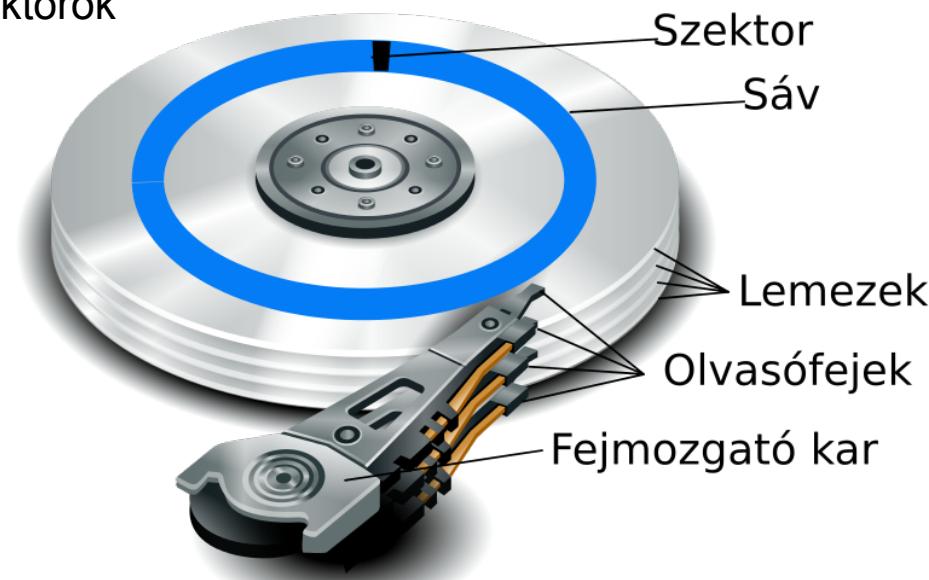
(HGST korábban Hitachi Global Storage Technologies)



Forrás: <https://www.backblaze.com/blog/hard-drive-reliability-q4-2015/>

Allokáció merevlemezes meghajtókon

- Szuperblokk, inode lista és adatblokkok elhelyezése
 - teljesítmény? megbízhatóság?
- Cilinder (blokk) csoport
 - azonos fejpozícióhoz tartozó sávok
 - a fej mozgatása nélkül olvasható szektorok
 - egyszerre sérülnek a fej miatt
- Allokációs elvek
 - szuperblokk másolása minden cilindercsoportba
 - inode lista és szabad blokkok csoportonként kezelve
 - egy könyvtár – egy csoport
 - kis fájlok egy csoportba
 - nagy fájlok „szétkenve” több csoportba
 - új könyvtárnak egy új, kevéssé foglalt csoportot keres



Diszk feladatok ütemezése (Linux)

Noop (FIFO): összevonhat szomszédos kéréseket

- minimális terhelést jelent
- ha a tároló maga is ütemez (pl. HW RAID, NCQ, virtualizált rendszerek stb.)
- ha nincs értelme ütemezni (pl. RAM diszk, SSD)
- ha nincs nagy I/O terhelés (CPU-intenzív rendszer)

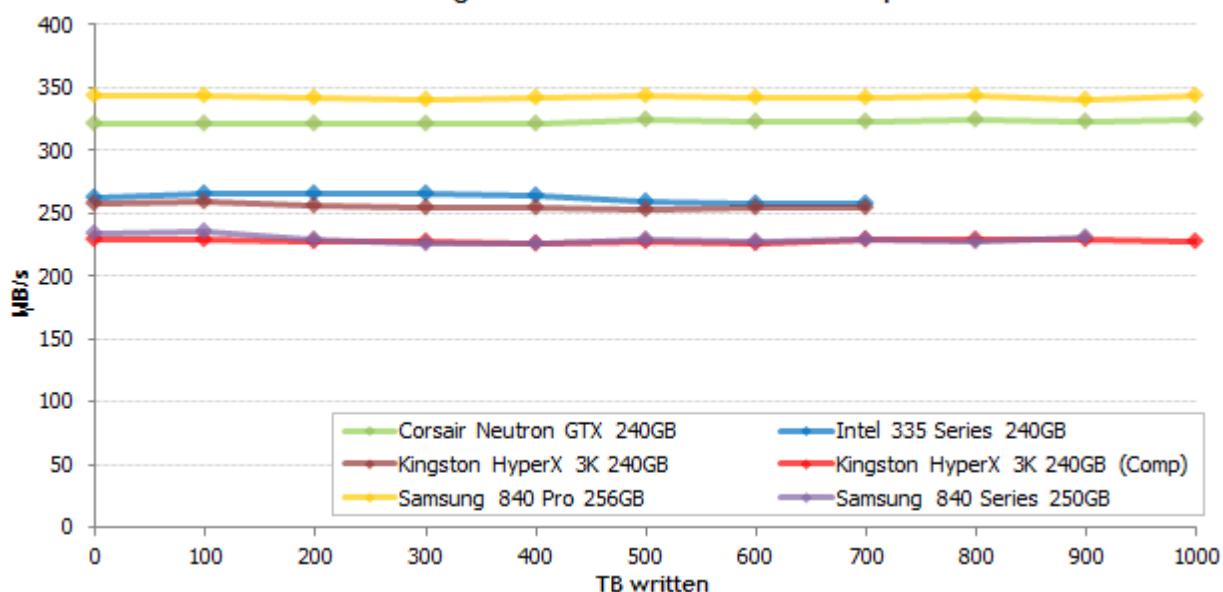
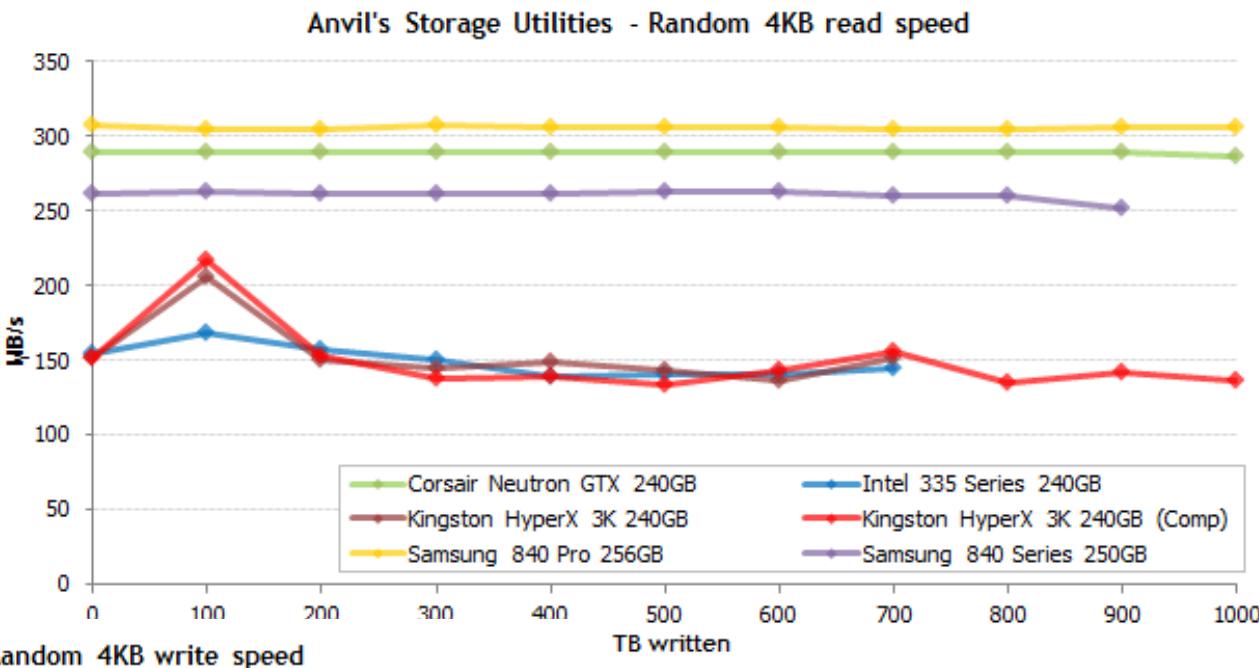
Deadline: késleltetésre optimalizál

- blokkcím szerint rendezett írási vagy olvasási kötegek
- nagy I/O terhelésnél **globálisan** jó (egy taszkra nem feltétlenül előnyös)

CFQ (Completely Fair Queuing): egyenletes kiszolgálás

- folyamatonkénti sorok, azokhoz rendelt I/O időszeletek
- a sorokhoz prediktív előrejelzés
- általános célra egy kiegyensúlyozott ütemező
- jellemzően ez az alapértelmezett
- konfigurálható: man ionice

SSD megbízhatóság

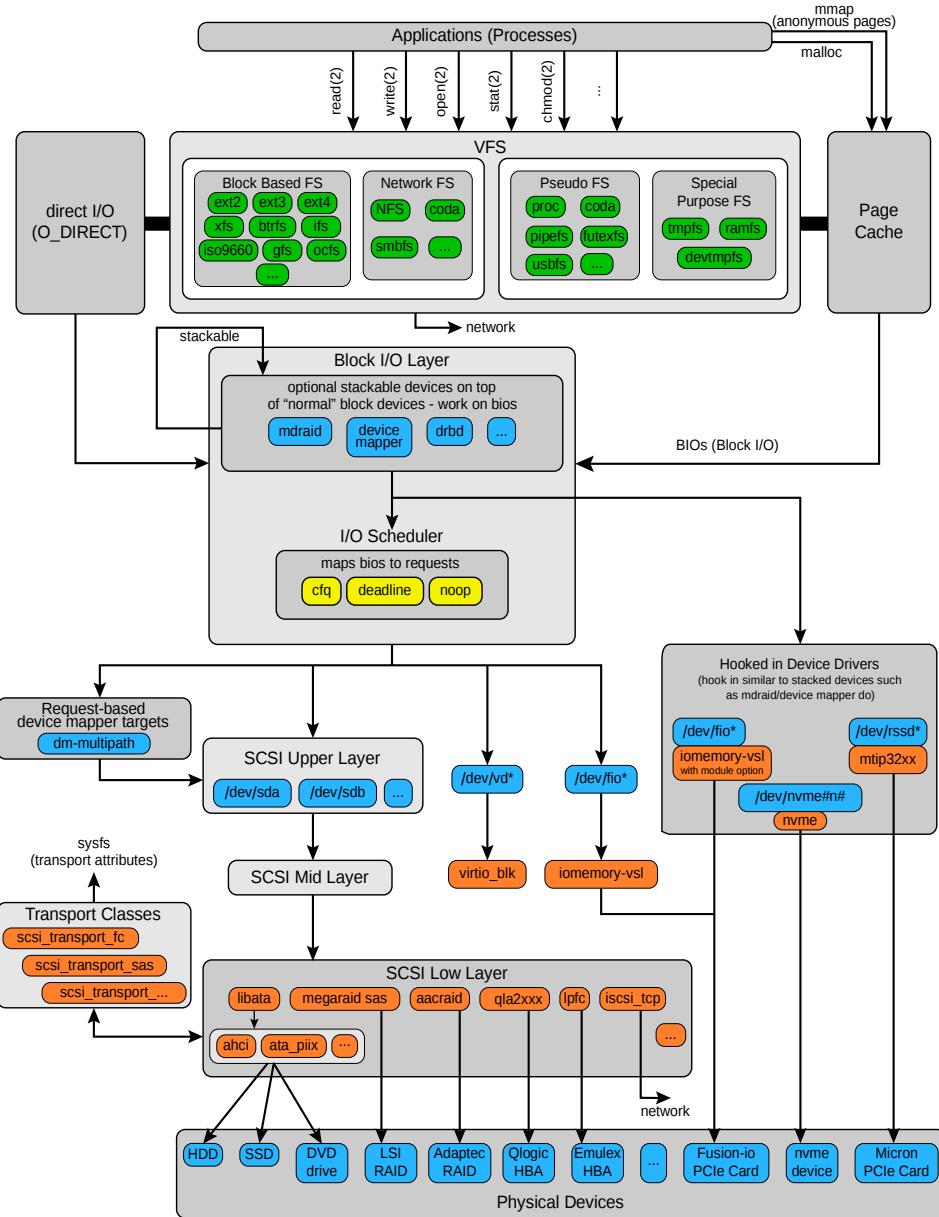


Napi 50GB írás esetén is kb. 40 év a leggyengébb SSD élettartama.

Forrás: <http://techreport.com/review/24841/introducing-the-ssd-endurance-experiment>

Áttekintés

- Felhasználói szemmel...
 - végfelhasználó
 - adminisztrátor
 - programozó
- Belső működés
 - fájlrendszer interfések
 - kernel adatstruktúrák
 - a háttértár szervezése
 - virtuális fájlrendszerek
- Adattárolás
 - fizikai tárolók (HDD, SSD)
 - I/O ütemezés
 - tárolórendszer-virtualizáció:
 - helyi (RAID, LVM)
 - hálózati (SAN, NAS)
 - elosztott fájl- és tárolórendszerek



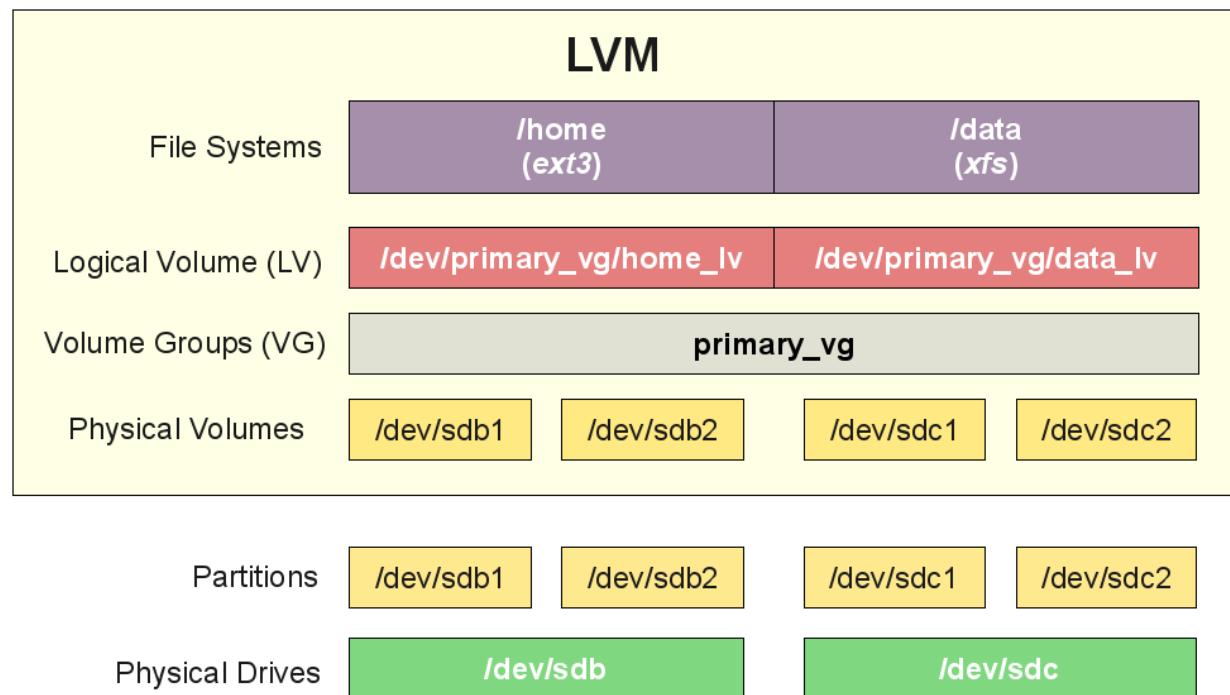
Virtuális tárolórendszerek

- Fizikai tárolórendszerek korlátai
 - kapacitás, teljesítmény, megbízhatóság
 - menedzsment (rugalmatlan)
 - hibaelhárítás
- Virtualizáció
 - a fizikai eszközökre épít egy (vagy több) további szolgáltatási réteget
 - összeolvasztás (kapacitásbővítés)
 - szolgáltatásbővítés
 - jobb menedzsment
- Virtuális tárolórendszer
 - a fizikai eszközök határain átnyúló tárolórendszer
 - példák: LVM, RAID stb.
 - építőelemei
 - fizikai tárolók: diszk, partíció
 - más virtuális tárolók, pl. RAID diszkek

Logikai kötetkezelés (logical volume management)

Windows: Logical Disk Manager Linux: Logical Volume Manager

- **fizikai kötet** (physical volume, PV): diszk, partíció stb.
részei: **physical extent (PE)**
- **logikai kötet** (logical volume, LV): virtuális diszk partíció ← **fájlrendszer**
részei: **logical extent (LE)**, amelyek PE-kre képződnek le
- (logikai) **kötetcsoport** (logical volume group, VG): LV-k halmaza, **a virtuális tároló**

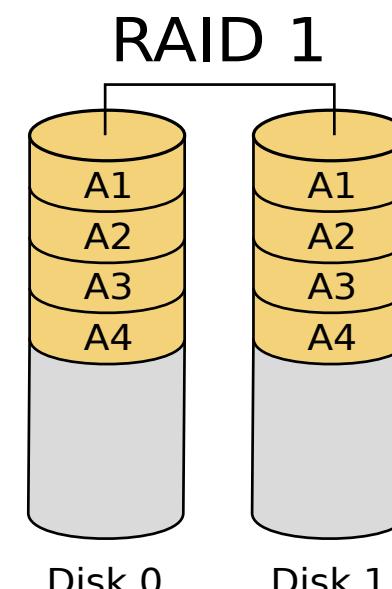
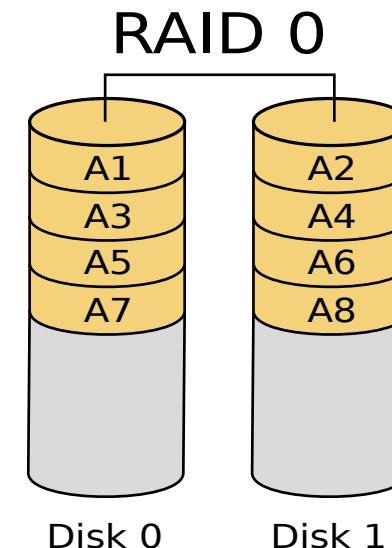


RAID

- Tárolórendszerek megbízhatósága
 - a fizikai eszközök számának növekedésével **nő** a hiba esélye is
 - pl.: 1 diszk MTTF 100 000 óra, 100 diszk MTTF 1000 óra (41 nap)
 - több eszköz → **kisebb nagyobb** megbízhatóság
- Adatredundancia beépítésével elérhető
 - **tükrözés** (mirroring)
 - költséges (a kapacitás jelentősen csökken)
 - **paritás**: hibajelzés mellett javítására is szolgálhat
 - pl. N egyforma blokk mellé 1 blokk paritást rendelünk
- Redundant Array of Inexpensive Disks
 - virtuális tárolórendszer
 - „olcsó” („kis”) merevlemezek egybeolvasztásával
 - I = Independent, a RAID diszkek nem olcsók
 - cél: **redundancia** (megbízhatóság) és a **teljesítmény**
 - hardveres és szoftveres megvalósítása is létezik
 - az alaplapi RAID szoftveres
 - hardveres RAID nem olcsó

RAID szintek: 0 - 1

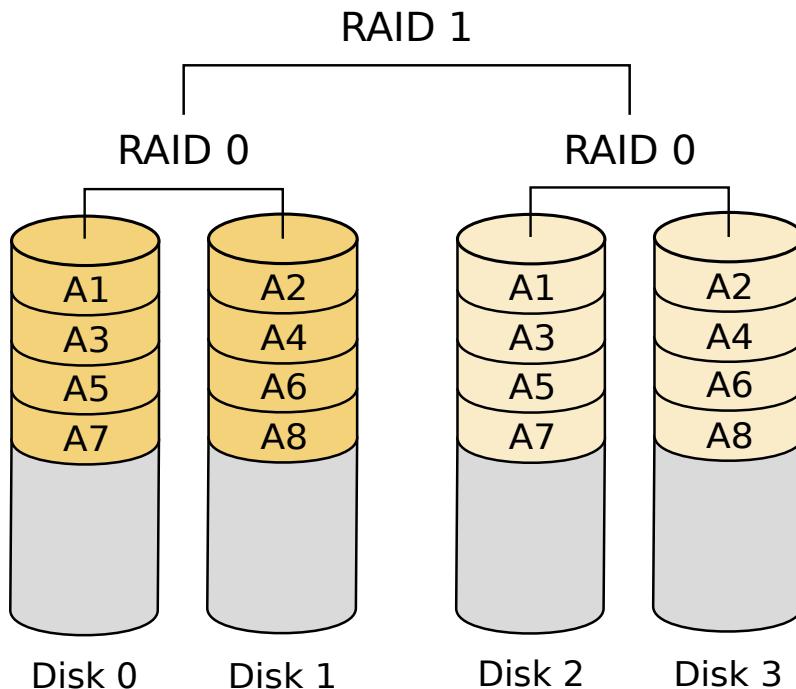
- **RAID szint (RAID level)**
 - az egybeolvasztás módja
- **RAID 0 (stripe, „csíkozás”)**
 - N diszken egyenletesen terít
 - cél: a teljesítmény növelése
 - a diszkek kapacitása összeadódik
 - diszkhiba esetén az adat elvesz
 - átgondoltan alkalmazandó
- **RAID 1 (mirror, tükrözés)**
 - az adatokat többszörözve tárolja
 - cél: megbízhatóság
 - mérete egy diszk kapacitása
(a többi másolatot tárol)
 - az írás lassul (másolatok)
 - az olvasás mérsékelten gyorsabb



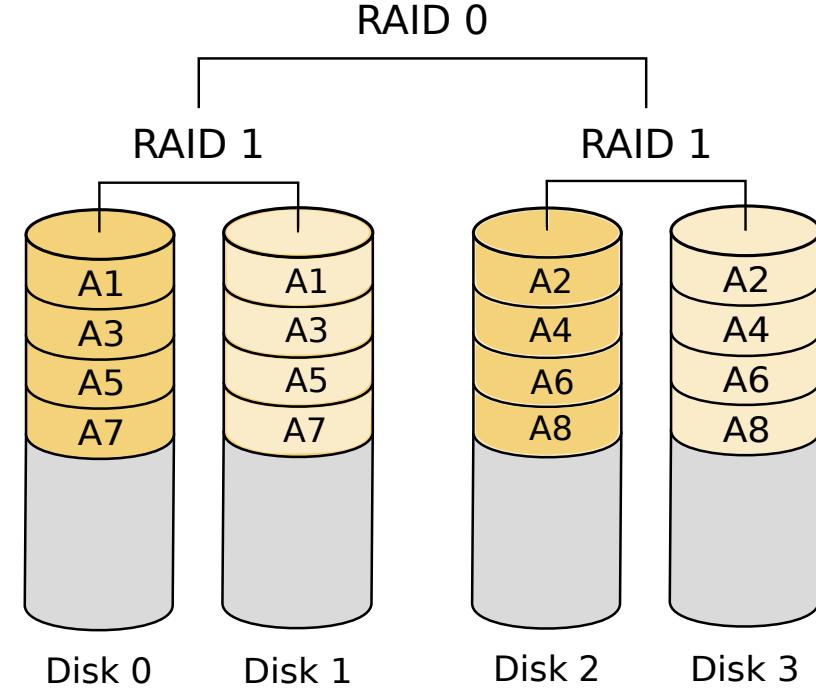
RAID szintek: 01 és 10

- A két alap RAID szint ötvözhető is:
 - RAID 01 (0+1)**: „mirror of stripes”
 - elvi felépítés, a gyakorlatban nem használt
 - RAID 10 (1+0)**: „stripe of mirrors”
 - egyszerűbb I/O-intenzív rendszerekben ajánlott

RAID 0+1



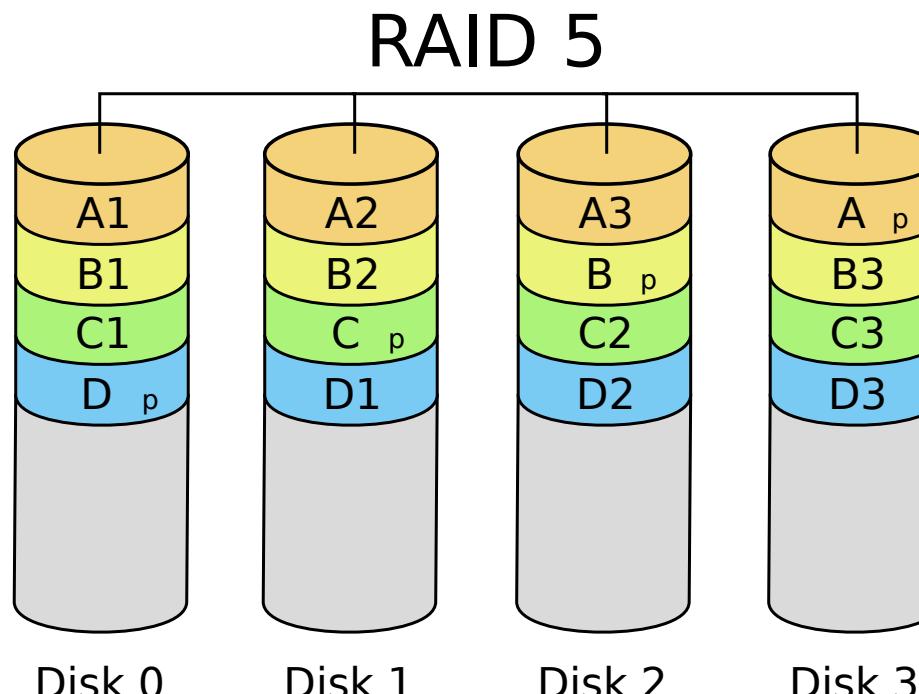
RAID 1+0



RAID 5: Blokkszintű csíkozás egy paritással

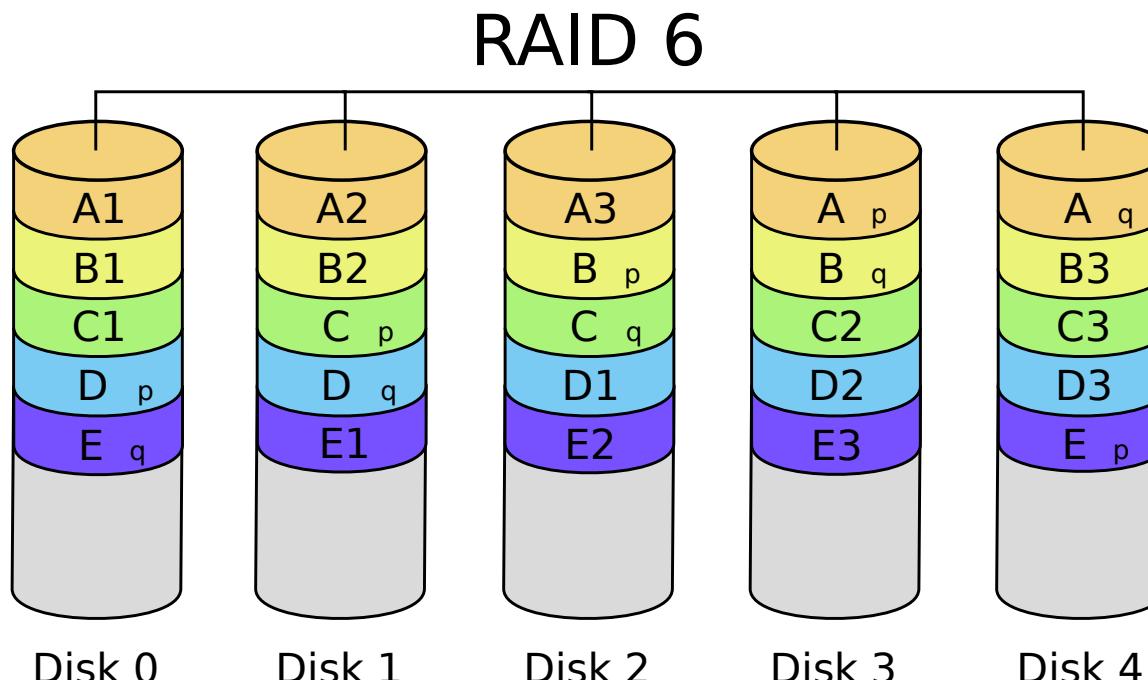
- Felépítés
 - N adatblokk + 1 paritásblokk ($N+1$ diszk)
 - a paritásblokkokat egyenletesen („csíkozva”) helyezi el a fizikai diszkeken
- Jellemzés
 - a teljesítménye a RAID0-hoz közelí
 - a kapacitás egy diszk méretével csökken
 - egy diszk meghibásodása ellen véd

„néma hiba” (silent error)



RAID 6: blokkszintű csíkozás két paritással ($N+2$ diszk)

- Felépítés
 - RAID5 + második paritásblokk
- Jellemzők
 - két diszk hibája ellen véd
 - jó teljesítmény, mérsékelt kapacitáscsökkenés
 - a helyreállításkor jelentkező néma hiba ellen is véd



A RAID korlátai

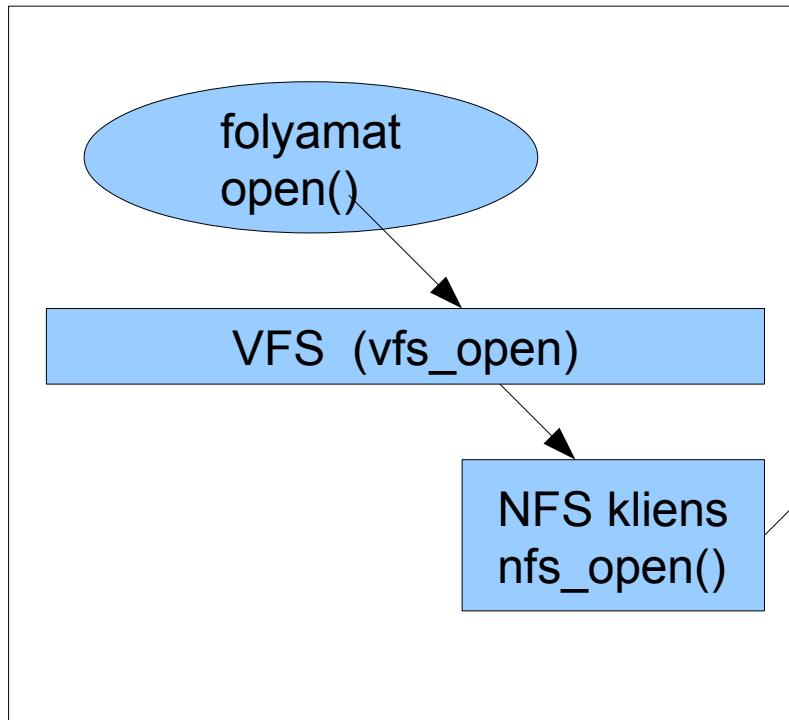
- Mennyi ideig tart egy RAID5 hiba javítása 4+1 diszk esetén?
 - 150GB-os diszkek: kb. 10 óra (egy hosszú éjszaka)
 - 6TB-os gyorsabb diszkek: kb. 80 óra (több nap)
 - meleg tartalék (hot spare) + RAID6 a legjobb, de nem elég
- Sok egyforma diszket kíván
 - egyre nehezebb a pótlás, a HW RAID nagyon érzékeny erre
- Kötött redundanciájú, nem rugalmas
 - RAID5 → RAID6 migráció futásidőben?
- A tárolókapacitás nem növelhető nagyra
 - 6-8 diszk / HW RAID kártya
 - a kiépítés fizikai korlátai
- Csak diszkhiba ellen véd
 - alaplap, CPU, memória, táp stb.?

Hálózati és elosztott tárolórendszerek

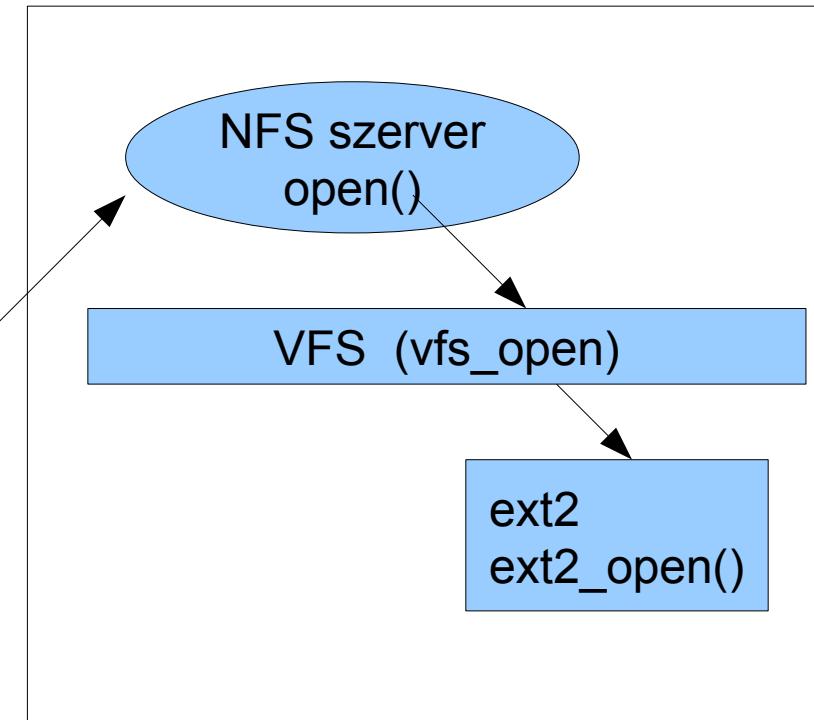
- Kliens-szerver modell
 - szerver: helyi tároló → hálózat → kliens
 - fájltároló: **NAS (Network Attached Storage)**
 - NFS** (Network File System)
 - SMB/CIFS** (Common Internet File System)
 - blokktároló: **SAN (Storage Area Network)**
 - iSCSI** (internet SCSI): SCSI parancsok IP-alon
 - FCP** (Fibre Channel Protocol) és **FCoE**: SCSI átvitel FC / ethernet-alapon
- Elosztott fájlrendszer (distributed file system)
 - elosztott rendszerként működő tárolórendszer
 - Ceph** (Inktank, Red Hat, SUSE), Google **GFS**, RedHat **GlusterFS**,
 - Windows **DFS**, PVFS (parallel virtual FS) → **OrangeFS**
- Kérdések, problémák
 - késleltetés
 - hálózati hibák
 - konzisztencia

Az NFS megvalósításának egyszerűsített felépítése

gép 1



gép 2



Hálózati fájlrendszerk elméleti kérdései (ízelítő)

- Hogy/hol érjük el az adatainkat?
 - **elhelyezkedés-átlátszóság** (location transparency)
 - az adatok címzése (fájlok megnevezése) nem utal az elhelyezkedésükre
 - **elhelyezkedés-függetlenség** (location independence)
 - a címzés, elnevezés nem változik az adatok áthelyezésével
- Ki teljesíti a kéréseket?
 - távoli szolgáltatás (központi)
 - ahol az adat → egyszerű
 - kommunikációs gondok
 - késleltetés, csomagvesztés, sorrend változása
 - (részben) helyi átmeneti tárral
 - helyi másolaton → nehezebb
 - tárolható helyileg?
 - írás, konzisztencia (több másolat)?
- Hogyan működik a hálózati kiszolgáló?
 - **állapotot tároló** (stateful): a fájlműveletek előélettel rendelkeznek, gyorsabb
 - **állapotmentes** (stateless): örökifjú, megbízhatóbb

Elosztott, skálázható tárolórendszerek: Ceph

- Virtualizált tárolórendszer (szoftveres) többféle eléréssel
 - blokkos tárolóeszköz (SAN)
 - fájltárolás (NAS)
 - objektumtár (OSD, az adatok objektum-alapú tárolására, az alap)
- Skálázható és hibatűrő (nincs leállás)
 - nincs sebezhető pontja (*single point of failure*)
 - minden komponense futásidőben cserélhető, bővíthető (új diszk, gép)
 - futásidőben változtatható a replikáció mértéke (hány másolat legyen)
- További előnyei
 - PB (petabájt, 1000 TB, 10^{15} bajt, 76 évnyi 720p H.264 videó) kapacitás
 - sokkal gyorsabban felépül a hardver hibákból, mint a RAID tömbök
 - nem igényel speciális hardvereket (lásd RAID kártya és diszk)
 - nem szükséges tartalék hardverek beépítése (lásd RAID spare disk)
 - együttműködik a virtualizációs rendszerekkel ([OpenStack](#), [Amazon S3](#))
 - nyílt forráskódú (a technológia mögötti céget [megvette](#) a RedHat)

A Ceph alapja a **RADOS** tárolási rendszer

- RADOS: Reliable, Autonomic Distributed Object Store
- OSD (object storage device): adattárolási csomópont
 - CPU + memória + lokális diszk (jellemzően diszkenként egy OSD)
- Tárolóegység (placement group, PG): az objektumok tárolóhelye
 - az OSD-kre épülő elosztott (logikai) tárolási hely (pár OSD sok PG)
 - meghatározza a replikáció mértékét
- Klasztertérkép: a tárolási rendszer leírása (minden csomópontban)
 - az OSD-k és PG-k lista (milyen építőelemekből épül fel a klaszter)
 - a tárolt adatok elhelyezkedése (mit és hol tárol a rendszer)
- Monitor: felügyelő, menedzsment komponens
 - kezeli és szükség szerint módosítja a klasztertérkép mesterpéldányát
 - jellemzően annyi példány van belőle, ahány fizikai gép alkotja a klasztert
- Az objektumok (adatok) elhelyezése (CRUSH algoritmus)
 - kvázi véletlenszerűen egyensúlyozva a tárolóegységek (PG) között
 - a tárolóegységen belül az ún. OSD térkép segítségével (replikáció)
 - automatikus áttelepítés kiesett vagy újonnan belépő eszközök esetén

Merre tovább, fájl- és tárolórendszerek?

- Integrált fájl- és tárolórendszerek
 - a fájlrendszerek, LVM és RAID megoldások integrált megvalósítása
 - pl. zfs, btrfs
- Skálázhatóság
 - a tárolókapacitás dinamikus növekedése, különösen virtualizált rendszerek alatt
- Megbízhatóság
 - nagy tárolókapacitás, sok diszk → sok hiba jelentkezik
 - csökkenteni kell a hibajavítás (diszkcsere, adatmozgatás) okozta kiesést (nullára)
- Memóriaalapú tárolók
 - lásd bevezető előadás: a háttértárrak és a fizikai memória sebessége közelít
- **Data deduplication** (pl. zfs, btrfs)
- További ajánlott olvasmányok érdeklődőknek:
 - Microsoft **ReFS** (Resilient File System), [ezen az oldalon](#) is
 - Solaris **ZFS** (Z File System, eredetileg Zettabyte...), [FreeBSD-n](#) és [Linuxon](#) is
 - Linux **Btrfs** (B-Tree File System, „butter F S”)
 - **F2FS** (Flash-Friendly File System, Samsung)
 - **GPUfs** (fájlrendszerek elérése GPU-n, lásd heterogén rendszerek)

Operációs rendszerek: a virtualizáció alapjai

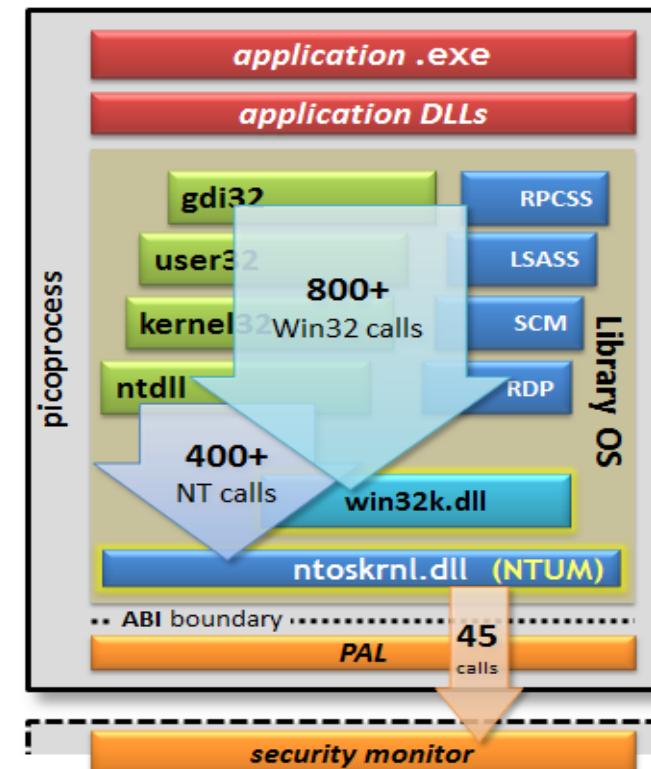
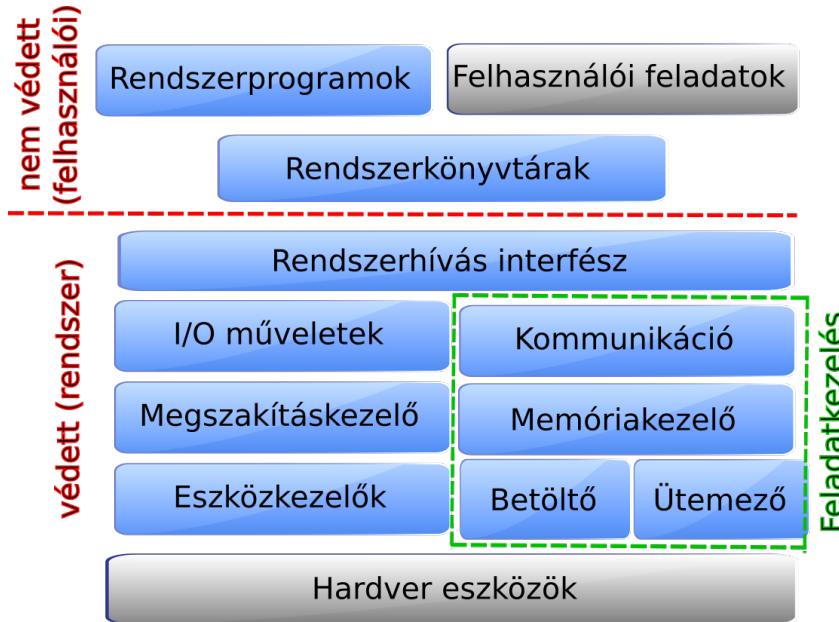
Mészáros Tamás
<http://www.mit.bme.hu/~meszaros/>

Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

Az előadásfóliák legfrissebb változata a tantárgy honlapján érhető el.
Az előadásanyagok BME-n kívüli felhasználása és más rendszerekben történő közzététele előzetes engedélyhez kötött.

Az eddigiekben történt...

- Az OS multiprogramozott
 - szeparálja a taszkokat
 - absztrakt virtuális gép koncepció
- Erőforrások
 - CPU + védelmi szintek
 - memória + MMU + VMM
 - tárolórendszer virtualizáció



- Komplexitás → hibák, költségek
 - fejlesztés (l. architekturális részek)
 - üzemeltetés (l. laborok)

Mi a virtualizáció?

File Virtual Machine Help

Home vSphereAdmin

Welcome To VMware Player

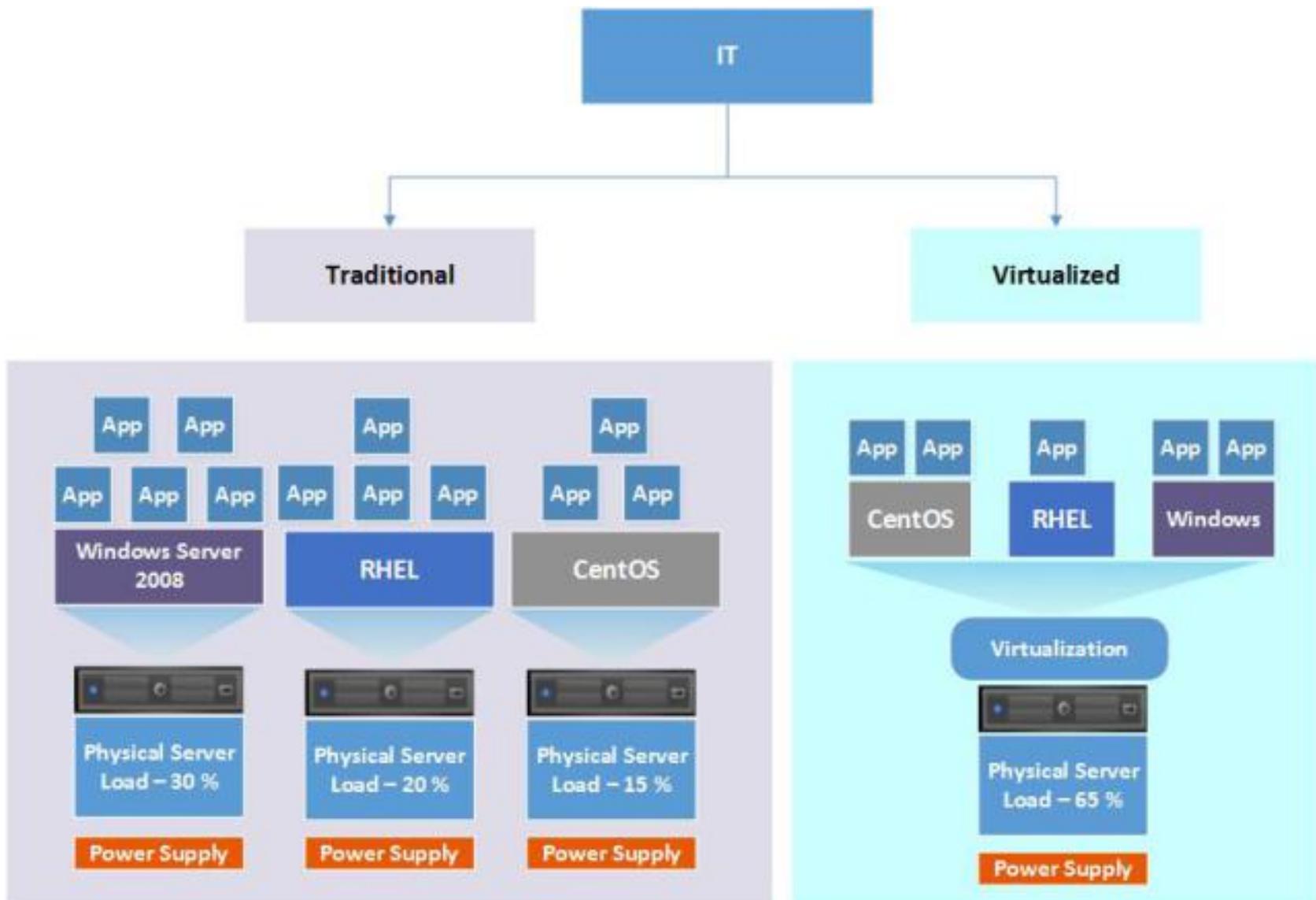
Server size: Small

	VMWare
Hypervisor	
Operációs Rendszer/Sablon	CentOS 7.x 64bit
Virtuális CPU	1
RAM	1 GB
Lemezterület	20 GB
Hálózati forgalom	Sávszélesség 1000 Mbit/s 2 TB/hónap ingyenes
Erőforrások költsége	300,00 Ft
Szoftver Licencek	--,-- Ft
Költség per Hónap	300,00 Ft + ÁFA

Search... 

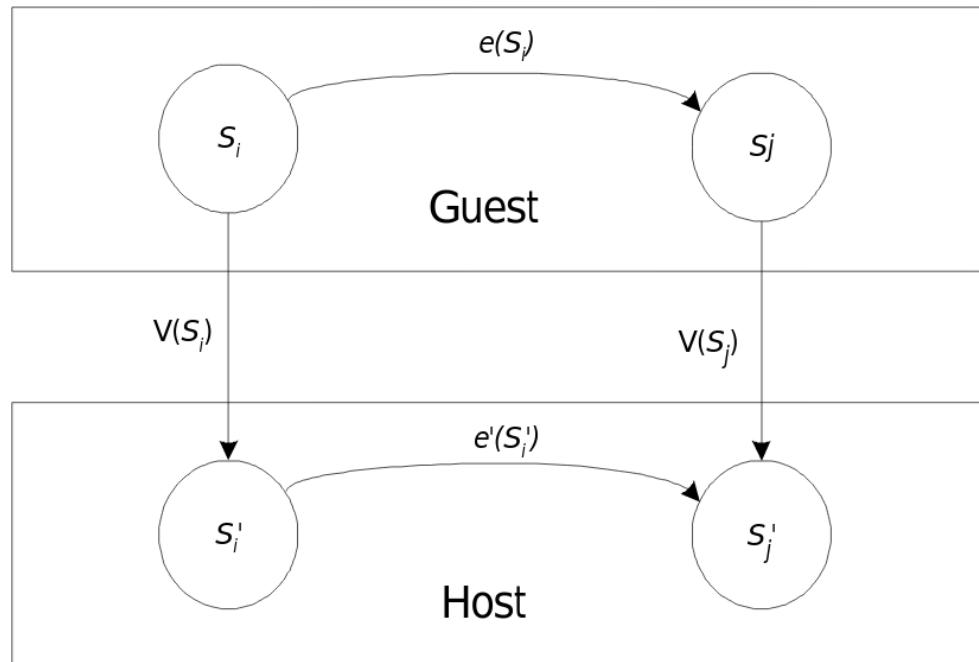
Mi a virtualizáció?



Mi a virtualizáció?

Erőforrás virtuális (szoftveres) változatának létrehozása, amely az eredetire támaszkodva, ahoz hasonlóan, de attól elválasztott módon működik.

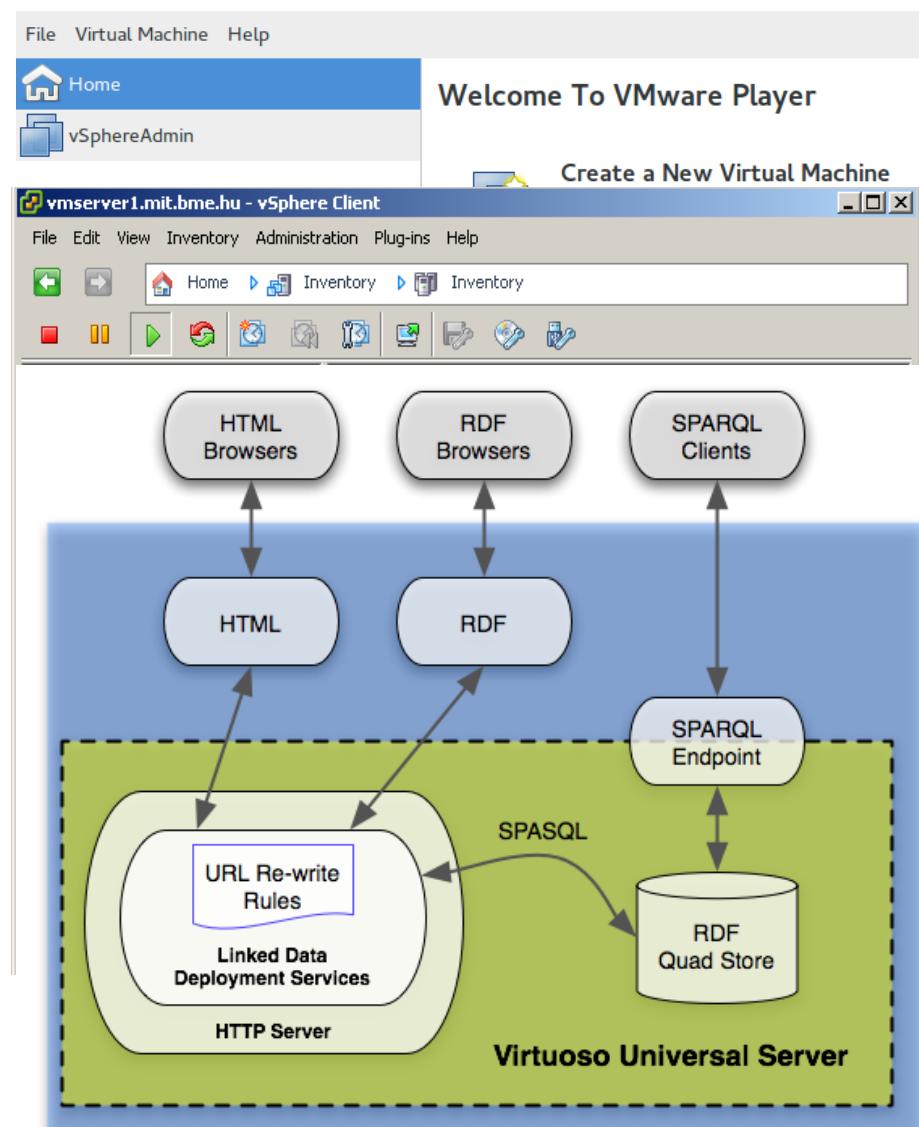
virtualizált erőforrás: számítógépes hardver vagy szoftver
host (gazda): a virtualizált erőforrást biztosítja
guest (vendég): az erőforrás felhasználója



Forrás: Smith, Nair: Introduction to Virtual Machines

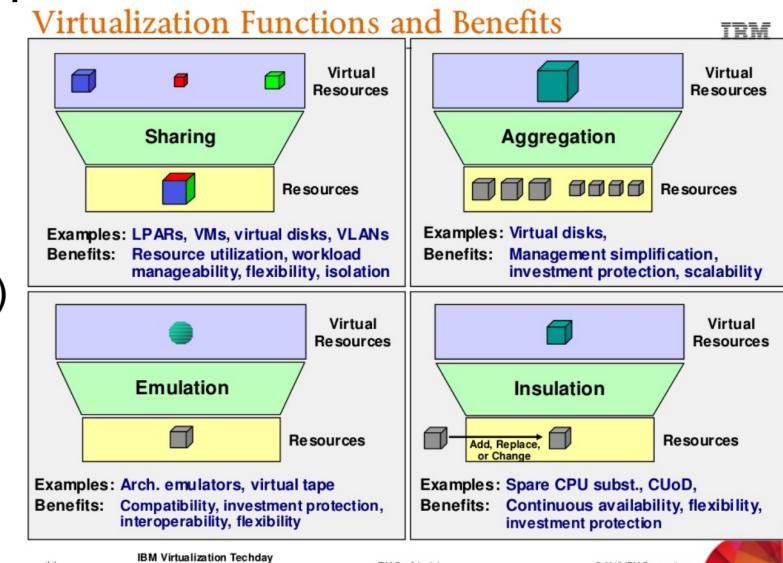
Mi virtualizálható?

- Hardver
 - teljes számítógép
 - számítógépes hálózat
 - grafikus kártya
 - stb.
- Szoftver
 - egy szolgáltatáshalmaz, azaz API
 - lehet rendszerkönyvtár (pl. GUI), kernel (vagy egy része) is
- Adat
 - formátum- és elhelyezkedésfüggetlen
 - hozzáférés és módosítás
- Egy már virtualizált rendszer is

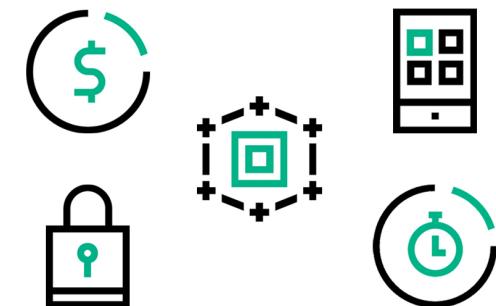


Mire jó a virtualizáció?

- Konkurens erőforrás-használat (→ multiprogramozott OS)
 - egyszerre többen használhatják az erőforrást
 - kezeli a versenyhelyzeteket
- Összeolvasztás
 - kapacitásbővítés (lásd még tárolórendszerek)
 - szolgáltatások fúziója
- Szolgáltatásbővítés és -szűkítés
 - csak ami kell, akár többfélét összegyűrva
 - újfajta aggregált szolgáltatásokat megvalósítva
- Felügyelet, menedzsment (→ OS)
 - szabályozott, automatizált
 - szereplők, jogosultságok
- Archiválás
 - „dobozba zárva” megőrizhető



5 benefits of virtualization



Miért jó a virtualizáció?

- Erőforrás-kihasználtság (→ OS)
 - több használó, kevesebb „üresjárat”
- Csökken a gyártófüggőség
 - helyettesíthető erőforrások
- Csökken az erőforrások száma
 - kevesebb hiba és energiafelvétel
- Jobb menedzsment
 - automatizálható, egyszerűsíthető
- Nagyobb izoláció
 - kisebb, szeparált támadási felületek
- Hatékonyabb rendszerfejlesztés
 - (fél)kész komponensek polcról
 - egyszerűbb tesztelés és archiválás

Csökkenő költségek

TCO: total cost of ownership
beruházás
fenntartás (menedzsment)

Növekvő rendelkezésre állás

kezelhetőbb hibák
megbízhatóbb rendszerek

Növekvő flexibilitás

rugalmasabb specifikáció
skálázható, adaptív rendszerek

Kockázatok és mellékhatások

támadható és hibaforrás (SPOF)
van rezsiköltsége
komplex lehet a kezelése

A virtualizáció főbb fajtái

- Rendszer (system / platform / full)
 - teljes rendszer virtualizációja
 - teljes környezetet („élettér”) biztosítása feladatok végrehajtására
 - az erőforrás egy teljes rendszer
 - feladatok: OS és taszkok
 - pl. VMware Player
- Folyamat (process / software)
 - API / ABI virtualizációja
 - taszkok működéséhez biztosít felületet
 - az erőforrás egy működéshez szükséges (futtatási) felület
 - pl. Java VM
- Infrastruktúra (infrastructure)
 - (jellemzően fizikai) infrastrukturális elemek virtualizációja
 - az erőforrás egy hardver/szoftver elem
 - pl. számítógépes hálózat, adattároló rendszer stb.

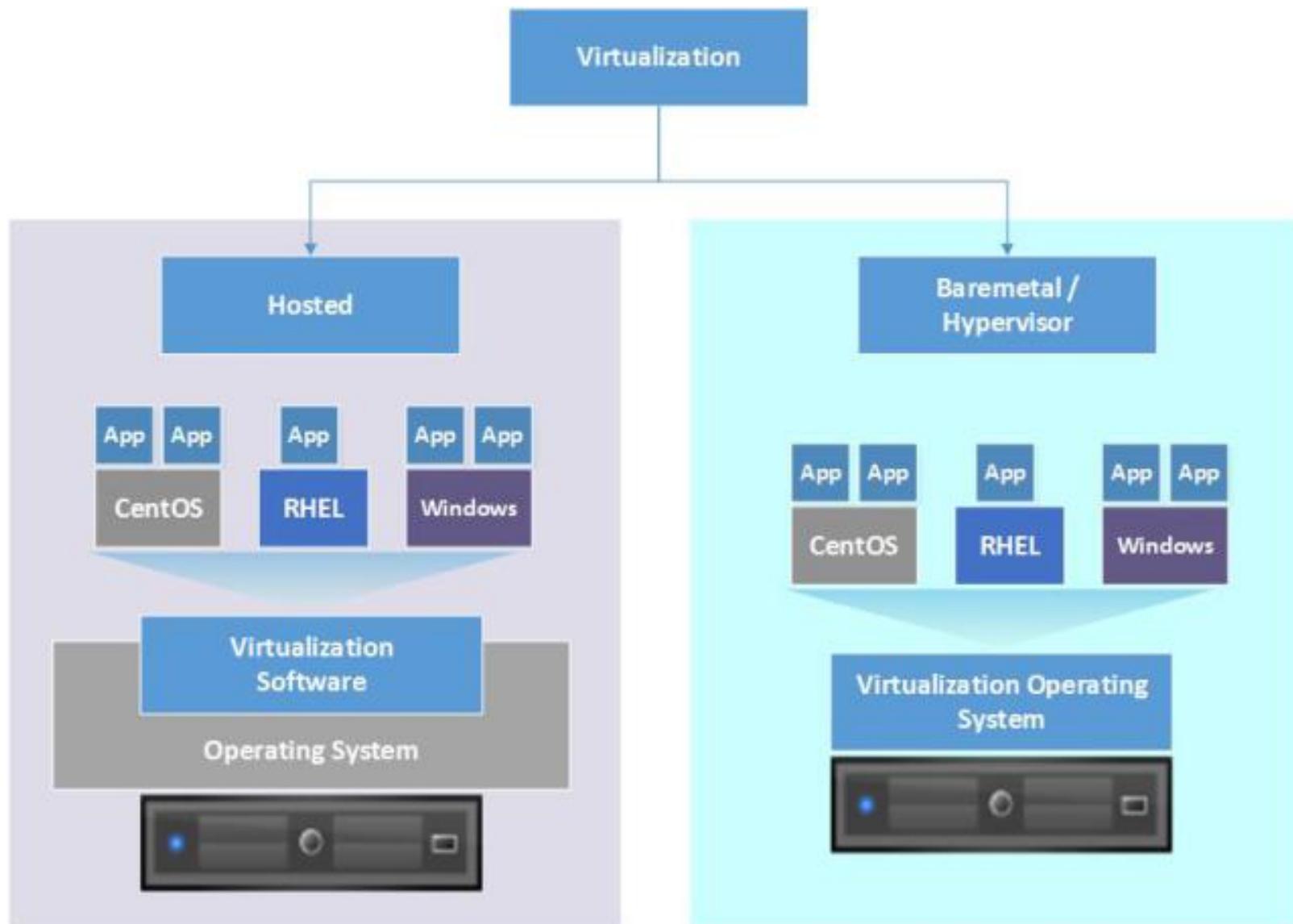
Rendszer virtualizáció

- Cél: teljes virtualizált környezet felépítése
 - (jellemzően) **hardver virtualizáció**: teljes számítógép virtualizálása
 - **virtuális gép**: a virtualizált hardveren futó rendszer
 - a virtuális gépek használata a fizikai gépekhez hasonló módon történik
- Összetevői
 - **gazda számítógép** (host): a fizikai gép, amelyen a virtuális gépek futnak
 - **vendég gép** (guest): a gazdagépen futó virtuális gép
 - **virtuális gép monitor** (VMM): a virtuális gépeket felügyelő program
- Sokféle altípus
- Példák
 - Vmware Player, Xen, KVM, Hyper-V és ezernyi más
- Értékelés
 - egyszerű → nagyon elterjedt
 - hardvertámogatás? teljesítmény?

A hardver virtualizáció fajtái

- *Bare metal (1. típusú)*
 - a hardvert a VMM kezeli
 - a gazdagépen nem futnak más alkalmazások
 - a VMM neve ebben az esetben **hypervisor**
 - fizikai – virtuális hardver megfeleltetése
 - transzparens módon: **natív virtualizáció**
 - hardveres támogatással, vagy futásidejű bináris átírással
 - más hardver képében: **paravirtualizáció**
 - a fizikai hardverhez hasonló, de nem megegyező virtuális hardver
- *Hosted (2. típusú)*
 - a hardvert egy OS kezeli
 - a VMM egy alkalmazás a gazdagépen (pl. VMware Player)
 - a gazdagépen más alkalmazások is futhatnak (több VMM is)
- *Hibrid megoldások*
 - a hypervisor-ral egybeépítve is működik egy kernel, így
 - a VMM egyes funkciót célszerű lehet az OS kernelre építve megvalósítani

Bare metal vs. hosted virtualizáció



A virtualizáció megvalósítása

A virtualizáció megvalósítása: elvárások

- Transzparencia
 - a vendég gép változtatás nélkül működjön
 - a programokat ne kelljen kézzel átírni
 - legyen automatikus és láthatatlan az utasítás-átírás

→ sok feladatot ró a virtualizációs rendszerre

- Védelem
 - vendég ↔ gazda, vendég ↔ vendég
 - pl. natív virtualizáció és a HALT utasítás → ne álljon le a gazdagép

→ felügyelet, jogosultságok megvalósítása

- Hatékonyúság
 - a VMM rezsiköltsége legyen kicsi
 - az átírás minél kevésbé csökkentse a teljesítményt

→ a hardvertámogatás minél teljesebb kihasználása

A virtualizáció megvalósítása: CPU

- Tiszta emuláció
 - virtuális hardveren (állapotgépen) hajtja végre az utasításokat
 - az utasításokat leképezi (lefordítja) a fizikai eszköze
 - **nem hatékony**
- Trap and emulate
 - utasítások válogatása **futásidőben** (végrehajtás közben)
privilegizált: elkapja és átírja; nem védett: közvetlenül végrehajtja
 - hatékony, de **hardvertámogatást igényel**
- Bináris átírás
 - a VMM privilegizált utasításokat **végrehajtás előtt** (de futásidőben) átírja
 - a CPU már a biztonságos utasításokat hajtja végre
 - az átírás valamelyest csökkenti a hatékonyságot
- Forráskód-átírás (paravirtualizáció)
 - a vendég OS forráskódját alakítják át fejlesztési időben
 - a privilegizált utasításokat VMM hívásokra cserélik
 - hatékony, de fejlesztői támogatást igényel (mai OS-ekben jellemző)

Példa a bináris átírásra

Guest Code

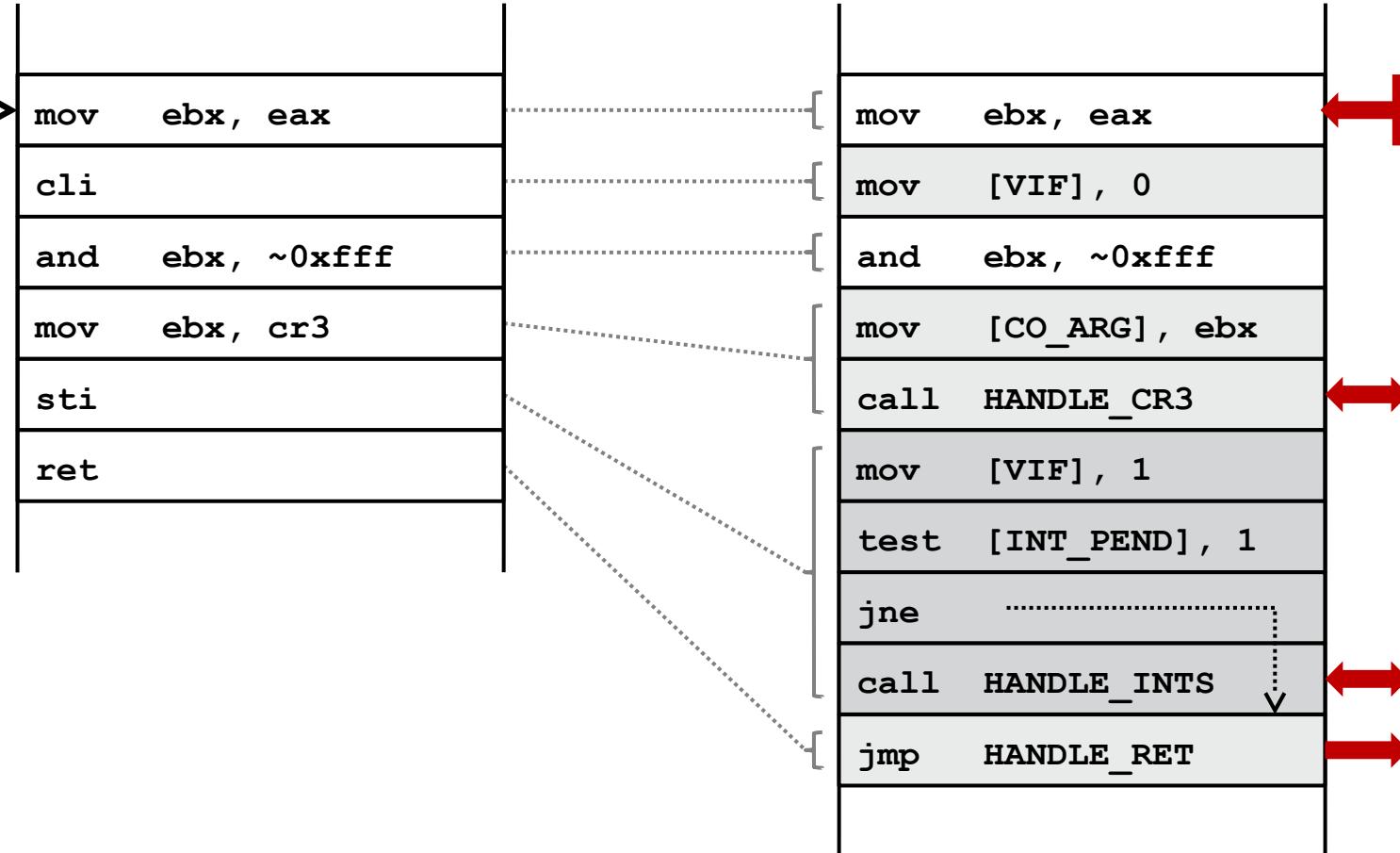
```
mov    ebx, eax  
cli  
and    ebx, ~0xffff  
mov    ebx, cr3  
sti  
ret
```

vEPC →

Translation Cache

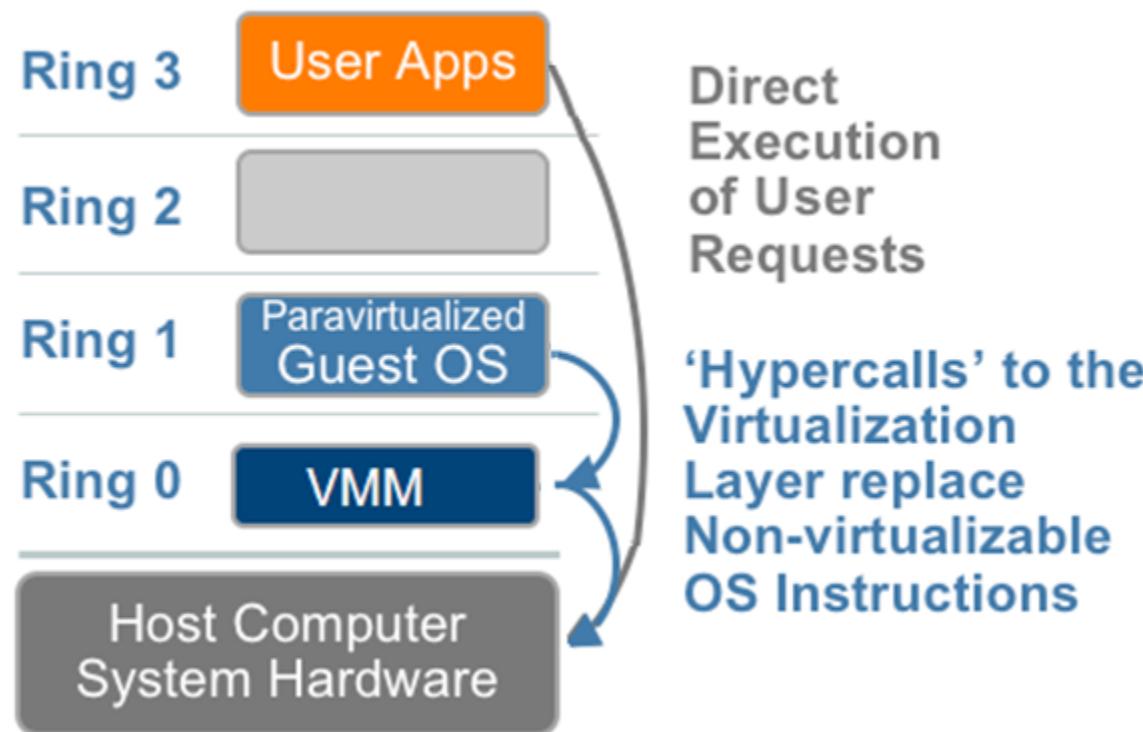
```
mov    ebx, eax  
mov    [VIF], 0  
and    ebx, ~0xffff  
mov    [CO_ARG], ebx  
call   HANDLE_CR3  
mov    [VIF], 1  
test   [INT_PEND], 1  
jne  
call   HANDLE_INTS  
jmp   HANDLE_RET
```

start

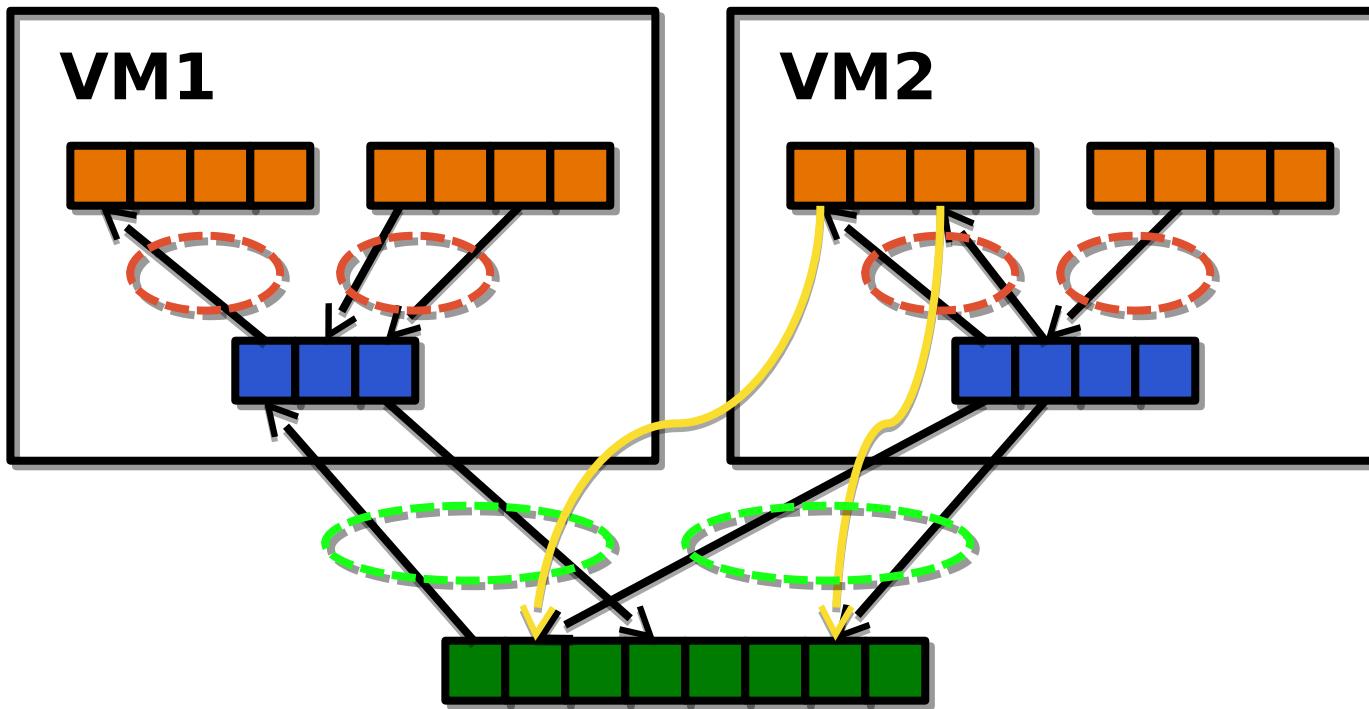


Forrás: Carl Waldspurger, Introduction to Virtual Machines

Paravirtualizáció



A virtualizáció megvalósítása: memória

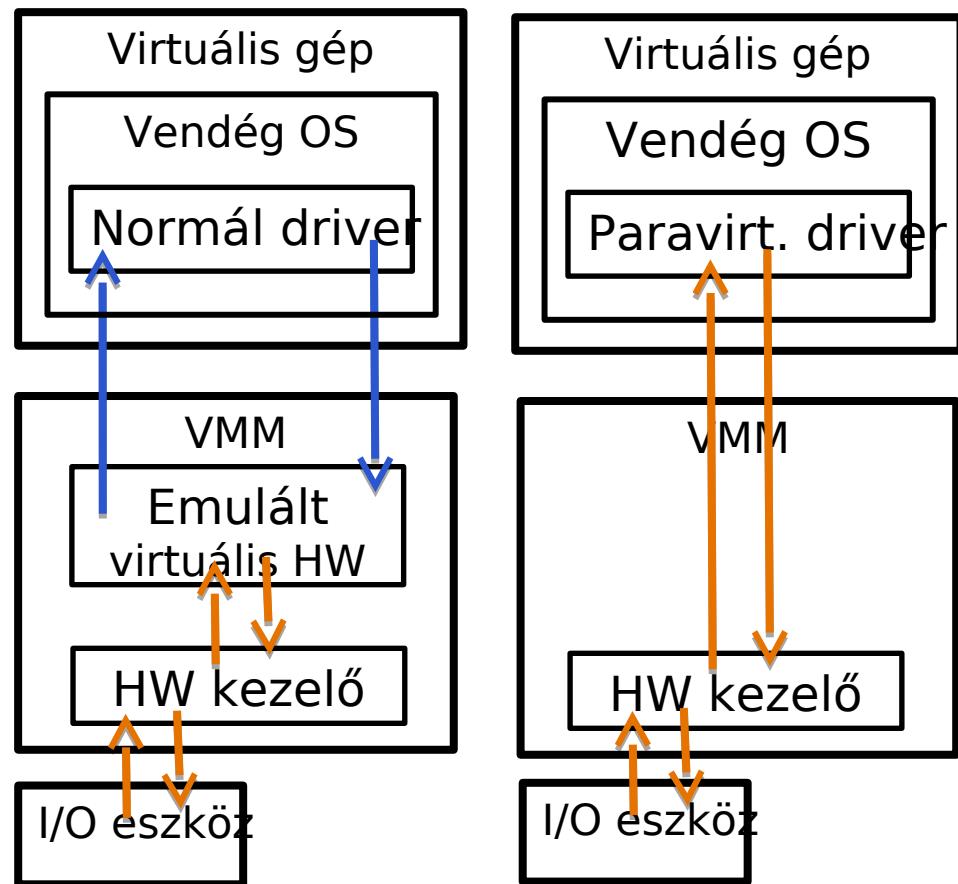


- Teljesítményérzékeny terület
- Kétszeres címfordítás
 - hardvertámogatás nélkül nagyon költséges
 - AMD Rapid Virtualization Indexing, Intel Extended Page Tables
 - beágyazott laptáblák és TLB címkézés

Az ábrát Micskei Zoltán, BME MIT készítette.

A virtualizáció megvalósítása: I/O

- Szoftveres emuláció
 - a teljes kommunikációt emulálja
 - egyszerű hardvert emulált
 - korlátozott képességek
 - transzparens, de **nem hatékony**
- Paravirtualizáció
 - a vendég a virtualizációs rendszer által felkínált eszközt használja
 - bizonyos hívások, adatmozgatások egyszerűsödnek a hardver felé
 - **speciális eszközmeghajtót kell telepíteni** a vendég OS-ben
 - nem annyira transzparens, de hatékonyabb az emulációnál
- hardveres virtualizáció
 - I/O eszközök megosztása
 - Intel VT-d, AMD IOMMU, PCI IOV



Termékek és szolgáltatások

Üzleti megoldások és piaci szereplők

- Saját kézben telepíthető rendszerek szállítói

VMware

XEN

Oracle Virtualbox

Microsoft Hyper-V, Virtual PC

Linux KVM

IBM PowerVM

Redhat EV

...

- Szolgáltatók ([felhő...](#))

Amazon EC2

Rackspace

Google Cloud Platform

Microsoft Azure

IBM Cloud

DigitalOcean

Felhőalapú szolgáltatások

- **IaaS:** infrastructure-as-a-service

- teljes hardvert nyújt
- operációs rendszert telepíthetünk
- sokféle sablonnal
- pl.: Amazon EC2, RackSpace, Microsoft Azure, Linode, DigitalOcean

- **PaaS:** platform-as-a-service

- futtatókörnyezetet nyújt
- saját alkalmazásainkat futtathatjuk
- pl.: Amazon AWS, Microsoft Azure, Google AppEngine, Heroku

- **SaaS:** software-as-a-service

- szoftverszolgáltatást nyújt
- előre telepített alkalmazás (pl. adatbázis, dokumentumkezelő, email)
- pl.: Microsoft Office365, Google Docs és Gmail

A virtualizáció kockázatai

- Támadások a virtualizációs rendszer ellen
 - a virtualizációs infrastruktúra lecserélésre (hyperjacking)
 - nagyon veszélyes, jelenleg inkább elvi lehetőség
 - támadás a virtualizációs mechanizmusok ellen
 - egy-egy mechanizmus (pl. hálózat, migráció) megfigyelése, megváltoztatása
 - a felügyelt rendszerek közötti adat- és kódszivárgás (VM jumping)
 - a szeparáció kijátszása megfigyelési vagy befolyásolási céllal
 - pl. vendég kitörése (guest breakout), a gyakorlatban is működik
- Auditálási nehézségek
 - az egyre bonyolultabb rendszer és annak nagyobb dinamizmusa miatt
- Bonyolultabb menedzsment
 - sokféle virtualizált erőforrás, összetett virtualizációs sémák
 - a rendszer telepítése és üzemeltetése esetenként nagyon összetett feladat
- Szakemberhiány
 - új és változó technológiák

Esettanulmányok: RHEV / oVirt (demo)

