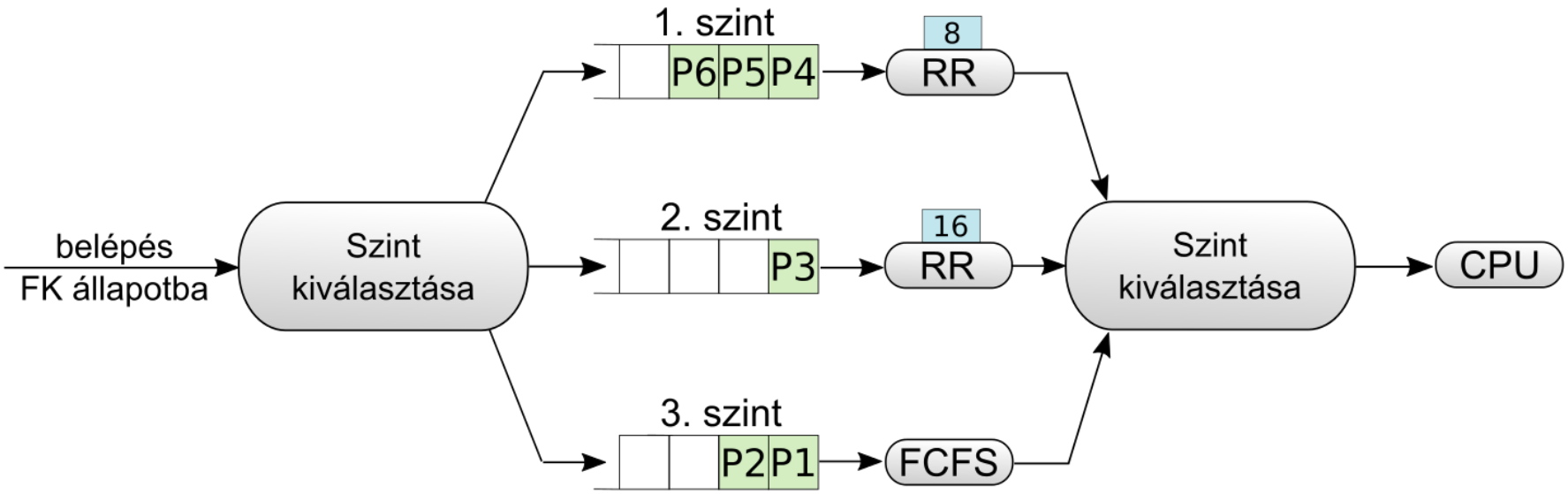
A többszintű ütemező alapműködése



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

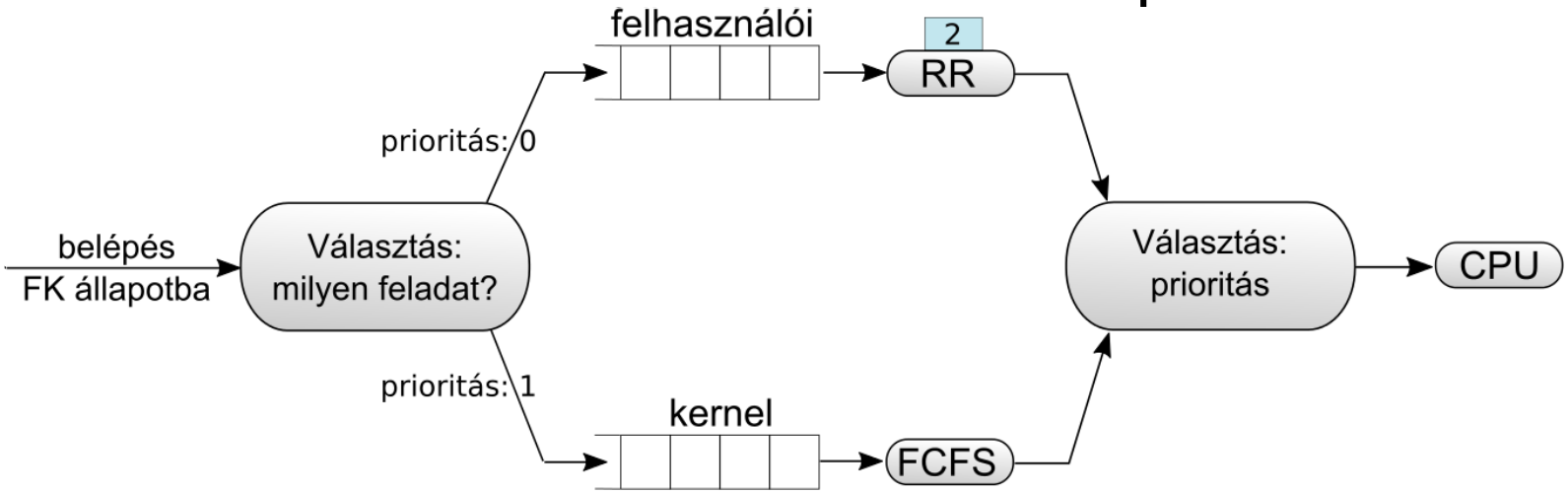
- Hogyan kerülhető el a kiéheztetés prioritásos szintütemező esetén?
  - megengedjük a szintek közötti váltást
    a „felfele” és „lefele” léptetés számára kell egy algoritmus (mikor?)

Statikus többszintű sorok (static multilevel queues)

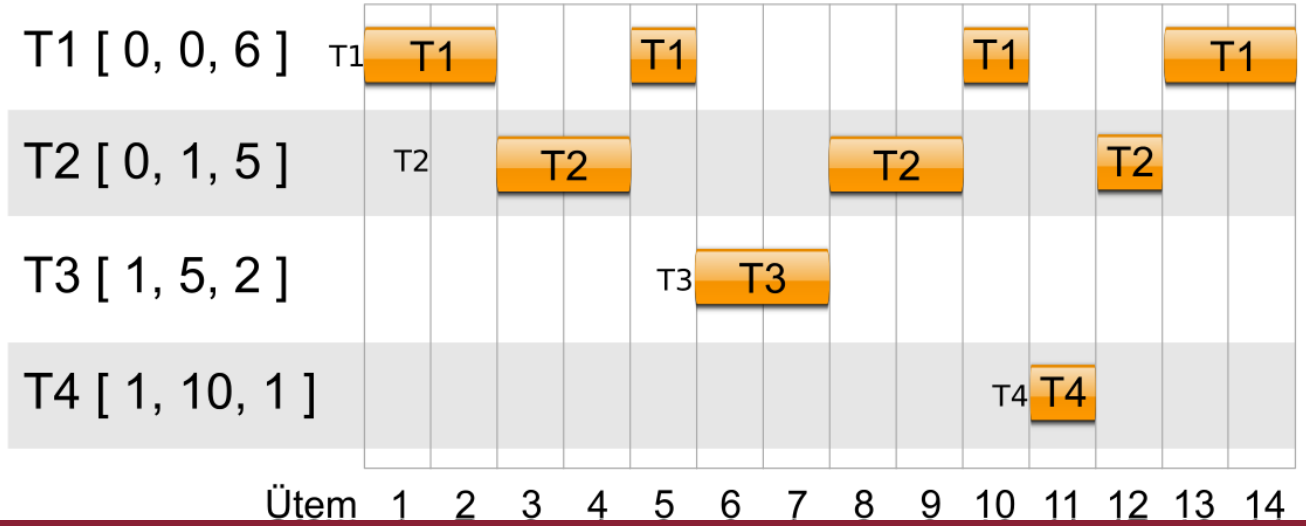
- A taszkokat statikus módon rendeljük a szintekhez (sorokhoz)
  - nem léphetnek át szintek között, pl. statikus prioritást kapnak
  - hozzákötődnek az adott ütemezési algoritmushoz
- A feladatok és az elvárások jellege szerinti alakítjuk ki a szinteket
  - valósidejű
  - rendszerfeladatok
  - interaktív
  - 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]



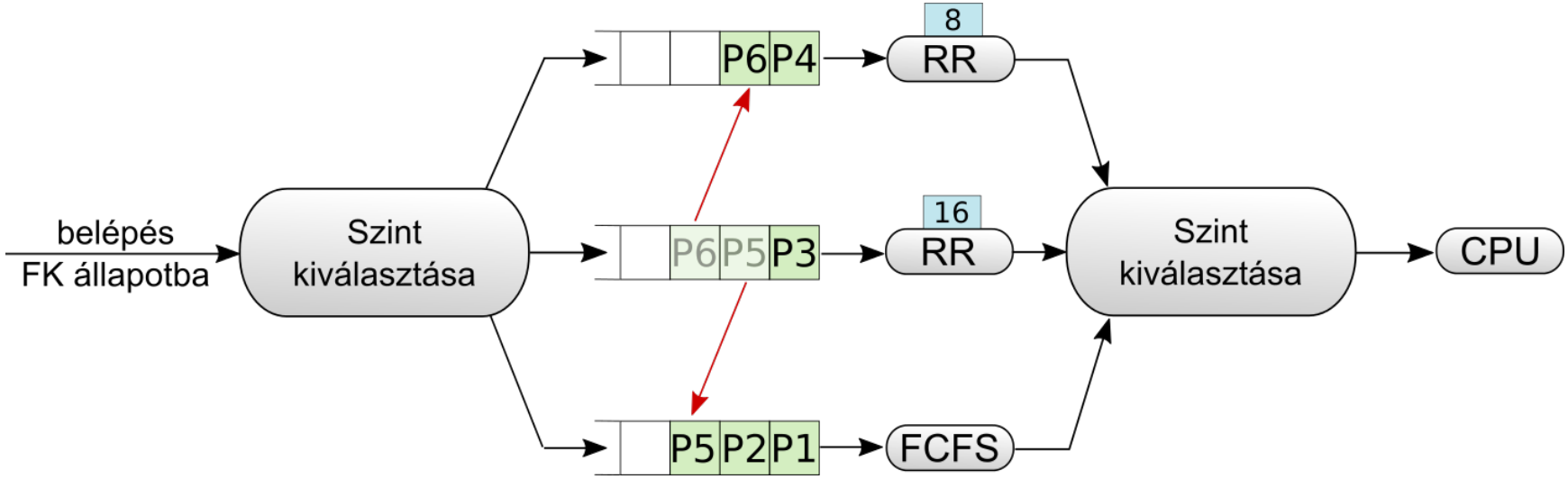
A statikus többszintű sorok értékelése

- 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



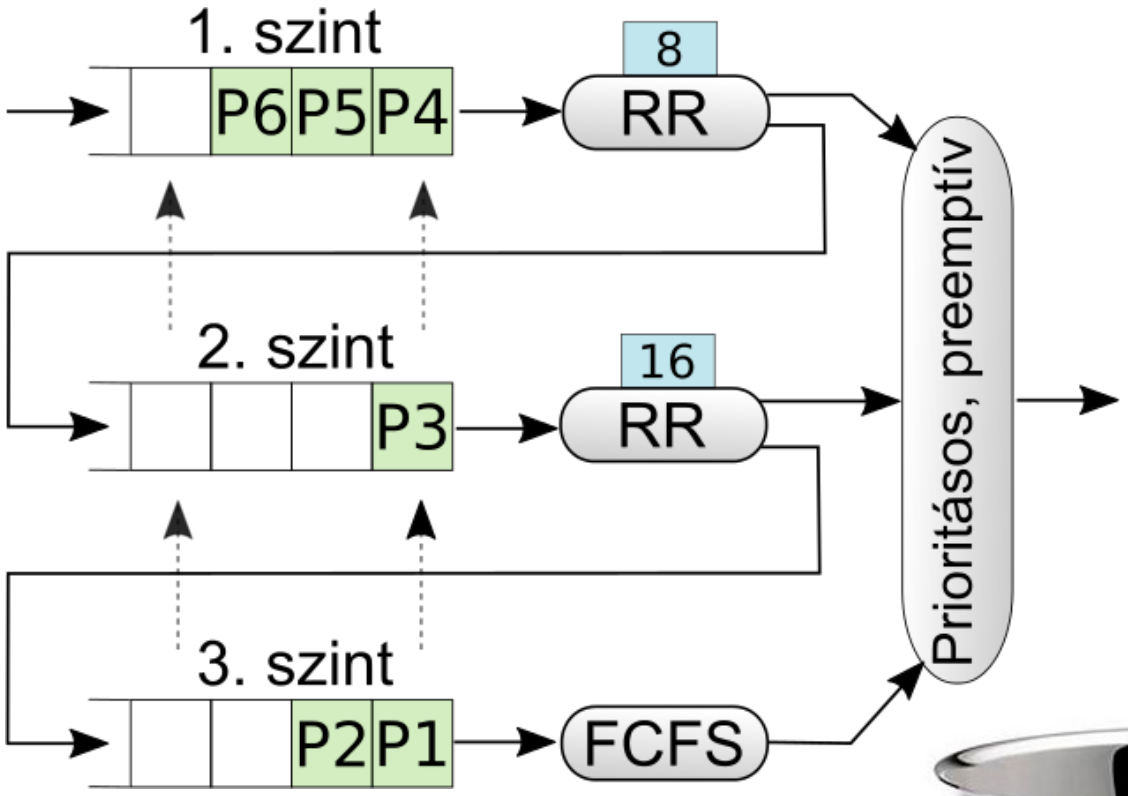
A dinamikus többszintű sorok értékelése

- 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)



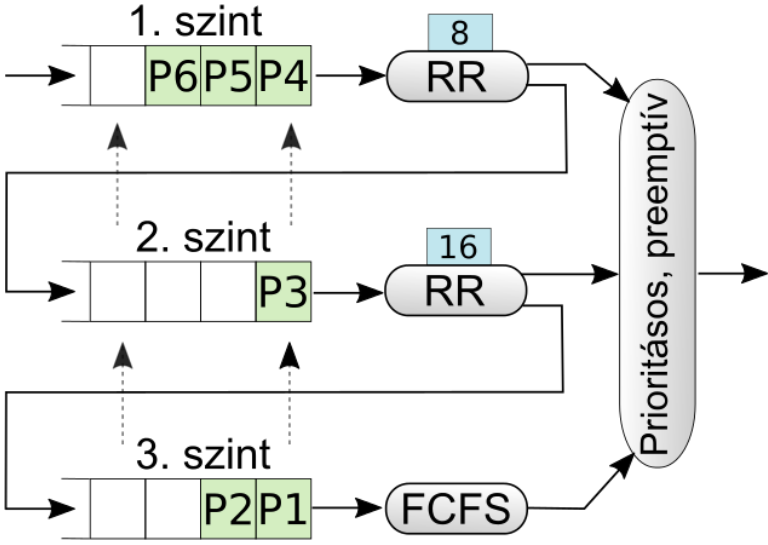
Taszkok szintlépései:

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

Megfigyeléseink 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 milyen 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 számítsuk 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 égig, „öregíteni” kell

  `p_cpu = p_cpu * KF` `KF` korrekciós faktor < 1

- A korrekciós faktor meghatározása

  `KF = 1/2` (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

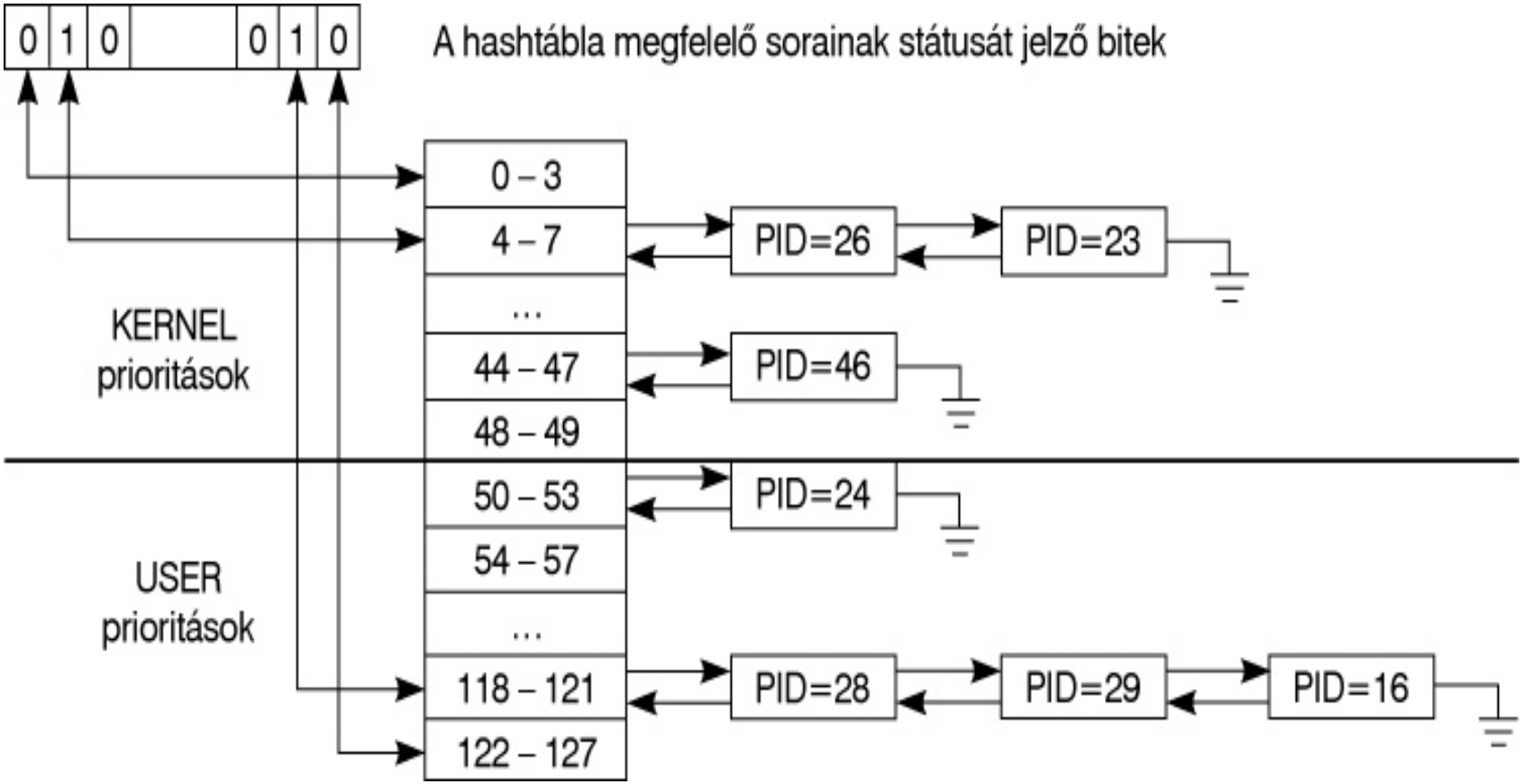
  A fentiek alapján mitől függjön a KF?

  ```
  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 ütemezőnk neve és értékelése

- Klasszikus Unix ütemező
  - többszintű, prioritásos, időosztásos
  - System V R3, BSD 4.3, korai Linux stb.

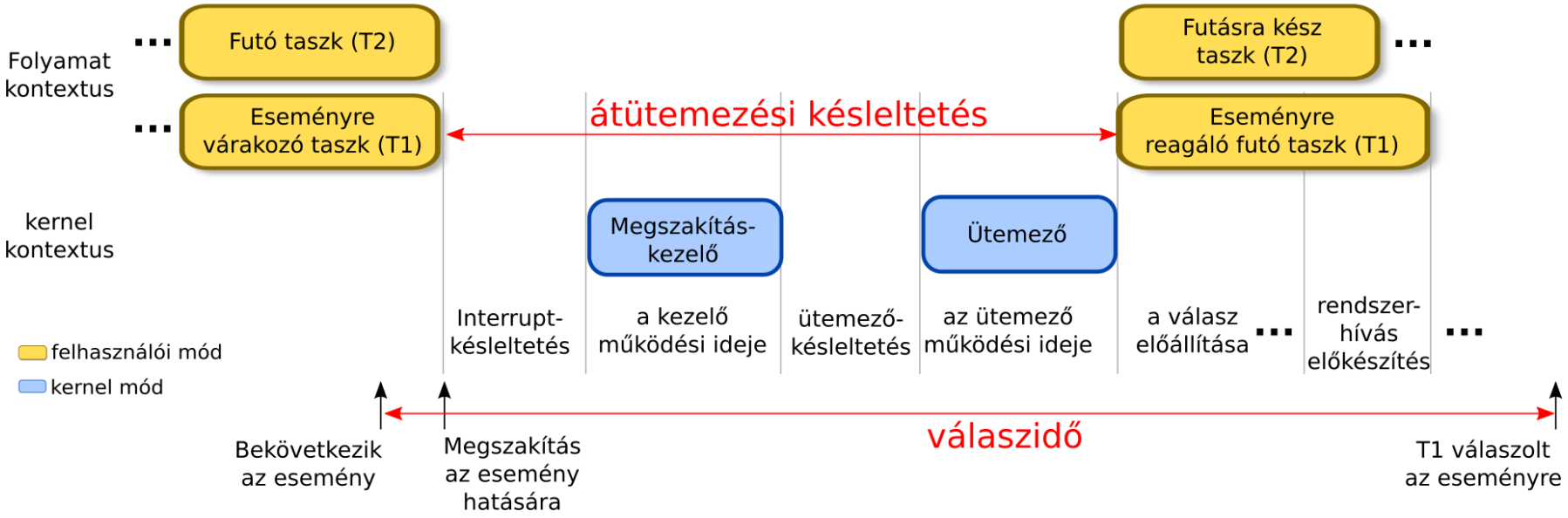
- Erősségei
  - kötegelt és interaktív taszkok keverékére jól működik
  - jó válaszidőt biztosít az interaktív taszkok számára, miközben
  - nem engedi a háttérben futó kötegelt munkák kiéheztetését

- Problémái
  - komplexitás?
    - pl. FK-sorok újrarendezése
  - késleltetés?
    - **valósidejű taszkok**
    - **prioritásinverzió** (priority inversion)
  - egyprocesszoros hardverre fejlesztették
    - a kernel belső konkurenciájával nem foglalkozik

Hogyan legyen kernel módban preemptív?

Hogyan működjön többprocesszoros környezetben?

A válaszidő és a késleltetések

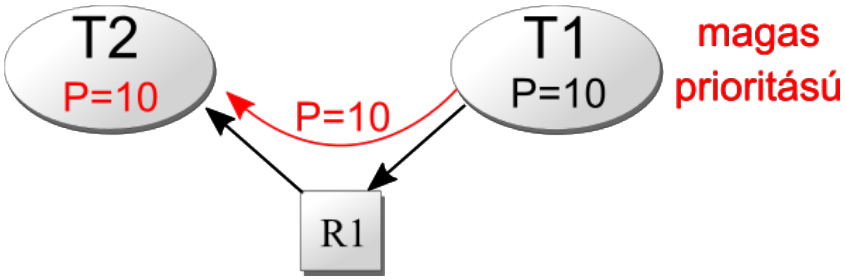


- Interrupt-késleltetés
  - pl. más megszakítások kiszolgálása
- Ütemező-késleltetés
  - más kernel tevékenységek zajlanak (pl. rendszerhívás)
- A válasz előállítása
  - további kernel-tevékenységeket igényelhet (rendszerhívás, memóriakezelés stb.)
- A válasz elküldése
  - rendszerhívás előkészítése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

Ezen lehetne javítani!

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

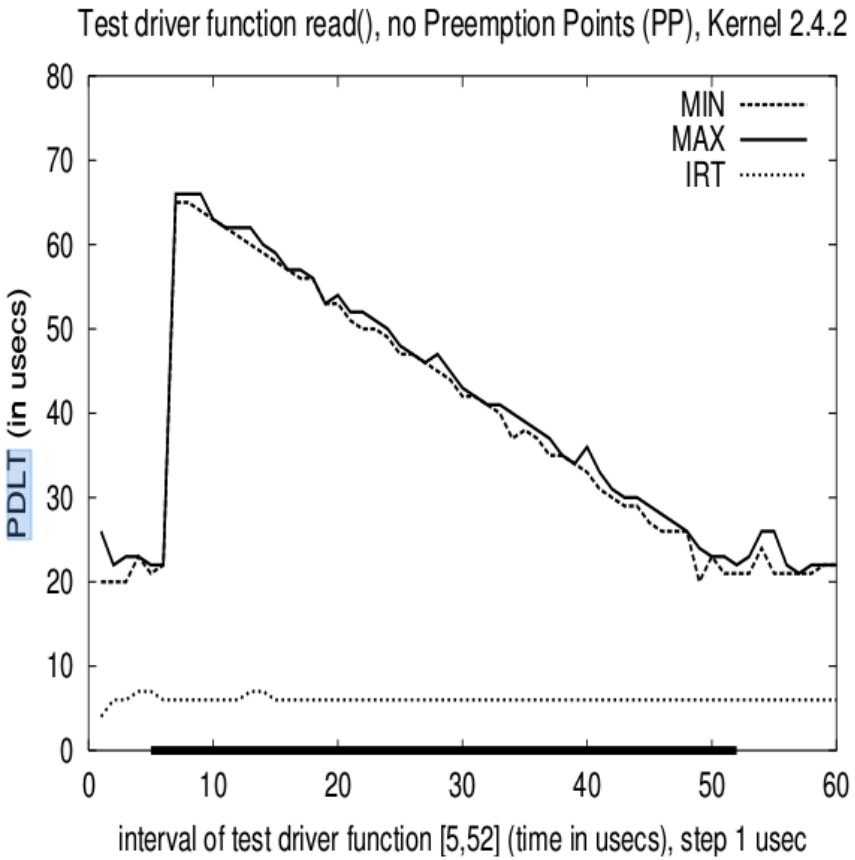
    Magas diszk I/O esetén a taszkok 150-800ms késleltetést is elszenvedhetnek.

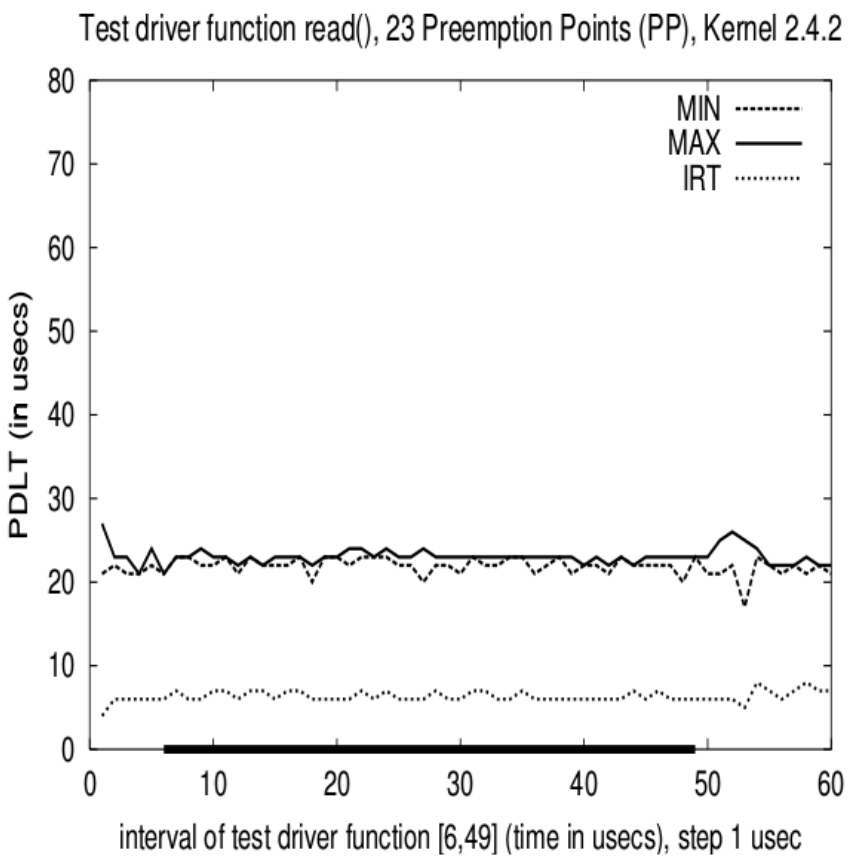
```
if (current->need_resched) {          // preeption point (PP)
   current->state = TASK_RUNNING;
   schedule();
}
```

- 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 opció
  - 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 projektté vált „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*)
  - előre ismert a határidő (*d*)

$$0 \le t \le d \le p$$

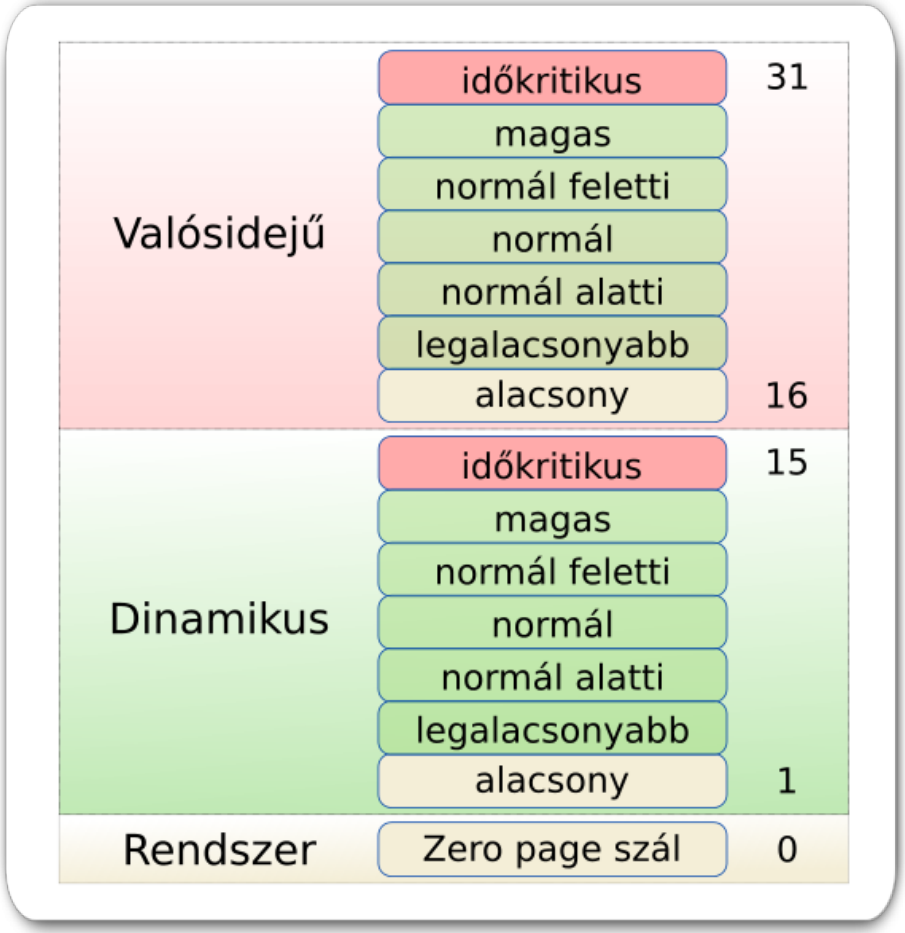
- „Legrövidebb periódusidejű előre” (Rate-Monotonic Scheduling, RMS)
  - statikus prioritás (~ 1/p), preemptív
  - feltételezi, hogy a CPU-löket állandó (= jósolható)
  - **belépési feltétel**: becsült *t*, ismert *p* és *d* alapján

- „Legkorábbi határidejű előre” (Earliest Deadline First, EDF)
  - dinamikus prioritás (~ 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 rendezi a taszkokat
  - a kisebb értékek balra, a nagyobbak jobbra
  - *O(log n)* komplexitás

- Cél: egyenletes `vruntime` minden taszkra
  - akinek a legkisebb, az fut
    `vruntime += idő_delta * (NICE_0_LOAD / curr->load.weight)`
  - ne ütemezzünk át túl gyakran:
    - target scheduling latency (TSL): maximális várakozás FK állapotban
      - pl. 2 egyforma taszk, 20ms TSL → 10 ms időszelet
    - minimum granularity (MG): minimális garantált futásidő
      - pl. 10db egyforma taszk, 20ms TSL, 5ms MG → max(2ms, 5ms) = 5ms 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>
```

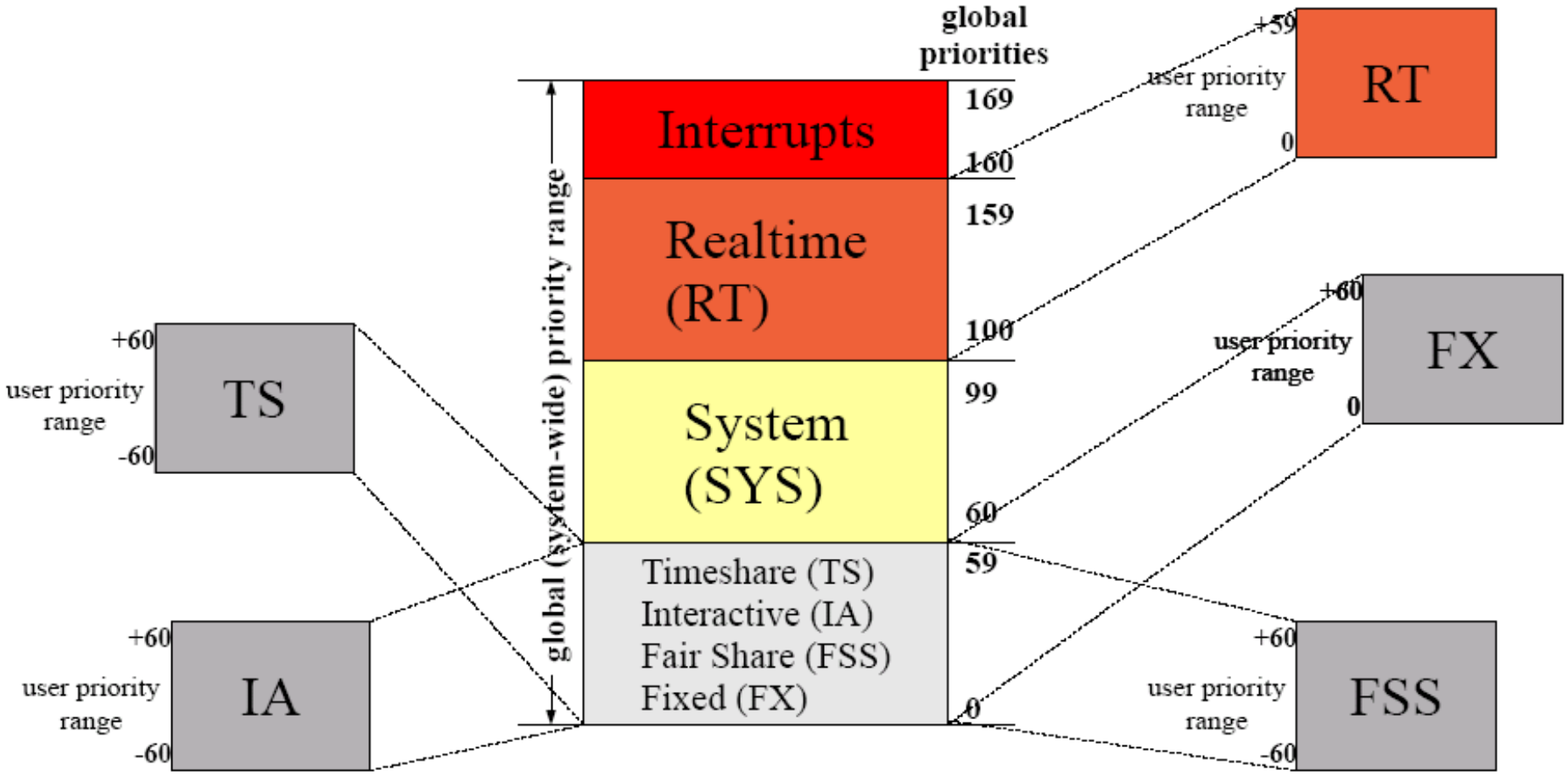
  - (A terhelés további növelése: `stress -q --cpu 2`)
  - Újra megfigyelni a CPU idő változását a két taszkra

- 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 átütemez
  - SJF: a legrövidebb löketidejű taszk fut legelőször (kooperatív)
  - SRTF: a legrövidebb hátralevő löketidejű taszk fut legelőször (preemptív SJF)
  - PRI: mindig a legnagyobb 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

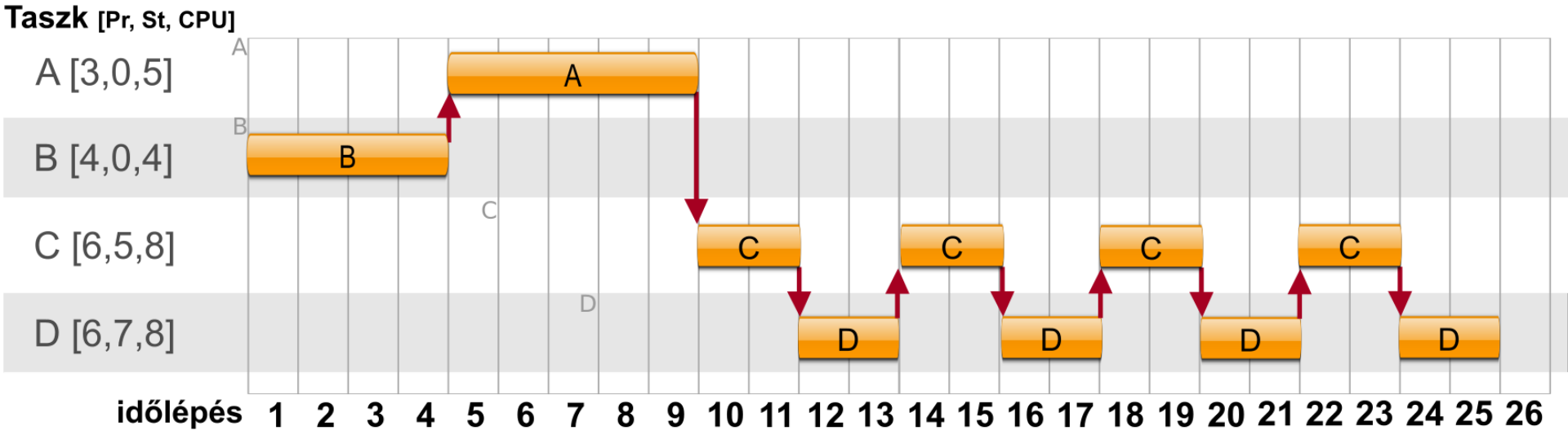
```
T1[0,0,6]  T2[0,0,5]  T3[0,2,6]  T4[0,2,2]  T5[0,4,8]
T6[1,1,3]  T7[1,3,4]  T8[1,4,2]  T9[1,5,7]
```

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