

# Operációs rendszerek

ELTE IK.

Dr. Illés Zoltán

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

# Miről beszéltünk korábban...

- ▶ Operációs rendszerek kialakulása
- ▶ Op. Rendszer fogalmak, struktúrák
- ▶ Fájlok, könyvtárak, fájlrendszerek
  - Fizikai felépítés
  - Logikai felépítés
- ▶ Folyamatok
  - Létrehozásuk, állapotuk
- ▶ Folyamatok kommunikációja
  - Kritikus szekciók, szemaforok.

# Mi következik ma...

- ▶ **Folyamatok kommunikációja**
  - Monitorok
  - üzenetküldés
- ▶ **Klasszikus IPC problémák**
  - Étkező filozófusok esete
- ▶ **Folyamatok ütemezése**
  - Elvek, megvalósítások
  - Szálütemezés

# Mi a baj a szemaforokkal?

- ▶ Semmi...viszont apró elírások nehezen felderíthető programhibákhoz vezetnek.
  - Pl. ha felcseréljük a 2 sort, akkor mikor a bolt tele van, a péket az üres szemafor blokkolja, a vásárlót emiatt pedig a szabad blokkolja, ezért mindkét folyamat blokkol, egymásra várnak! (holtpont)

```
void pék() /*eredeti recept*/  
{  
    int kenyér;  
    while (1)  
    {  
        kenyér=pék_süt();  
        down(&üres);  
        down(&szabad);  
        kenyér_polcra(kenyér);  
        up(&szabad);  
        up(&tele);  
    }  
}
```

```
void pék() /*cserélt recept*/  
{  
    int kenyér;  
    while (1)  
    {  
        kenyér=pék_süt();  
        down(&szabad); /*itt a csere*/  
        down(&üres);  
        kenyér_polcra(kenyér);  
        up(&szabad);  
        up(&tele);  
    }  
}
```

# Van baj a szemaforokkal?

- ▶ Alapvetően nincs, de ahogy az előbbi csere során láttuk, kicsi tévesztés, nagy nehézséget tud okozni.
- ▶ Ugyanígy bárhol az up, down utasítások felcserélése hasonló eredményt ad.
- ▶ Valamelyik elhagyása, hiba...
- ▶ Lehet valami jobb?
  - Kernel (gépi kód) szinten nem igazán.

# Monitorok

- ▶ Brinch Hansen (1973), Charles Anthony Richard Hoare (1974) magasabb szintű nyelvű konstrukciót javasoltak.
- ▶ Ezt nevezték el monitornak.
  - Kicsit a mai osztálydefinícióra hasonlít.

```
Monitor veszélyes_zóna
  Integer polc[];
  Condition c;
  Procedure pék(x);
  ...
End;
Procedure vásárló(x);
  ...
End;
End monitor;
```



# Monitorok tulajdonságai

- ▶ Monitorban eljárások, adatszerkezetek lehetnek.
- ▶ Egy időben csak egy folyamat lehet aktív a monitoron belül.
- ▶ Ezt a fordítóprogram automatikusan biztosítja.
  - Ha egy folyamat meghív egy monitor eljárást, akkor először ellenőrzi, hogy másik folyamat aktív-e?
    - Ha igen, felfüggesztésre kerül.
    - Ha nem beléphet, végrehajthatja a kívánt monitor eljárást.

# Monitor megvalósítása

- ▶ Mutex segítségével
- ▶ A felhasználónak nincs konkrét ismerete róla, de nem is kell.
- ▶ Eredmény: sokkal biztonságosabb kölcsönös kizárás megvalósítás
- ▶ Apró gond: mi van ha egy folyamat nem tud továbbmenni a monitoron belül?
  - Pl: a pék nem tud sütni mert tele van a bolt?
- ▶ Megoldás: állapot változók (condition)
  - Rajtuk két művelet végezhető: wait, signal



# Gyártó–Fogyasztó probléma megvalósítása monitorral. I.

## ► N elem

```
monitor Pék–Vásárló
    condition tele, üres;
    int darab;
    kenyeret_polcra_helyez(kenyér elem)
    {
        if (darab==N) wait(tele);
        polcra(elem);
        darab++;
        if (darab==1) signal(üres);
    }
    kenyér kenyeret_levesz_a_polcról()
    {
        if (darab==0) wait(üres);
        kenyér elem=kenyér_polcról();
        darab--;
        if (darab==N-1) signal(tele);
        return elem;
    }
end monitor
```

# Pék–Vásárló folyamata.

```
pék()
{
    while(1)
    {
        kenyér új;
        új=kenyér_sütés();
        Pék–Vásárló.kenyeret_polcra_helyez(új);
    }
}

vásárló()
{
    while(1)
    {
        kenyér új_kenyér;
        új_kenyér=Pék–Vásárló.kenyeret_a_polcról();
        lakoma(új_kenyér);
    }
}
```

# Más megoldások

- ▶ Az előző az un. Pidgin Pascal megoldás vázlat volt.
- ▶ C-ben nincs monitor
  - C++-ban igen, wait, notify
- ▶ Java:
  - Synchronized metódusok
  - Nincs állapotváltozó, de van wait, notify
- ▶ C#
  - Monitor osztály
    - Enter, TryEnter, Exit, Wait, Pulse ( ez a notify megfelelője)
  - Lock nyelvi kulcsszó
  - Példa: VS2008 párhuzamos solution, monitor projekt.

# Mi a baj a monitorokkal?

- ▶ Hm,...semmi.
- ▶ Sokkal biztonságosabb mint a szemafor használat.
- ▶ Egy vagy több CPU, de csak egy közös memória használatnál jók!
- ▶ Ha a CPU-knak önálló saját memóriájuk van, akkor ez a megoldás nem az igazi...

# Üzenetküldés

- ▶ A folyamatok jellemzően két primitívet használnak:
  - Send(célfolyamat, üzenet)
  - Receive(forrás, üzenet)
    - Forrás tetszőleges is lehet!
- ▶ Rendszerhívások, nem nyelvi konstrukciók
- ▶ Ha küldő–fogadó nem azonos gépen van, szükséges un. nyugtázó üzenet.
  - Ha küldő nem kapja meg a nyugtát, ismét elküldi az üzenetet.
  - Ha a nyugta veszik el, a küldő újra küld.
  - Ismételt üzenetek megkülönböztetése, sorszám segítségével.



# Gyártó–fogyasztó probléma üzenetküldéssel I.

## ► A gyártó (pék) folyamata:

```
#define N 100          // a pékségben lévő helyek száma, a
kenyeres polc mérete
void pék()             // pék folyamata
{
    int kenyér;         // „kenyér” elem tárolási hely
    message m;          // üzenet tároló helye
    while(1) // folyamatosan sütünk
    {
        kenyér= kenyeret_sütünk();
        receive(vásárló,m);          // vásárlótól várunk egy
                                     // üres üzenetet m -ben
        m=üzenet_készítés(kenyér);
        send(vásárló,m); // elküldjük a kenyeret a vásárlónak
    }
}
```

# Gyártó–fogyasztó probléma üzenetküldéssel II.

## ► Fogyasztó–vásárló folyamata:

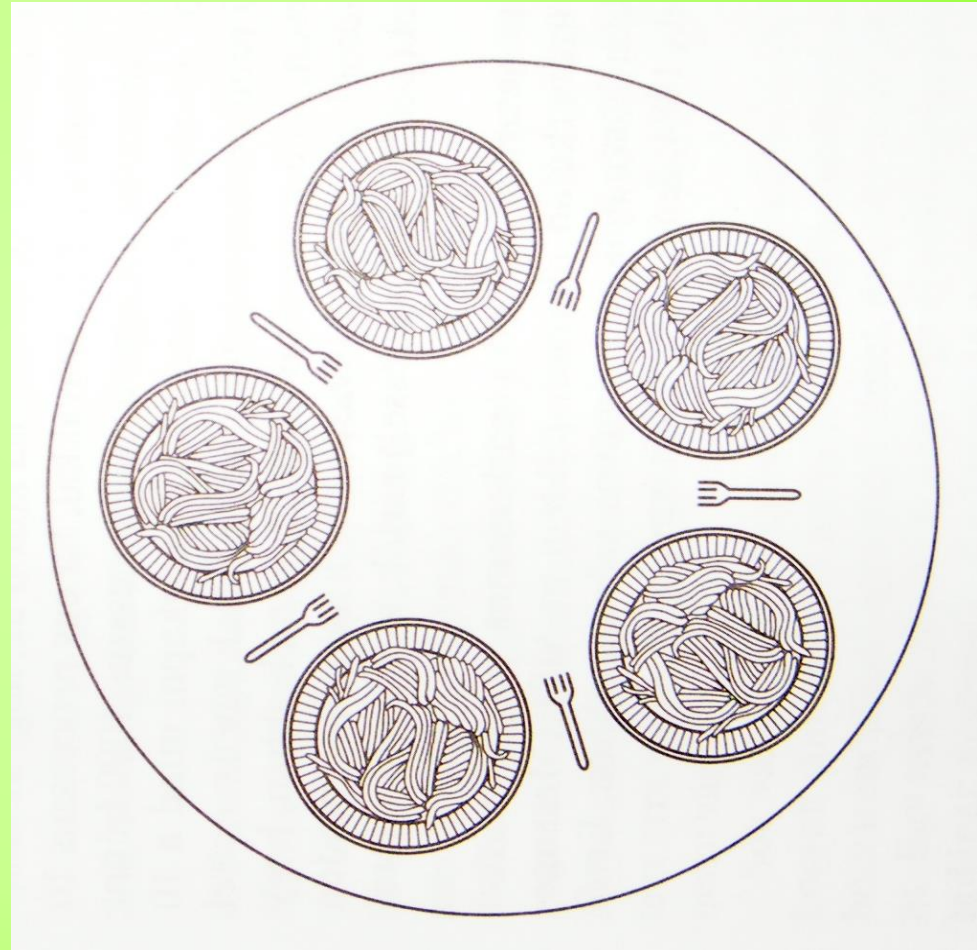
```
void vásárló()           // vásárló folyamata
{
    int kenyér;           // „kenyér” elem tárolási hely
    message m;           // üzenet tároló helye
    int l;
    for(i=0;i<N;i++) send(pék,m);    // N darab üres helyet
                                      // küldünk a péknek
    while(1) // a vásárlás is folyamatos
    {
        receive(pék,m); // várunk a péktől egy kenyeret
        kenyér=üzenet_kicsomagolás(m);
        send(pék,m);    // visszaküldjük az üres kosarat
        kenyér_elfogyasztás(kenyér);
    }
}
```

# Üzenetküldés összegzése

- ▶ Ideiglenes tároló helyek (levelesláda) létrehozása mindkét helyen.
- ▶ El lehet hagyni, ekkor ha send előtt van receive, a küldő blokkolódik, illetve fordítva.
  - Ezt hívják randevú stratégiának.
  - Minix 3 is randevút használ, rögzített méretű üzenetekkel.
  - Adatcső kommunikáció hasonló, csak az adatcsőben nincsenek üzenethatárok, ott csak bájtsorozat van.
- ▶ Üzenetküldés a párhuzamos rendszerek általános technikája. Pl. MPI

# Klasszikus IPC problémák I.

- ▶ Étkező filozófusok esete:
  - 2 villa kell a spagetti evéshez
  - A tányér melletti villákra pályáznak.
  - Esznek–gondolkoznak
  - Készítsünk programot, ami nem akad el!



# Megoldás I.

- ▶ A megoldásnak apró hibája, hogy pl. holtpont lehet, ha egyszerre megszerzik a bal villát és minden várnak a jobbra.
- ▶ Ha leteszi a bal villát és újra próbálkozik, még az se az igazi, hiszen folyamatosan felveszik a bal villát majd leteszik. (Éhezés)

```
Void filozófus(int i)
{
    while(1)
    {
        gondolkodom();
        kell_villa(i); // bal villa
        kell_villa((i+1)%N); //jobb
        eszem();
        nemkell_villa(i);
        nemkell_villa((i+1)%N);
    }
}
```



# Megoldás II.

```
Int s[5];           // eszik, éhes, gondolkodom értékei lehetnek
Szemafor safe_s=1; //jelző az s tömb használatához
Szemafor filo[5]={0,0,0,0,0}; //1 szemafor minden filozófushoz, 0=tilos
Void filozófus(int i)
{
    while(1) {
        gondolkodom();
        down(safe_s); //csak én módosítom s[]-t
        s[i]=éhes;    //
        if (s[bal]!=eszik && s[jobb]!=eszik) //vajon szabad a 2 szomszed villa?
            { s[i]=eszik; up(filo[i]); }; //i eszik, filo[i] szabad jelzést mutat
        up(safe_s);    // s[]-t más is elérheti
        down(filo[i]); // blokkol, ha nincs 2 villa, ha nem eszik az i. filozófus
        spaghetti_evés();
        down(safe_s); // evést befejeztem,újra védem s[]-t, mert módosítom
        s[i]=gondolkodom;
        if (s[bal]==éhes && s[bal2]!=eszik) { s[bal]=eszik;up(filo[bal]);}
        if (s[jobb]==éhes && s[jobb2]!=eszik) { s[jobb]=eszik;up(filo[jobb]);}
        up(safe_s);
    }
}
```

# Megoldás III.

- ▶ Legyen 5 villa szemaforunk az egyes villákra.
- ▶ Max szemafor
- ▶ Ez korlátozott erőforrás megszerzésre példa.

```
Int N=5;
Szemafor villa[]={1,1,1,1,1}; //mind
szabad
Szemafor max=4; //max 4 villa használt
//egyszerre

Void filozófus(int i)
{
    while(1)
    {
        gondolkodom();
        down(max);
        down(villa[i]); // bal villa
        down(villa[(i+1)%N]); //jobb
        eszem();
        up(villa[i]);
        up(villa[(i+1)%N]);
        up(max);
    }
}
```

# Olvasók–Írók probléma

- ▶ Adatbázist egyszerre többen olvashatják, de csak 1 folyamat írhatja:

```
// író folyamat
Szemafor database=1;
Szemafor mutex=1;
int rc=0;
Void író()
{
    while(1)
    {
        csinál_valamit();
        down(database); // kritikus
        írunk_adatbázisba();
        up(database);
    }
}
```

```
Void olvasó()
{
    while(1)
    {
        down(mutex);
        rc++;
        if (rc==1) down(database);
        up(mutex);
        olvas_adatbázisból();
        down(mutex); // kritikus
        rc--;
        if (rc==0) up(database);
        up(mutex);
        adatot_feldolgozunk();
    }
}
```

# Ütemezés

- ▶ Korábbiakban láttuk több folyamat képes „párhuzamosan” futni.
- ▶ Egyszerre csak 1 tud futni.
- ▶ Melyik fusson?
- ▶ Aki a döntést meghozza: Ütemező
- ▶ Ami alapján eldönti, hogy ki fusson: ütemezési algoritmus

# Folyamatok I/O igénye

- ▶ Egy folyamat jellemzően kétféle tevékenységet végez:
  - Számolgat magában
  - I/O igény, írni, olvasni akar adatot perifériára
- ▶ Számításigényes folyamat
  - Hosszan dolgozik, keveset várakozik I/O-ra
- ▶ I/O igényes folyamat
  - Rövideket dolgozik, sokszor várakozik I/O-ra



# Mikor váltunk folyamatot?

- ▶ Biztosan van váltás:
  - Ha befejeződik egy folyamat
  - Ha egy folyamat blokkolt állapotba kerül (I/O vagy szemafor miatt)
- ▶ Általában van váltás:
  - Új folyamat jön létre
  - I/O megszakítás bekövetkezés
    - I/O megszakítás után jellemzően, egy blokkolt folyamat, ami erre várt, folytathatja futását.
  - Időzítő megszakítás
    - Nem megszakítható ütemezés
    - Megszakítható ütemezés

# Ütemezések csoportosítása

- ▶ Minden rendszerre jellemzők:
  - Pártatlanság, mindenki hozzáférhet a CPU-hoz
  - Mindenre ugyanazok az elvek érvényesek
  - Mindenki „azonos” terhelést kapjon
- ▶ Kötegelt rendszerek
  - Áteresztőképesség, áthaladási idő, CPU kihasználtság
- ▶ Interaktív rendszerek
  - Válaszidő, megfelelés a felhasználói igényeknek
- ▶ Valós idejű rendszerek
  - Határidők betartása, adatvesztés, minőségromlás elkerülése

# Ütemezés kötegelte rendszerekben

## I.

- ▶ Sorrendi ütemezés, nem megszakítható
  - First Come First Served – (FCFS)
  - Egy folyamat addig fut, amíg nem végez vagy nem blokkolódik.
  - Ha blokkolódik, a sor végére kerül.
  - Pártatlan, egyszerű, láncolt listában tartjuk a folyamatokat.
  - Hátránya: I/O igényes folyamatok nagyon lassan végeznek.
- ▶ Legrövidebb feladat először, nem megszakítható ez se, (shortest job first–SJB)
  - Kell előre ismerni a futási időket
  - Akkor optimális, ha a kezdetben mindenki elérhető

# Ütemezés kötegetelt rendszerekben

## II.

- ▶ Legrövidebb maradék futási idejű következzen
  - Megszakítható, minden új belépéskor vizsgálat.
- ▶ Háromszintű ütemezés
  - Bebocsátó ütemező
    - A feladatokat válogatva engedi be a memóriába.
  - Lemez ütemező
    - Ha a bebocsátó sok folyamatot enged be és elfogy a memória, akkor lemezre kell írni valamennyit, meg vissza.
    - Ez ritkán fut.
  - CPU ütemező
    - A korábban említett algoritmusok közül választhatunk.

# Ütemezés interaktív rendszerben I.

- ▶ Körben járó ütemezés–Round Robin
  - Mindenkinek időszelet, aminek végén, vagy blokkolás esetén jön a következő folyamat
  - Időszelet végén a körkörös listában következő lesz az aktuális folyamat
  - Pártatlan, egyszerű
  - Egy listában tárolhatjuk a folyamatokat (jellemzőit), és ezen megyünk körbe–körbe.
  - Egy kérdés van: Mekkora legyen az időszelet?
    - Processz átkapcsolás időigényes
    - Kicsi az idő → sok CPU megy el a kapcsolgatásra
    - Túl nagy → interaktív felhasználóknak lassúnak tűnhet pl a billentyűkezelés



# Ütemezés interaktív rendszerben II.

## ► Prioritásos ütemezés

- Fontosság, prioritás bevezetése
  - Unix: 0–49 → nem megszakítható (kernel) prioritás
  - 50–127 → user prioritás
- Legmagasabb prioritású futhat
  - Dinamikus prioritás módosítás, különben éhenhalás
- Prioritási osztályok használata
  - Egy osztályon belül Round Robin
  - Ki kell igazítani a folyamatok prioritását, különben az alacsonyak nagyon ritkán jutnak CPU-hoz.
  - Tipikusan minden 100 időszeltnél a prioritásokat újraértékeli
    - Jellemzően a magas prioritások alacsonyabbra kerülnek, majd ezen a soron megy RR. A végén újra felállnak az eredeti osztályok.

# Ütemezés interaktív rendszerben III.

- ▶ Többszörös sorok
  - Szintén prioritásos és RR
  - Legmagasabb szinten minden folyamat 1 időszeletet kap
  - Következő 2-t, majd 4-et, 8, 16, 32, 64-et.
  - Ha elhasználta a legmagasabb szintű folyamat az idejét egy szinttel lejjebb kerül.
- ▶ Legrövidebb folyamat előbb
  - Bár nem tudjuk a hátralévő időt, de becsüljük meg az előzőekből!
  - Öregedés, súlyozott átlag az időszeletre.
    - $T_0, T_0/2 + T_1/2, T_0/4 + T_1/4 + T_2/2, T_0/8 + T_1/8 + T_2/4 + T_3/2$

# Ütemezés interaktív rendszerben IV.

- ▶ **Garantált ütemezés**
  - Minden aktív folyamat arányos CPU időt kap.
  - Nyilván kell tartani, hogy egy folyamat már mennyi időt kapott, ha valaki arányosan kevesebb időt kapott az kerül előbbre.
- ▶ **Sorsjáték ütemezés**
  - Mint az előző, csak a folyamatok között „sorsjegyeket” osztunk szét, az kapja a vezérlést akinél a kihúzott jegy van
  - Arányos CPU időt könnyű biztosítani, hasznos pl. video szervereknél
- ▶ **Arányos ütemezés**
  - Vegyük figyelembe a felhasználókat is! Mint a garantált, csak a felhasználókra vonatkoztatva.

# Ütemezés valós idejű rendszerben I.

- ▶ Mi az a valós idejű rendszer?
  - Az idő kulcsszereplő. Garantálni kell adott határidőre a tevékenység, válasz megadását.
  - Hard Real Time (szigorú), abszolút, nem módosítható határidők.
  - Soft Real Time (toleráns), léteznek a határidők, de ezek kis mértékű elmulasztása tolerálható.
  - A programokat több kisebb folyamatra bontják.
  - Külső esemény észlelésekor, adott határidőre válasz kell.
  - Ütemezhető: ha egységnyi időre eső  $n$  esemény CPU válaszidő összege  $\leq 1$ .
- ▶ Unix, Windows valós idejű?

# Ütemezési elvek, megvalósítás

- ▶ Gyakori a gyermek folyamatok jelenléte a rendszerben.
- ▶ A szülőnek nem biztos, hogy minden gyermekével azonos prioritásra van szüksége.
- ▶ Tipikusan a kernel prioritásos ütemezést használ (+RR)
  - Biztosít egy rendszerhívást, amivel a szülő a gyermek prioritását adhatja meg
  - Kernel ütemez – felhasználói folyamat szabja meg az elvet, prioritást. (nice)



# Szálütemezés

- ▶ Felhasználói szintű szálak
  - Kernel nem tud róluk, a folyamat kap időszeletet, ezen belül a szálütemező dönt ki fusson
  - Gyors váltás a szálak között
  - Alkalmazásfüggő szálütemezés lehetséges
- ▶ Kernel szintű szálak
  - Kernel ismeri a szálakat, kernel dönt melyik folyamat melyik szála következzen
  - Lassú váltás, két szál váltása között teljes környezetátkapcsolás kell
  - Ezt figyelembe is veszik.



# Folyamatok, prioritások

- ▶ A Unix, Linux prioritás alapú folyamat ütemezést végez!
- ▶ POSIX4 – IEEE1003.1b (1993), Valós idejű kiterjesztés megjelenés
- ▶ Két prioritás lista
  - nice -20–19, ahogy láttuk
  - Valós idejű prioritási lista, 0–99 közti értékekkel.(100 prioritási szint)
    - Ebben a listában a nagyobb szám jelenti a nagyobb prioritást
  - A lista közös értelmezése a következő:
    - 99,98...1,-20,-19,..0,1...,19, ezt gyakran 140-es prioritás intervallumnak neveznek, amiben különböző eltolások(mapping) lehetségesek.

# Windows prioritás osztályok

- ▶ Windows operációs rendszer 32 prioritás szintet használ.
  - 0...31-ig, a 0. szintet az ún. zero page thread használja csak!
    - Feladata: kitörölni a memórialap tartalmát!
  - 1–31-ig használt a standard folyamatok számára
  - 1–15 -ig használják a normál folyamatok
  - 16–31-ig valós idejű osztályhoz tartozó folyamatok
- ▶ A prioritási szintek prioritási osztályokba vannak csoportosítva!

# Linux ütemezés – $O(1)$

- ▶ Korábban láttuk, prioritásos, preemptív ütemezést használnak a mai interaktív rendszerek!
- ▶  $O(1)$  ütemezés (ordó 1 – konstans keresési idő, kb 2003-tól, 2.6 kerneltől)
  - Molnár Ingo (ELTE, fizikus, Red-Hat kernel fejlesztő)
  - Processz számtól független, konstans idő alatti ütemezés.
  - SMP támogatás – runqueue
  - I/O, CPU igény szerinti heurisztikus szétválasztás. (bonyolult, nem ad biztos eredményt)

# $O(1)$ ütemezés

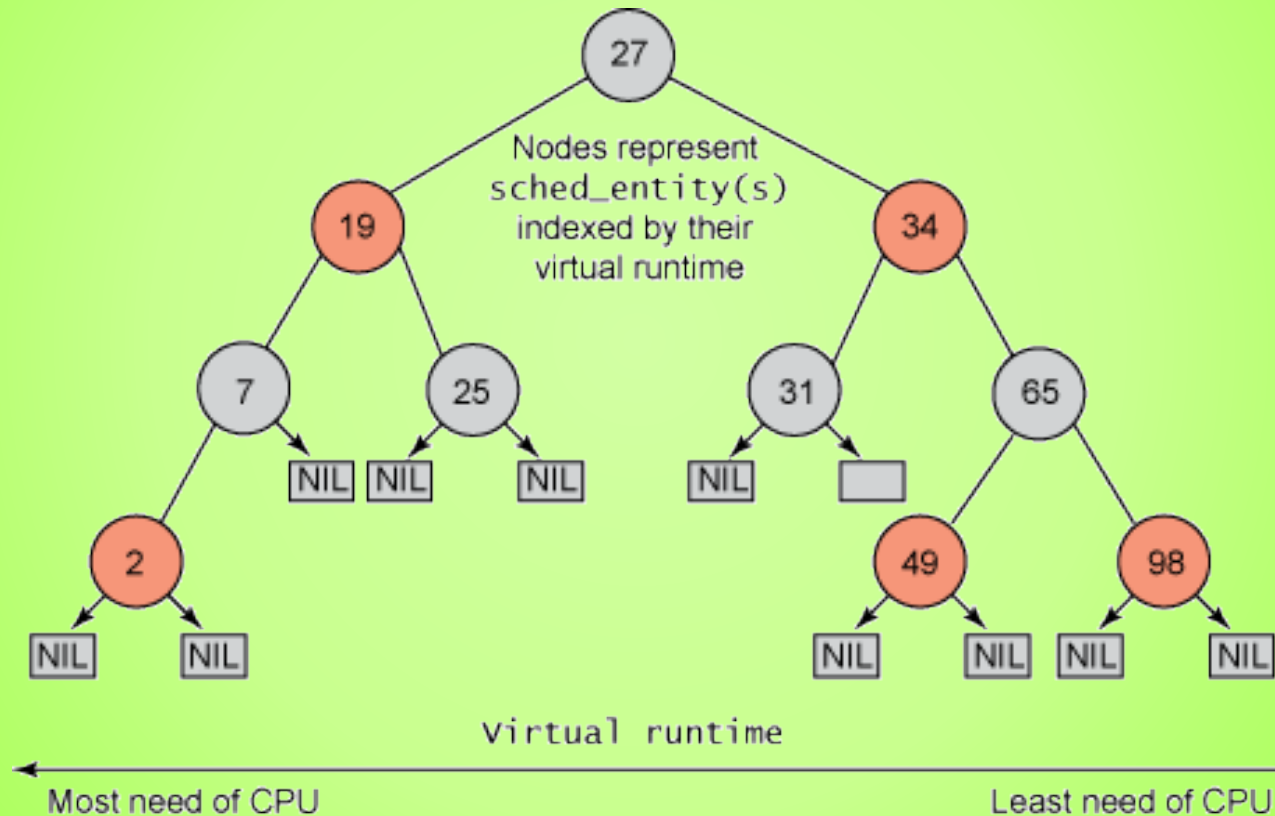
- ▶ Processzoronkénti futási sor (runqueue)
  - Minden futási sor 140 elemű láncolt lista, minden elem egy „dupla” tömbmutató!
  - Minden prioritási szinthez tartozó folyamatokból egy aktív és lejárt tömb mutatót tart nyilván.
  - Végigmegy az aktív tömbön, ha egy folyamat időszelete lejár, átkerül a lejárt tömbbe!
  - Ha kiürül az aktív tömb, akkor helyet cserélnek!
  - Ütemezés esetén

# CFS– Completely Fair Scheduler

- ▶  $O(1)$ – továbbfejlesztése (Molnár Ingo)
- ▶ 2.6.23 kerneltől kezdve
- ▶ Kon Colivas, Rotating Staircase Deadline Scheduler (RSDL) elemeket is használ.
- ▶ A CPU idő „fair” kiosztása, hasonlít a garantált ütemezésre!
- ▶ A CPU idők nyilvántartása egy fa struktúrában, balra kisebb, jobbra nagyobb idejű folyamatok (redblacktree)



# CFS – Sched-entity fa





# Prioritások CFS esetén

- ▶ Nincs direkt prioritás CFS-ben!
- ▶ Mindenki azonos, fair módon részesül a CPU erőforrásokból, de:
  - A nagyobb prioritású folyamat kisebb idő csökkenést szenved el.
  - Az alacsonyabb prioritás nagyobbbat!
- ▶ Így érvényesül a prioritási elv, nincs kiéheztetés!
- ▶ Nem kell prioritásonkénti folyamat nyilvántartás!

# CFS – moduláris ütemezés

- ▶ Alaposztály: sched\_class
  - Ebből származik: rt\_sched\_class, fair\_sched\_class(sched\_other), idle\_sched\_class, batch\_sched\_class

Diagram showing the hierarchy of scheduling classes:

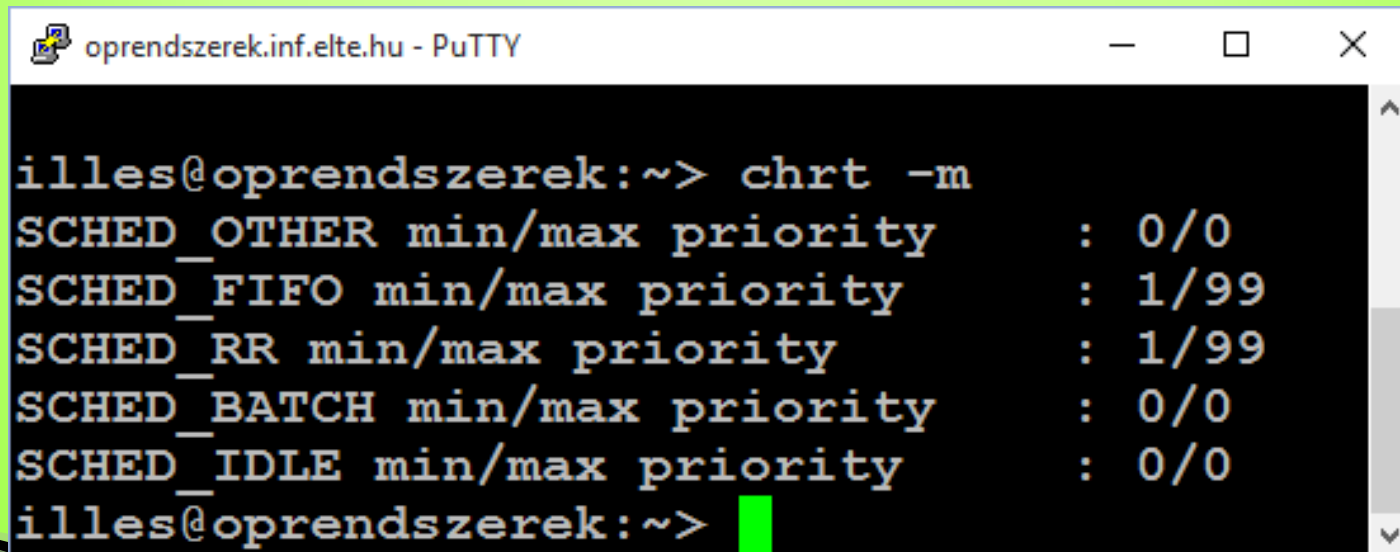
```

    sched_class_highest
    |
    +--> kernel/sched_rt.c
    |
    +--> kernel/sched_fair.c
    |
    +--> kernel/sched_idletask.c
    
```

sched_class	rt_sched_class	fair_sched_class	idle_sched_class
	next	next	NULL
enqueue_task	enqueue_task_rt	enqueue_task_fair	NULL
dequeue_task	dequeue_task_rt	dequeue_task_fair	dequeue_task_idle
yield_task	yield_task_rt	yield_task_fair	NULL
check_preempt_curr	check_preempt_curr_rt	check_preempt_wakeup	check_preempt_curr_idle
pick_next_task	pick_next_task_rt	pick_next_task_fair	pick_next_task_idle
put_prev_task	put_prev_task_rt	put_prev_task_fair	put_prev_task_idle
...	...	...	...
	SCHED_FIFO /SCHED_RR	SCHED_OTHER	

# Ütemezési jellemzők – chrt

- ▶ Chrt -m
- ▶ SCHED\_FIFO, SCHED\_RR (Round Robin), klasszikus ütemezések RT folyamatokra!
- ▶ SCHED\_OTHER– Alapértelmezett CFS!



The screenshot shows a terminal window titled "oprendszer.inf.elte.hu - PuTTY". The user "illes@oprendszer" has executed the command "chrt -m". The output lists the minimum and maximum priority values for various scheduling policies:

```
illes@oprendszer:~> chrt -m
SCHED_OTHER min/max priority      : 0/0
SCHED_FIFO min/max priority       : 1/99
SCHED_RR min/max priority         : 1/99
SCHED_BATCH min/max priority      : 0/0
SCHED_IDLE min/max priority       : 0/0
illes@oprendszer:~>
```

# Köszönöm a figyelmet!

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