



Operációs rendszerek nem hivatalos jegyzet

BME - VIK - mérnök informatikus képzés

< szerzői jogok helye, kis bevezető, ismertető helye >Készítette: Szabó Csaba

Tárgy előadói: Dr. Kovácsházy Tamás (általános rész), Mészáros Tamás (UNIX), Micskei Zoltán (Windows), Naszály Gábor (uC/OS)
Vendégelőadó: Fischer Erik (Solaris, dTrace)

Operációs rendszerek

Tartalomjegyzék

1. Bevezető.....	3
2. UNIX bevezető:.....	6
3. Windows bevezető	7
4. Windows hibakeresés.....	10
5. Scheduling (ütemezés, task kezelés multiprogramozott OS-en)	10
6. Scheduling (ütemezés)	15
7. Feladatok együttműködése	19
8. UNIX folyamatok kezelése	22
9. UNIX: folyamatok ütemezése.....	26
10. Ütemezés Windowsban.....	29
11. uC/OS-II.....	32
12. Feladatok együttműködése (kölsönös kizárás, szinkonizáció, kommunikáció)	33
13. Feladatok együttműködése (üzenet alapú kommunikáció)	40
14. Feladatok együttműködése (holtpont).....	43
15. Windows: Feladatok együttműködésének ellenőrzése	45
16. UNIX folyamatok kommunikációja.....	46
17. Memóriakezelés.....	50
18. Memóriakezelés a Windowsban.....	63
19. Virtualizáció	65
20. dTrace:	71
20. A permanens tár kezelése	72
21. UNIX fájlrendszerek alapismeretei	81
22. Felhasználó- és jogosultságkezelés	85
23. Biztonsági alrendszer a Windowsban.....	90

1. Bevezető

Operációs rendszer: nincs jó definíció, összetett fogalom

Korai operációs rendszerek:

1) Korai batch rendszerek:

- lyukkártyák, átlapolt feldolgozás, puffereelés
- Spooling: nagyobb kapacitású, gyors, véletlen hozzáférésű memóriák
 - több feladat, melyek egy időben futhatnak
 - elmozdulás a multiprogramozás felé
- *multiprogramozás*
 - nagyobb kapacitású, gyorsabb memóriák
 - a feladatok nem beérkezés sorrendje szerint dolgozhatók fel
 - optimalizáció lehetősége, megjelenik az ütemezés (*scheduling*)

2) 1960-as évek vége:

- C és modernebb nyelvek, ARPANET, miniszámítógépek
- *időosztásos rendszerek*
 - time sharing/multitasking
 - online felhasználók -> fontos a válaszidő
 - pl.: klasszikus UNIX erre vállalkozik

3) 1970-es évek:

- PC, IBM PC, x86 CPU architektúra, memória, HDD, grafikus képernyő, ...
- lépés az *elosztott rendszerek* felé
 - decentralizálás - funkciók térbeli elosztása
 - előny/hátrány: biztonság, skálázhatóság, megbízhatóság
- többprocesszoros rendszerek: homogén / heterogén processzorok

Operációs rendszer típusok:

- alkalmazás specifikus : kliens/server/mainframe, beágyazott, mobil OS
- tulajdonság specifikus: valós idejű, nagy megbízhatóságú, konfigurálható

4) Beágyazott rendszer: (PC-t használhatunk, de nem erre lettek kitalálva)

- jól meghatározott feladatra alkalmas
- külvilággal intenzív információs kapcsolat
- sokszor biztonságkritikus környezet (közlekedés, egészségügy, ...)

5) Valós idejű rendszerek:

- Adott eseményre a rendszer adott időn belül adott valséggel válaszol
- *lágyszál valós idejű*: $\text{valség} < 1$, nincs katasztrofális következménye a késésnek
- *kemény valós idejű*: $\text{valség} = 1$, rendszer specifikus, meghatározott időkorláton belül válaszolnak, vagy hibásnak tekinthetők
- valós idejű operációs rendszerek: ~~Windows, Linux~~ (sajnos az OS X sem)
- nem biztos hogy a felhasználónak látszólag azonnal válaszolnak, alkalmazástól függ
- a lágyszál valós idejű rendszerek nem csak prioritásos ütemezőt használnak, lehet statikus is
- a valós idejű OS nem garantálja, hogy a felhasználói feladatok valós időben futnak le, csak az egyes OS szolgáltatásokat garantálják

HW architektúrák:

- sokféle, x86 PC esetén is
- különbségeket elfedi az operációs rendszer, de figyelni kell
- pl.: Linux - ARM különbségek: HW közeli cserével oldhatóak meg

Védelmi szintek a CPU-ban: (CPU privilege levels)

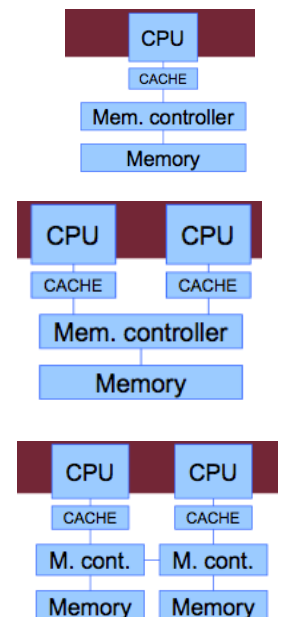
- hierarchikus, 2 vagy 4 szint, általában: *user/kernel mode*
- alacsony szintű CPU erőforrásokhoz történő hozzáférést szabályozza
- felhasználói módban futó feladat nem érheti el direkt módon a perifériákat

Memory Management Unit (MMU):

- speciális HW a CPU-ban
- memória állapotának nyilvántartása, memória védelem
- virtuális memória leképzése fizikai memóriára
- csak a folyamatok közötti memória hozzáférést kezeli, felhasználótól függetlenül

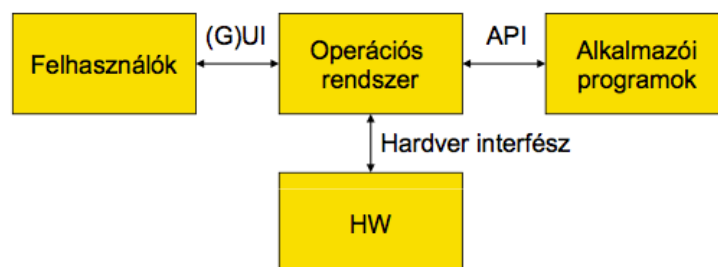
Számítógép architektúrák: (homogén többprocesszoros rendszerek esetében)

- single CPU
 - DMA¹, ha párhuzamosan kezeli a memóriát a CPU-val
 - versenyhelyzet a DMA vezérlő és a CPU között
 - CACHE koherencia sérülhet
- DMA-val történő helyes átvitelnek nem szükséges feltétele a CACHE koherens DMA támogatás, van rá megoldás
 - CACHE ürítése DMA-kor
 - CACHE-elés részleges vagy teljes tiltása
- SMP (Symmetric Multiprocessing)
 - több azonos CPU, külön cache, közös buszon a memória
 - OS támogatás kell hozzá (különben 1 CPU látható)
- NUMA (Non-Uniform Memory Access)
 - összefüggő fizikai memória, ccNUMA (cache koherenciával)
 - csak a memória vezérlők között van speciális kommunikáció



OS és környezete:

- a felhasználók, alkalmazói programok és a HW nincs direkt kapcsolatban



¹ Direct Memory Access

OS-ek átlalános belső felépítése:

- réteges szerkezet: rétegek határán egy virtuális gép
- kernel, HW közeli réteg, rendszerprogramok, rendszerhívások fogadása

OS-ek kialakítása:

- *monolitikus kernel* - összes funkció
- *moduláris kernel* - dinamikusan betölthető kernel modulok
- *mikrokernel* - minimális funkció, kliens/server arch.-ban csatlakozás, erőforrás igényesebbek, de védettek az eszközmeghajtók és a nem alapvető kernel szolgáltatások hibáitól

Linux: monolitikus kernel, modulárisan betölthető kernel modulokkal

OS X: mikrokernel + Free BSD UNIX

Hardver kezelés: I/O portok, mem. írás/olvasás, DMA, megszakítás (HW/SW)

Rendszerhívás: alkalmazói program - IT - kernel - alk. program, nagy overhead!

I/O műveletek: rendszerhívásokkal, kernel hajtja végre

OS elindulása (PC/server):

- bootstrap process
- Init/Reset vector
- BIOS/EFI
- boot sector
- 2. szintű boot loader
- OS betölt majd indul

2. UNIX bevezető:

Miért UNIX? : (>30 év), nyílt forráskód, minden rétegben megtalálható

Történet: AT&T Bell Lab, 1969, Ken Thompson és Dennis Ritchie

UNIX fejlesztési modellje:

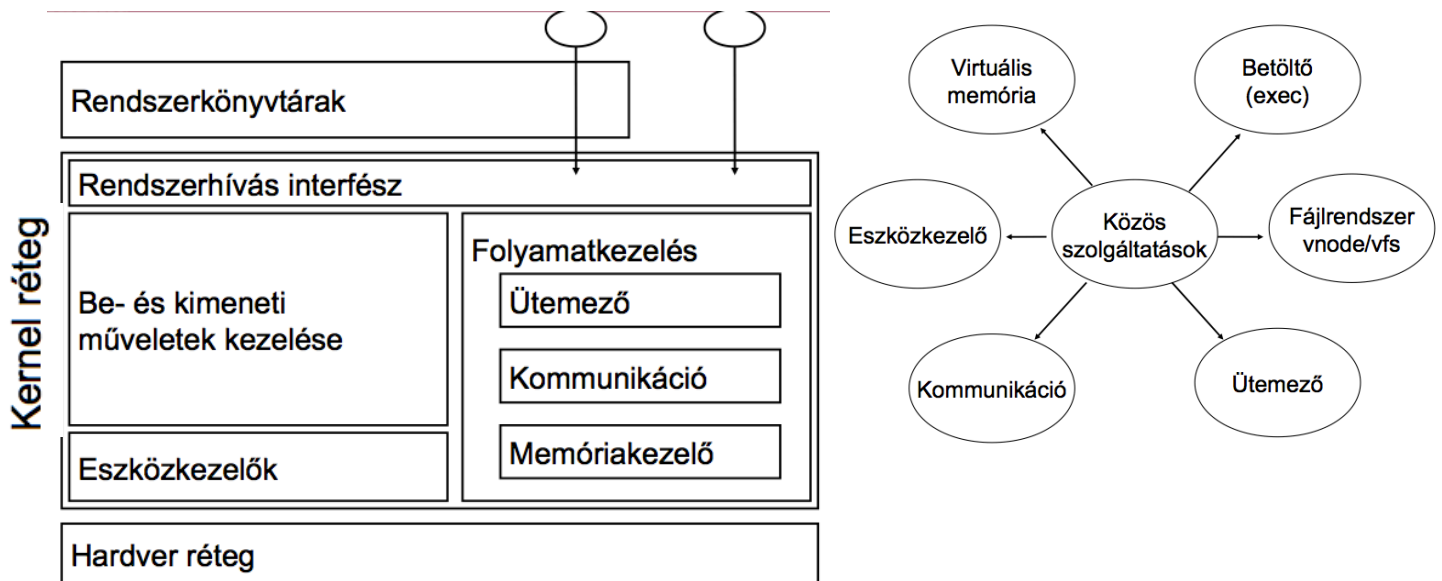
- iteratív, több fejlesztő, több HW platform
- forráskódok egy része nyílt, vannak zárt forráskódú változatok is
- előny: gyorsan fejlődik/terjed, hátrány: inkompatibilis, szakértelmet igényel

Családfa és szabványosítás

- System V: AT&T változat: Sun Solaris, SCO, ...
- BSD: Berkley változat: SunOS, OpenBSD, FreeBSD, OS X :)
- szabványosítás: IEEE POSIX, AT&T SVID, X/Open, Unix95, Unix98,...
- ma: kliens elenyésző, server platformon jelentős, beágyazott terület

Felépítés:

- *réteges, moduláris monolitikus kernel*



Rendszergazdai szemmel:

- karakteres (grafikus) felület, felhasználó azonosítás és hozzáférés-szab.
- naplózás, eszközkezelés, hálózati/vállalati szolgáltatások, virtualizáció, ...

Felhasználói szemmel:

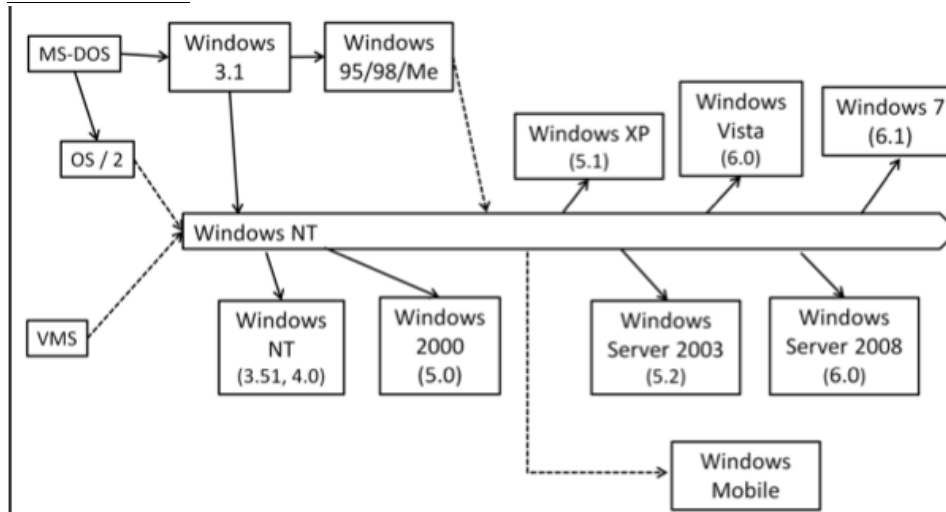
- grafikus felület, könyvtárrendszer, beépített parancsok, telepített alk.-ok

Jelentősebb UNIX disztribúciók:

- server: CentOS, OpenSolaris, IBM AIX, Suse Linux E. Server, openSUSE
- kliens: Ubuntu, Debian, Fedora, SUSE Linux E. Desktop, openSUSE

3. Windows bevezető

Történelem:



Windows felépítése:

- hordozhatóság a cél: többféle CPUra, kernel: C nyelven
- kiterjeszthetőség: moduláris, jól def. interfészek, unicode használata
- megbízhatóság: biztonsági szabványok
- teljesítmény: 32 bites, preemptív, többszálú, SMP, aszinkron I/O, újrahívható
- kompatibilitás: POSIX, DOS és 16 bites Win Api támogatás, OS/2

Újrahívható (reentrant): a rendszerhívásokat több alkalmazás is meghívhatja egyszerre, nem blokkolódnak, ha már valakit éppen kiszolgálja az adott rendszerhívást

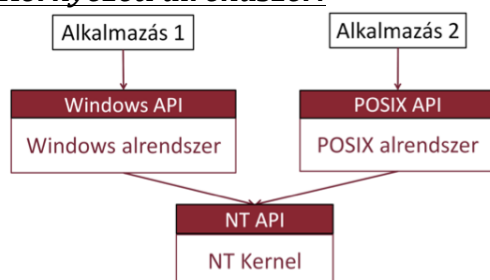
Preemptív: egy szálát fel lehet függeszteni futás közben.

A Windows kernel *teljesen preemptív*, a kernel szálak is felfüggeszthetőek.

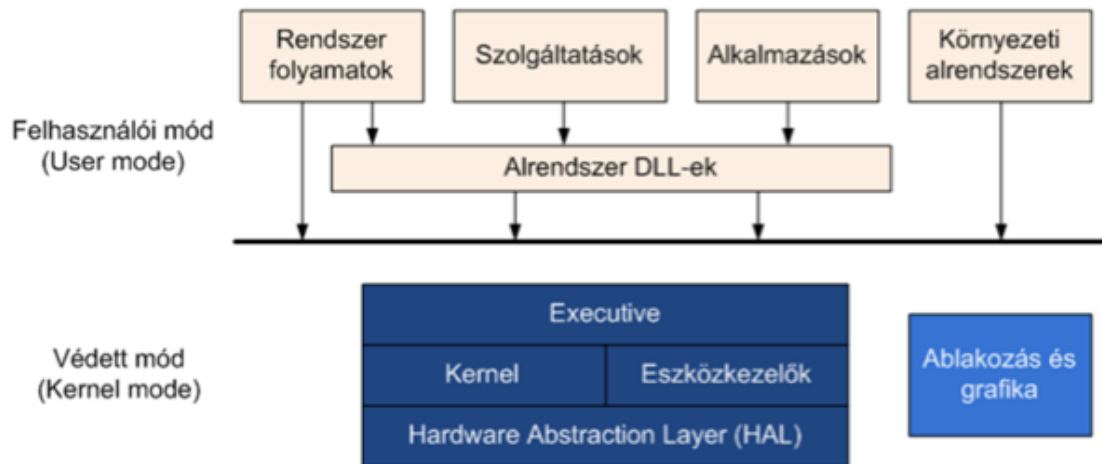
API (Application Programming Interface):

- kívülről milyen függvényeken, adat struktúrákon keresztül érhető el egy alkalmazás (pl: fork(), select(), fopen())
- a Windows API publikus, az NT API változhat a kiadások között

Környezeti alrendszer:



Egyszerűsített architektúra:



HAL: HW részletek elrejtése, egységes felület, hal.dll

Eszközkezelők (Device driver): kernel módú modulok, rétegzett struktúra, fájlrendszer, hálózatkészítés, *.sys

Kernel: Alap szolgáltatások, ütemezés, megszakítás, multiprocesszor szinkron, ntoskrnl.exe

Executive: OS szolgáltatások, memória, folyamatkezelés, OO szemlélet, Biztonság, I/O, ntoskrnl.exe

Processor Access Mode: CPU támogatás, védelem, kernelt felhasználói folyamatoktól, felhasználói folyamatokat egymástól. Más mint a környezetváltás ahol elmentjük az éppen futó szál adatait (regiszterek, program számláló, stb.), és betöltünk, majd futtatunk egy újat: itt nem változik, hogy melyik szálat hajtjuk végre.

Rendszerfolyamatok (System processes): alapvető rendszerfunkciók, induláskor ezek

Szolgáltatások (Services): háttérben futó folyamatok, hasonló mint a UNIX daemon

Környezeti alrendszerek: folyamatok felh. módú kezelése, állapottárolás, csrss.exe

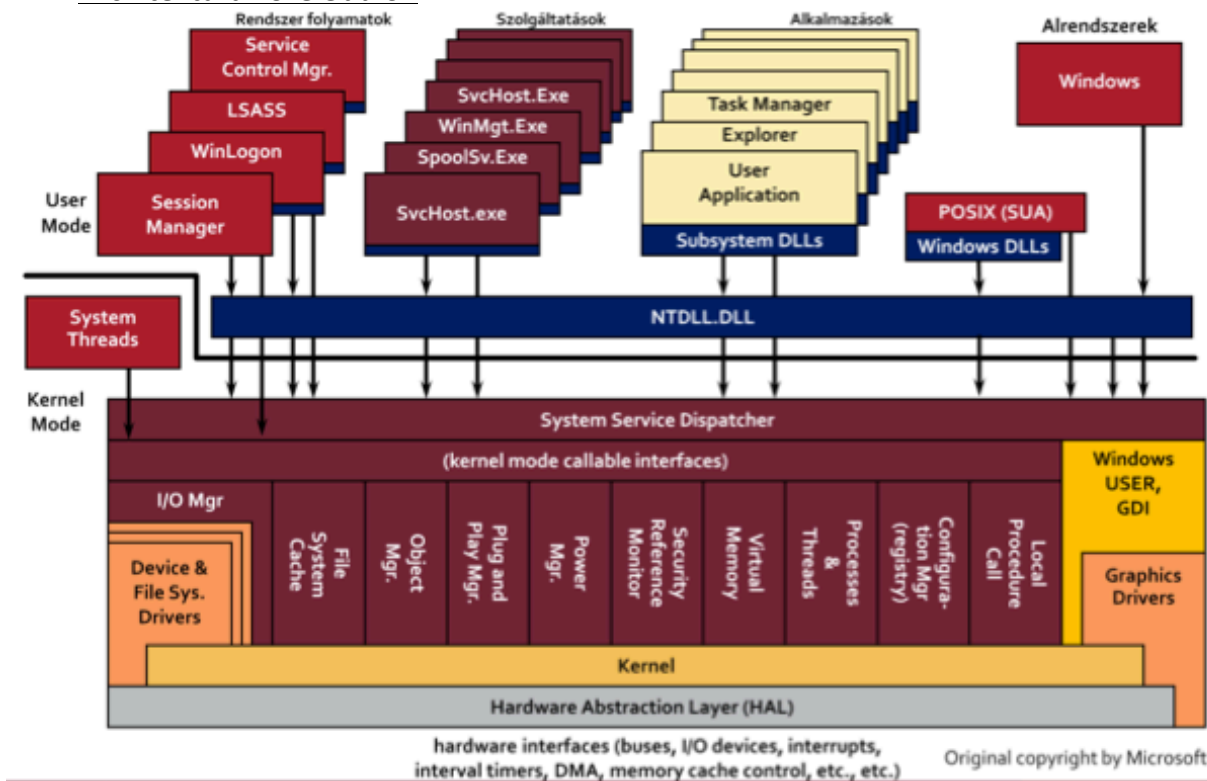
Alrendszer DLL-ek: alrendszer API hívásokat fordítják az Executive hívásaira, kernel32.dll

Ablakozás és grafika: GUI kernel módban fut, ablakkezelés, rajzolás

Munkamenet(session): több felh. bejelentkezve egy gépen, ID 0: rendszerfolyamat

Windows kernel: monolitikus

Architektúra közelebbről:



Wininit: rendszer folyamatok indítása

Winlogon: bejelentkezés, Ctrl+Alt+Del kezelés

LSASS: biztonság

Service Control Manager: szolgáltatások elindítása, leállítása

Szolgáltatások: felhasz. bejelentkezés nélkül is futnak

SvcHost: általános futtató folyamat

Alkalmazások: legtöbb program alrendszer DLL-eken keresztül használja az OS-t

NTDLL.DLL: executive függvény csokjai, belső függvények az alrendszereknek

System Service Dispatcher: rendszerrhívások elkapása, paraméter ellenőrzése, továbbítás a megfelelő komponensbe

System threads: kernel és meghajtók számai, feladatok amikhez várni kell a háttérben futnak, ezek is felfüggeszthetőek

Objektum Kezelés:

- a Windows objektumokkal dolgozik: executive, kernel
- adatkezelés, interfész használat

Windows verziók:

- ugyanaz a kernel forrás skálázódik

4. Windows hibakeresés

Process Monitor, eseménynapló, esemény részletei

Eventlog online help

Crash dump

Safe mode

5. Scheduling (ütemezés, task kezelés multiprogramozott OSen)

Történeti háttér:

- batch rendszerek, polling, multiprogramozás
- multiprogramozás:
 - 1 CPU - M feladat
 - feladattípusok: rendszer-, batch-, on-line feladat
 - feladat készlet (job pool)
 - a feladatok végrehajtásához erőforrások szükségesek
- multiprogramozás során az a cél, hogy a feladat készletet minél optimálisabban hajtsa végre a rendszer
- a feladatoknak alapesetben nem szabad tudniuk egymásról: külön "virtuális" gépen futnak, de osztozniuk kell a rendszer erőforrásain, szinkronizálódniuk kell, kommunikálniuk kell => OS feladata

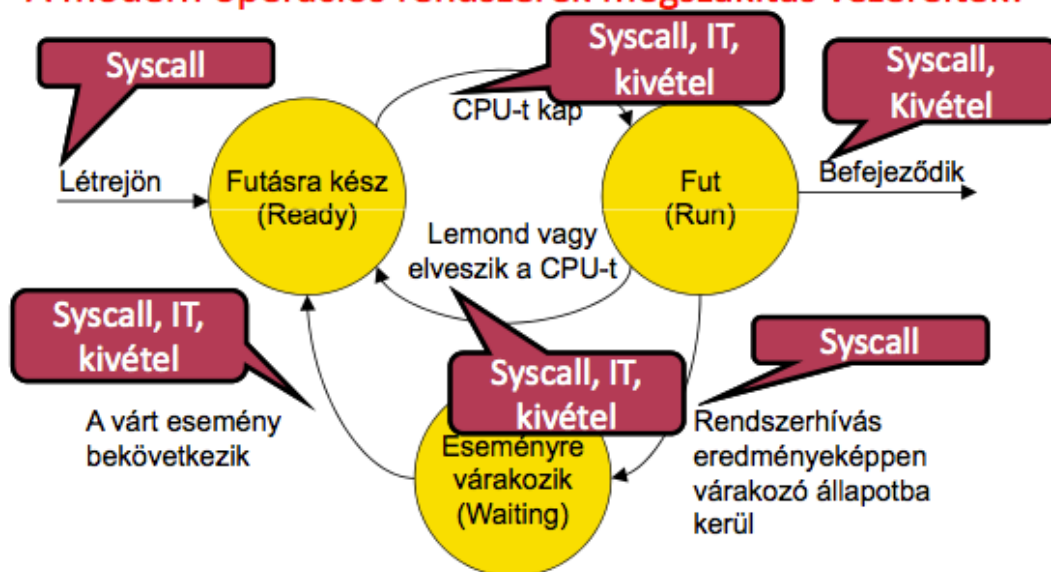
Alapfogalmak:

1. Esemény: rendszer életében lezajló változás, történés, SW IT, HW IT

2. Feladat(task): folyamat, szál, process, job

- az elnevezések keverednek
- multiprogramozott OS legfontosabb alapfogalma
- műveletek meghatározott sorrendben történő végrehajtása
- *a feladat több mint program:* állapota van, aktív, adatszerkezetekkel vannak feltöltve

- feladatok állapota:



- A modern operációs rendszerek megszakítás vezéreltek.
 - egy megszakítás lehet hardver vagy szoftver megszakítás, de van kivétel ami se nem hardver se nem szoftver

- feladat leíró (Task control block, TCB):

- adatstruktúra a feladattal kapcsolatos adatok OS-en belüli tárolására
- ez alapján végzi az OS a feladatok kezelését
- task ID, állpot, PC, CPU regiszterek, ...

- kontextus váltás (context switch):

- OS feladata hogy a feladat futásakor a lementett kontextust helyreálljon

3. Feladatok ütemezése (task scheduling):

- kiválasztani a futásra kész feladatok közül a futót
- a feladatok feladat sorokban (*task queue*) vannak: pl.: FIFO
- ütemezés időskálái:
 - rövid távú/CPU ütemezés: futó folyamat kiválasztása a futásra kész feladatok közül
 - hosszú távú ütemezés batch rendszerekben: sokkal több feladat mint amennyit hatékonyan párhuzamosan végre tudunk hajtani, általában percenként fut
 - középtávú ütemezés: swapping (memória kiírása háttértárra), majd visszatöltés

4. Preemptív ütemezés: A futó feladattól az OS elveheti a CPU-t

- a futó feladattól másik feladat nem veheti el a futás jogát, csak az OS
- nem preemptív (kooperatív) ütemezés is létezik, egy folyamat addig használja a CPU-t ameddig akarja
- bizonyos OS feladatok nem szakíthatóak meg

5. Mértékek: algoritmusok összehasonlítására

- CPU kihasználtság (%): $t_{CPU} = t_{CPU,munka} + t_{CPU,admin} + t_{CPU,idle}$ (adminisztráció és a henyélés is bele tartozik), CPU kihasz. = $\text{Sum}(t_{CPU,munka}) / \text{Sum}(t_{CPU}) * 100$ [%]
- átbocsátó képesség: [munka/s] v. [1/s], adott időegység alatt elvégzett feladatok száma, rendszerfeladatokat nem számoljuk
- várakozási idő: $t_{waiting} = t_{ready} + t_{other, non-running}$ [s]
- körbefordulási idő: egy feladat rendszerbe helyezésétől a teljesítésig eltelt idő
 $t_{CPU,végrehajtás} + t_{várakozás}$
- válaszidő: feladat megkezdésétől az első kimenet produkálásáig eltelt idő

Ütemezés követelmények:

- ütemező algoritmus valós idejű - alacsony overhead
- célfüggvény szerint legyen optimális
- matematikai modell, szimulációk vagy mérések alapján vizsgálódnak
- korrektség
- kiéhezés elkerülése
- jósolható viselkedés
- alacsony adminisztratív veszteségek
- maximális átbecsítő képesség, minimális várakozási idő
- erőforrás használat figyelembe vétele

Egyéb szempontok:

- prioritás
- fokozatos leromlás/összeomlás (graceful degradation)
 - a rendszer elérve a könyökkapacitást nem egyből omlik össze, hanem csak fokozatosan romlik le
- statikus/dinamikus ütemezési algoritmus
 - statikus: tervezési időben eldöntött hogy melyik feladat mikor fut legrosszabb esetre tervezés, biztonságkritikus rendszerekben lehet
 - dinamikus: futási időben dől el melyik feladat és mikor fut gyakorlatban ezt használják, dinamikus erőforrás kihasználás

CPU ütemezési algoritmusok:

- 1 CPU, 1 feladat fut egy időben
- a futásra kész feladatok valamilyen várakozási sorban tároljuk (task queue)
- a feladatok CPU vagy I/O löketekből (burst) állnak
 - CPU burst: fizikai memóriával dolgozik, illetve feladatokat hajt végre, aktívan dolgozik a CPU
 - I/O burst: különböző input/output műveletek, passzívan várkozik a CPU (hatékonyság növelhető ha más feladatot végez ilyenkor a CPU)
- FIFO,FCFS: legegyszerűbb
 - feladat leíróra mutató referenciákat tároló FIFO (First Input First Output) sor
 - put()-tal bekerül a folyamat a várakozó sorba
 - get()-tel levesszük a futásra kerülőt
 - nem preemptív, I/O-ra várhat, feltarthatja a nagyobb CPU löketű folyamat a kisebb löketű folyamatokat

- átlagos várakozási idő nagy lehet
- kis adminisztrációs overhead
- hosszú CPU löketű feladat feltartja a rövid CPU löketű feladatokat:
konvoj hatás
- Round-robin:
 - időosztásos rendszer számára, FIFO javítása
 - kedvezőbb az on-line felhasználók számára
 - Round-robin = FIFO + egyszeri óra megszakítás => preemptiv
 - megszakítás adott időszetenként: quantum/time slice (általában 10-20 ms)
 - futás előtt az órát elindítjuk, az egy adott idő lejártá után megszakítás
 - periodikus óra interrupt
 - időszület > átlagos CPU löket -> átmegy FIFO-ba
 - nem jár le az időszület, a feladatok CPU löketideje ütemez
 - időszület = átlagos CPU löket -> normál működés
 - a átlagosnál hosszabb CPU löketű feladatokat szakítanak csak meg
 - időszület < átlagos CPU löket -> sok adminisztrációs overhead
 - ökölszabály: a CPU löketek 80%-a kisebb az időszületnél
- Prioritásos ütemezők: ütemező család
 - prioritás = fontosság (0 a legkisebb vagy a legnagyobb? - változó)
 - külső/belső prioritás
 - külső: operátor vagy a feladat maga állítja be
 - belső: a rendszer adja
 - megállapítható: felhasználó által adott kiinduló prioritás, időkorlát, memória igények, használt erőforrások és azok száma, CPU és I/O löket, futási idő, tervezett futási idő
 - attól függően hogy milyen célunk van, ugyanazt a mértéket kezelhetünk úgy hogy mint egy negatív prioritást csökkentő tényező és úgy is kezelhetjük mint egy prioritást növelő tényező
 - statikus/dinamikus prioritás
 - statikus: tervezési időben dől el
 - dinamikus: OS futása során változtatja a feladatok prioritását
 - jellegzetesen preemptiv, de lehet nem preemptiv is
 - nem korrekt (fair), de nem is akarjuk hogy az legyen
 - nem korrekt, kiéheztesítés előfordulhat (feladatok öregítése segíthet)
 - feladat öregítése: ha régóta vár, akkor egyre fontosabb feladat lesz
 - egyszerű prioritásos ütemező - egy szinten egy feladat
 - gyakorlatban: egy prioritási szinten tetszőleges számú feladat
 - legrövidebb löketidejű (Shortest Job First - SJF)
 - nem preemptiv, a legrövidebb löketidejűt választja futásra
 - optimális az átlagos várakozási és körülfordulási idő szempontjából
 - becsülni kell a löketidőt: korábbi löketidő/felhasználó által megadott érték alapján
 - legrövidebb hátralévő idő (Shortest Remaining Time First - SRTF)
 - SJF preemptiv változata, amely egy feladat beérése esetén újraütemezi a futásra kész folyamatokat, és a legrövidebb maradék lökediszével rendelkezőt választja ki futásra
 - kontextus váltást is figyelembe kell venni (megéri-e a váltás)

- itt se tudjuk a löketidejeket, becsülni kell
- a legjobb válaszarány (Highest Response Ratio - HRR)
 - kiéhezést próbálja megoldani
 - SJF-ből indul ki, CPU löketet és a várakozási időt is figyelembe veszi

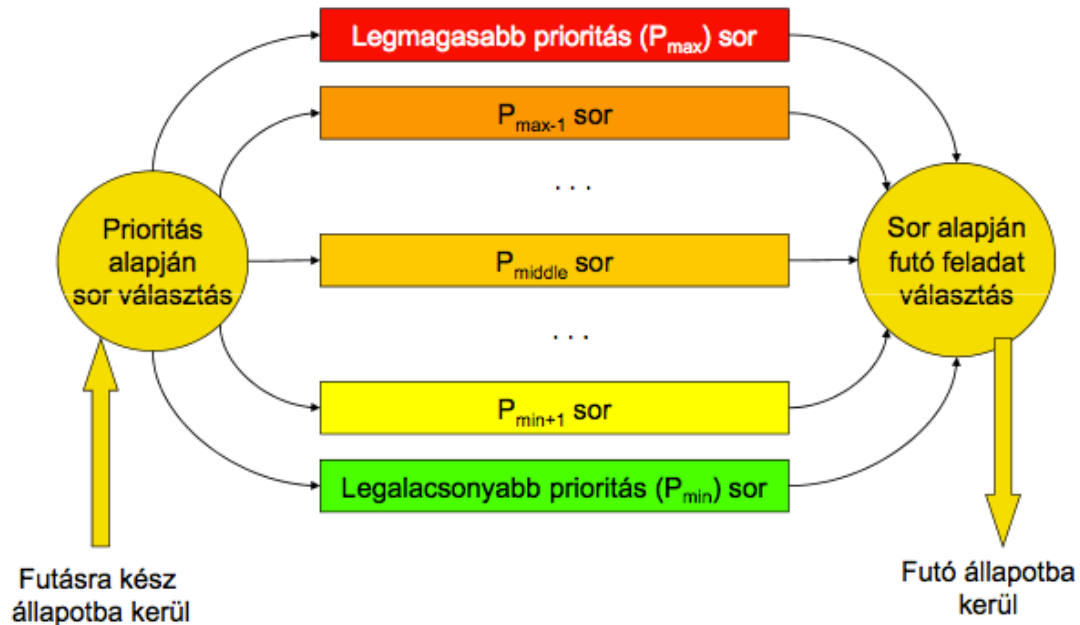
6. Scheduling (ütemezés)

Számolási feladatok ütemezéshez: (1.2)

<http://home.mit.bme.hu/~micskeiz/opre/files/opre-feladatok-segedanyag.pdf>

Többszintű sorok (multilevel queue):

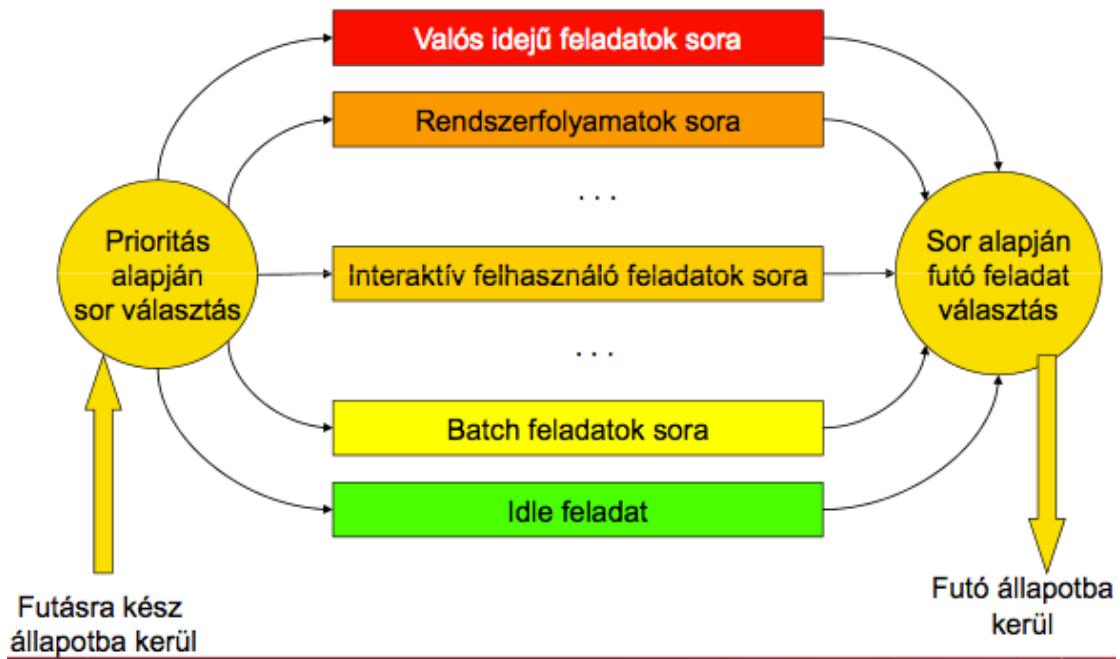
- komplexebb ütemezési algoritmusok
- minden prioritási szinthez task queue
- különböző szinteken különböző algoritmusok lehetnek
- általában 8-16-32 prioritási szint



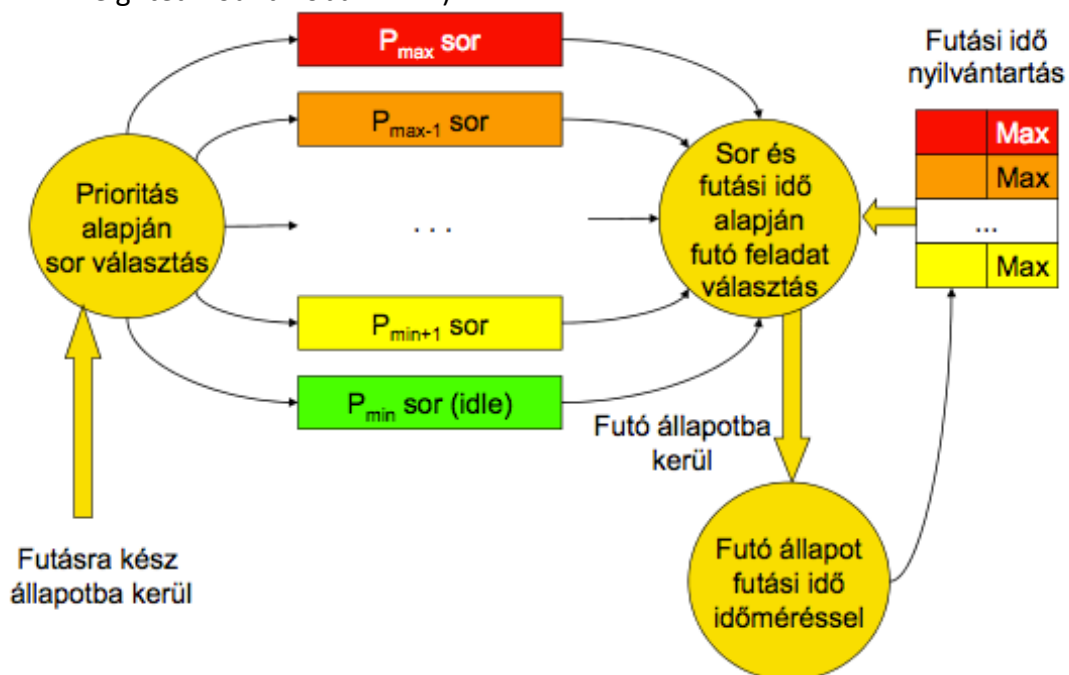
- működése: érkeznek be feladatok, prioritás alapján választunk egy prioritáshoz tartozó sort, ebbe a sorba belerakjuk a feladatot, majd amikor futó állapotra ki kell választani az ütemezőnek egy feladatot, akkor sor alapján választ, mindig a legmagasabb prioritású sorból választ, addig amíg abban van, ha abban nincs akkor az egyel alacsonyabból választ, és így tovább
- bemeneti döntés: prioritás alapján
- kimeneti döntés: sor alapján
- kétszer döntenek, kétszer futnak

Prioritás feladathoz rendelése:

- idle legalacsonyabb prioritással
- batch feladatok alacsony prioritással
- on-line feladatok közepes interaktivitással
- rendszer feladatok magas prioritással
- valós idejű feladatok legmagasabb prioritással
- tipikus RR minden prioritási szinten



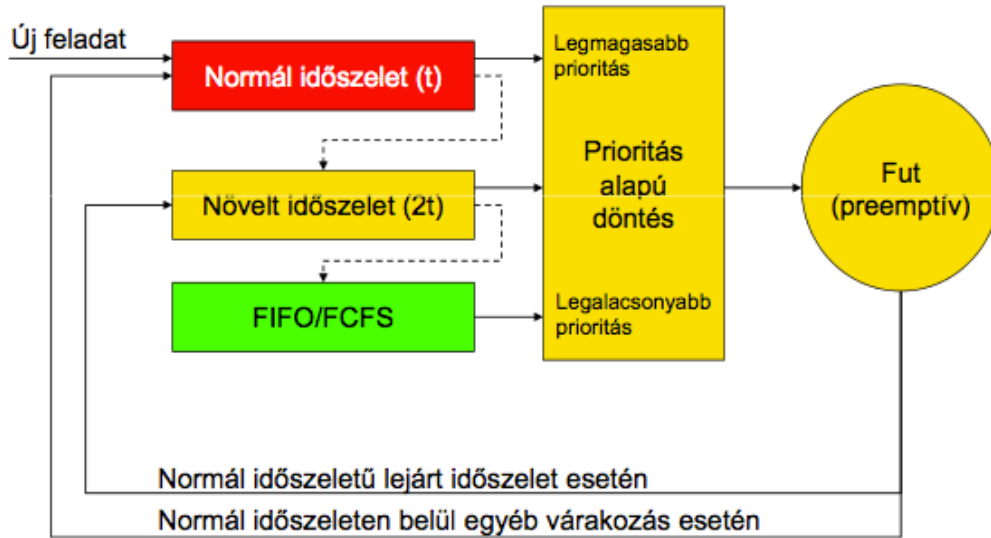
- **Probléma:** alacsony prioritási szinten éhezés ha a felsőbb szinten túl sok és sokat futó feladat van => időosztás prioritási szintek között, tárolunk egy futás idő nyilvántartást, összetett adminisztrációt igényel (Weighted fair queuing WFQ, Weighted Round-robin WRR)



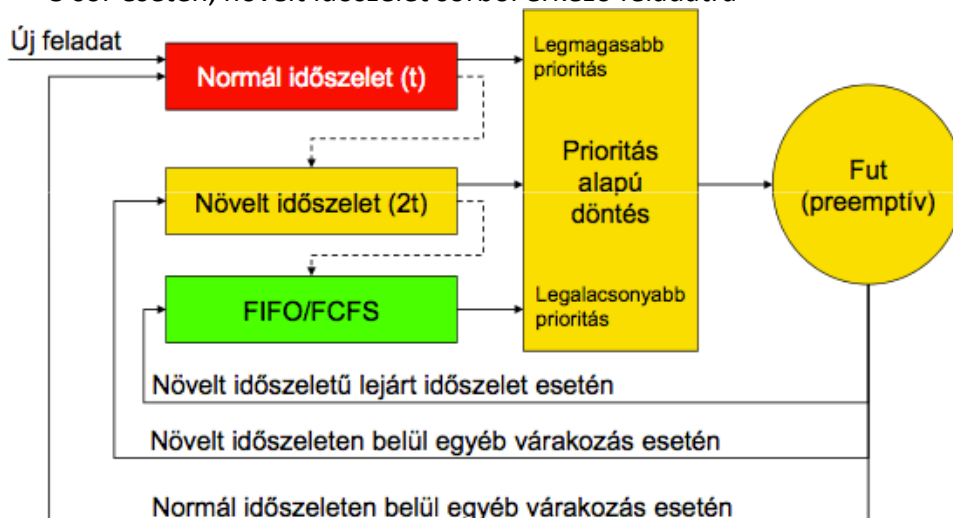
- futási időt is figyelembe vesszük a választás során, mérnünk kell az egyes feladatok és egyes prioritások által elhasznált időt
- az idle folyamatokat nem kell nyilvántartani, a többit igen
- tényleges - maximálisan felhasznált időt tároljuk, a tényleges időt időnként nullázzuk (jellegzetesen másodpercenként nullázzuk)
- bonyolult algoritmus, általános OS-ekben nem használjuk

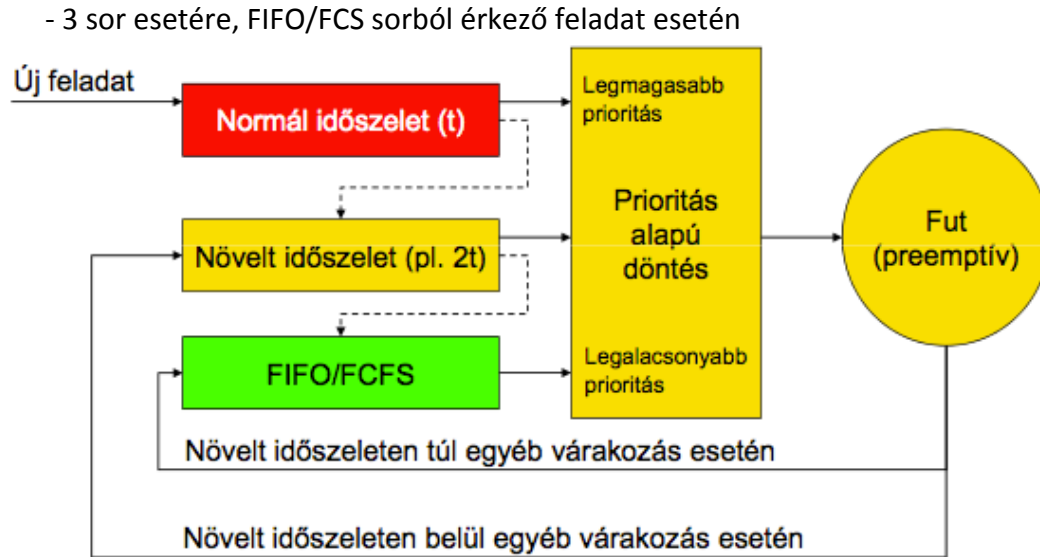
Visszacsatolt többszintű sorok: Multilevel Feedback Queues (MFQ)

- széles körben alkalmazzák, részleteiben eltérő algo
- 3 sor esetén, normál időszel sorból érkező feladatra



- működése: a normál időszelű sorból érkezik a feladatunk, akkor neki a legnagyobb prioritása van, az ütemező mindig innen fog választani, és azt behelyezi futó állapotba, amikor az fut, akkor egy idő után az időszel lejárt neki/lejárhatna neki. Ha nem jár le, hanem ő I/O löketet hajt végre vagy megszakítják akkor visszamegy a normál időszelű sorba. Ha őt meg tudja szakítani az OS az időszel alapján átkerül a növelt időszelű várakozási sorba, ahol az OS már alacsonyabb prioritással fogja kezelni ezt a feladatot. Ha egy feladat a növelt időszelű sorból érkezik, akkor lefut időszelén belül lefut vagy nem. Ha megszakítják akkor átkerül a normál időszelű sorba. Ha a növelt időszelén belül megy el várakozni, akkor növelt időszelű sorban. Ha megint túllépi a növelt időszelét, akkor lekerül a FIFO sorba. Hasonlóan a FIFO-ban.
- CPU burst alapján a feladat más módon prioritással futtat le
- sok helyen használják: Windows, Linux kernelben
- 3 sor esetén, növelt időszel sorból érkező feladatra



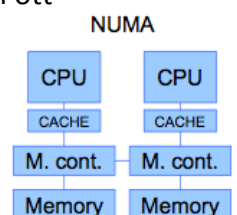
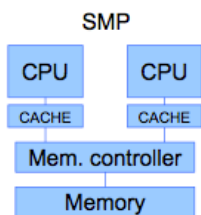


Többprocesszoros ütemezés (multiple-processor scheduling):

- homogén többprocesszoros rendszerek esetén (SMP vagy NUMA)
- egy I/O periféria egy adott CPU-hoz van rendelve (arra csatlakozik)
- két lehetséges megoldás:
 - masters and slaves (egy CPU osztja ki a feladatokat)
 - a master futtatja az ütemezési algoritmust, irányítja a slaveseket
 - self-scheduling / peering (minden CPU ütemez)
- self-scheduling-ot használják Windows, Linux OS-eknél
- master/slaves esetben sok az adminisztráció/kommunikáció, kis számú CPU esetén rossz kihasználtság

Processzor affinitás (Processor Affinity):

- oka: CPU-nak cacheiben az adatok ott vannak, a rajta futó folyamatok adatai ott vannak, és ha azt a feladatokat átrakjuk másik CPU-ra, akkor a másik CPU cache-ében nem lesz ott, és amíg át nem kerül addig lassabban tudja végrehajtani a másik CPU a folyamatot
- a cache a futó feladat utasítás és adat tartalmának egy részét tartalmazza
- cél: a feladatot ugyanazon a CPU-n tartani: *laza/kemény CPU affinitás*
- SMP esetén a CPU affinitás csak a cache tárolók szempontjából érdekes, azért van rá szükség, hogy a CACHE találati arány magas legyen
- NUMA esetén a feladat által használt fizikai memória is érdekes: a feladat CPU-hoz közeli memóriából fusson, távoli mem. elérés minimalizálás



Terhelés megosztás (load balancing):

- egy globális futásra kész sor vagy processzoronként futásra kész sor
- Processzoronkénti task queue:
 - push and/or pull
 - push: OS kernel folyamat mozgatja a sorok között a feladatokat
 - pull: az idle állapotban CPU próbál a többi sorából feladatot kapni
- összefüggő, párhuzamosan futtatható feladatok optimalizálása: Gand scheduler
 - elsősorban erősen párhuzamos feladatok esetén szükséges

- nagyobb CPU számig lineáris skálázódás érhető el vele

7. Feladatok együttműködése

Feladat (task) fogalma:

- feladat fogalma eredetileg folyamat (process) értelemben került használatra
- folyamat = végrehajtás alatt álló program
 - ugyan abból a programból több folyamat is létrehozható
 - saját kód, adat, halom (heap), verem (stack) és szabad memória területtel rendelkezik, a verem/halom dinamikusan nő
 - védettek a többi folyamattól (szeparáció, virtuális gép, sandbox)

Folyamatok szeparációja:

- Virtuális CPU-n futnak: nem férhetnek hozzá a többi folyamat és az OS futása során előálló processzor állapothoz, nem férhetnek hozzá másik folyamat regisztereikhez
- saját virtuális memóriaterületük van: nem férhetnek hozzá más folyamatok virtuális memóriájához vagy direkt módon a fizikai memóriához, ennek megoldása az MMU feladata

Folyamatok létrehozása

- OS specifikus rendszerhívás (pl.: `CreateProcess()`, `fork()`, ...)
- szülő/gyerek viszony a létrehozó és létrehozott között => process fa (process tree), a szülő erőforrásaihoz való hozzáférés többnyire konfigurálható, a szülő megvárhatja a gyerek terminálódását vagy futhat vele párhuzamosan, paraméterezhető a gyerek
- sok adminisztráció, erőforrás igényes

Folyamatok kommunikációja:

- a folyamatoknak együtt kell működniük => kommunikáció kell
- két tetszőleges folyamat nem tud közös memórián keresztül kommunikálni -> MMU, virtuális memória, OS rendszerhívásokon keresztül tudnak csak
- hatékony a védelem/szeparáció szempontjából
- nem hatékony módja a párhuzamos, erősen összefüggő feladatok megoldásának

Folyamatok befejezése:

- OS specifikus rendszerhívás (pl.: `TerminateProcess()`, `exit()`, ...)
- nyitott, használatban lévő erőforrásokat le kell zárni (pl.: fájlokat)
- szülő megkapja a visszatérési értéket (általában integer)
- a szülő megszűnik de a gyerek nem: OS specifikus, például: gyerek is megszűnik, alapértelmezett szülő (UNIX-ban `init` alapértelmezett szülő, neki nincs szülője)
- sok adminisztráció, erőforrás igényes

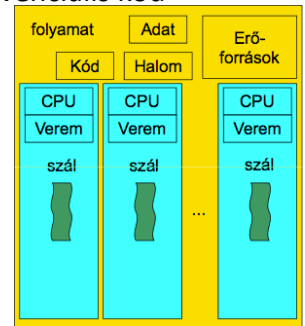
Folyamatok értékelése:

- védelmi/szeparációs szempontból jó megoldás, de erőforrás igényes (létrehozás, kommunikáció, erőforrás megosztás, megszüntetés)

- megoldás: szál (thread):

- a szál a CPU használat alapértelmezett egysége, magában szekvenciális kód
- egy szálon belül nincsenek párhuzamosan végrehajtott részek
- saját virtuális CPU-ja van, saját verem
- kód, adat, halom, egyéb erőforrások osztozik a többi szálakkal, melyekkel azonos folyamat kontextusában fut

- folyamat = nehézűyű folyamat, szál = nehézűyű folyamat



Szálak támogatása:

- jelenleg natív módon támogatják az OS-ek a szálak létrehozását
- Windows: az ütemező szálakat ütemez
- Linux: az ütemező taskokat ütemez, melyek lehetnek folyamatok vagy szálak

Felhasználó módú szálak:

- korábban UNIX/Linux alatt: green threads
- ha az OS csak a folyamatot tudja ütemezni, ha az fut akkor azon belül a felhasználói módú szál könyvtár saját ütemezője fut

Szálak létrehozása:

- Win32 API: CreateThread() bonyolult paraméterezéssel
- Pthreads: POSIX threads
- JAVA thread: VM a folyamat, VM-en belül szál, Thread osztályból származtatta, runnable interface megvalósítása, platform-specifikusan
- a JAVA virtuális gép egy folyamat a hoszt OS-en belül
- natív OS specifikus szál (one-to-one, tipikus)
- JAVA specifikus szálak (many-to-one) egy natív OS szála vagy folyamatra leképezve
- many-to-many leképezés

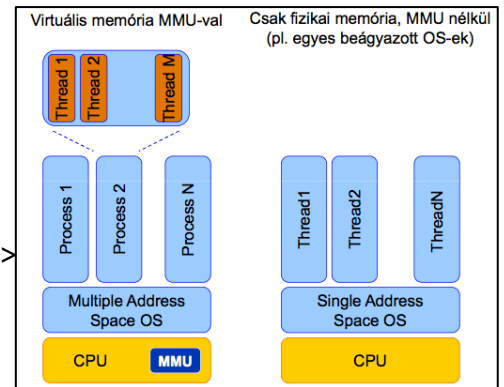
Szálak alkalmazásának előnyei:

- kis erőforrás igényűek a létrehozásuk és megszüntetésük
- egy nagyságrenddel gyorsabb mint a folyamatok létrehozása
- alkalmazáson belüli többszálúság támogatása
- gyors kommunikáció közös memórián az azonos folyamat kontextusában futó szálakra, csak a verem szál specifikus
- hátrány: megtörténhet az kedvezőtlen esetben, hogy egy szál átírja egy másik szálnak a vermet, hozzáférhet, hisz ugyanazon a memóriaterületen van
- skálázhatóság, alkalmazáson belül kihasználható több CPU
- a szál és a biztonság kritikusság között semmi kapcsolat nincs

Szálak alkalmazásának következményei:

- a közös memórián keresztüli kommunikáció veszélyes, a kommunikációra használt memóriaterület konzisztenciája sérülhet
- megoldás: kölcsönös kizárás
- eltérő folyamatok kontextusában futó szálak kommunikációja az OS-en keresztül történik (ritkán van rá szükség)

HW támogatás - - - - -



Coroutine és fiber (rost):

- kooperatív multitasking: folyamaton vagy szálon belül, ütemezési algoritmus a felhasználó kezében
- coroutine: programnyelvi szinten
 - subroutine általánosítása
 - subroutine: LIFO, egy belépési pont, több kilépési pont, verem használata
 - coroutine: belépési pont azonos a subroutine-nal, legutolsó kilépési ponttal tér vissza, átlépés a "yield to Coroutine_id" utasítással történik, nem használhat vermet (mert megtelne a sok lépkedéstől)
 - a coroutine emlékszik arra hogy hol léptek ki belőle utoljára, és mindig ott folytatja a végrehajtást ahol kiléptek belőle
 - egy coroutine-on belül több yield is lehet, és bármelyikből léptem ki, az utáni utasításhoz fogok visszalépni
- fiber: rendszerszintű eszköz, Win32 API, Symbian
- verem alapú környezetben (C/C++) nehéz az implementációja a coroutine-nak
- nem szükséges osztani az erőforrásokon
- egyetlen végrehajtó egységet tudnak csak használni

8. UNIX folyamatok kezelése

Alapismeretek:

- folyamat: egy felhasználói program futás alatt álló példánya
- mikor indulnak a folyamatok? : rendszerhíváskor, felhasználó által
- hogyan kezeljük a folyamatokat?
 - listázás: ps parancs (ps -ef), top parancs (atop - grafikus)
 - vezérlés (pl.: leállítás): jelzésekkel (kill <JELZÉS> <FOLYAMAT AZONOSÍTÓ>),
prioritás állítás: renice
- folyamatok elkülönítése a kerneltől
 - végrehajtási mód: különbség a kernel és a folyamat programkódja között
 - kontextus: kernel és folyamat adatok közötti különbség
- végrehajtási (futási) módok:
 - kernel mód: védett (kernel) tevékenységek végrehajtása
 - felhasználói mód: a folyamat programkódjának végrehajtása
- végrehajtási környezet:
 - kernel kontextus: kernel saját feladatainak ellátásához szükséges adatok
 - folyamat kontextus (virtuális-memória kezelés): folyamat futásainak adatai,
folyamat kezeléséhez, vezérléséhez szükséges adatok

Folyamatok futtatása:

- felhasználói mód:
 - folyamat kontextus: alkalmazás
 - kernel kontextus: nincs
- kernel mód:
 - folyamat kontextus: rendszerhívás
 - kernel kontextus: megszakítások, rendszer feladatok

A végrehajtási mód váltása:

- felhasználói mód és a kernel mód között átmenet lebonyolítása
- jellemzően rendszerhívások meghívása esetén zajlik
 - a folyamat kernel módban végrehajtandó tevékenységet szeretne végezni
 - meghívja a megfelelő rendszerhívást
 - a libc kiadja a SYSCALL utasítást (megszakítást generál)
 - a CPU kernel módba vált a megszakítás hatására
 - a kernel SYSCALL kezelője előkészíti a rendszerhívás végrehajtását
 - végrehajtódik a kernel módú eljárás
 - a kernel visszatér a megszakításból
 - a CPU felhasználói módba vált a visszatérési utasítás hatására
 - a libc visszatér a folyamat által meghívott függvényből
- hardver megszakítások és kivételek esetén is kell módváltás

Rendszerhívások:

- nyomkövetése: strace parancs
- virtuális rendszerhívások
 - probléma: túl sok rendszerhívás, sok interrupt, sok kontextusváltás
 - az egyszerű esetekben próbáljuk lerövidíteni az utat: bizonyos kernel funkciókat felhasználói címtérben lévő eljárásokként teszünk elérhetővé
 - virtuális rendszerhívások (Linux): minden folyamat címtérében megjelenik egy "kernel" lap, biztonságosnak ítélt rendszerhívások érhetőek el rajta

UNIX folyamatok kontextusa részletesebben:

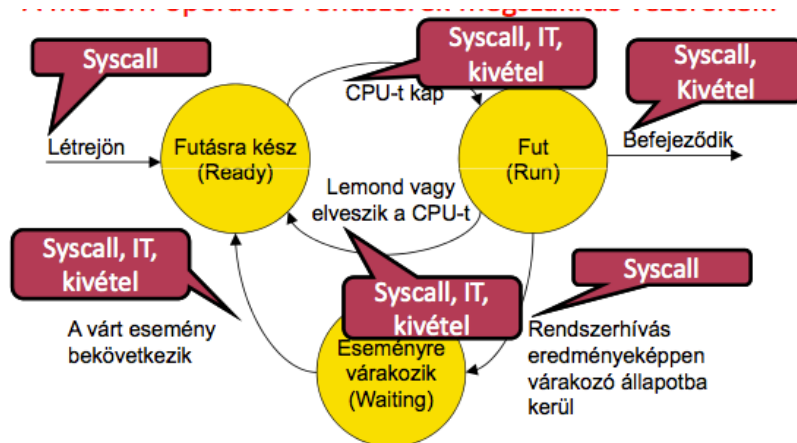
- folyamat adatok: programszöveg, adatok, veremk, ...
- hardver kontextus: cpu, mmu, fpu, ...
- adminisztratív adatok:
 - folyamat futása során szükségesek (folyamat címtér része/u terület): hozzáférés-szabályozás adatai, rendszerhívások állapotai és adatai, nyitott fájl objektumok, számlázási és statisztikai adatok
 - folyamatok kezeléséhez szükségesek (kernel címtér része/proc struktúra): azonosítók, futási állapot és ütemezési adatok, memóriakezelési adatok
- környezeti adatok (folyamat indításakor megörökölt adatok): set, setenv, expert

Folyamatok főbb adminisztratív adatai:

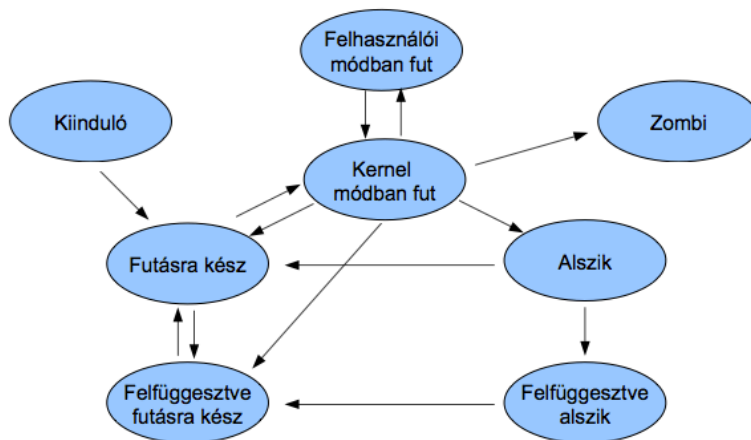
- PID (Process ID), PPID (Parent PID): szülő folyamat azonosítója
- folyamat állapota: fut, alszik, ...
- ütemezési információk (prioritás, nice, ...)
- hitelesítők:
 - UID (User ID), UID = 0 a root vagy superuser
 - GID (Group ID)
 - valós/effektív azonosítók
- memória-kezelési adatok: címleképzési térkép
- kommunikációs adatok: fájlleírók, jelzés információk
- statisztikák: erőforrás használat

Folyamatok életciklusa:

- létrehozás:
 - fork(): új folyamat létrehozása
 - exec(): új programkód betöltése egy folyamat címtérébe
- futás elindítása, futás megállítása(várkozás, felfüggesztés, leállítás, átütemezés)
- leállítás: exit(), zombi állapot, szülő értesítése, gyerekek adoptálása



UNIX folyamatok állapotai



Zombi állapot:

- azt fejezi ki, hogy a folyamat leállt, de a szülőt még nem értesítették róla
- a folyamatok adatait nem tároljuk, csak a proc struktúrában van pár adata

Felfüggesztett állapot:

- hosszútávú felhasználói ütemezés
- a felhasználó kiemelheti az ütemezésből, és visszahelyezheti

A fork() és exec() rendszerhívások:

- fork: megduplázza az éppen futó folyamatot, de valójában csak adminisztratív bejegyzéseket másol/módosít, az új folyamat az előző folyamat memórialapjait használhatja, ha ütköznek akkor kap sajátot
- a fork() eltérő értékkel tér vissza a szülő és a gyere esetében
- az exec() sikeres végrehajtás esetén nem tér vissza
- fork() variációk: clone():osztott címtér, vfork():exec, fork1():szál
- exec() variációk: exec1(), execv(), execl(), execve(), execvp(),...
- kód minta:

```

if ((res = fork()) == 0) {
    // gyerek
    exec(...);
    // ha visszatér, exec hiba történt
} else if (res < 0) { /* fork hiba */ }
// res = CHILD_PID (>0), szülő kódja fut tovább
    
```

A folyamatok családfája:

- folyamatot csak egy másik folyamat tud létrehozni, a szülő változhat
- a `fork()` megadja a szülőnek a gyerek azonosítóját (PID)
- az ős folyamat (PID 1): rendszer leállásáig fut, örökli az árva folyamatokat
- a szülő értesítést kap a gyerek folyamat leállításáról

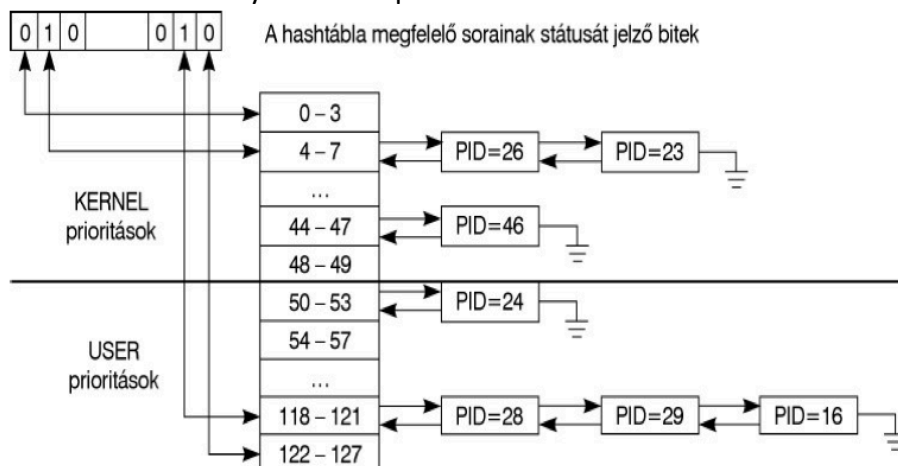
9. UNIX: folyamatok ütemezése

Feladatok ütemezése: (ismétlés)

- feladat: kiválasztani a futásra kész feladatok közül a következő futót
- alapfogalmak: preemptív, mértékek (CPU kihasználtság, átbocsátó képesség, várkozási idő, körülfordulási idő, válszidő), prioritás, statikus/dinamikus
- működés alapelve: task queue-ba rendezzük a feladatokat (FIFO)
- az ütemező a sorokból választja ki a következő futó folyamatot
- követelmények: kis erőforrásigény, optimalitás, determinisztikus, fair, elkerüli a kiéhezést, összeomlás elkerülése

Tradicionális UNIX ütemezők:

- felhasználói módban: preemptív, időosztásos, időben változó prioritásos
- kernel módban: nem preemptív, nincs időosztás, rögzített prioritás
- adatstruktúra:
 - prioritás 0-127 között, 0 legmagasabb, 127 legkisebb
 - 0-49:kernel prioritási szintek, 50-127: felhasználói szintek
- az ütemező a folyamatokat prioritásuk szerint 32 db sorba rendezi:



Ütemezés kernel módban:

- prioritás statikus, nem preemptív
- prioritás nem függ a felhasználói módban lévő prioritástól és CPU időtől
- a prioritást a folyamat elalvásának az oka határozza meg: alvási prioritás

Ütemezési felhasználói módban:

- a számított prioritás az ütemezés alapja
- ütemez minden óraciklusban: (van-e magasabb szinten folyamat)?(átütemez):(-)
- ütemez minden időszelet végén (10 óraciklus): Round-Robin algoritmus

Ütemezés további adatai (prioritás számításhoz):

- p_{pri} a folyamat aktuális prioritása
- p_{usrpri} a folyamat felhasználói módban érvényes prioritása
- p_{cpu} a CPU használat mértékére vonatkozó szám
- p_{nice} a felhasználó által adott prioritás módosító értéke
- kernel módba váltáskor a prioritás értéke elmentődik a p_{usrpri} -ba

Prioritás számítása felhasználói módban:

- minden óraciklusban a futó folyamat p_cpu értékét növeli: p_cpu++
- minden 100. óraciklusban a kernel kiszámítja a prioritás értékét
 - minden folyamatnál "öregíti" a p_cpu értékét : $p_cpu = p_cpu * KF$
 - kiszámolja a prioritást: $p_pri = P_USER + p_cpu/4 + 2*p_nice$
- korrekciós faktor: $KF = 1/2$, 4.3 BSD: $KF = 2 * load_avg / (2 * load_avg + 1)$
 $load_avg$ = futásra kész feladatok száma

Felhasználói módú ütemezés összefoglalója:

- minden óraütésnél:
 - ha van magasabb prioritási sorban folyamat, akkor átütemezés
 - a futó folyamat p_cpu értékének növelése (csökkenni fog a prioritása)
- minden 10. óraütésnél:
 - Round-Robin ütemezés a legmagasabb futási szinten lévő folyamatok között
- minden 100. óraütésnél
 - korrekciós faktor kiszámítása, az elmúlt 100 ciklus átlagos terhelése alapján
 - a p_cpu "öregítése"
 - a prioritás újraszámolása
 - szükség szerint átütemezés

Ütemezési mintapéldák: (1.3)

<http://home.mit.bme.hu/~micskeiz/opre/files/opre-feladatok-segedanyag.pdf>

Tradicionális UNIX ütemező értékelése:

- + egyszerű, hatékony
- + általános célú, időosztásos rendszerben megfelelő
- + jól kezeli az interaktív és batch típusú folyamatok keverékét
- + elkerüli az éhezést
- + jól támogatja az I/O műveleteket végrehajtó folyamatokat
- nem skálázódik jól: sok folyamat -> sok számítás
- nem lehet folyamat (csoportok) számára CPU-t garantálni
- nem lehet a folyamatokat igényeik szerint eltérően ütemezni
- nincs garancia a válaszidőre
- a többprocesszoros támogatás nem kidolgozott
- a kernel nem preemptív

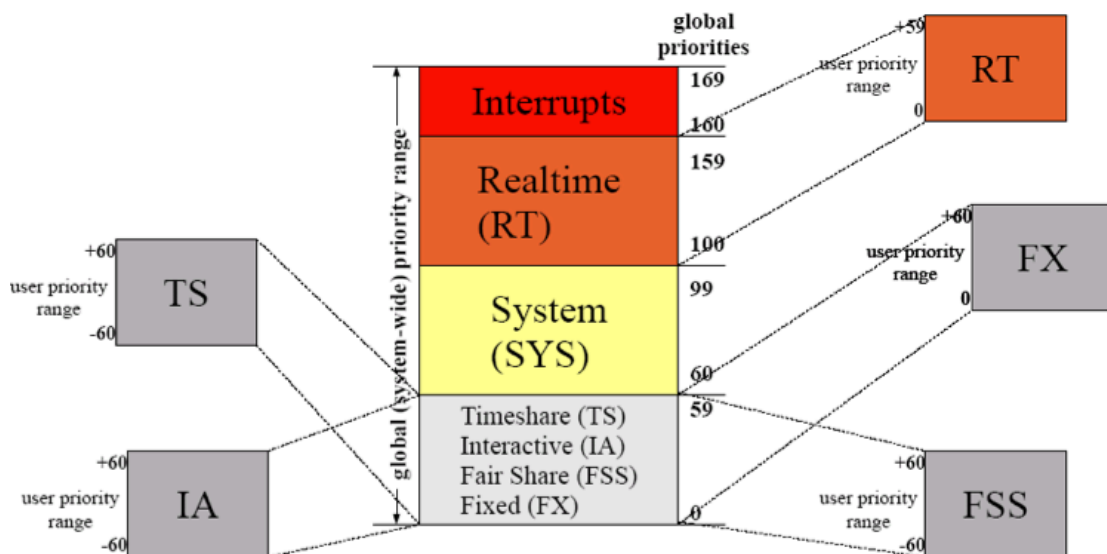
Modern UNIX ütemezők szempontjai:

- új ütemezés osztályok: speciális igényekre, "fair share", a kernelek is egyre több saját folyamatot (szálat) futtatnak, batch real-time kernel és interaktív folyamatok keveréke fut egy időben, moduláris ütemező rendszer szükséges
- kernel megszakíthatóság: többprocesszoros ütemezésnél elengedhetetlen
- az ütemező erőforrásigénye: az ütemezés egyre összetettebb feladat, az ütemezési algoritmusok komplexitása lineáris, de inkább $O(1)$ legyen
- szálak kezelése

Solaris ütemező tulajdonságai:

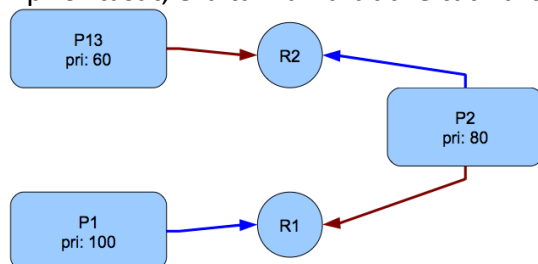
- száralapú, kernel preemptív, támogat többprocesszoros rendszert/virtualizációt
- többféle ütemezési osztály:
 - időosztásos (TS): ütemezés alapja hogy ki mennyit várt/futott
 - interaktív (IA): TS + az éppen aktív ablakhoz tartozó folyamat kiemelése
 - fix prioritás (FX): állandó prioritású folyamatok
 - fair share (FSS): CPU erőforrások folyamatcsoportokhoz rendelése
 - real-time (RT): a legkisebb késleltetést biztosító legmagasabb szint
 - kernel szálak (SYS): az RT szint kivételével mindennél magasabb szint

A Solaris ütemezési szintjei:



Örökölt prioritások (Solaris): a prioritás inverzió problémája
(kék nyíl:vár, piros:foglal)

Megoldás: prioritás emelés, avagy örökölt prioritások, P13 meg fogja örökölni P1 prioritását, ezáltal hamarabb le tud futni, és nem tartja fel a többi folyamatot



Linux ütemezők története, tulajdonságai:

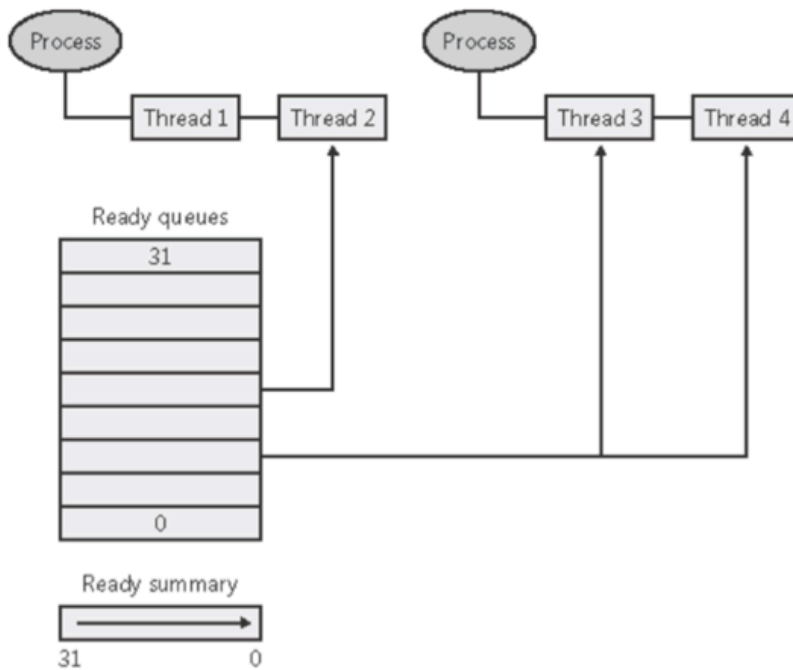
- első változatok (v2 előtt) tradicionális ütemezőre épültek
- 2.4 előtt: ütemezési osztályok: RT/nem-preemptív/normál, O(n) ütemezés, egy ütemezési sor, nem preemptív kernel
- kernel v2.6: O(1) ütemezés, processzoronkénti futási sorok, I/O és CPU terhelésű folyamatok heurisztikus szétválasztása
- 2.6.23: CFS (Completely Fair Scheduler): CPU idő biztosításának kiegyensúlyozása egy speciális adatstruktúra segítségével (PF fa)

10. Ütemezés Windowsban

Ütemezési alapelvek:

- preemptív ütemező (kernel és user módban is)
- 32 prioritási szint: legmagasabb fut mindig, azonosok között Round-Robin
- a szálak adott ideig futnak (quantum)
- nincs mindig futó központi ütemező, ütemezést események indítják
- szálak prioritása válthozhat a futás során, kernel módú szálak prioritása is
- 31-16: 16 "realtime" szint: nem igazi valós idejű, csak nincs prioritási szint változás
- 15-1: 15 dinamikus szint: OS változtathajta a prioritást futás közben
- 0: zero page thread: felszabadított memórialapok kinullázása
- idle szál(ak) - ha nincs futásra kész, üres idő számlálására
- szálak: 7 féle relatív prioritás
- a szálakhoz a processzoronkénti prioritáson kívül I/O prioritás is rendelhető:
Vista kernel módosítás, 5 féle prioritás az I/O kéréseknek, I/O sávszélesség foglалás

Várakozás sorok - kész szálak:

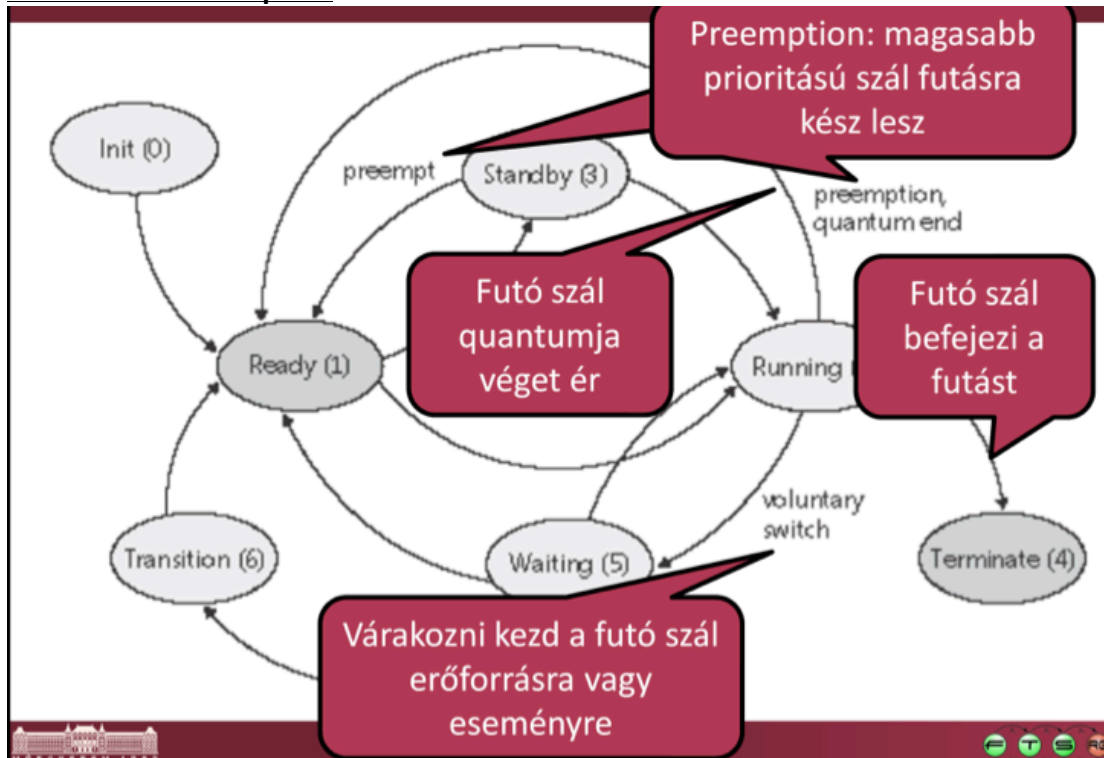


- kétszeresen láncolt listák a futásra kész szálakról, lista elejéről veszi le az ütemező a következő szálát -> nem $O(n)$ komplexitású

Quantum: RR ütemezésnél az időszelet

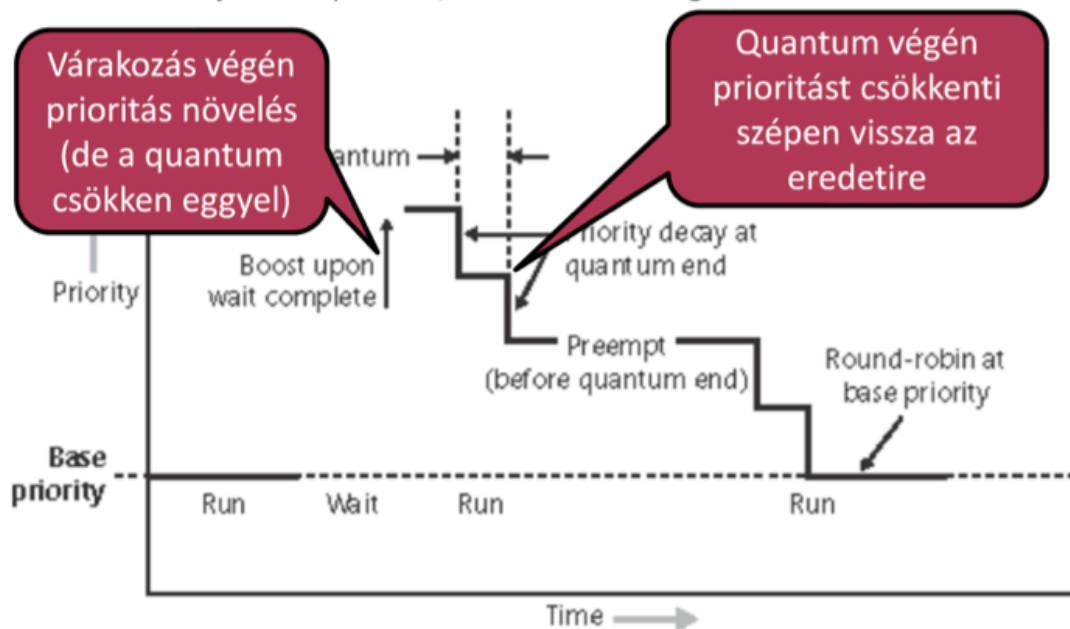
- óra megszaításban mérik (clock tick), tárolás: "3*clock tick száma, futó szálak quantumja 3-mal csökken minden óraütéskor (Vista előtt)
- Quantum hossza: kliens esetén 2-6 clock tick (előtérben lévők hosszabbak), szerver esetén mindenkinek 12

Ütemezés életbe lépése:



Prioritás módosítása:

Adjunk esélyt annak, akinek most ért véget a várakozása!



Éhezés elkerülése:

- OS mp-enként megnézi a futásra kész szálakat
- aki nem futott már 300 óraútés óta: 15-ös prioritás, XP-n quantumját megduplázza, Serveren 4-re állítja, egy quantumnyi futásig

SMP (Symmetric Multiprocessing):

- minden CPU egyenrangú: közös memória címtér, IT-t bármelyik CPU kiszolgálhat
- CPU-k maximális száma a registry-ben tárolva
- implementációs limit (bitvektor hossza): 32 CPU 32 bites, 64 CPU 64 bites
- Windows 7/Server 2008 R2 : logikai CPU csoportok, 4*46 CPU, NUMA támogatás

Multiprocesszoros ütemezés:

- szálak bármelyik CPU-n futhatnak, de megpróbálja az OS ugyanazon a CPU-n tartani (*"soft affinity"*), beállítható hogy melyiken futhat (*"hard affinity"*)
- elosztott: bármelyik CPU bármelyik futást megszakíthatja
- ütemezési adatok tárolása:
 - Windows Server 2003 előtt: rendszerszintű futásra kész sorok
 - Windows Server 2003-tól: CPU-nkénti sorok

Windows 7 módosítások:

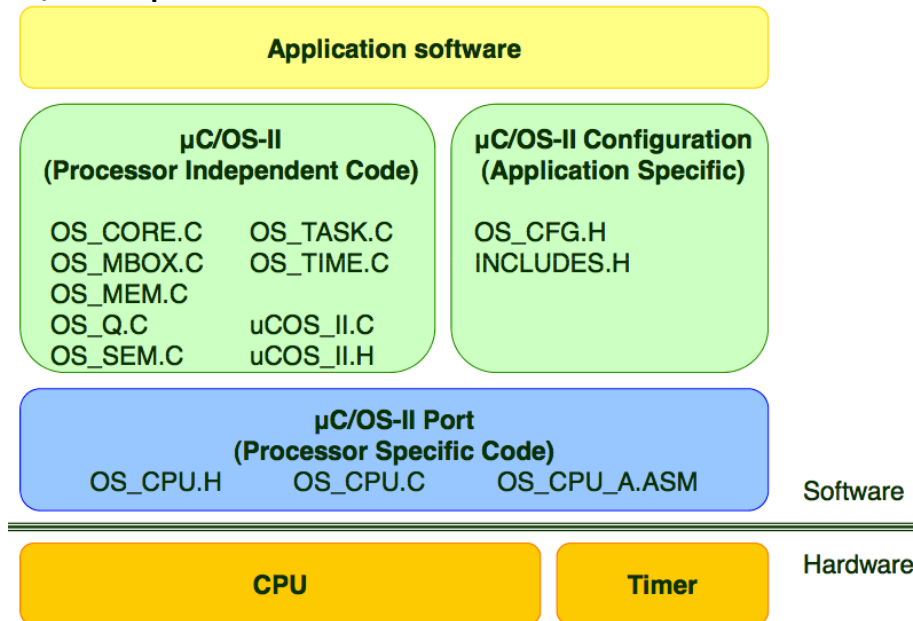
- *Core Parking (server)*: minél kevesebb logikai CPU használata, nem használt socketek standby üzemmódban
- time coalescing: azonos periódusú időzítők összevonása
- *DFSS (Dynamis Fair Share Scheduling)*: remote desktop szerverekhez, minden munkamenet "CPU keretet" kap, ha elhasználja csak idle CPU-t kaphat
- globális zárok megszüntetés

11. uC/OS-II

Tulajdonságok:

- nyílt forráskód, hordozható, skálázható, multi-tasking, preemptív ütemező
- determinisztikus futási idő, különböző mértű stack-ek
- rendszer szolgáltatások: mailbox, queue, semaphore, idő...
- IT management, robosztus, megbízható
- jó dokumentáció, kiegészítő csomagok
- C és assembly nyelven megült megírásra
- az OS-t az alkalmazás indítja el

uC/OS felépítése:



Taszk állapotai: Running, ISR(IT rutin), Waiting, Ready, Dormant("szunnyadó"), ha nincs futásra kész task, akkor az OSTaskIdle() fut

uC/OS ütemezője:

- 2D bitmap struktúrában tárolódnak a futásra kész taszkok
- (+): gyors beszúrás
- (-): taszkoknak egyedi prioritás -> RR kizárva, lookup táblázat a bitmap mellé
- az ütemező lényegében egy függvény:
 1. taszk kivétele a futásra készek közül
 2. taszk futásra készvé tétele
 3. legmagasabb prioritású, futásra kész taszk megtalálása
 4. kontextus váltás

Időzítő: hardver timer, ami periodikus megszakításokat generál

Egyéb szolgáltatások:

- taszk kezelés, idő kezelés
- memória partíciók kezelése, szinkronizáció/kommunikáció

12. Feladatok együttműködése (kölcsonös kizárás, szinkonizáció, kommunikáció)

Együttműködés lehetőségei:

- közös memórián keresztül (RAM v. PRAM modell) : szálak esetén
- üzenetekkel

RAM modell:

- klasszikus Random Access Memory (egy végrehajtó egység)
- tárolórekeszekből áll, 1D-ban, rekeszenként címezhető, írás és olvasás
- írás: teljes rekesztartalmat felülír, olvasás: nem változtatja a rekesz tartalmát

PRAM modell:

- Parallel Random Access Memory (sok végrehajtó egység)
- több végrehajtó egység olvashatja/írhatja párhuzamosan
- változtatások a RAM modellhez képest:
 - olvasás - olvasás => ugyanazt olvassák
 - olvasás - írás, írás - írás => versenyhelyzet
- gyakorlatban ezt használjuk

Erőforrás (resource):

- olyan eszköz amire a párhuzamos programnak futás közben szüksége van
- legfontosabb erőforrás a végrehajtó egység
- memória és annak tartalma, perifériák, ...

Közös erőforrás (shared resource):

- egy időintervallumban több, párhuzamosan futó feladatnak lehet rá szüksége
- az erőforráson osztoznak a feladatok
- egy időben egy vagy maximum megadott számú feladat tudja helyesen használni
- fontos feladat: felismerni a közös erőforrásokat, és helyesen használni
- az OS szolgáltatásokat nyújt probléma megoldására, de a megoldás a programozó kezében van

Kölcsonös kizárás (mutual exclusion):

- annak biztosítás, hogy a közös erőforrást egy időben csak annyi magában szekvenciális feladat használja, amely mellett a helyes működés garantálható
- a kölcsönös kizárást meg kell oldanunk a programban
- többnyire a használt erőforrást lock-oljuk (elzárjuk)

Kritikus szakasz (critical section):

- a magában szekvenciális feladatok azon kódrészletei, amely során a kölcsönös kizárást egy bizonyos közös erőforrásra biztosítjuk
- a kritikus szakaszt a hozzá tartozó erőforrásra atomi műveletként kell végrehajtani, nem megszakítható módon

Atomi művelet (atomic operation):

- nem megszakítható művelet, a CPU egy utasításként hajtja végre
- TAS, RMW, speciális CPU utasítások az IT tiltás/engedélyezés elkerülésére
- a közös erőforrás lock-olást, kritikus szakasz megvalósítást atomi műveletekre vezetjük vissza

Közös erőforrások védelme:

- közös erőforrásokhoz hozzáférhetek: ISR (IT service routine), feladat, DMA
- lehetőségek: IT tiltás/engedélyezés, locking, ütemező lista tiltása/engedélyezése
- a lock megoldás sokak szerint nem jó, de jobb mint bármi eddigi megoldás

Újrahívhatóság (reentrancy):

- a közös erőforrás problémájának egyfajta kiterjesztett esete egy függvényen belül is felléphet, amennyiben ezt a függvényt egyszerre többen is meghívhatják
- előfordulása:
 - ugyanazt a függvényt hívjuk taszkból is és egy IT rutinból is
 - ütemezés preemptív, és ugyanazt a függvényt hívjuk két taszkból
- újrahívhatóság feltételei: újrahívott függvényben használt változók, perifériák, függvények, stb. közös erőforrásnak minősülnek, és ha azok, akkor azokat közös erőforrásként megfelelően kezeli-e a függvény?
- pl.: PC BIOS függvényei nem újrahívhatóak, preemptív OS API függvényei újrahívhatóak

Várakozás közös erőforrásra: más feladat által használt közös erőforrásra is "eseményre várakozik" állapotban várakozik a feladat, az OS nyújt a közös erőforrások védelmére szolgáltatásokat

Lock-olás és az ütemező:

- a közös erőforrást használni kívánó feladat meg próbálja szerezni az erőforrást egy OS hívással
- ha az erőforrás szabad :
 - az OS lefoglalja az erőforrást, visszatér "fut"/"futásra kész" állapotba, majd CPU megkapás után a hívásból visszatérve tud tovább
 - használja az erőforrást
 - használat végén felszabadítja azt egy OS hívással
- foglalt erőforrás esetén:
 - a feladat az adott erőforrást váró feladatok várakozási sorába "eseményre vár" állapotba kerül
 - fut az ütemező a következő futó feladat kiválasztására
 - ha egy másik feladat később felszabadítja az erőforrást:
- fut ismét az ütemező, és kiválaszta az erőforrásra váró feladatok közül valamelyiket
 - annak számára lefoglalja az erőforrást
 - majd "futásra kész" állapotba helyezi azt

- lock-olás részletessége
 - kölcsönös kizárás megvalósítása erőforrás használatával jár, minimalizálni kell
 - túl nagy egységekben végzett kölcsönös kizárás is erőforrás pazarlással jár

Hibák:

- versenyhelyzet (race condition):
 - párhuzamos program lefutása során a közös erőforrás helytelen használata miatt a közös erőforrás vagy az azt használó program nem megfelelő állapotba kerül
 - a hibás állapot részletei erősen függenek az azt használó szekvenciális részfeladatok lefutási sorrendjétől
- kiéheztetés (starvation):
 - ha a párhuzamos rendszer hibás működése miatt egy feladat soha nem jut hozzá a működéséhez szükséges erőforrásokhoz, akkor ki van éheztetve
 - nem csak a CPU-ra, de más közös erőforrásokra is felmerülhet
- holtpont:
 - a közös erőforrások hibás beállítás vagy használata miatt a rendszerben a részfeladatok egymásra várnak
 - nincs futásra kész folyamat
 - nem jöhet létre belső esemény
 - a rendszer nem tud előrelépni
- livelock:
 - többnyire a hibás holtpont feloldás eredménye
 - a rendszer folyamatosan dolgozik, de nem lép előre
- prioritás inverzió (priority inversion):
 - prioritásos ütemezőkben fordul elő, de az erőforrás használatával is összefügg
 - kedvezőbb esetben csak a rendszer teljesítménye csökken, a válszidők nőnek
 - rosszabb esetben kiéheztetés, vagy akár holtpont is lehet
 - a legegyszerűbb előfordulásához kell:
 - 3 feladat, különböző statikus prioritással
 - egy közös erőforrás, amelyet a 3 feladat közül a legmagasabb és a legalacsonyabb is használni kíván
 - a közepes prioritású feladatnak CPU intenzívnek kell lennie

Prioritás inverzió megoldások:

- Prioritás öröklés (PI, Priority Inheritance)
 - az alacsony prioritású feladat megörökli az általa kölcsönös kizárással tartott feladat prioritását a kritikus szakaszából való kilépésig
 - csak részben oldja meg a problémát
- Prioritás plafon (PC, Priority Ceiling)
 - majdnem ugyan az, de az adott közös erőforrást használó legnagyobb prioritású feladat prioritását örökli meg
 - az adott erőforrást máskor használó többi feladat sem tud futni
 - az alacsony prioritású feladat akadály nélkül le tud futni
- modern beágyazott OS-ekben választható
- a prioritás plafon protokoll tervezési időben használható

Prioritás inverzió más szempontból:

- sokak szerint a prioritás inverzió alapvetően egy rendszertervezési hiba eredménye
- nem speciális protokollokkal kell megoldani, hanem jól kell tervezni a rendszert

Hogyan lock-olunk:

- passzív várakozás az OS szolgáltatásaink felhasználásával
- aktív várakozás

Lock feloldásra várkozás passzívan: sleeplock, blocking call, ...

- ütemező által karbantartott várakozási sorok
- ha az erőforrás nem lockolt, akkor megkapja az erőforrást a feladat lezárva, és fut tovább
- ha az erőforrás lock-olt, akkor a feladat megy az erőforráshoz tartozó várakozási sorba, a futásra kész feladatok közül egy futó állapotba kerül, ezután ha az erőforrás felszabadul, akkor az erőforráshoz tartozó sor elején álló megkapja az erőforrást lezárva, és futásra kész állapotba kerül
- erőforrás-takarékos, de van overhead-je
- utána csak futásra kész sorba kerül a részfeladat, pontos időzítés nehezen megoldható, alsó korlátot ad
- a processzor aludhat, ha nincs feladat: HW IT-re felébred

Lock feloldásra várkozás aktívan: livelock, spinlock, busy wait, spinning, ...

- aktív várakozás az erőforrás felszabadulására, megszerzésére - CPU pazarlás
- fogyasztás is nő, hiszem a CPU folyamatosan fut, nem tud aludni
- Compiler optimalizáció kiszedheti/eltávolíthatja a kódot
- függ a CPU sebességétől

Hogyan várakozunk?

- rövididejű várkozáshoz a spinlock elkerülhetetlen: garantáltan rövid idejű kölcsönös kizárási problémák kezelésére, periféria kezelés során
- HW Timer is használható adott időtartamú várakozásra, korlátolt számú, IT overheadje megjelenik, kombinálható aktív várakozással
- nehéz jó kompromisszumot találni
- az ütemező nem használ a feladatok ütemezése során spinlock jellegű hívásokat
- maga az OS kernel viszont gyakran használhat

Probléma: adott idejű várkozás: $delay(N)$

- hívásra és futásra kész állapotba kerülés között eltelt időnek a felső korlátját adja meg többnyire
- az OS rendszeróra óraütésnek a felbontásával dolgozik

Probléma: idő mérés:

- ha az OS timer felbontása nem elég (1-10-20 ms)
- időintervallum mérés két esemény között
 - szabadon futó Timer periféria értékének lekérdezése és különbség képzés
 - Timerstamp counter
 - események közötti idő, stb mérhető vele
 - időmérés többprocesszoros/elosztott rendszerben

Eszköz RAM/PRAM modell esetén:

- kölcsönös kizárás megoldására: lock bit, szemafor, kritikus szakasz obj., mutex
- minden OS-ben hasonló eszközöket fogunk találni
- tipikus hibák kivédésére és kölcsönös kizárási probléma megoldása: monitor

Lock bit:

- legegyszerűbb forma, TRUE=használhat, FALSE=nem használhat erőforrást

Atomi utasítások:

- a lock bit alkalmazása során egy súlyos hiba jelenhet meg:
 - IT a lock bit tesztelése során: a következő utasítás már nem fog lefutni, vagyis a kritikus szakaszba lépést nem jelezzük, pedig már abban vagyunk
 - az IT-ben vagy az utána futó más részfeladatokban az erőforrás szabadnak látszik
 - a védett erőforrás inkonzisztens állapotba kerülhet
- megoldás:
 - IT tiltás a teszt előtt, IT engedélyezés a beállítás után

Szemafor:

- bináris és counter típusú: egy feladat/több feladat a kritikus szakaszban
- OS hívás, a bináris szemafor egy magas szintű lock bit
- két művelet értelmezett rajta: belépés, kilépés
- a counter típusú esetén a belépés és kilépést számoljuk
- ha egynél több erőforrásra lenne szükségünk, akkor azokat vagy egyben mind megkapjuk, vagy a töredékeket nem foglalkozunk le

Kritikus szakasz objektum és Mutex:

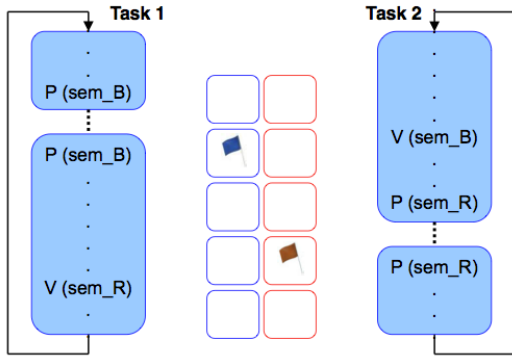
- bináris szemafor-szerűen működnek
- kritikus szakasz objektum: CriticalSection obj: létrehozás, Enter(), Leave()
- Mutex (Mutual Exclusion): Acquire(), WaitOne(), Release()

Miért lock-olunk még?

- kölcsönös kizárást lezártuk
- részfeladatok közötti szinkronizáció visszavezetése kölcsönös kizárásra: nincs védendő objektum igazából, részfeladatok együttműködése a cél

- pl.: randevú
 - két vagy több feladat összehangolt végrehajtása
 - ebben az esetben az OS által nyújtott szolgáltatásokat használunk, feladatok passzívan várnak egymásra
 - a szemaforok alapesetben foglalként vannak inicializálva ebben az esetben
- memórián keresztül történő kommunikáció

Kétoldalú randevú szemaforral, B és R foglalként inicializálva:



Kommunikáció: Védett memóriaterület:

- szálak közötti kommunikáció során:
 - folyamatok így nem tudnak kommunikálni (MMU)
 - UNIX SystemV shared memory nem valódi osztott memória, OS szolgáltatás, folyamatok közötti kommunikációt tesz lehetővé
- tömb, struktúra, objektum
- egy vagy kétirányú kommunikáció
- kölcsönös kizárást lock-bit, szemafor, mutex, kritikus szakasz objektum oldja meg

Lock-olás során tipikusan elkövetett hibák:

- belépés/kilépés elmarad
- többszöri be-/kilépés
- más erőforrás lefoglalása
- az erőforrás indokolatlanul hosszan történő lezárása
- prioritás inverzió
- deadlock, livelock

Monitor (hibák elkerülésére):

- lokalizáljuk a lock-olással kapcsolatos feladatokat a közös erőforrást körülvevő API-val
- monitor fejlődése: Hoare és Mesa szemantika²
 - Hoare szemantika: azonnal az erőforrást megszerző részfeladat fut, erőforrás igényes, és nem kompatibilis a preemptív ütemezőkkel
 - Mesa szemantika: a futásra kész feladatok közé kerül és később az ütemező futtaja

² működés, jelentés

Szemafor, Kritikus szakasz objektum, Mutex OS-en belül kerül megvalósításra:

- Egy folyamaton belül futó szálak kommunikációja közös memórián keresztül (gyors, alacsony erőforrás igény).
- kölcsönös kizárás és szinkronizáció üzenetekkel
- van overheadje: próbálkozások

13. Feladatok együttműködése (üzenet alapú kommunikáció)

Üzenetek:

- nem azonos a hálózaton küldött üzenettel, annál általánosabb
- lehet:rendszerhívás,TCP/IP kapcsolat vagy üzenet gépen belül/gépek között
- többnyire az alkalmazás kódjában OS API függvényként jelenik meg
- OS valósítja meg szolgáltatásaival

Üzenettovábbítás tulajdonságai:

- nagyobb késleltetés, kisebb sávszélesség
- a csatorna nem megbízható elosztott rendszerek esetén: a közös memória 1 valószínűséggel megbízható, de az OS-en belül történő üzenetküldésre ez nem igaz (túlterhelés), a számítógép hálózaton történő üzenetküldés definíció szerint nem megbízható
- minimum egy küldő folyamat
- minimum egy vevő folyamat: vétel után törlődik/megmarad
- lehet tulajdonosa: OS / folyamat

Üzenetek címezése:

- számítógép hálózatok...
- egy adott folyamat (unicast cím), egy folyamat (anycast cím)
- minden folyamat (broadcast cím): teljesítmény menedzsment események
- folyamatok csoportja (multicast cím)

Direkt kommunikáció:

- szimmetrikus üzenet alapú kommunikáció
 - send(P,message), receive(Q,message), P/Q folyamat azonosítók
- asszimmetrikus üzenet alapú kommunikáció
 - send(P,message), receive(id,message), id visszatérési érték, bárkitől fogadhat

Indirekt kommunikáció:

- köztes szereplő: postaláda (mailbox), üzenetsor, port, ...
- interface: létrehozás és megszüntetés + send(A,msg), receive(A,msg), A:postaláda

Blokkolás:

- nem blokkoló hívás = aszinkron
 - az eredmények és mellékhatások visszatéréskor még nem jelentkeznek
 - csak a végrehajtás kezdődik el a hívásra
 - visszatérési érték kezelése csak más értesítés után lehetséges
- blokkoló hívás = szinkron
 - az eredmények és mellékhatások a visszatérés után jelentkeznek
 - visszatérési érték kezelés egyszerű

Blokkolás a küldő oldalon:

- blokkoló send(): visszatér ha az üzenetet vették/postaládába került/időtúllépés
- nem blokkoló send(): azonnal vissza tér, és értesül hogy megkapták-e

Blokkolás a fogadó oldalon:

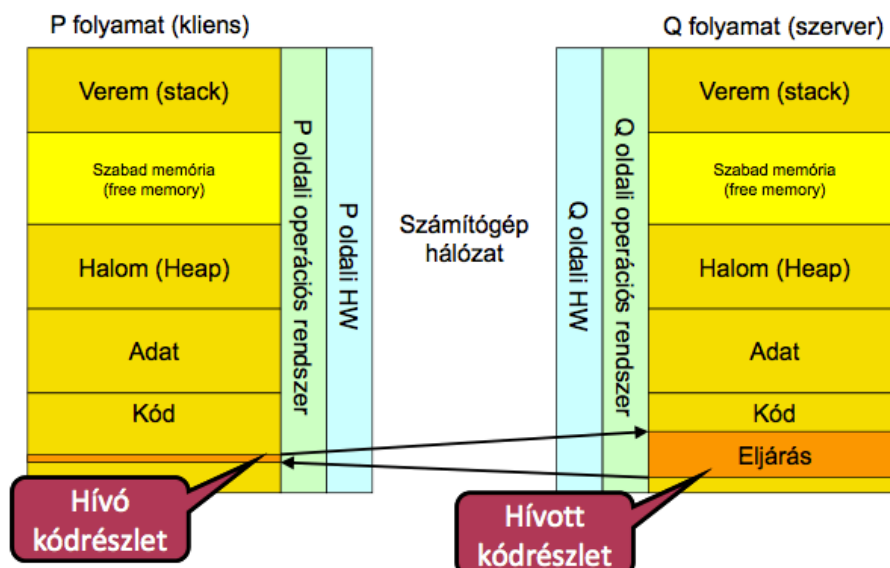
- blokkoló receive(): nem tér vissza amíg nem érkezik üzenet, pl.: listen()
- nem blokkoló receive(): azonnal visszatér ha van üzenet, ha nincs üzenet akkor azt jelzi, ha nincs üzenet akkor végtelen ciklusban arra vár

Implementációk:

- mailbox:
 - indirekt kommunikáció
 - egy vagy több üzenet tárolása, de véges számú
 - OS szintű postaláda is létezhet, illetve folyamatok is hozhatnak létre
- MessageQueue:
 - indirekt kommunikáció
 - szinte végtelen számú üzenet tárolására alkalmas
 - üzenet alapú middleware-ek
- TCP/IP TCP vagy UDP port:
 - direkt kommunikáció
 - socket interface
 - gépen belül a localhost-on (127.0.0.1/8)
 - alacsony szintű, számos más middleware alapul rajta:
 - távoli eljárás hívás
 - távoli metódus hívás
 - üzenet alapú middleware-ek
- különböző folyamatok és csővezetékek
- System V Shared Memory (UNIX, Linux): direkt, memóriaszerű interface

Távoli eljárás hívás (Remote Procedure Call, RPC):

- alapja a napjainkban használt távoli metódus hívást alkalmazó megoldásnak
- egy másik folyamat memóriaterületén elhelyezkedő függvény hívás
- a hívó fél blokkolva(passzívan) vár a távoli hívás lefutására
- a meghívott függvény az őt tartalmazó folyamat egyik vezérlési szálában fut



Az RPC a programozó számára:

- hívása azonos egy "lokális" függvény hívással: lényegében egy programkönyvtárban található függvény
- a ténylegesen végrehajtott függvény megírása sem különbözik egy lokális végrehajtandó függvény megírásától

RPC működése:

- típusos a hívás és a visszatérési érték
 - struktúrált adatküldés
 - platform függetlenség megvalósítása az OS és a környezet feladata
- a kliens program egy teljesen normális lokális függvényt hív meg
 - automatikusan generált csont (stub)
 - feladata a kommunikáció részleteinek elrejtése a hívó fél elől
 - feltételezzük hogy a hívó fél megtalálja a szervert
- a kliens oldali csont feladata a hívás során:
 - a hívási paraméterek összecsomagolása platform független formára és elküldése a végrehajtó folyamatnak
 - ehhez felhasználja a lokális OS és a hálózat szolgáltatásait
- a szerver oldalon az RPC-t használó szolgáltatás megkapja a szabványos formátumban megérkező hívást
 - platform független formától lokális formára hozás
 - lokális függvény hívása
 - visszatéréskor a visszatérési érték platform függetlenné alakítása
 - a visszatérési érték küldése a kliensnek
- a kliens oldali csont feladata a hívás visszatérése során:
 - a szerverről megkapott visszatérési érték konvertálása lokális formára
 - visszatérés a lokális csontból a visszatérési értékkel
- a kliens szál az RPC hívás során várakozik
 - eseményre várakozik a hívás során
- a szerver szál vár a beérkező hívásokra, és amikor van kiszolgálandó hívás, akkor fut, eseményre várakozik a hívás beérkezéséig

14. Feladatok együttműködése (holtpont)

Holtpont (deadlock)

- a holtpont a legsúlyosabb hiba, ami multiprogramozás során előfordulhat
- Def.: egy rendszer feladatainak H részhalmaza holtponton van, ha a H halmazba tartozó valamennyi feladat olyan eseményre vár, amelyet csak egy másik, H halmazbeli feladat tudna előállítani
- nehéz felismerni: versenyhelyzet formájában fordul elő
- szükséges feltételek:
 - 1. Kölcsönös kizárás (Mutual Exclusion): vannak olyan feladatok melyeket csak kizárólagosan lehet használni, de azokat több feladat használná
 - 2. Foglalva várakozás (Hold and Wait): olyan feladat mely foglalva tart erőforrásokat, miközben más erőforrásokra várkozik
 - 3. Nincs erőszakos erőforrás elvétel a rendszerben
 - 4. Körkörös várakozás

Holtpont kezelése:

- 1. holtpont észlelése és feloldása
 - észlelés:
 - megkülönböztetjük az egypéldányos/többpéldányos esetet
 - egypéldányos: Wait-for gráf: azt rögzítik melyik feladat vár melyikre
 - többpéldányos: Coffmann-féle holtpontdetektáló algoritmus
 - mikor fusson az algoritmus?
 - egyik szélsőség: amikor az erőforrás kérelem nem kielégíthető azonnal
 - másik szélsőség: periodikusan, alacson CPU kihasználtság idején, amikor sok feladat vár erőforrásra
 - a gyakorlatban használt algoritmusok nagy futás idejűek
 - feloldás
 - többféle megoldás: radikális, kíméletes
 - feladatok: áldozatok kiválasztása, feladatok visszaállítása
 - el kell kerülni hogy ne mindig ugyanazt a feladatot állítsuk vissza
 - a holtpont észlelése és feloldása csak akkor kísérelhető meg, ha az erőforrások és a feladatok visszaállítási pontokkal rendelkeznek, vagyis holtpont esetén a holtpont előtti helyzetükbe visszaállíthatóak
- 2. holtpont megelőzése tervezési időben
 - holtpont feltételeiből legalább egy ne teljesüljön
 - nincs futás idejű erőforrás használat
 - foglalva várakozás kizárása
 - az erőforrást birtokló feladat kér újabb erőforrást
 - minden szükséges erőforrást rendszerhívással egyben kell foglalni
 - erőforrás kihasználás romlik
 - erőszakos erőforrás elvétel lehetővé tétele: erőforrás menthető állapottal, nem kell rollback feladat szinten
 - körkörös várakozás elkerülése
 - fejlesztés során az erőforrás használat korlátozása

- teljes sorrendezés algoritmus

- 3. holtpont elkerülése futási időben

- erőforrás igény kielégítése előtt mérlegelni kell, hogy az erőforrás igény kielégítésével nem kerülhet-e holtpontra a rendszer? Bankár algoritmus

Bankár algoritmus, feltételezések:

- holtpont elkerülési algoritmus, nem pedig holtpont megelőzési algoritmus!
- N feladat, M erőforrás (többpéldányos)
- a feladatok előzetesen bejelentik hogy az egyes erőforrásokból maximálisan mennyit használnak futásuk során: MAX NxM-es mátrix
- a feladatok által lefoglalt erőforrások száma: FOGLAL NxM-es mátrix
- szabad erőforrások száma: SZABAD M vektor
- MAXr az egyes erőforrásokból rendelkezésre álló maximális példányszám
- FOGLALr az egyes erőforrásokból foglalt példányszám
- egy feladat által még maximálisan bejelenthető kérések száma:
MÉG = MAX - FOGLAL NxM-es mátrix
- a feladatok várakozó kérései: KÉR NxM-es mátrix

Bankár algoritmus, működése:

- 1. lépés: Ellenőrzés (van-e elég erőforrás)
- 2. lépés: Állapot beállítása
- 3. lépés: Biztonság vizsgálata
 - 1. lépés: Kezdőérték beállítása
 - 2. lépés: Továbblépésre esélyes folyamat keresése:
 - a maximális igény kielégíthető a jelenlegi erőforrásokkal
 - ha igen: lefuthat, és az erőforrásai visszakerülhetnek SZABAD-ba
 - ha van még feladat, akkor 2. pont, egyébként 3. pont
 - 3. lépés: Kiértékelés: feladatok listája, amelyek holtpontra juthatnak
- 4. lépés: Ha nem biztonságos a rendszer, az állapot visszaállítása

Bankár algoritmusra feladatok: (2.1)

<http://home.mit.bme.hu/~micskeiz/opre/files/opre-feladatok-segedanyag.pdf>

Bankár algoritmus értékelése:

- a holtpont lehetőségét bizonyítja, nem biztos hogy holtpontra is jut
- az algoritmus a legkedvezőtlenebb esetet vizsgálja
- MAX mátrix nem mindig áll rendelkezésünkre

Holtpont kezelése a gyakorlatban:

- többnyire tervezési időben történik, holtpont megelőzése
- a holtpont észlelése és megszüntetése többnyire humán operátorokkal
- okok: a futás idejű algoritmusok komplexek, erőforrás-igényesek, és a gyakorlatban nem feltétlenül állnak rendelkezésre a futásukhoz szükséges adatok

15. Windows: Feladatok együttműködésének ellenőrzése

Lehetnek-e ketten egyszerre a kritikus szakaszban?

- Hyman algoritmusa
- Peterson algoritmusa

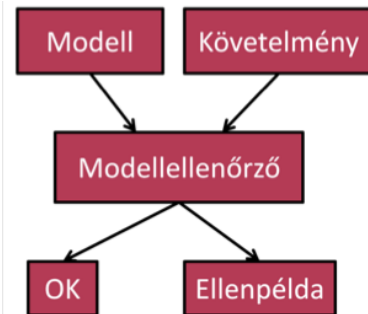
Algoritmusok helyességének ellenőrzése:

- Modellezőkkel

- modell: - rendszer működésének leírása
 - tipikusan állapotgép-szerű
- követelmény:
 - mit akarunk ellenőrizni (holtpont, kölcsönös kizárás)
 - logikai kifejezés
- modellező: fekete doboz, automatikus, TRUE/FALSE

- UPPAAL modellező:

- globális változók, automata (állapot, átmenet, órák), rendszer
 - átmenet kiválasztása, változók állapota, automaták képe
 - trace: szöveges/grafikus
- követelmény: logikus forma
 - elemei: állaputra hivatkozás, NOT, AND, OR



Példa: Étkező filozófusok.

További eszközök:

- *Java Pathfinder*
- *CHESS*: .NET, párhuzamosságból adódó hibákra
- *jchord*: java, versenyhelyzet és holtpont detektálásra
- *Static Driver Verifier*: Windows eszközmeghajtók ellenőrzésére

16. UNIX folyamatok kommunikációja

Ismétlés:

- kernel
 - felügyeli a folyamatokat, menedzseli az erőforrásokat
 - elkülöníti egymástól a folyamatokat (külön virtuális gép)
 - réteges, moduláris felépítésű
 - a folyamatok a kernellel rendszerhívásokon keresztül kommunikálnak
- folyamatok
 - a felhasználói programok futás alatt álló példányai
 - UNIX alatt szülő-gyerek kapcsolatban állnak egymással
- kommunikáció elmélete
 - közös memórián keresztül, üzenetekkel, távoli eljárással (RPC)
 - erőforrások védelme, kritikusz szakasz, szemafor
 - blokkoló (szinkron), nem blokkoló (aszinkron)

UNIX jelzések:

- cél: egy folyamat értesítése
 - a kernelben, más folyamatokban, önmagában bekövetkezett eseményről
 - szinkronizálása más folyamatokhoz
- jelzés típusa (*SIGINT*, *SIGCHLD*, *SIGKILL*)
 - rendszer: kivételek, hibák, kvóta, riasztás, értesítés
 - felhasználói: emberek (*CTRL+C*, *CTRL+V*), folyamatok
- a működés áttekintése
 - jelzést keltése (rendszerhívással / esemény bekövetkeztével)
 - a kernel értesíti a címzetteket a jelzésről
 - a címzett egy jelzéskezelő eljárásban fogadja a jelzést
- problémák a megvalósítással:
 - a keltés és a kézbesítés időben szétválík
 - sokféle implementáció, némelyik nem túl megbízható
- jelzés keltése

```
#include <signal.h>
kill(pid, SIGUSR1); // jelzés küldése
```
- jelzések kezelése:
 - többféle kezelési eljárás lehetséges, bizonyos keretek között állítható
 - core: core dump és leállítás (*exit()*)
 - term: leállítás (*exit()*)
 - ign: figyelmen kívül hagyás
 - stop: felfüggesztés
 - cont: visszatéréés a felfüggesztett állapotból
 - az alkalmazás saját kezelő függvényt is megadhat

```
void alarm(int signum) { ... }
signal(SIGALRM, alarm);
```
 - a jelzés típusától függ, hogy a fentiek közül mi az alapértelmezett, illetve minek a beállítása engedélyezett: pl.: a *SIGKILL* nem hagyható figyelmen kívül és nem definiálható rá jelzéskezelő függvény

UNIX jelzés példa:

```
#include <signal.h>           // signal(), kill()
#include <unistd.h>           // getpid()
#include <sys/types.h>        // pid_t
pid_t pid = getpid();         // saját PID
kill(pid, SIGSTOP);          // STOP jelzés küldése
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
void alarm(int signum) { ... } // az eljárás
signal(SIGALRM, alarm);      // jelzéskezelő függvény beállítása
alarm(30);                   // alkalmazás: ALARM jelzés 30 mp múlva
```

UNIX csővezetékek: *pipe()*

- cél: folyamatok közötti adatátvitel
- jellemzők:
 - csak szülő-gyerek (leszármaztatott, testvér) viszonylatban
 - adatfolyam (nincs üzenethatár, tipizálás)
 - egyirányú adatfolyam (egy/több író -> olvasó), limitált kapacitás
- a megvalósítás:
 - egy folyamat létrehoz egy csővezeték (*pipe()*)
 - a kernel létrehozza az adatstruktúrát és olvasási/írási leírókat ad vissza
 - (a folyamat továbbadja a leírókat a gyerekeinek)
 - a leírók segítségével kommunikálnak a folyamatok (*read()*, *write()*)
- korlátok, problémák:
 - nincs címzés, tipizálás, üzenethatár
 - csak a "rokonságban" működik

UNIX elnevezett csővezetékek (named pipe, FIFO):

- az egyszerű csővezetékek legkomolyabb problémájának megoldása
 - független folyamatok kommunikációja
 - avagy egy már létező csővezeték elérése egy másik folyamat által
- jellemzők
 - nem csak szülő-gyerek viszonylatban működik
 - ugyanúgy működik, mint a csővezeték
 - a létrehozása csővezeték segítségével történik
 - lehetséges a kétirányú kommunikáció (olvasás és írás)

UNIX System V IPC:

- cél: folyamatok közötti "szabályos", egységes kommunikáció
 - adatátvitel és szinkronizáció
- közös alapok (fogalmak)
 - erőforrás: a kommunikáció eszköze
 - kulcs: azonosító az erőforrás eléréséhez
 - közös kezelőfüggvények: **ctl()*, **get(... kulcs ...)*
 - jogosultsági rendszer: létrehozó, tulajdonos és csoportjaik / a szokásos hozzáférés-szabályozási rendszer (felhasználó, csoport, mások)
- erőforrások: szemaforok, üzenetsorok, osztott memória

UNIX System V IPC: szemaforok:

- cél: folyamatok közötti szinkronizáció: $P()$ / $V()$, szemaforcsoportok kezelése
- használat:
 - $sem_id = semget(kulcs, szám, opciók);$
 - adott számú szemaforhoz nyújt hozzáférést (adott kulccsal és opciókkal)
 - az *ops* struktúrában leírt műveletek végrehajtása
 - $status = semop(sem_id, ops, ops_méret);$
 - egyszerre több művelete, több szemaforon is végrehajtható
 - blokkoló és nem blokkoló $P()$ operáció is lehetséges
 - egyszerű tranzakciókezelésre is lehetőség van

UNIX System V IPC: üzenetsorok:

- cél: folyamatok közötti adatátvitel:
 - diszkét/tipizált üzenetek
 - nincs címzés, üzenetszórás
- használat:
 - $msgq_id = msgget(kulcs, opciók);$
 - adott kulcsú üzenetsorhoz nyújt hozzáférést (adott opciókkal)
 - üzenetküldés: $msgsnd(msgq_id, msg, méret, opciók);$
 - vétel: $msgrcv(msgq_id, msg, méret, típus, opciók);$
 - a típus beállításával szűrést valósíthatunk meg:
 - = 0 a következő üzenet
 - > 0 a következő adott típusú üzenet
 - < 0 a következő üzenet, amelynek a típusa kisebb-egyenlő

UNIX System V IPC: osztott memória:

- cél: folyamatok közötti egyszerű és gyors adatátvitel
 - a kernel helyett közvetlen adatátviteli csatorna
 - a fizikai memória elkülönített része, amely közösen használható
- használat:
 - $shm_id = shmget(kulcs, mérete, opciók);$
 - adott kulcsú osztott nyújt hozzáférést (adott opciókkal)
 - hozzárendelés saját virtuális címtartományhoz: az adott változót hozzákötjük a visszkapott címhez
 - $változó = (típus) shmat(...);$
 - lecsatolás: $shmdt(cím);$
 - a kölcsönös kizárást meg kell valósítani (pl: szemaforokkal)

UNIX "hálózati" (socket) kommunikáció:

- cél: címzéssel és protokollokkal támogatott adatátvitel
 - tetszőleges folyamatok között kliens - szerver architektúra
 - többféle célra (folyamatok közötti / gépek közötti)
 - sokféle protokollt támogat
 - többféle címzés
- fogalmak
 - hálózati csatoló avagy azonosító (socket): a kommunikáció végpontja
 - IP cím és portszám

- használat
 - sfd = socket(domén, típus, protokoll)*
 - szerver: bind(sfd, cím, ...);*
 - kliens: connect(sfd, cím, ...);*
 - szerver: listen(sfd, sor_mérete);*
 - szerver: accept(sfd, cím, ...);*
 - send(sfd, üzenet, ...);*
 - recv(sfd, üzenet, ...);*
 - shutdown(sfd);*

Távoli eljárás UNIX módra: (SUN) RPC:

- a socket kommunikációra épülő "elosztott rendszer" infrastruktúra
- cél:
 - folyamatok közötti magas szintű kommunikáció
 - távoli eljárások meghívása
 - programozói támogatás: interfész leírás + programgenerálás
- fogalmak
- RPC nyelv: a hívható eljárások és típusaik (interfész) leírása
 - azonosítók: a leírásban megadott egyedi számok (program, eljárás)
 - portmapper: a programazonosítók és a hálózati portok összerendelése
 - rpcgen: a leírásból C programkódot generaló program
- a Sun RPC technológia részei:
 - interfész leírás
 - programgenerátor
 - kommunikációs infrastruktúra

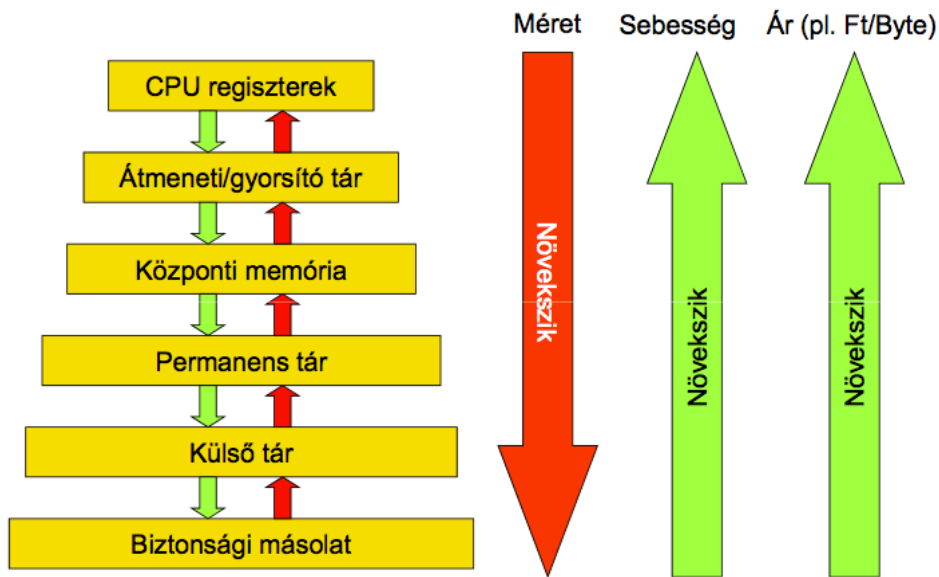
RPC interfész leírás és programgenerátor:

- RPC nyelv (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_src.c: a szerver date implementációját meghívó függvények
 - (...)

17. Memória kezelés



CPU regiszterek:

- jellegzetes D tároló: 10-100 gépi szó, nem általános felhasználású egyik része
- sebesség: az utasítás végrehajtása alatt akár többször is elérhető (pl: ADD)
- ár: nehezen kategorizálható, architektúra függő
- felejtő memória (tápfeszültség)

Átmeneti/gyorsító tár (Cache):

- jellegzetes SRAM, többnyire többszintű (1.:64-128KB|2.:1-8MB|3.:4-32MB)
- sebesség: $n \cdot 10 \text{ GB/s}$ (1.: egy/néhány órajel ciklus|2-3.: $10 - n \cdot 10$ órajel ciklus)
- ár: magas, az SRAM cella nagy félvezető területet foglal
- felejtő memória (tápfeszültség nélkül)

Központi (fizikai) memória:

- jellegzetes DRAM, $n \cdot 10 \text{ MB} - n \cdot 100 \text{ GB}$
- sebesség:
 - memory wall, max. és random access
 - max.: $n \cdot 1 \text{ GB/s} - 10 \text{ GB/s}$
 - késleltetés: $n \cdot 10 \text{ ns}$ (worst case)
- ár: erősen sebesség, méret, technológia és piac függő
- felejtő memória: tápfeszültség + idő

Permanens/háttér tár (HDD, Flash):

- HDD/flash memória, mérete: $n \cdot 1 \text{ MB} - n \cdot 100 \text{ TB}$
- fájl alapú elérés: blokk elérésű eszköz, fájl-ként látjuk a tartalmat, kivéve NOR
- sebesség:
 - max. és random access más, olvasás és írás más
 - max.: $n \cdot 10 \text{ MB/s} - n \cdot 1 \text{ GB/s}$
 - késleltetés: $n \cdot 10 \text{ ns} - 10 \text{ ms}$
- ár: erősen sebesség, méret, technológia és piac függő
- nem felejtő, de meghibásodhat: HDD: MTBF, Flash: véges számszor írható

További lépések:

- központi memória kezelése
 - fizikai, logikai, virtuális memória fogalma
 - ezeknek a megfeleltetése és kezelése
- permanens tár kezelése
 - a virtuális tárkezelésen keresztül kapcsolódik majd a központi memória kezeléséhez
 - a permanens tár kezelése, szerkezete, partíció, fájlrendszer, fájl fogalma
 - RAID, NAS, SAN

Memória:

- a CPU használja: utasítások betöltése és adatok írása/olvasása
- egymás után következő memória műveletek folyamáról van szó
 - olvasások és íráskok adott sorrendben
 - ahogy ezt a program kód előírja
- folyamat támogatás (szeparáció)
- tekintsünk el az átmeneti tárolótól (cache): hozzáférés sebességén befolyásol

Logikai cím (logical address): A CPU generálja a folyamat futása közben.

Fizikai cím (physical address): egy adott memória elem címe, ahogy az a fizikai memória buszon megadásra kerül a memória vezérlő által.

Logikai/Fizikai címtartomány: egy adott folyamathoz tartozó logikai/fizikai címek összessége

Logikai cím = Virtuális cím, ha a fizikai és logikai cím eltér, ebben az esetben a futás idejű leképezést az MMU valósítja meg.

Címkepzés: mikor történik meg a fizikai és logikai címek közötti leképezés?

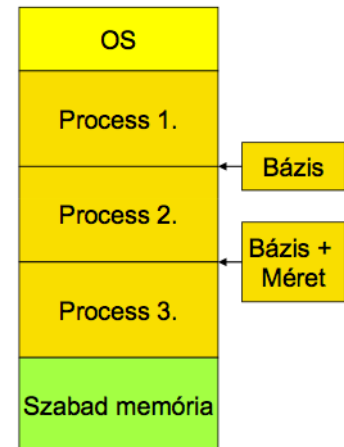
- fordítási idejű (compile/link time)
 - ismert a betöltés helye, abszolút címezés
 - a program fizikai címeket tartalmaz
- betöltési idejű (load time)
 - áthelyezhető kód
 - betöltés során oldódik fel a leképezés
 - a program fizikai címeket tartalmaz
- futás időben is változhat a leképezés:
 - folyamat számára tanszparens módon más fizikai címterületre kerülhet
 - speciális HW (MMU) szükséges ehhez
 - a folyamat nem látja (láthatja) hogy melyik fizikai címeken található a memória, amin elfér: mindig logikai/virtuális címeket használ
- a fizikai címekhez kötött logikai címek használata alapvető a modern OS-ek működése szempontjából
- dinamikus betöltés: bizonyos funkciót nem töltenek be amíg nem használjuk
 - gyors program indulás
 - alacsonyabb memória használat
 - a program feladata a dinamikus betöltés megvalósítása: OS nem tud róla

Dinamikus kapcsolatszerkesztés (dynamic linking)

- a dinamikus betöltés operációs rendszer támogatással
 - dynamically linked/loaded library (Windows *.dll)
 - shared object (UNIX/Linux *.so)
- a dinamikusan beszerkesztett programkönyvtárak több program számára is elérhetőek (code sharing)
- megvalósítás:
 - a program csak egy csokot (stub) tartalmaz
 - a csok feladata a dll/so megtalálása és betöltése az OS felhasználásával
 - egy adott funkciójú dll/so több eltérő verzióban is lehet a rendszerben

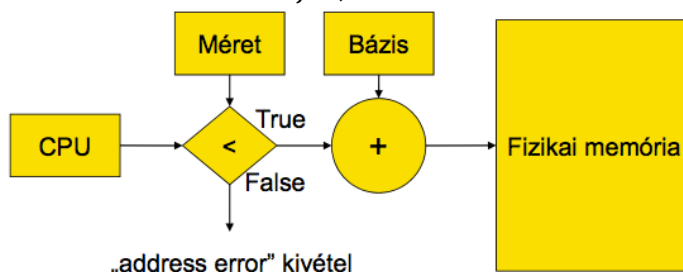
Alapmegoldás (változó mértű partíciók):

- minden egyes folyamat egy összefüggő, statikus, előre kiválasztott fizikai címtartományba kerül
 - báziscímtől kezdődik
 - adott mértű
 - bázisrelatív címzés
- a folyamatnak ebbe a memóriaterületbe kell beleférni élettartama során
- nem túl hatékony, de első iterációnak jó



Egyszerű megvalósítás:

- implementációhoz szükséges
 - folyamatonként 2 védett módban állítható regiszter
 - 1 összeadás és egy vizsgálat
- ha a folyamat kicímézi a rendelkezésre álló memóriából, akkor kivétel jelzés
 - a CPU-nak küldjük, és azt a CPU kernel módban kezeli



Problémák:

- külső tördelődés: Process 2 kilép, a helye felszabadul, helyére egy kisebb Process 4 töltődik, Process 4 és Process 3 közötti memória terület nem használható: nem fér bele folyamat, elveszik az OS számára
- belső tördelődés: ugyanaz mint a külső, csak P. 4 az egész területet megkapja
- a programok által igényelt memória dinamikusan változik, nehéz felső korlátot adni
- a programok rendelkeznek az alábbi tulajdonságokkal:
 - a programkód kis része használja az adatok kis részét nagyrészt
 - egyes részek soha nem kerülnek végrehajtásra: dinamikus kapcsolatszerkesztés, csak a ténylegesen használt kódot töltjük be

Foglalási stratégiák:

- *First fit*: tár elejéről indulva az első elégséges méretű területet foglalja le
- *Next fit*: legutolsó allokált területtől kezdi a keresést
- *Best fit*: legkisebb szabad hellyel járó helyre tesszük be, minimális külső tördeléses
- *Worst fit*: a legnagyobb szabad hellyel járó helyre tesszük be, talán a maradék helyre még fér be valami

Foglalási stratégiák értékelése:

- külső tördelést befolyásolják
- "First/Next/Best fit" kb a teljes memória terület 33%-át, a használt memória 50%-át nem tudja allokálni, az kihasználatlanul marad
- a "worst fit" algoritmus esetén a teljes memória 50%-a kihasználatlan marad
- "first/next fit" a legkevésbé erőforrás igényes, másik 2 teljes listát végignézni

Számolási feladatok: (3.1)

<http://home.mit.bme.hu/~micskeiz/opre/files/opre-feladatok-segedanyag.pdf>

Szabad helyek tömörítése (Compaction, Garbage collection):

- megoldja a külső és belső tördelést
 - átrendezi a futó folyamatokat az optimális memória használat elérésére
 - egy, összefüggő szabad területet hoz létre
- nagyon erőforrás igényes

További lehetőségek:

- fizikai memória kezelést általánosabban, hatékonyabban kell megvalósítani
- tárcsere (swapping)
- szegmens szervezés
- lapcsere

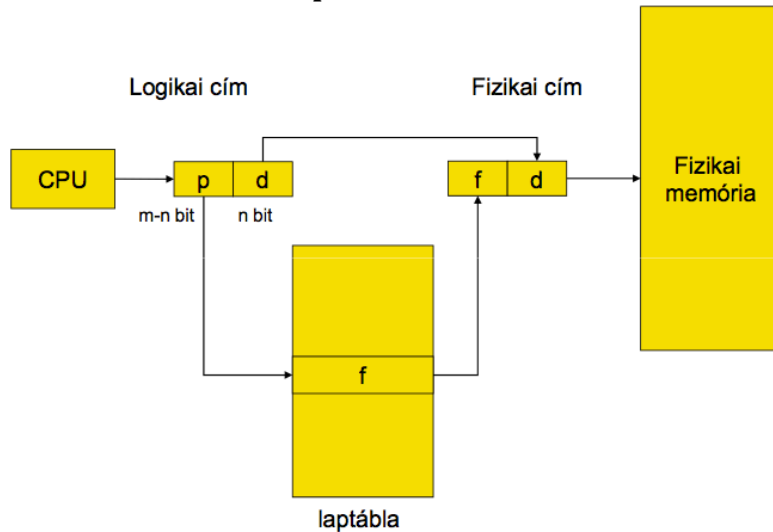
Tárcsere (swapping):

- a teljes folyamat háttértárolóra is kiírható
- ha nincs futó állapotban és nincs folyamatban lévő I/O művelet (kivéve az OS területén lefoglalt pufferekbe/pufferekből végzett I/O)
- ha a címképzés futási időben történik: akár még más fizikai címre is kerülhet a visszatöltött folyamat
- a tárcserével összekapcsolt kontextus váltás nagyon időigényes: a teljes folyamatot ki kell írni, majd visszatölteni

Lapszervezés (paging):

- a folyamathoz tartozó fizikai memória terület lehet nem folytoson is
- a fizikai memóriát *keretekre (frame)* osztjuk
- a logikai memóriát *lapokra (page)* osztjuk
- minden egyes logikai címet két részre osztunk: lap szám (p), lap eltolás (d)
- a lap szám a laptábla indexelésére szolgál: a laptábla az egyes lapokhoz tartozó fizikai báziscímet tárolja (f)
- a keret tábla tartja nyilván az üres kereteket

Címtranszformáció laptáblán:

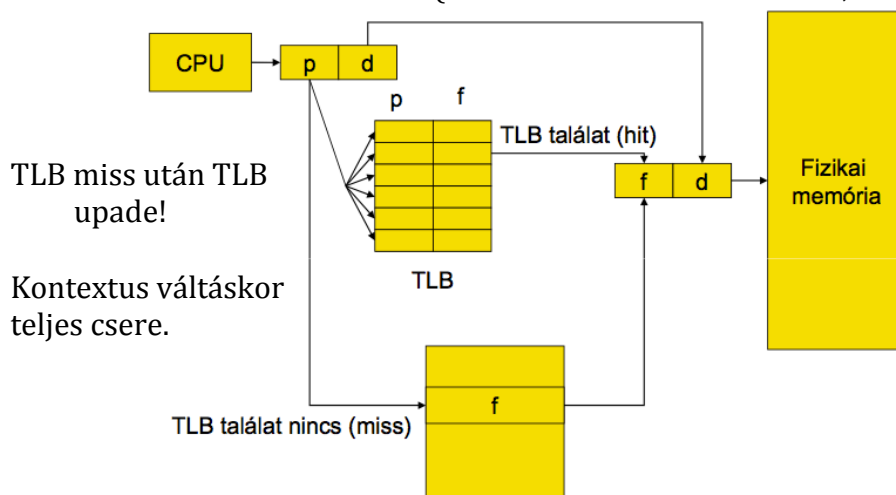


Lapszervezés tulajdonságai:

- a leképezést hardver végzi
- szeparáció megvalósul (folyamat létrehozható)
 - a folyamat által látott logikai címtartomány, és a tényleges használt fizikai címtartomány teljesen elkülönülnek
 - a folyamatok nem férnek hozzá egymás lapjaihoz
 - az OS kezelheti a lap- és kerettáblát
- külső tördelés nincs, belső tördelés átlagosan fél lapnyi
- a modern gépekben sok memória van -> nagy lapméret lenne jó

Keresés a laptáblában:

- a lap- és kerettábla nagy lehet a modern OS-ekben
 - 4KB-os lapok, 32 bites címzés -> 2^{20} bejegyzés van -> min 4MB memóra
 - többféle lapméret támogatás
- megoldás:
 - többszintű laptáblák (hierarchical paging)
 - hash-elt laptáblák (hashed page table)
 - inverted page table
- a keresés a laptáblában lassú:
 - 2x annyi idő (laptábla indexelés, olvasás majd adatelérés)
 - asszociatív lekérdezés (Translation look-aside buffer, TLB):



TLB hatása:

- a kérdés a találati arány (hit ratio):
 - a TLB méretétől és a végrehajtott kódtól függ
 - tipikusan 80-90% körül van
- hatása a teljesítményre: ADM Phenom
 - a TLB hibás, a BIOS-ból kikapcsolható, eltérés: 20% teljes, 14% alkalmazás, memória benchmarkok esetén akár 50% is

Egyéb információk a laptáblában:

- kiegészítő bitek a laptáblában:
 - valid/invalid bit: benne van-e az adott lap a fizikai memóriában
 - read/read-write bit, execute bit: végrehajtható memória műveletek
 - referenced/used bit: memóriaművelet esetén az MMU beállítja, Second Chance algoritmus használja, használat esetén beállítódik
- a HW(MMU) beállítja/ellenőrzi, ha a szabályokkal ellentétes használatot talál, akkor kivételt generál, amit az OS kezel
- az OS is felhasználja ezeket

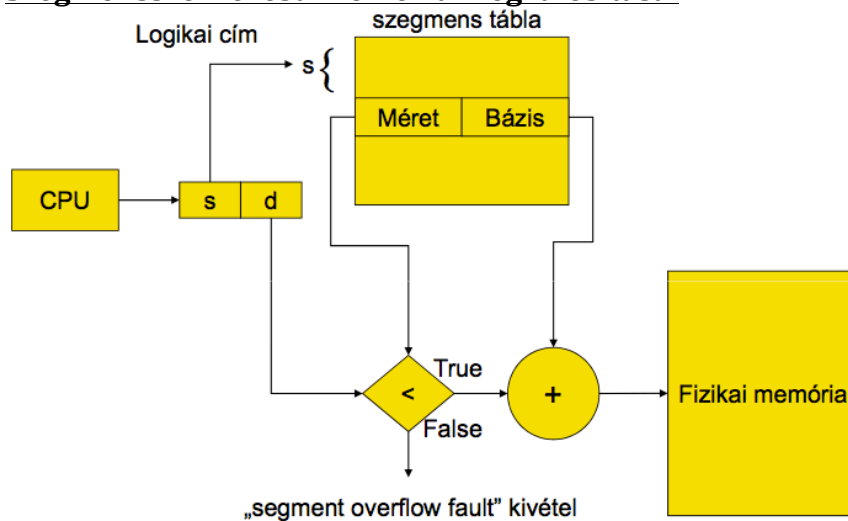
Lapok megosztása:

- a folyamatok memória tekintetében szeparálva vannak egymástól
- de teljesítmény okokból nem teljes a szeparáció
- OS által kontrollált módon megosztható memória:
 - közös kód lapok
 - Copy on Write (COW):
 - szülő-gyerek kapcsolatokban lévő folyamatok esetén
 - a két folyamat először egy közös fizikai lapot használ
 - ha azt bármelyik írni próbálja, akkor a gyerekek létrejön egy saját lapja, az írási kísérlet előtti tartalommal
 - közösen olvashatóak és írhatóak (UNIX System V Shared Memory)
- HW (MMU) támogatás szükséges a megvalósításhoz

Szegmensszervezésű memória:

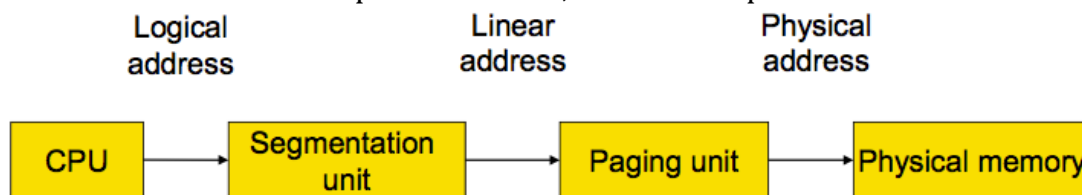
- a programozó:
 - adat, kód, bizonyos programkönyvtárak, stack, heap, stb.
 - nem egy összefüggő lineáris memória területben gondolkodik
- a szegmensszervezésű memória ennek az elképzelésnek felel meg:
 - a logikai címtartomány szegmensekre van osztva
 - a programozó egy szegmenst, azon belül egy szegmens offsetet ad meg
 - a lapozásnál egy címet ad meg, és azt a HW bontja ketté
 - itt két részt ad meg, és azt a HW rakja össze
- a szegmensek mérete adott, azt is tárolni kell
 - adott szegmensen kívüli címzés esetén "segment overflow fault" kivétel
- a szegmensek folytonosan tárolhatóak, belső tördelődés nincs
- a szegmenseket a fordító és a linker állítja össze, és adja meg a programot betöltő loader-nek

Szegmensszervezésű memória megvalósítása:



Szegmens és lapszervezés együtt:

- egyes rendszerek mint a kettőt támogatják
- a szegmensszervezést minimálisan használják a x86 HW-t támogató OS-ek
- a lapozás viszont alapvető CPU szolgáltatás
 - x86 HW esetén 4KB (2 szintű tábla) és 4MB-es lapok (1 szintű tábla)
 - a Linux 3 szintű laptáblát használ, ebből a középső üres x86 HW esetén



Virtuális tárkezelés (Virtual memory):

- logikai cím != fizikai cím (nem erről lesz szó)
- össze szokták keverni a swapping-gel (nem erről lesz szó)
- a korábban ismert memória menedzsment módszere alapján kidolgozott komplex memória menedzsment megoldás a virtuális tárkezelés

Alapgondolatok, megfigyelések:

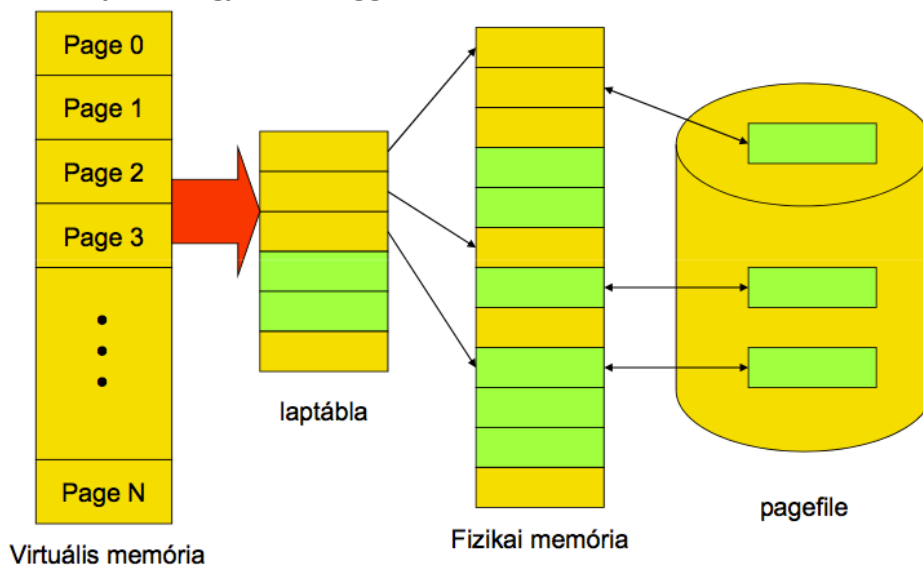
- a folyamatok fizikai memóriában történő végrehajtása:
 - szükségesnek és célszerűnek tűnik
 - komoly és kellemetlen következményekkel jár
1. A teljes folyamat nem szükséges annak végrehajtásához, többnyire az aktuális utasítás számláló "környezete", és az éppen használt adatszerkezetek elégségesek (lokalitás)
 2. A folyamatok kódrészleteinek nagy részét soha nem hajtjuk végre vagy nagyon ritkán hajtjuk végre (error handling, software/feature bloat)
 3. A folyamatok elindulásához nem szükséges a teljes program betöltése
 4. Egyes kódrészletek, erőforrások megoszthatóak a folyamatok között
 5. A párhuzamosan futó folyamatok egyes, már használt kódrészleteire sokáig vagy akár soha többé nincs szükségük, míg más folyamatoknak nem elég a memória

Elvárások:

- jó lenne nagyobb memóriát használó folyamatokat futtatni, mint amennyi fizikai memória rendelkezésre áll
 - virtuális memória, a programozónak nem kell foglalkozni a rendelkezésre álló memóriával
 - architektúra függő határig lehetséges
 - következmények: komplexitás és sebesség
- ha a folyamataink csak a tényleges szükséges fizikai memóriát tartják a fizikai memóriában, több folyamatot tudunk egy időben betölteni
- a programok gyorsabban induljanak el, csak a szükséges kódrészleteket töltsse be az OS
- képes legyen osztozni a közös kód és adatszerkezeten, erőforrásokon

Virtuális tárkezelés:

- az alapjai a lapozás: a folytonos virtuális címteret egy tábla (memory map, page table, laptábla) képezi le
- nem direkt módon fizikai memóriára történik leképzés, hanem:
 - részben fizikai memóriára
 - részben a permanens táron (HDD, Flash tár) kialakított speciális terület
 - pagefile (windows), swap file (UNIX/Linux)
- a folyamat egy összefüggő virtuális címteret lát



Egyéb tulajdonságok:

- folyamatok megoszthatnak memóriaterületeket olvasásra/írásra
 - hozzáférési jogosultságok
- az ilyen memória területek több folyamat virtuális címtartományába vannak belapozva
- módosítás nyilvántartása (modified/dirty bit)
 - minden laphoz tartozik egy HW által kezelt bit: betöltéskor törlik, módosításkor beállítódik
- hivatkozások nyilvántartása (referenced/used bit)
 - OS adott időnként és/vagy adott eseményekre törli
 - használat esetén beállítják

Következmények:

- a folyamat virtuálisan összefüggő memóriát lát (virtuális memória)
- valójában az összefüggő memóriaterület ritka (sparse address space)
 - nagy része mögött nincs fizikai memória vagy pagefile bejegyzés
 - ha az szükséges, akkor dinamikusan kerül mögé memória, amit a folyamat elérhet

Működés:

- ha egy éppen használt virtuális memória lapon található cím bent van a fizikai memóriában, akkor a kód vegrehajtható
- ha nincs bent (valid/invalid bit)
 - laphiba ("page fault") kivételt generált az MMU
 - érvényes, de éppen nem fizikai memóriában lévő lap
 - nem hiba, normális működés
 - az OS ezt kezeli, lehetőségek:
 - HDD-re ki van írva: be kell hozni a pagefile-ból
 - soha nem lett betöltve: be kell tölteni
 - kérdés: hová, főleg ha tele van a fizikai memória?
 - az OS visszaadhatja a vezérlést a megszakított folyamatnak
 - a hozzárendelés során a folyamat passzívan várkozik

Lapozási stratégiák (fetch strategy):

- igény szerinti lapozás (demand paging)
 - csak laphiba esetén, és csak a laphibát megszüntető lapot hozza be a fizikai memóriába
 - csak a szükséges lapok vannak a fizikai memóriában
 - új lapra vonatkozó hivatkozás mindig hosszú várkozást eredményez (be kell hozni)
- előretekintő lapozás (anticipatory paging)
 - előre tekintve (becslés) az OS megpróbálja kitalálni, hogy mely lapokra lesz szükség, és azokat is behozza
 - feltétel: szabad erőforrások (CPU, HDD, fizikai memória)
 - ha a behozott lapokra tényleg szükség lesz, akkor csökken a laphibák száma
 - ha nem, akkor feleslegesen használtunk erőforrást

Lapcsere stratégiák (replacement strategies):

- tele van a fizikai memória, és laphiba történik
 - ki kell válsztani a permanens tárra kikerülő lapot
 - ennek a helyére kerül be a lap a permanens tárról
- algoritmusok:
 - optimális algoritmus
 - legrégebbi lap (FIFO)
 - újabb esély algoritmus (second chance)
 - legrégebben nem használt (LRU, last recently used)
 - legkevésbé használt (LFU, last frequently used)
 - utóbbi időben nem használt (NRU, not recently used)

Optimális algoritmus:

- előre néz, és teljes információval rendelkezik a jövőben használt lapokról
- nem realizálható, de jó összehasonlítási alap

FIFO:

- a behozott lapokat egy FIFO-ba rendezi, és mindig a legrégebben behozott lapot cseréli
- egyszerű, a múlt alapján dönt, hátranézve
- azokat a lapokat is cseréli amiket a folyamat gyakran használ
- Bélády anomália: több folyamat, több laphiba

Second chance algoritmus:

- a sor elején lévő lapot csak akkor cseréli le, ha arra nem hivatkoztak: used bit
- a used bitet (referenced bitet) az MMU állítja be, ha a lapot használták
- a used bitet törli, ha az be van állítva:
 - ezért újabb esély a neve
 - ezután a lapot a sor végére rakja
 - egyébként végtelen ciklusba kerülhetne
- bonyolultabb mint a FIFO, de jobb algoritmus
- hátra néz, és a behozás sorrendje, és a használat alapján dönt
 - a program lokalitását figyelembe veszi

Legrégebben nem használt (LRU):

- bonyolult, de jól közelíti az optimális algoritmust: a lokalitás miatt jó a közelítés
- hátrafelé néz
- megvalósítása:
 - számláló: minden laphoz egy "last used" timespamot kerül
 - láncolt lista: a lista végére kerül a legutoljára használt
 - kétdimenziós tömb: NxN-es mátrix, N a lapok száma
- sokszor a közelítéseit szokták számolni

Legkevésbé használt (LFU):

- a közelmúltban gyakran használt lapokat a lokalitás miatt nagy valószínűséggel újra fogjuk használni
- a ritkán használtakat kis valószínűséggel fogjuk használni
- az R bit értékét hozzáadja időnként egy laphoz tartozó számláléhoz, és törli az R bitet
- a kisebb számláló értéket tartalmazó lapokat cseréljük le
- az algoritmus nem felejt
- az algoritmus a frissen behozott lapokra fogja lecserélni - azokat be kell fagyasztani a memóriába kis időre

Utóbbi időbe nem használt (NRU):

- a hivatkozott és módosított biteket is használja
- R törölhető, viszont M-et meg kell hagyni
- prioritást rendel a lapokhoz M és R alapján:
 - 0. prioritás : M=0, R=0 (legalacsonyabb)
 - 1. prioritás: R=0, M=1
 - 2. prioritás: R=1, M=0
 - 3. prioritás: R=1, M=1 (legmagasabb)
- mindig a legkisebb prioritású csoportból választ, ahol még van lap

Lapcsere szempontok:

- globális csere: a teljes fizikai memória potenciálisan cserélhető
- lokális csere: a folyamat által használt fizikai memória lapok között történik a csere
- lapok tárba fagyasztása (lock bit):
 - I/O műveletek hivatkoznak rá
 - ott fizikai címet kell használnunk
 - lehet kernel szinten pufferelni, és akkor ez is megkerülhető
- LFU algoritmus frissen behozott lapjai

Számítási feladat: (3.2)

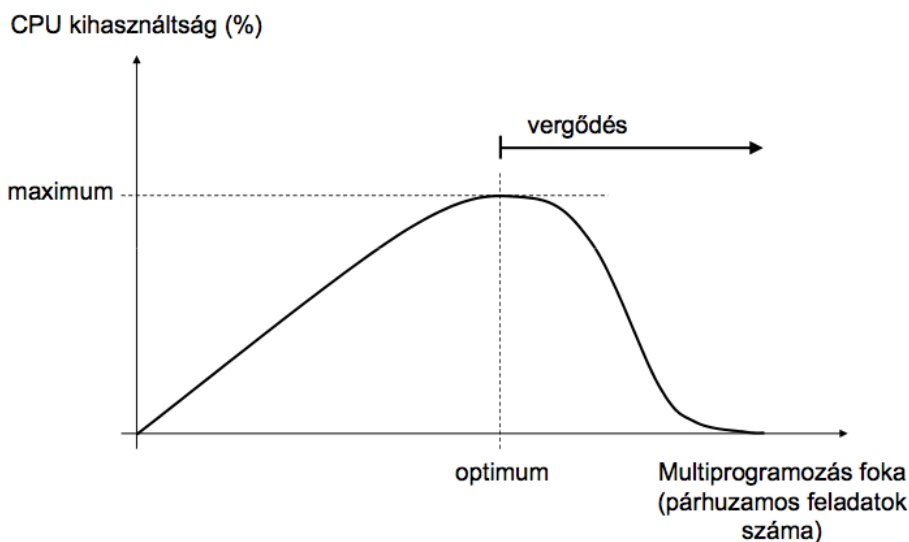
<http://home.mit.bme.hu/~micskeiz/opre/files/opre-feladatok-segedanyag.pdf>

Virtuális memória teljesítménye:

- fizikai memória sebessége:
 - n*1GB/s adatátviteli sebesség
 - n*10ns késleltetés
 - ha cache-elve van, akkor még gyorsabb
- permanens tároló sebessége:
 - tipikusan 100MB/s adatátviteli sebesség, de random elérés esetén és sok párhuzamos felhasználó esetén sokkal lassabb
 - 10 ms legrosszabb hozzáférési idő (fejmozgás)
 - Flash esetén az írás (törlés) lassú és problémás, hamar tönkremegy
 - több nagyságrend a különbség
- ha egy lap nincs bent a fizikai memóriában:
 - több nagyságrenddel lassabb a permanens tárolón lévő lapok elérés, mint egy fizikai memóriában lévő lap elérése
 - az elérhető sebességet alapvetően a laphibák gyakorisága befolyásolja
 - gyakori laphiba => teljesítmény drasztikusan csökken
 - sikeres előretekintő lapozás javíthat a teljesítményen

Vergődés:

- hogyan válasszuk meg, hogy egy folyamathoz hány memória keretet rendeljünk a fizikai memóriából?
 - kevés: nagyszámú/állandó laphiba (trashing)
 - sok: más feladatnak nem marad memória
- DEF: a gyakori laphibák által okozott rendszer teljesítmény csökkenését vergődésnek (trashing) nevezzük
 - a laphiba kezelése során újabb laphiba jelenik meg
 - a laphibák felgyűlnek a háttértár várkozási sorában
 - a CPU a laphibák megoldására vár
 - a hosszú távú ütemező (ha van), ezt I/O intenzív folyamatként is értelmezheti, újabb folyamatokat beengedve a rendszerbe
- ábra:



Vergődés elkerülése:

- cél: alacsonyabb laphiba gyakoriság (PFF, Page Fault Frequency)
- egy laphiba kezelése során ne jöjjön létre újabb laphiba: a háttértár várakozási sora csak csökkenhet
- lokális lapcsere stratégia:
 - a folyamatok nem tudják egymástól elvenni a fizikai memória keretet
 - nem terjedhet át a hatás más feladatokra
 - csökkenti a problémát, de nem oldja meg
- hány keretre van szükség a fizikai memóriában egy folyamatnak a hatékony futáshoz?

Lokalitás:

- statisztika: egy időintervallumban a folyamatok a címtartományuk csak egy kis részét használják: időbeli / térbeli
- vergődés: megfelelő számú fizikai memória keret allokálása
 - nincs vergődés
 - laphibák a lokalitás váltásakor

Munkahalmaz (Working-set):

- a lokalitáson alapul
- a folyamat azon lapjainak halmaza, amelyekre egy időintervallumban a folyamat hivatkozik
- a munkahalmaz alapján megadható az adott folyamat munkahalmaz mérete (WSS)
- a futó folyamatok teljes fizikai keret memória igénye kiszámolható (D)
$$D = \text{SUM}(WSS_i)$$

Alkalmazása:

- az OS méri a WSS-t folyamatonként
 - ha van szabad memória keret:
 - akkor az igények kielégíthetőek
 - új folyamatok engedhetőek be a rendszerbe
 - ha nincs szabad memóriakeret:
 - akkor ki kell választani egy "áldozat" folyamatot
 - azt fel kell függeszteni
 - az áldozat folyamat fizikai memória kereteit fel lehet használni
- a vergődés elkerülhető a multiprogramozás fokát közel optimálisan tartva
- a gyakorlatban erőforrás igényes a megvalósítása

PFF alapú optimalizáció:

- vergődés: magas PFF érték
- folyamatonként egyszerűen mérhető PFF érték
 - alacsony: túl sok fizikai memória keret
 - magas: túl kevés fizikai memória keret
- felső és alsó PFF határértékek megadása
 - felső határértéket túllépi: kap egy fizikai memória keretet
 - alsó határértéket túllépi: egy fizikai memória keret elvonása

Beágyazott rendszerek:

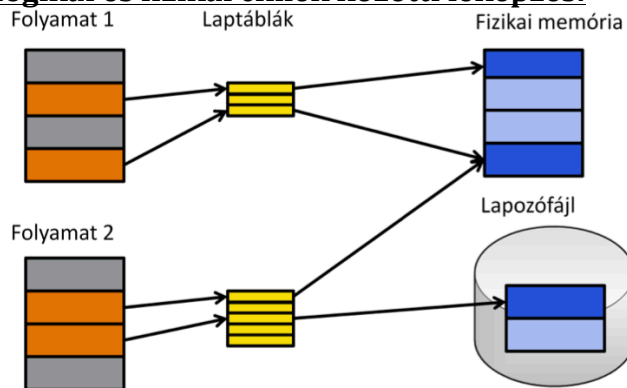
- nagyon eltérő a memóriakezelés
- a beágyazott rendszerek CPU-inak egy részében nincs MMU:
 - lapozás, virtuális memória kezelés, szeparáció, stb ki van zárva
- ha van MMU, és használják:
 - a háttértár korlátos, a pagefile-nak nincs hely sem
 - elsősorban a feladatok szeparációja és virtualizáció a feladat

18. Memória kezelés a Windowsban

A Windows memóriakezelésének alapelvei:

- virtuális tárkezelés
 - lapszervezés (4KB/2KB mértű lapok, x86/x86+PAE/x64:2/3/4 szintű)
 - lapozófájl használata
- hatékonyság
 - igény szerinti lapozás + lustering + prefetch
 - lapozás: kezdetben igény szerinti lapozás
 - clustering: a kért lap környékén lévő lapokat is behozza (lokális elv)
 - prefetch: rögzíti, hogy a programok induláskor miket szoktak igényelni, és azokat előre behozza
 - memória megosztás, copy-on-write
 - fájl cachelés memóriában (memory mapped file)
- biztonság
 - minden folyamatnak külön címtartomány
 - elérés leírókon keresztül (hozzáférési token)
- x86 (32 bites rendszer)
 - maximum 4 GB fizikai memória, de néha kevesebbet lát az OS
 - alapesetben egy felhasználó folyamat maximum 2 GB-os címtartományt használhat fel, a címtartomány másik részén a rendszer memória van
 - boot.ini-ben /3GB kapcsolóval 3GB-ra növelhető (kényszer megoldás)
 - Physical Address Extension (PAE): 32 helyett 36 címbit van, akár 64 GB memóriát is képes így kezelni
- x64 (64 bites rendszer)
 - lényegesen nagyobb memória: max. 8/192 GB (Win 7 basic/prof)
 - 8GB felhasználói folyamat tartomány / 6657 GB rendszer tartomány
- Virtual Address Space (VAS)
 - felhasználó címtartomány:
 - egyedi minden folyamatra
 - a futó alkalmazás (.exe és .dll-ek)
 - felhasználói módú verem minden szálnak
 - alkalmazás adatstruktúrái
 - rendszer tartomány:
 - rendszerszinten közös
 - executive, kernel és a HAL
 - rendszerszintű adatstruktúrák
 - laptáblák (virtuális -> fizikai leképezés, folyamatonként különböző)
 - executive heap-ek (pools)
 - védett módú eszköz meghajtók
 - védett módú verem minden folyamat minden szálnak
- folyamatok memóriafoglalása (két lépésben)
 - reserve: virtuális címtartomány lefoglalása
 - commit: virtuális memória lefoglalása
 - előny: csak annyit foglal, amennyi ténylegesen kell a folyamatnak

Logikai és fizikai címek közötti leképezés:



x86 PAE címfordítás:

- hiába van PAE, a folyamatunk továbbra is csak 2^{20} fizikai memória keretet tud megcímezni, így a 12 bites offsettel együtt is 4 GB memóriát kezel. A különbség ott van a PAE-nél, hogy a fizikai memórialap címezésére 24 bitet használunk -> 64 GB memóriát tud kezelni

x64 PAE esetén a PTE:

- 64 bites, 24 bit a lap címének
- flagek: P(present), A(access), D(dirty), U/S(user/system), W/R(write/read)

Munakészlet (Working set)

- egy folyathoz tartozó fizikai memóriában lévő lapok
- ezeket éri el laphiba nélkül
 - *soft laphiba (soft page fault)*: memóriában volt még a lap
 - *hard laphiba (hard page fault)*: lemeztől kell beolvasni

Working set limit:

- ennyi fizikai memóriát birtokolhat egyszerre
- ha eléri, lapcsere kell
- ha a szabad memória lecsökken: *trimming* (egy rendszer szál megvizsgálja a folyamatot, és akinek sok fizikai memórialapja van vagy régóta nem használta őket, azoktól elvesz párat)

Lapozófájl (page file):

- csak a módosított adat kerül bele, kód nem, ha:
 - van szabad memória
 - folyamatok nem foglalhatnak bármennyi memóriát
 - tartalék az új/többi folyamatnak
- meghajtóként egy darab, ajánlott nem a rendszerlemezre rakni, de maradjon egy kicsit ott is a memory dumpnak
- ajánlott méret: 1 vagy 1,5-szer a fizikai memória

Optimalizáció:

- *Prefetch (XP)*: első tíz másodperc lapbetöltéseit megjegyzi, következő induláskor hivatkozott lapok betöltése aszinkron módon
- *Superfetch (Vista)*: 8 prioritás a memórialapokhoz, lapok használatának követése, memória felhasználása esetén lassan visszahoz lapokat a standby listára, amik kellenek még

19. Virtualizáció

Virtualizáció alkalmazásai:

- vékony kliensek
- egygépes termékek
- OS szintű virtualizáció
- alkalmazás becsomagolás
- teljes számítógép virtualizálás
- tárolórendszer felépítésének elfedése
- dinamikus adatközpont, életciklus kezelés, konvertálás, telepítés

Virtuális gép osztályozása:

- *System VMs* : VM csak egy hardvert lát
- *Process VMs*: VM egy ABI-t (Application Binary Interface) lát

Platform virtualizáció:

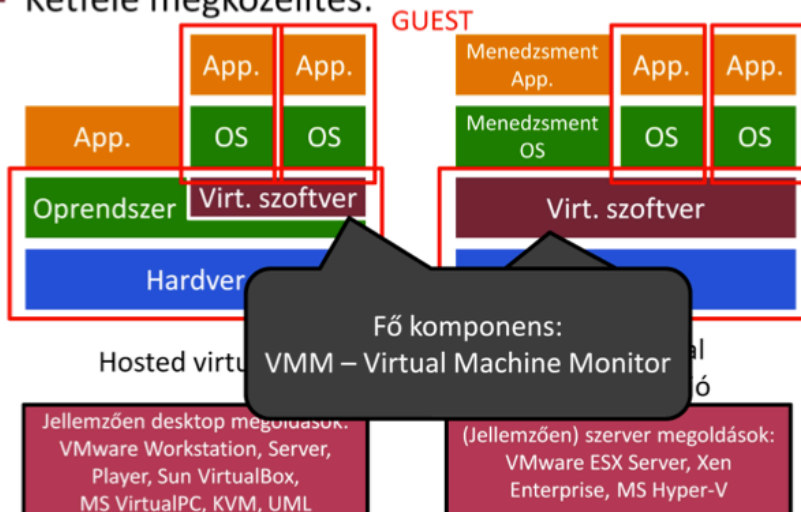
- teljes számítógép virtualizálása, egy gépen több OS futhat
- elemek:
 - gazda gép (host machine) = fizikai gép
 - vendég gép (guest machine) = virtuális gép
 - Virtual Machine Monitor (VMM) = virtuális gépeket kezelő program

Miért jó a platform virtualizáció?

- tesztrendszer kiépítése
- alacsonyabb kihasználtságú szervereket, amiket a rajtuk lévő egymással nem kompatibilis alkalmazások miatt nem lehet egy fizikai gépre rakni, összevonja egy gépre
- Régi rendszer, pl.: DOS, régi UNIX
- dinamikus adatközpont
- rendelkezésre állás, katasztrófa védelem
- hordozható alkalmazások
- új terület: mobil virtualizáció

Platform virtualizáció:

▪ Kétféle megközelítés:



Hosted virtual/bare-metal virtualizáció:

- hosted virtual
 - asztali számítógépeknél jellemzően
 - az operációs rendszeren fut a VMM
- bare metal
 - egy hypervisor feladata, hogy hardver erőforrások kiosztását elvégezze

1. Elméleti alapok, követelmények egy virtualizációs megoldástól:

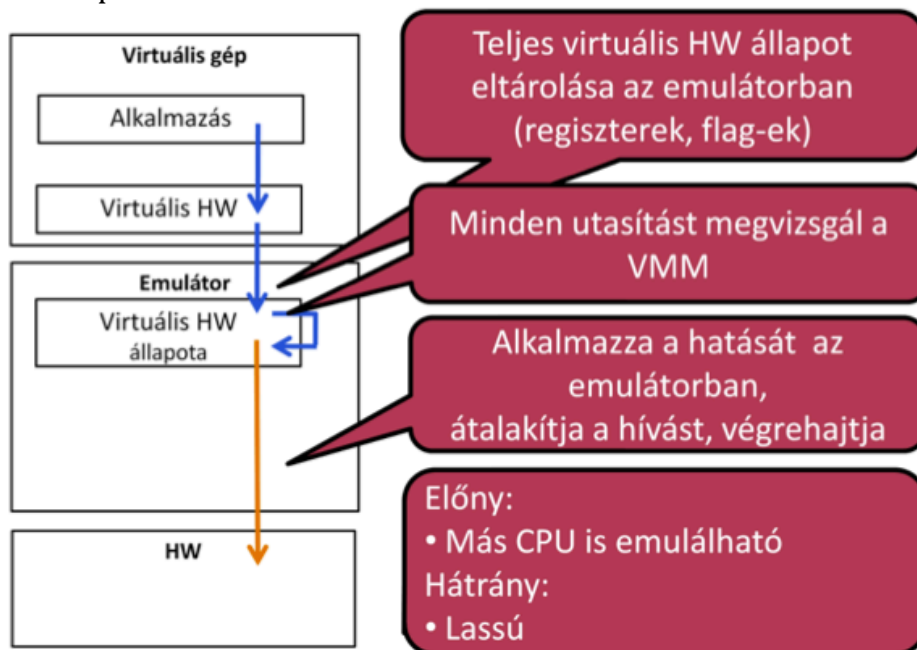
- azonosság: virtuális gépen futtatott programok ugyanazt az eredményt adják (Popek és Goldberg féle követelmény)
- biztonságosság: a VMM kezeli az össze hardver erőforrást
- hatékonyság: a vendég gép utasításainak nagy része beleavatkozás nélkül fut

Alapvető probléma:

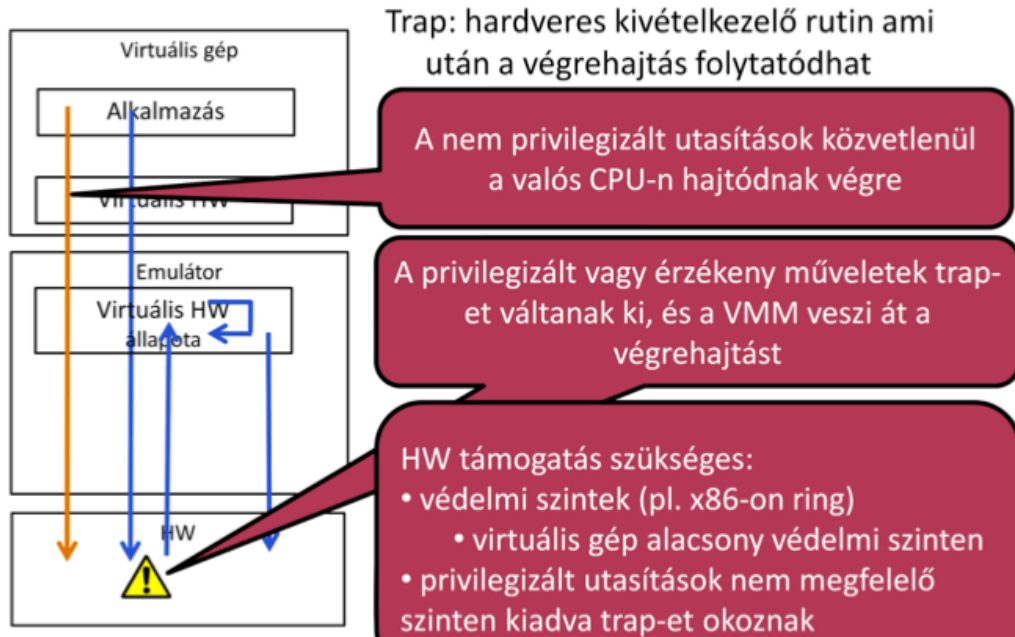
- vendég gépektől védeni kell a rendszert
- megoldás: VMM felügyeli a vendég utasításait
 - privilegizált (érzékeny regisztert/flaget módosít) utasítások kezelése

2. Elméleti alapok, CPU virtualizáció:

- alapvető módszerek: **tiszta emuláció**



- alapvető módszerek: **trap and emulate**
 - processzor támogatás is szükséges hozzá
 - privilegizált műveleteknek trap-et kell kiváltaniuk



x86 virtualizációk korlátai:

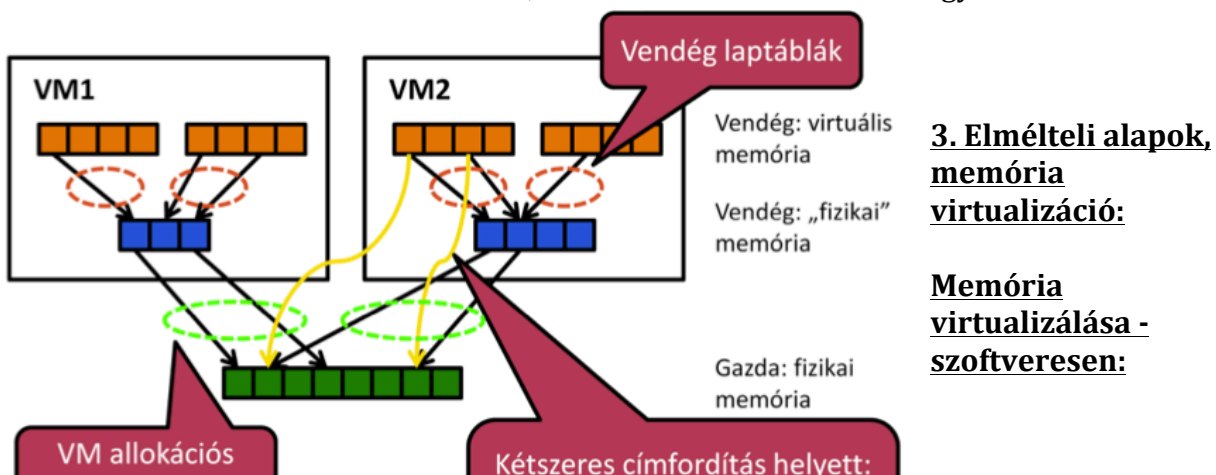
- kb. 250 utasításból 17 megsérti a klasszikus feltételeket
- következmény: nem használható a trap&emulate módszer klasszikus x86-on

Megoldások az x86 CPU virtualizációra:

- binary transation (szoftveres)
 - utasítások nagy része közvetlenül fut
 - privilegizált utasítások átírása futás közben, ezeket eltárolja
 - nem igényel forráskódot
 - vendég OS nem tud arról, hogy virtualizált
- paravirtualizáció
 - vendég OS forrásának módosítás
 - problémás utasítások lecserélése
 - hypercall: CMM-et hívja közvetlen
- hardveres virtualizáció
 - HW támogatás: root mode, VMCS
 - működik a trap&emulate módszer

Melyik a legjobb/leggyorsabb módszer?

- folyamatosan változik
- összemosódnak a határok, több módszert használnak vegyesen



Kétszeres cím-fordítás kell: kétféle laptáb-lákat kell fenntarta- ni. A régebbi hardve-rekben csak egy le-képzés támogatás van, fenntartanak egy harmadik féle struk-túrát is (árnyék lap-táblák). VMM-nek gondoskodnia kell ar-ról, hogy az árnyék laptáblák szinkron-
ban legyen a vendég
laptáblákkal.

Memória virtualizálás - paravirtualizáció:

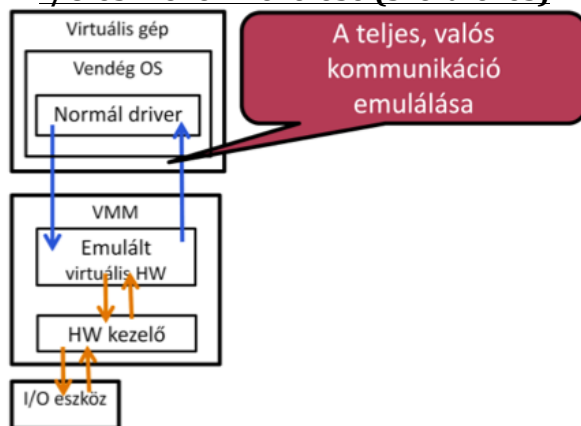
- árnyék laptáblák
- vendég OS forrásának módosítása
- ha a vendég módosítja a laptábláit, akkor értesítse a VMM-t is erről

Memória virtualizálás - harveres:

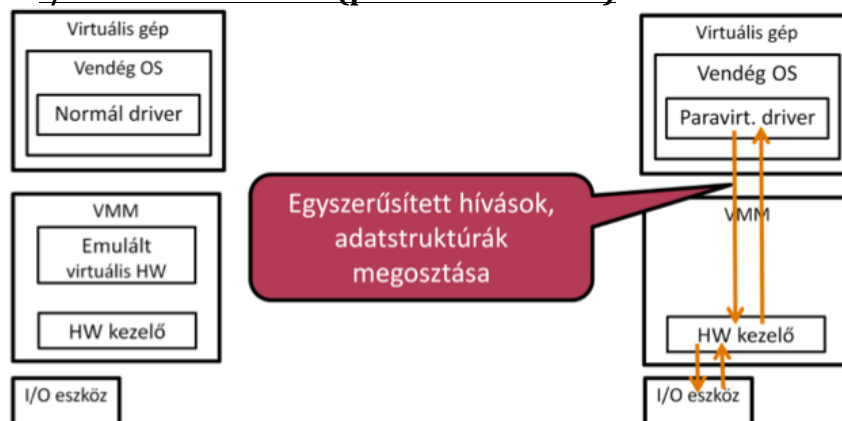
- HW támogatás az újabb CPU-kban (AMD Rapid Virtualization Indexing, Intel Extended Page tables)
- beágyazott laptábla (Nested page table)
 - vendég fizikai -> gazda fizikai leképezés eltárolása
 - cím leképezési rutin ezt is bejárja
- TLB bejegyzések azonosítóval ellátása
- nagy teljesítménynövekedés

4. Elméleti alapok, I/O virtualizáció:

I/O eszközök kezelése (szoftveres)



I/O eszköz kezelése (paravirtualizáció)



- speciális csomag telepítése a vendégben
- VMware Tools, Virtual PC Additions
- mindig telepítsük a vendég gépen!

I/O eszközök kezelése (hardveres)

- hardveres támogatás: Intel VT-d, AMD IOMMU
 - PCI szabvány kiegészítése: I/O Virtualization (IOV)
- I/O eszközök: megosztás VM-ek között, közvetlen hozzárendelés VM-hez

Cloud computing:

- amazon, windows azure, google apps, dropbox, rackspace, ...

Számítási felhő rétegei:

- *IaaS (Infrastructure as a Service)*
 - virtuális gépet kapunk
 - Amazon EC2, RackSpace, ...
- *PaaS (Platform as a Service)*
 - futtatható környezetet kapunk: java konténer, .NET adatbázis, ...
 - MS Azure, Google AppEngine, ...
- *SaaS (Software as a Service)*
 - szolgáltatást érünk el
 - Google Docs, Salesforce CRM, ...

Virtual Application:

- adott célra összeállított virtuális gép
- előny:
 - nincs telepítés, függőség
 - csak a feltétlen szükséges komponensek vannak telepítve
- JeOS: Just Enough Operation System

20. dTrace:

A feladat: hibakeresés, diagnosztika, optimalizáció

- végzetes/tranziens/percepcionális("lassú a gép") hibák okainak felderítése

Korábbi megoldások: gyenge megfigyelhetőség és invazív technikák

- egyszerű, nem túl rugalmas megfigyelő parancsok (vmstat, iostat, stb.)
- bináris vagy forráskódú beavatkozások szükségesek a hibakereséshez
- rendszerkönyvtár és kernel debug változatok használata szükséges
- erőszakosak, nehézkesek, időigényesek, korlátozottak, élő rendszeren nem alkalmazhatók, új hibákat vihetnek a rendszerbe

A DTrace összetevői:

- mérőrendszer: mérőpontok függvény be- és kilépési pontokon és adatfeldolgozók (fogyasztók)
- a mérőrendszer programozási nyelve (D)
- a megvalósító kernel modul

A mérési helyek (ún. provider-ekben):

- felhasználói függvények
- rendszerhívások
- kernel függvények
- összesen > 50 ezer mérőpont

Provider-ek:

- lista: dtrace -l
- fbt: kernel függvények (~45 ezer)
- syscall: rendszerhívások (~ 400 db mérőpont)
- I/O, processz, ütemezés, zárolás, stb.

Fogyasztók:

- DTrace parancs (szkriptek)
- programozási nyelvekben
- stb.

Működési mód:

- dinamikus mérőkód beszúrás a megfelelő fv. be/kilépési pontokra
- kernel szintű adatgyűjtés a meghatározott mérőpontokról
- a fogyasztók (pl. "dtrace" parancs) lekérdezik az adatgyűjtőt

A DTrace programozási nyelve: D

- C, awk, perl keverék, szokásos adattípusok
- speciális típusok: asszociatív tömb
- aggregációs műveletek, beépített változók (pid, ppid, execname, stb.)
- szálakra lokális változók: this->..., nanoszekundum felbontású időmérés
- hozzáférés a megfigyelt függvény argumentumához és visszatérési értékéhez
- megjelenítés: printf, pinta (asszociatív tömbökre)
- akciók (destruktívak is, pl. stop(), panic(), ha engedélyezzük)

20. A permanens tár kezelése

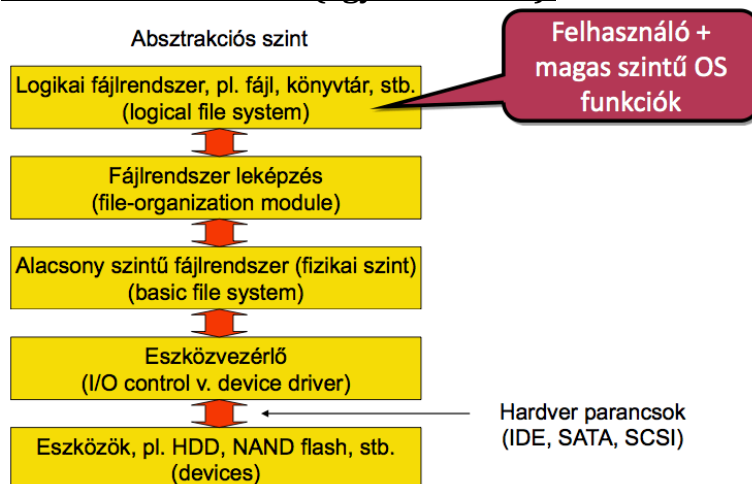
Permanens tár vagy háttértár:

- a központi memóriával összehasonlítható
- nagyságrendekkel nagyobb tároló terület
- nagyságrendekkel lassabb: adatátviteli sebesség, késleltetés
- nem felejtő tároló
- blokk alapú szervezés
 - az OS ebben kezeli, ennél kisebb egységekben nem gondolkodik
 - blokkonként olvasható, írható, törölhető
 - kivéve egyes beágyazott rendszereket
 - NOR flash memória szervezésű
 - NOR flash-ből direkt módon futtatható az OS és a programok

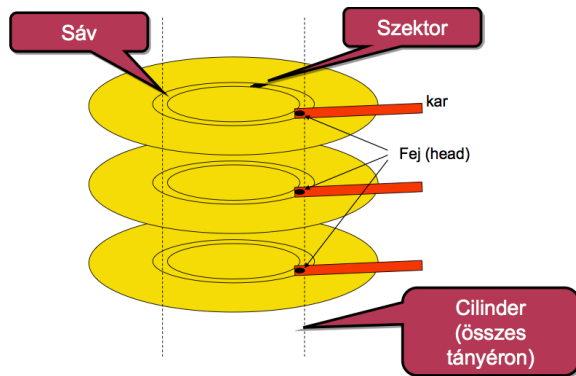
Fájl leképzése a HW tároló elemre:

- a fájl a permanens táron az adattárolás logikai egysége
 - névvel rendelkezik, a nevével lehet rá hivatkozni
- az OS feladata a logikai egységes (fájlok) leképzése valódi fizikai egységekre
- ezt az OS egy többszintű hierarchikus/réteges rendszer, különböző absztrakciós szintekkel oldja meg
- legalacsonyabb szinten többnyire valamilyen speciális HW van (HDD, SSD, ...)
 - kivétel az u.n. RAM drive
- a Linux esetén szinte az összes szabad fizikai memória keretet felhasználja a rendszer diszk cache-ként

Absztrakciós szintek (egyszerűsített):



HDD szerkezete:



HDD (merevlemez):

- forgó mágnesezhető tányérok (platter)
- egy forgatható karra (arm) vannak szerelve író/olvasó fejek (head)
- a tányérok cylinder (cylinder), és azokat sávokra (track) azokat meg szektorokra (sector) osztjuk
- a cylinder, sáv és szektor együtt azonosítja az írható/olvasható adatblokkot
- gyakorlatban már a fizikai eszközök egy logikai leképezést alkalmaznak (Logical Block Addressing, LBA: 48 vagy 64 bit napjainkban)
- 512B / 4KB szektor méret áttérés jelenleg folyik

A HDD tényleges sebessége:

- erősebb függ attól hogy éppen hol helyezkedik el a fej, és ahhoz képest az elérendő adatblokk (szektor), és milyen sebességgel forognak a tányérok
- több szintű optimalizáció:
 - diszk ütemezés (disk scheduling)
 - HDD szintjén (SATA NCQ, SCSI)
 - OS szintjén a párhuzamos írák/olvasások ütemezése során
 - prefetch
- több szintű cache:
 - HDD szintjén (16-64 MB jelenleg)
 - OS szintjén: disk cache, dinamikusan változó méret

NAND Flash tároló:

- alacsony szintű interface azonos a merevlemezével
 - Solid State Disk (SSD) SATA vagy IDE interfésszel
 - PEN drive USB interfésszel: kártya olvasók is így működnek, cserélhető tárolóval, más-más tároló foglalat
- olvasás gyors, független az adatot tároló blokk elhelyezkedésétől
 - nem kell fejeket mozgatni és szektort pozícióba forgatni
 - RAM jelleggel érthető el
- az írás (valójában a törlés) problémás
 - véges számú alkalommal törölhető egy blokk
- az írás/törlés lényegesen lassabb: párhuzamosítható több blokk írása gyorsíthat, ha az eszköz támogatja

Eszköz csatlakozás:

- a host számítógéphez csatlakozó megoldások (Host Attached Storage):
 - direkt csatlakozás: SATA/eSATA, IDE, SCSI, SAS, stb.
 - indirekt csatlakozás:

- USB, Firewire alapú alagút (tunnel)
- RAID (Redundant Array of Inexpensive Disks)
- hálózati tároló eszközök (Storage-Area Networks, SAN)
 - hálózati alagút (tunnel) a hoszt és a tároló eszköz között
 - speciális protokollok: Fibre channel
 - Ethernet és/vagy TCP/IP alapú: iSCSI, AoE
- A NAS (Network-Attached Storage, File megosztás, stb.) nem ezen a szinten valósulnak meg

USB (USB mass storage device class):

- SCSI transzparens parancs készlet kerül átküldésre az USB buszon
- az OS számára egy SCSI buszon keresztül csatlakozó eszköznek tűnik
- az USB csupán egy alagutat képez az eszköz és az OS között

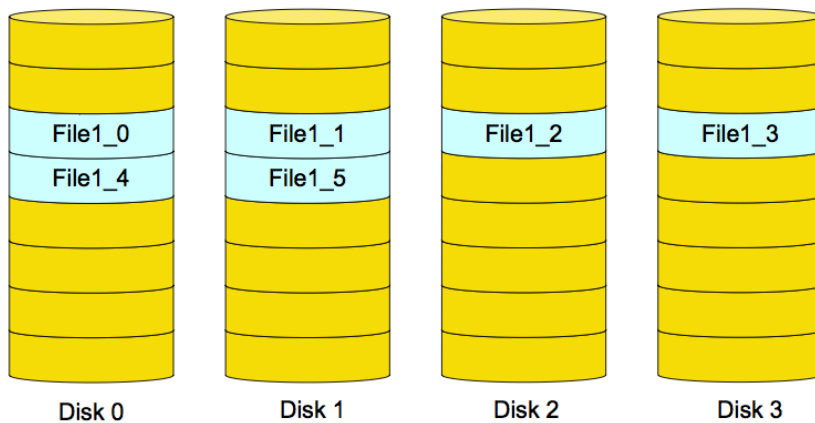
RAID:

- tények:
 - a merevlemezek olcsók
 - nem megbízhatóak (morgó alkatrészek, érzékenyek)
 - lassúak
- ötlet: használjuk belőle többet egyszerre
 - több redundáns alkalmazása növeli a megbízhatóságot
 - több párhuzamos használata növeli a sebességet
 - hozzunk létre egy virtuális diszket a fizikai diszkekből (OS majd azt kezeli)
- megvalósítások:
 - HW RAID vezérlők
 - SW RAID vezérlők
 - alaplapi megoldások is ilyenek szinte kivétel nélkül
 - szerver alapokban esetleg van HW RAID
- szintek:
 - RAID 0-6 és egymásba ágyazott (nested) szintek
 - RAID 0-1 szabványok általában SW implementációval és kevés diszkkal
 - RAID 2-4 szabványokat ritkán használjuk
 - RAID 5 és 6 alkalmazása tipikus nagyobb számú diszk esetén
 - egymásba ágyazott szintek: RAID 1+0, RAID 0+1
 - vannak gyártó specifikus nem szabványos megoldások

RAID level 0 (striped disks):

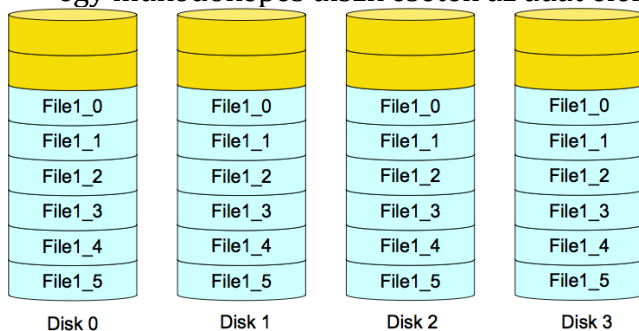
- több diszk párhuzamos használata
- a file részei N diszkre kerülnek, az egyes részek egymástól függetlenül érhetőek el
 - a diszkek tároló kapacitás összeadódik
 - N azonos diszk esetén a RAID 0 virtuális diszk olvasási és írási adatátviteli sebessége maximum N-szeres közelében nő
 - a hozzáférési idő közel eléri egy diszk hozzáférési idejét

- bármelyik diszk meghibásodás esetén elveszik az adat



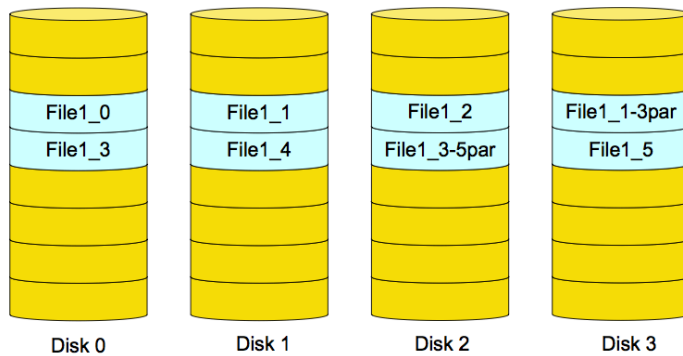
RAID level 1 (mirroring):

- több diszk redundáns használata
- a file minden része minden (N) diszkre kikerül
- azonos diszkeket feltételezve a tároló terület egy diszk tároló területével azonos
- az adatátviteli sebesség lassabb mint egy diszk sebessége
- a hozzáférési idő nő
- speciális esetben az olvasási sebesség N-szeresre nőhet
- egy működőképes diszk esetén az adat elérhető



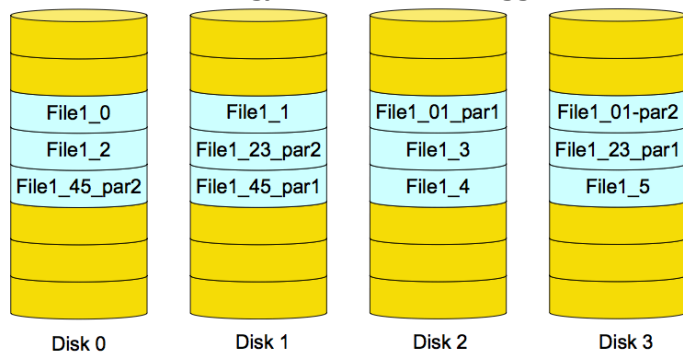
RAID level 5 (block interleaved distributed partity)

- több diszk redundáns és párhuzamos használata
- adat és paritás elosztása N+1 diszkre:
 - a sebesség tekintetében közel áll az N diszket használó RAID 0-hoz (HW támogatás esetén)
 - 1 diszk meghibásodása esetén az adat elérhető
 - 2 vagy több diszk meghibásodása esetén az adat elveszik
 - az adat nem feltétlenül állítható helyre
 - csendes/néma hibák (silent error)
 - a 2. meghibásodás észlelése a tömb újraépítése során



RAID level 6 (block interleaved dual distributed parity):

- több diszke redundáns és párhuzamos használata
- adat és paritás elosztása N+2 diszkre
 - a sebesség tekintetében közel áll az N diszket használó RAID 0-hoz (HW támogatás esetén)
 - 2 diszke meghibásodása esetén az adat elérhető
 - 3 vagy több diszke meghibásodása esetén az adat veszik
 - az adat nagyobb valószínűséggel állítható helyre a RAID 5-höz képest



RAID kritikája:

- hamis biztonságérzet:
 - csak a merevlemez egyedi, véletlen meghibásodása ellen véd
 - nem véd a SW hibáktól, illetéktelen hozzáféréstől, stb.
- nem pótolja a biztonsági másolatokat, csak a rendelkezésre állási időt és a sebességet növelheti
- A HW RAID vezérlők drágák:
 - 8 portos SATA RAID RAID5 és RAID 6 támogatással kb 200e FT
 - drágább, mint a hozzá csatlakozó diszkek
 - komplett gépet lehet venni ekkora összegből
- a SW raid megoldások elsősorban RAID 0 és RAID 1 esetén alkalmazhatóak
 - lassú a RAID 5 és 6 bonyolult kódolásának SW megvalósítása

RAID előnyei:

- a RAID 1 és RAID 5,6 megvéd a tipikus véletlen HDD hibák által okozott azonnali rendszer leállástól
 - a HDD a leggyengébb láncszem
 - SMART hatásos hibák előrejelzésére, de nem jelez biztosan előre hibákat
- a RAID 0 és RAID 5,6 gyorsíthatja a diszke hozzáférést

Hálózati tároló eszközök (SAN):

- Hálózati "tunnel" a host és a tároló eszköz között
 - alacsony, blokk szerű megoldás
 - általában egy kliens férhet hozzá egy időben
 - többnyire SCSI parancsokat küldenek át
 - a tároló eszköz virtualizációja:
 - teljes értékű, tetszőlegesen skálázható, partícionálható, bootolható,...
 - mintha lokálisan lenne a merevlemez csatlakoztatva
 - általában egy géphez csatlakoztathatók csak (kivéve fürtők/cluster)
- megoldások:
 - speciális protokollok: Fibre channel (drága, dedikált HW)
 - Ethernet és/vagy TCP/IP alapú: iSCSI, AoE
 - olcsó/ingyenes, részben SW, de legalább firmware támogatás is kell
- konvenciók:
 - target: hálózati tároló eszköz, amihez a fizikai tároló eszközök direkt módon, vagy további SAN szinteken keresztül csatlakoztatva vannak
 - initiator: a kliens, ami használja a tároló eszközöket
 - elnevezési konvenciók a használandó tároló eszköz azonosítására
 - initiator szintű hozzáférés ellenőrzése
- SAN-in elérhetővé tett permanens tárról lehetséges az OS indítása, de ehhez firmware (pl. BIOS) és OS támogatás is szükséges

Alacsony szintű fájlrendszer:

- fizikai diszk blokkok írását és olvasását végzik
- egyben feladata a működés során használt információk cache-elése is
- puffer cache (buffer cache): a lapozás során külön cache szolgál a pagefile cache-elésre, és külön a blokkok cache-elésére (beleértve a pagefile cache tartalmát is)
- egységes puffer cache (unified buffer cache): a cache a blokk szinten működik, nincs külön pagefile cache
- egységes virtuális memória (unified virtual memory)
 - a lapozás és a fájlrendszer szinten megvalósított, OS egészére vonatkozó disk cache összevonása
 - a file lényegében virtuális memóriára van leképezve
 - pl.: Linux ezt használja

Fájlrendszer leképezése:

- **logikai blokkok leképezése fizikai blokkokra** (allocation)
 - folytosonos allokáció (contiguous allocation)
 - láncolt listás allokáció (linked allocation)
 - indexelt tárolás (indexed allocation)
- **az üres helyek menedzselése** (free-space management)
 - bit vektor (bit vector)
 - láncolt lista (linked list)
 - szabad helyek csoportjainak listája (grouping)
 - számlálás (counting)
 - egybefüggő szabad területek nyilvántartás (space maps)

Folytonos allokáció (contiguous allocation):

- a fájl egy folytonos fizikai blokk sorozatot foglal el
 - hozzáférés egyszerű és gyors HDD esetén
- Növekvő mértű fájloknak a helyfoglalás problémás: milyen mértű szabad helyet foglaljunk?
 - új fájlok számára megfelelő szabad hely megtalálása nehéz, külső tördelés lép fel
 - fájl törlés után a méretének megfelelő számú blokk felszabadul
 - erre a helyre kisebb vagy egyenlő méretű fájl írható
 - first fit, next fit, best fit, worst fit
- növekedő fájlok
 - a best fit allokáció stratégia különösen veszélyes
 - a fájl nagyobb szabad helyre másolása - worst fit
- külső tördelődés csökkentése:
 - teljes másolás egy üres diszke majd vissza (off-line)
 - rendszerleállással jár
 - hosszú ideig tart és erőforrás igényes
 - futás idejű (on-line) töredezettség csökkentés (defragmentation)

Láncolt listás allokáció (linked allocation):

- a könyvtárakat leíró adatstruktúrák tartalmazzák az első és az utolsó blokk azonosítóját
- minden blokk tartalmazza a következő blokk azonosítóját
- a fájlhoz tartozó blokkok tetszőleges helyen lehetnek a diszken
- nincs külső töredezettség
- problémák
 - szekvenciális fájl elérésére alkalmas, a fájlban indexelni nehéz
 - a blokkban levő azonosítók helyet foglalnak
 - sérülékeny (azonosítók fűzik a blokkokat össze)
 - sok fejmozgást okoz (seek), ha a blokkok el vannak szórva a diszken
- pl.: FAT fájlrendszer ezt használja
- töredezettség mentesség ebben az esetben mást jelent:
 - cél a fejmozgás minimalizálása egy file olvasása során
 - a fájlok egymás utáni blokkokon történő tárolását tűzi ki célul
 - ezt is defragmentation-nak hívják
 - ilyen célból ajánlott időnként töredezettségmentesítést futtatni
 - írást és olvasást is jelentősen gyorsíthat

Indexelt tárolás (indexed allocation):

- index blokkok használata: egyes blokkokat fájlukhoz tartozó indexek tárolására allokálunk
- szekvenciális és indexelt elérésre is alkalmas
- sérülékeny (az index blokkok sérülése a fájl elérhetetlenné teszi)
- sok fejmozgást okoz (seek), ha a blokkok el vannak szórva a diszken
 - itt is lehet töredezettségmentesítést futtatni

Logikai fájlrendszer:

- OS specifikus API a tetjén (tipikus API függvények: create, delete, read, write, set/get attributes, stb.)
- metaadat tárolása (minden, kivéve a tényleges adat)
- fájlok:
 - absztrakt adattípus (objektum, fájl mutató)
 - adat, név, típus tulajdonságok (attributes)
 - kölcsönös kizárás (file locking)
- könyvtárak (Directory/folder)
- kötetek (volume/drive)

Fájl:

- a fájl a permanens táron az adattárolás logikai egysége
- tulajdonságok: név, típus, tulajdonságok, jogosultságok, hozzáférési idő

Könyvtárak:

- az információ hierarchikus tárolására
- kialakítások az idő múlásával:
 - egyszintű => kétszintű => fa => aciklikus irányított gráf => ált. gráf

Aciklikus irányított gráf struktúra: (pl: UNIX/Linux hard és symbolic links)

- egy fájl vagy alkönyvtár több könyvtárban megtalálható, de csak egy példányban létezik

Általános gráf struktúra:

- OS-eken ritkán használjuk, web-en annál inkább
- gondok: keresés, keresés leállási pontja

Kötetek:

- a logikai fájlrendszer szintén legmagasabb egység
- megfelel egy fizikai vagy logikai partíciónak a fizikai tárolón
- Windowsban C: | UNIX/Linux mount

Adatszerkezetek az eszközön:

- alacsony szintű adatstruktúrák
- boot szektor (boot control block)
- partíciós tábla (volume control block)
- fájlrendszer specifikus információ (könyvtárstruktúra leírói, fájl leírók)

Adatvesztés:

- a fájlok egy időben a memóriában és a permanens táron is jelen vannak
 - eltérő állapotban lehetnek
 - a metaadatokat, az allokációs struktúrákat is módosítják
 - meghibásodás/tápfeszültség elvesztése inkonzisztenciát okozhat
- konzisztencia ellenőrzése
- a konzisztencia visszaállítása: tranzakció orientált fájlrendszerek
- biztonságos rendszerleállítás, szünetmentes táplálás (UPS)
- adatmentés és visszaállítás
 - a mestérből a helyreállítást tesztelni kell
 - ha nincs helyreállítási teszt, nem beszélhetünk adatbiztonságról

Széles körben elterjedt fájlrendszerek:

- FAT (File Allocation Table)
 - 8+3 karakteres fájlnev, a hosszú fájlnev külön fájlba tárolva
 - FAT16: max 2GB partíció, 32767 könyvtár bejegyzés
 - FAT32: 2TB partíció méret, file méret: 4GB-1Byte
- NTFS (New Technology File System)
 - 2^{64} Byte(16 EB) - 1KB max fájl méret, 2^{32} -1 fájl
 - 256 karakter hosszú fájlnev
 - tranzakció alapú
 - töredezés mentesítés ennél is szükséges
- EXT2
 - alapértelmezett Linux fájlrendszer korábban
 - van Windows driver is, lehet Windows alatt is használni
 - max fájl méret: 16GB - 2TB, max 10^{18} fájl
 - max fájlnev hossza: 255 Byte, max partíció: 2-32 TB, lassú töredezés
- EXT3
 - EXT2 javított, tranzakció kezeléssel kiegészített verziói
 - Htree alapú indexelés: több könyvtárat tesz lehetővé
 - ez a javasolt Linux fájlrendszer, kivéve a flash eszközöket
 - EXT2 és EXT3 között egyszerű a konverzió
- EXT4: további bővítések (nagyobb táruk kezelése, extents, stb.)
- CD-ROM/DVD fájlrendszerek
(extra anyag)
- HFS Plus (Mac OS Extended):
 - metaadat gazdag, naplózó fájlrendszer
 - törekedik az automatikus töredezettségmentesítésre
 - fájlnevek legfeljebb 255 karakter hosszúak lehetnek (Unicode)
 - 3 féle link:
 - hard-link (UNIX féle)
 - szimbolikus link (UNIX féle)
 - álnévek (képes fenntartani a kapcsolatot a fájllal, akkor is ha áthelyezték, átnevezték)
 - FAT-ot képesek írni/olvasni, NTFS-t csak olvasni tudnak az OS X újabb verziói

NAS (Network-Attached Storage):

- fájlrendszer szintű hálózati fájlmegosztás (többnyire nyomtató is)
- pl: Network File System (NFS)
- fájlrendszer szintű megosztás
 - a hálózaton a könyvtárakra és fájlokra vonatkozó utasításokat küldünk
 - jellegzetesen párhuzamosan több felhasználó érheti el
 - felhasználó szintű jogosultságok
 - problémák a kliens és szerver fájlkezelési konvenciójának eltéréséből
- HTTP nem ilyen, komplett fájl osztható meg, nem lehet indexelni

21. UNIX fájlrendszerek alapismeretei

Ismétlés:

- a folyamatok
 - a felhasználói programok futás alatt állópéldányai
 - a programokat permanens tárból töltjük
- a permanens táruk
 - nem felejtő, nagyságrendekkel nagyobb és lassabb a memóriánál
 - blokkos fizikai tárolás és fájl-alapú logikai szervezés
 - többféle megoldás egyedi jellemzőkkel (HDD, flash, usb, RAID, SAN)
- a kernel
 - kezeli a hardver erőforrásokat (köztük a permanens tárukat)
 - a hardverkezelő réteg felett többszintű fájlrendszer réteg található
 - háttértár kezelés, fájlrendszer szervezés, logikai felépítés
- adminisztrálja a fájlok blokkjait és az üres helyeket a permanens tárban
 - elvégzi a fizikai és logikai szervezés közötti leképezést
 - programozói interfészt nyújt az alkalmazásfejlesztők számára

Alapfogalmak:

- fájl : adattárolási hely
- fájlrendszer: fájlok tárolásának szervezése, hozzáférés biztosítása
- fájlrendszerek felhasználói felülete:
 - programozói (API, rendszerhívások)
 - parancssori (illetve grafikus)
- fájlrendszerek szervezési felülete: diszk szervezés

UNIX fájlrendszer történelmi áttekintése:

- System V első fájlrendszere: s5fs
- 4.2 BSD Fast File System (FFS)
 - megnövelt teljesítmény
 - új szolgáltatások
 - akkori diszk hardver felépítéshez optimalizált rendszer
- virtuális fájlrendszerek (vnode/vfs)
 - moduláris, objektum-orientált
 - cserélhető szervezési modulok, akár hálózati is
- elosztott fájlrendszerek
 - NFS: transzparens hálózati fájlrendszer RPC megvalósítással
- modern fájlrendszerek:
 - ext3, ext4, xfs, ReiserFS, Solaris ZFS
 - felhasználói fájlrendszerek gnome-vfs, fuse: ftp, smb, dav, stb. célra
 - klaszter fájlrendszerek, pl. Red Hat GFS

A fájlrendszer felhasználói szemmel:

- OS felhasználó:
 - parancssori grafikus felület
 - könyvtárszervezés, speciális könyvtárak
 - fájlok és könyvtárak kezelése, attribútumaik
 - fájlrendszerek menedzselése (rendszergazda)
- Programozó (alkalmazás fejlesztő):
 - programozói interfészek (rendszerhívások, rendszerkönyvtárak)
 - fájlleírók, nyitott fájl objektumok és kezelésük
 - zárolási módszerek: kötelező, ajánlott

Felhasználói interfész:

- diszkek, partíciók, fájlrendszerek
- tipikus UNIX könyvtárszerkezet
- fájlrendszerek csatlakoztatása a könyvtárszerkezethez
- fájl attribútumok:
 - típus (- d p l b c s)
 - linkek (hard, szoft)
 - eszköz, inode, méret, stb.
 - időbélyegek (ctime, mtime, atime)
 - azonosítási és hozzáférés-szabályozási adatok
 - listázási parancs: ls, ls -all, ls -al, ...

Programozói interfész:

- fájlok megnyitása (létrehozása)
 - open() rendszerhívás és paraméterei
 - a fájlleíró és a nyitott fájl objektum
 - több folyamat által megnyitott fájl és fork()
- írás és olvasás: read(), write()
- fájlok zárolása:
 - kötelező (mandatory): fcntl(), lockf()
 - ajánlott (advisory): flock()
- fájlok lezárása: close()

Fájlrendszerek szervezése:

- csatlakoztatás:
 - csatlakozási pont
 - eldefés
- szervezés a háttértáron:
 - blokkos tárolás
 - fájlok leírói (diszk inode)
 - szabad helyek kezelése
- szervezés a memóriában
 - csatlakoztatás nyilvántartása
 - fájlok leírói (memória inode)
 - kapcsolt a nyitott fájl objektummal

A tárolás megvalósítás:

- a diszken elhelyezett fájlrendszer részei:
 - szuperblokk (fájlrendszer metaadatok)
 - inode lista (fájl metaadatok)
 - tárolt adatok
- szuperblokk
 - a fájlrendszer mérete
 - szabad blokkok jegyzéke
 - zárolási információk
 - módosítás jelzőbit

Az index node (inode)

- hitelesítési információk (UID, GID)
- típus, hozzáférési jogosultságok, időbélyegek, méret
- adatblokkok elhelyezése (címtábla)
 - 10-15 db direkt blokkcím
 - 1x, 2x, 3x indirekt blokkcímek
- inode a memóriában:
 - a nyitott fájl objektumhoz kapcsolódik
 - diszk inode tartalma bekerül a memóriába
 - az aktív használat információval bővül
 - státusz (zárolt, módosított, stb)
 - háttértár eszköz (fájlrendszer) azonosítója
 - hivatkozás számlálók (fájlleírók)
 - csatlakozási pont adminisztrációja, ...

Allokáció a diszken:

- szempontok: teljesítmény, megbízhatóság
- Cylinder (blokk) csoport (FFS, ext2, ...)
- allokációs elvek
 - szuperblokk másolása minden csoportba
 - inode lista és szabad blokkok csoportonként kezelve
 - egy könyvtár - egy csoport
 - kis fájlok egy csoportba
 - nagy fájlok "szétkenve" több csoportba
 - új könyvtárnak egy új, kevésbé foglalt csoportot keres

A virtuális fájlrendszer:

- impelentáció-független fájlrendszer absztrakció
 - a modern unix fájlrendszerek alapjai
- célok:
 - többféle fájlrendszer egységes egyidejű támogatása
 - egységes kezelés a csatlakozás után (programozói feladat)
 - speciális fájlrendszerek (hálózati, processzor, stb)
 - modulárisan bővíthető rendszer
- absztrakció:
 - inode -> vnode
 - fs -> vfs

A vnode absztrakció:

- adatmezők
 - közös adatok (típus, csatlakoztatás, hivatkozási száml.)
 - v_data: állományrendszertől függő adatok (inode)
 - v_op: az állományrendszer metódusainak táblája
- virtuális függvények
 - állományrendszertől független: vop_open, vop_read, ...
 - a tényleges metódusokra helyettesíthetődnek be
- segédrutinok, makrók

A vfs absztrakció:

- adatmezők
 - közös adatok (fájlrendszer típus, csatlakozás, hivatkozás, vfs_next)
 - vfs_data: állományrendszertől függő adatok
 - vfs_op: az állományrendszer metódusainak táblája
- virtuális függvények
 - állományrendszertől független: vfs_mount, vfs_umount, vfs_sync, ...
 - a tényleges metódusokra helyettesíthetődnek be
- segédrutinok, makrók

22. Felhasználó- és jogosultságkezelés

Fontos!

- Személyes adatok megvédése. Vissza lehet velük élni.
- Banki, üzleti szféra.
- beágyazott, zárt rendszerekben: megpróbálnak majd belepiskálni
- alkalmazások: felhasználó által elvárttól eltérő viselkedés

Mikor fontos a számítógépes biztonság?

- szoftverfejlesztésben minden szinten foglalkozni kell a biztonsággal
- *"ha egy rendszert nem terveztek biztonságosra, akkor később lehetetlen azzá tenni."*
- a rendszer biztonsága a leggyengébb láncszem biztonságával azonos. *"Az OS nem csodaszer, rosszul megírt alkalmazás ellen nem véd"*



Biztonság fogalma:

- cél: garantálni, hogy a rendszer mindig az elvárt módon viselkedjen
- egy technológia magában kevés
- a biztonság mindig csak a célkitűzés függvényében értelmezhető
- bizalmasság, sértetlenség, rendelkezésre állás

Biztonságot biztosító eszközök:

- kriptográfia: kommunikáció sértetlenségéhez, bizalmassághoz kell
- platform szintű behatolás elleni védelem:
 - rendszeren futó alkalmazások sértetlensége
- hálózati behatolás elleni védelem
- redundancia, újrakonfigurálás: rendelkezésre állás
- hitelesítés, engedélyezés

Tartalom:

- számítógépes biztonság bevezető
- felhasználó kezelés, hitelesítés: UNIX/Windows alatt
- engedélyezés:
 - engedélyezés általános sémái
 - szerep alapú hozzáférés-vezérlés
 - hozzáférési jogosultság listák
 - engedélyezés UNIX, Linux alatt
 - engedélyezés Windows alatt
 - biztonsági alrendszer alapok
 - központosított hozzáférés-vezérlés

Hitelesítés:

- mi alapján döntjük el, hogy ki kicsoda?
 - amit tud (pl.: jelszó)
 - amije van (pl.: kulcsa, belépőkártya)
 - ami ő (pl.: ujjlenyomat, arckép)
- ezek alapján egy (sértetlen) gép el tudja dönteni ki a személy aki előtte ül
 - mi a helyzet, ha nem sértetlen a gép?
 - mi a helyzet a gép-gép közötti szolgáltatásokkal?
- hitelesítés 3 szinten kerül elő:
 - ember és gép közötti interakció
 - gép és gép között valamilyen hálózaton át
 - gépen belül futó alkalmazások valamint az OS között
- hitelesítési protokollok kellenek
- gépen belül ill. gépek között csak az "amit tud" séma lehetséges
 - itt már feltételezhető bonyolult kriptográfiai számítás elvégzése

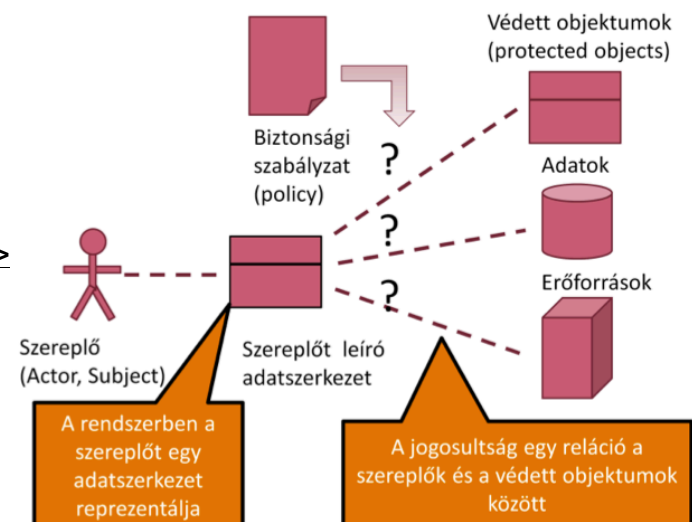
Miből áll egy felhasználói fiók:

- a rendszer száma'ra a felhasználó egy objektum
- a felhasználói fiókot azonosítja:
 - Linux, UNIX alapú rendszerek alatt UID
 - interger (root 0, felhasználó 1000-...)
 - /etc/passwd,shadow,group tárolja az attribútumokat
 - (csoporthoz is van hozzárendelt jelszó)
- mik ezek az attribútumok:
 - login név
 - jelszó (megváltoztathatóság, lejáratási idő)
 - home könyvtár
 - alapértelmezett shell (illetve shell belépés megtiltása)
 - alapértelmezett csoporttagság
 - komment (valódi név)

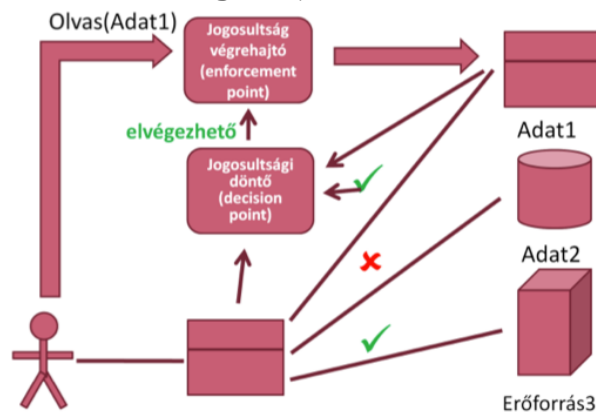
Azonosítás Linux alatt:

- gépen belül
 - felhasználó UID-név hozzárendelés feloldása gyakran kell -> /etc/passwd-t mindenkinek kell tudnia olvasni
 - jelszó ezért nem itt van, hanem /etc/shadow alatt, hash-elve
- többi kód között?
 - pl.: ssh-nál?
 - felhasználó név/jelszó
 - kriptográfiai kulcs alapján

Engedélyezés általános sémái ---->



Hozzáférés végrehajtása:



Jogosultságkezelés alapjai:

1. A rendszer működése során:

- a szereplők műveleteket kezdeményeznek
- a műveletek kontextusa tartalmazza a szereplő azonosítóját, a célobjektumok és az elvégzendő művelet fajtáját
- a jogosultsági döntő komponens kiértékeli a kontextust
 - engedélyezi vagy megtiltja a műveletet
- a jogosultsági végrehajtó komponens biztosítja, hogy a döntő által hozott döntés érvényre jusson

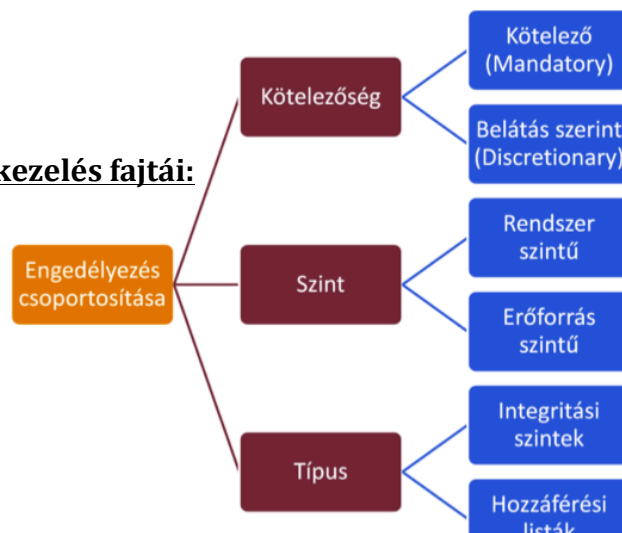
2. A rendszer karbantartása során:

- jogosultságok beállítása, módosítása történik
- a jogosultságot leíró adatszerkezet maga is egy védett objektum
 - tehát lehetnek olyan jogosultságok, amik saját magára hatással vannak
- általában a rendszer folyamatosan üzemel, nincs elkülönített karbantartási idő, a jogosultság módosítások azonnal érvényre jutnak

Jogosultságkezelés gyakorlati kihívási:

- sok szereplőt kell kezelni a rendszerben
 - különböző rendszerek különbözőképpen azonosítják őket
- sok védett objektumot kell kezelni
 - különböző rendszerek ezeket is különbözőképp azonosítják
- jogosultsági relációk:
 - (szereplők) X (objektumok) X (művelettipusok)
 - az ilyet teljes *hozzáférési mátrixnak* nevezzük
 - manuálisan (automatizáltan is) kezelhetetlen méretű adathalmaz

jogosultságkezelés fajtái:



Felhatalmazás fajtái - kötelezőség:

- kalasszikus fogalmak (US DoD szbvány)
- kötelező (mandatory)
 - csak központi jogosultság osztás
 - felhasználók nem módosíthatják a házirendet
- belátás szerint (discretionary)
 - megfelelő jogú felhasználó továbboszthatja a jogokat

Felhatalmazás fajtái - típus:

- integritás védelem
 - objektum címkézése: alacsony, közepes, magas... integritási szint
 - ellenőrzés: alacsonyabb szintű nem olvashat/írhat magasabb szintűt
 - Bell LaPadula (bizalmassági) és Biba (sértetlenségi) modellek
- Hozzáférés vezérlési listák
 - objektum -> (szereplők, engedélyek)
 - engedély: adatok írása, attribútumok olvasása...
 - hozzáférési maszk (access mask) tartalmazza, hogy pontosan melyik műveletre vonatkozik az engedély
 - szerep alapú hozzáférés-vezérlés (RBAC, Role-based Access Control)
 - a szerep foalom hierarchikus szereplő csoportosítási lehetőséget ad
 - a szükséges engedélyek száma kezelhető szintre csökken

Felhasználói fiók: A felhasználó csoporttagság valójában egy RBAC megvalósítási lehetőség.

Engedélyezés Linux alatt: POSIX fájlrendszer jogosultságok

- alapelvek:
 - szereplő: user
 - szereplő hierarchia: group
 - minden user tetszőleges sok group tagja lehet
 - minden group tetszőleges sok usert tartalmazhat
 - group további groupot nem tartalmazhat
- jogok
 - 3x3 bit, olvasás, írás, végrehajtás (könyvtárba belépé)
 - első a tulajdonos felhasználónak
 - második a tulajdonos csoportnak
 - harmadik mindenkinek
 - speciális bitek
 - setuid, setgid: futtatásnál átveszi a file tulajdonos uid-, gid-jét
 - sticky: újonnan létrejött fájlok tulajdonosát állítja
 - az execute bit tiltó hatása implicit módon öröklődik a könyvtárakon, más öröklés nincs

Fájlrendszeren kívüli engedélyek:

- speciális kiváltságok root felhasználó nevében futó folyamatoknak
 - kérhetnek valós idejű ütemezési prioritást
 - hozzáférhetnek közvetlenül a perifériákhoz
 - kell előtte memória, illetve I/O tartomány allokáció
 - a közelmúltig így működtek a grafikus felületet adó X Window server eszközmeghajtó programjai
 - 1024 alatti TCP/UDP porton hallgathatnak
 - kernel bizonyos konfigurációs beállításait megváltoztathatják, új modult tölthetnek be, stb.
- nem előnyös, ha ezek nem szabályozhatóak külön-külön
 - legkevesebb jog elve
 - POSIX Capabilities mechanizmus - globális rendszerszintű erőforrásokra vonatkozó jogosultságok (ún. privilégiumok)

Kitekintés: finomabb felbontású jogosultságkezelés végrehajtható fájlokra:

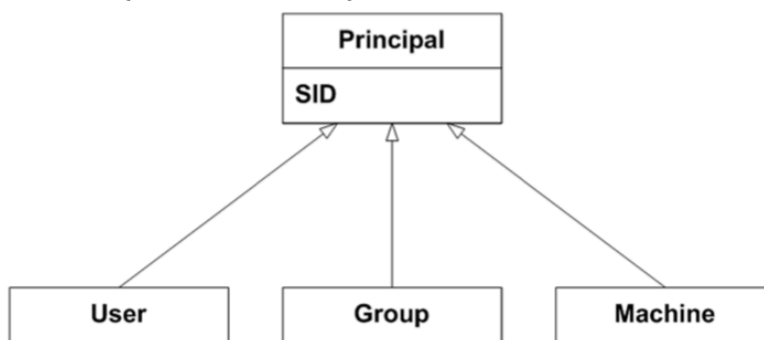
- platform szintű behatolás elleni védőmechanizmusok támogatása (PAX, grsecurity)
- a védőmechanizmusok számos egyébként sértetlen programot tesznek működésképtelenné (JavaVM)
- speciálisan kivételezni kell az ilyen alkalmazásokat fájlrendszerbe írt címkékkel (SELinux Security Labels)
- alkalmazásokhoz hozzárendelt rendszerhívás profilok (AppArmor) - felfedi ha a "szokásoshoz" képest megváltozik az alkalmazás futása

23. Biztonsági alrendszer a Windowsban

Biztonsági feladatok a Windowsban:

- 1. azonosítás (authentication)
 - birtok/tudás/biometria
 - pl. bejelentkezési képernyő, hitelesítő ablakok
- 2. engedélyezés (authorization)
 - alapelv: mindig csoportnak osztunk jogot
 - pl. biztonsági házirend, fájl ACL
- 3. auditálás
 - biztonsági naplózás

Azonoítás (authentication):



Principal: biztonsági rendszer által kezelt entitások összefoglaló neve

Security Identifier (SID):

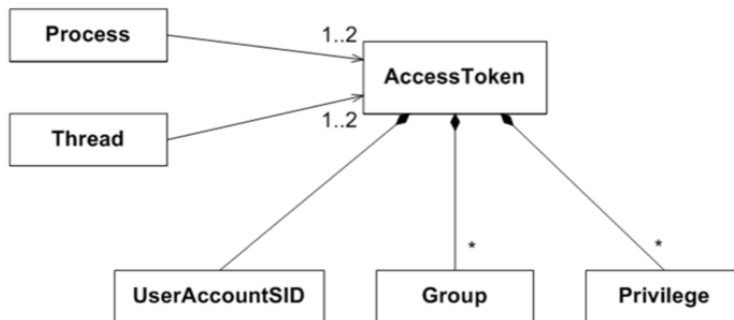
- felhasználó / számítógép azonosítója
- felhasználó, csoportok
 - <Gép SID>-<RID>
 - RID: relative identifier
- jól ismert SID-ek
 - everyone: S-1-1-0, administrator: S-1-5-domain-500
- vista: szolgáltatások is kapnak SID-et

1. Azonosítás:

- belépés:
 - winlogon saját ablakán keresztül
 - Secure Attention Sequence: Crtl+Alt+Del
- jelszavak tárolása:
 - hash a registry-ben
- hálózati azonosítás
 - NTLM: NT LAN Manager
 - Kerberos: Windows 2000 óta, tartományi (domain) környezetben

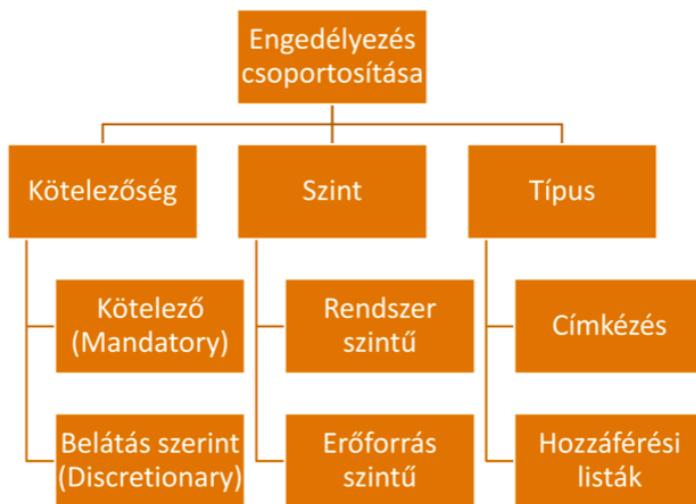
Azonosítás - Hozzáférési token:

- megismerés
- belépéskor hozzárendel egy hozzáférési token a felhasználóhoz, későbbi műveletek során az ebben tároltakat ellenőrzi a rendszer



2. Engedélyezés

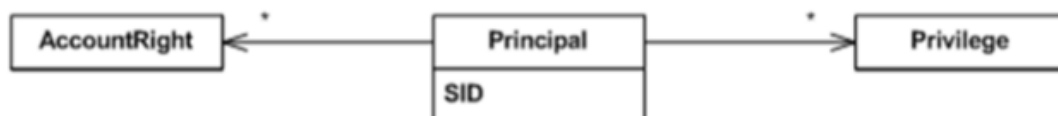
- engedélyezés fajtái



- a csoportosítás esetleges, rengeteg egyéb szempont van, ezek csak a Windowsban szereplő fogalmak

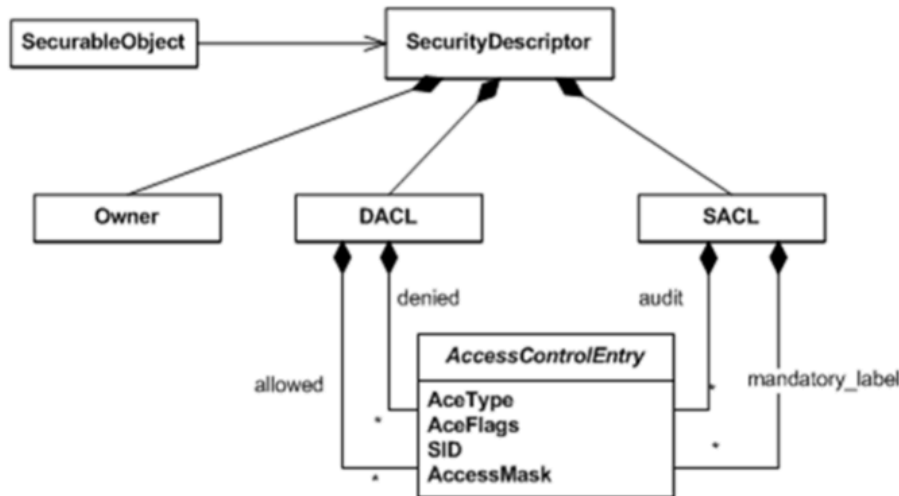
Engedélyezési lehetőségek a Windowsban:

- rendszerszintű jogosultságok



- jogosultság (privilege)
 - OS szintű jog
 - pl.: számítógép leállítása, eszközmeghajtó betöltés
 - név: SeShutdownPrivilege, SeLoadDriverPrivilege
- fiók jog (account right)
 - ki hogyan léphet be / nem léphet be
 - pl.: interaktív, hálózaton keresztül...

- Discretionary Access Control
 - belátás szerinti, erőforrás szintű, hozzáférési lista



- SecurableObject: Windowsos objektum, pl.: fájl, registry kulcs, pipe...
- SecurityDescriptor: összefogja a többi elemet
- Owner: megváltoztathatja az objektum engedélyeit, akkor is ha nincs explicit joga
- DACL: Discretionary Access Control List, hozzáférés szabályozása
- SACL: System Access Control List, biztonsági naplózás szabályozása
- AccessControlEntry:
 - típus: megengedő, tiltó, audit
 - flag: pl.: öröklődés
 - SID: kire vonatkozik
 - maszk: végrehajtás | törlés | tulajdonos írása...

- hozzáférési listák:
 - öröklődés flag: konténer típusú objektumnál, gyerek objektum megkapja azt az ACE-t
 - kiértéklés menete:
 - egy SID-re több ACE is érvényes lehet
 - ACE-kból kapott engedélyek UNIÓJA számít
 - kivéve a tiltást, az mindig magasabb prioritású

- Mandatory Integrity Control
 - kötelező, erőforrás szintű, címkézés

3. Auditálás (biztonsági másolat)

Eseménynapló:

- rendszeres és alkalmazás üzenetek
- bejegyzés: típus, idő, forrás, ID, leírás
- napló felülírása: ciklikus, időnként, soha