Operációs Rendszerek

Elméleti zh

A jegyzetet Umann Kristóf írta Csonka Szilvia segítségével. A jegyzet első számú forrása az előadásdiák, valamint nagyban a következő pdf struktúráját követi: `http://people.inf.elte.hu/memsaai/000`.

2. Előadás

Regiszter: A regiszterek a számítógépek központi feldolgozó egységeinek (CPU-inak), illetve mikroprocesszorainak gyorsan írható-olvasható, ideiglenes tartalmú, és általában egyszerre csak 1 gépi szó (word) (rövid karakterlánc, 1-2 szó általában 2-4 bájt) feldolgozására alkalmas tárolóegységei.

Processzor védelmi szintek:
Intel 80286 - minden utasítás egyenlő
Intel 80386 - 4 védelmi szint, ebből 2-őt használ, kernel mód (védett, protected mód) és felhasználói mód

Megszakítások: A megszakítások nagyon fontos elemei a számítógépek működésének. Amikor a mikroprocesszornak egy eszközt, vagy folyamatot ki kell szolgálni, annak eredeti tevékenységét felfüggesztve, megszakítások lépnek életbe. Létezik szoftveres, és hardveres megszakítás.

- *Hardveres megszakítás* akkor következik be, amikor egy eszköznek szüksége van az operációs rendszer figyelmére. Ilyenkor a processzor erőforrást (processzoridőt) igénylő eszköz megszakításkérést küld a mikroprocesszornak. Amennyiben a megszakítás lehetséges, a kérést kezdeményező eszköz használhatja a mikroprocesszort.
- *Szoftveres megszakítás* esetén a főprogram futását egy alprogram szakítja meg. Ebben az esetben a főprogram futásállapota elmentésre kerül, majd miután a megszakítást kérő program befejezte a műveletet a főprogram folytatja a futását a megszakítás előtti pozícióból. Példa erre, amikor a folyamat egy rendszerhívást tesz meg, ekkor a futása félbeszakad, végbemegy a hívás, és a folyamat futása folytatódik.
- Egy harmadik megszakítási típus lehet a *csapdák (traps)*, melyek hibás szoftverműködés esetén lépnek fel (pl. nullával történő osztás).

Megszakítás maszkolása: Maszkolással egyes megszakításokat figyelmen kívül tud hagyni az operációs rendszer. Vannak azonban nem maszkolható megszakítások is (NMI, Non-maskable interrupt), pl. azok melyek memóriahiba, vagy tápfeszültség kimaradás esetén keletkeznek.

Operációs rendszer: olyan program ami egyszerű felhasználói felületet nyújt, eltakarva a számítógép (rendszer) eszközeit.

Kommunikáció a perifériákkal:

- Lekérdezéses átvitel (polling) – folyamatos lekérdezés.
- Megszakítás (Interrupt) – nem kérdezgetjük folyamatosan, hanem az esemény bekövetkezésekor a megadott programrész kerül végrehajtásra. Pl.: aszinkron hívások esetén.
  Aszinkron hívások: Olyan adatátviteli mód, amikor a két kommunikáló fél nem használ külön időzítő jelet, ellentétben a szinkron átvitellel. Éppen ezért szükséges az átvitt adatok közé olyan információ elhelyezése, amely megmondja a vevőnek, hogy hol kezdődnek az adatok.
- DMA: közvetlen memória elérés (6. EA-ban van részletezve).

API (application programming interface): Egy program vagy rendszerprogram azon eljárásainak (szolgáltatásainak) és azok használatának *dokumentáció*ja, amelyet más programok felhasználhatnak. Egy nyilvános API segítségével lehetséges egy programrendszer szolgáltatásait használni anélkül, hogy annak belső működését ismerni kellene. Általában nem kötődik programozási nyelvhez. Az egyik leggyakoribb esete az alkalmazásprogramozási felületnek az operációs rendszerek programozási felülete: annak dokumentációja, hogy a rendszeren futó programok milyen – jól definiált, szabványosított – felületen tudják a rendszer szolgáltatásait (pl. kiíratás) használni.

POSIX (Portable Operating System Interface for uniX): Valójában *egy minimális API készlet, vagy szabvány*, aminek témaköreibe tartozik pl.: fájl, könyvtárműveletek, folyamatok kezelése, szignálok, szemaforok stb. Ma gyakorlatilag minden OS POSIX kompatibilis.

Firmware: Hardverbe a gyártó által épített szoftver (merevlemezbe, billentyűzetbe, monitorba, memóriakártyába, de pl. távirányítóba vagy számológépbe épített is). Amelyek olyan rögzített, többnyire kis méretű programok és/vagy adatstruktúrák, melyek különböző elektronikai eszközök vezérlését végzik el.

Middleware: *Operációs rendszer feletti réteg* (pl. Java Virtual Machine, JVM).
Általánosan véve egy olyan számítógépes szoftver, amely az operációs rendszerek mögötti, azok számára nem elérhető szoftveralkalmazásokat biztosítja, de nem része egyértelműen az operációs rendszernek, nem adatkezelő rendszer és nem része a szoftveralkalmazásoknak sem.
Megkönnyíti a szoftverfejlesztők dolgát a kommunikációs és az input/output feladatok végrehajtásában, így a saját alkalmazásuk sajátos céljára tudnak összpontosítani.

Operációs rendszer generációk:

- ***Történelmi generáció** (1792-1871)*:
  - *elektromechanikus* számítógépek
  - nincs oprendszer, operátoralkalmazás
- ***Első generáció**(1940-1955)*:
  - a programot kapcsolótáblán kellett beállítani, *elektroncső*vel működött, programozása *kizárólag gépi nyelv*en történt, lukkártyák megjelenése
  - *Neumann-elv* (Neumann János): kettes számrendszer alkalmazása, memória, programtárolás, utasításrendszer
- ***Második generáció**(1955-1965)*:
  - már *tranzisztor*okat tartalmaztak (ami lecsökkentette a méretüket)
  - memóriaként *mágnestár*at használnak (mágnesszalag, majd mágneslemez)
  - *operációs rendszer* megjelenése
  - magas szintű prognyelvek pl. fortan
  - kötegelt rendszer megjelenése (5. EA-ban van részletezve)
- ***Harmadik generáció**(1965-1980)*:
  - integrált áramkörök megjelenése
  - azonos rendszerek, kompatibilitás megjelenése
  - mutliprogramozás, multitask (több feladat a memóriában egyidejűleg) megjelenése
  - időosztás megjelenése
  - bonyolultabb operációs rendszerek
- ***Negyedik generáció**(1980-tól napjainkig)*:
  - *személyi számítógépek*, MS Windows
  - áramkörök, CPU (processzor) fejlődés
  - hálózati, osztott rendszerek

Rendszerhívások: azok a szolgáltatások amelyek az operációs rendszer és a felhasználói programok közötti kapcsolatot biztosítják. Két fajtája van: process kezelő és fájlkezelő.

Process: egy végrehajtás alatt lévő program. Saját címtartománnyal rendelkezik, megszüntetés, felfüggesztés, és process-ek kommunikációja is lehetséges.

Processz táblázat: Az operációs rendszer által nyilván tartott táblázat, melynek minden sora tartalmazza egy éppen futó process adatait, pl. cím, regiszter, munkafájl adatok.

Operációs rendszer struktúrák:

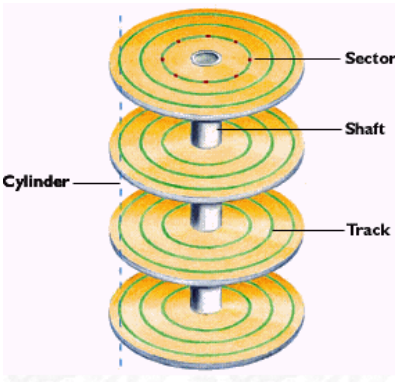
- ***Monolitikus rendszerek***: nincs különösebb struktúrája, a rendszerkönyvtár egyetlen rendszerből áll, így mindenki mindenkit láthat. Információelrejtést nem igazán valósítja meg. Létezik modul, modulcsoportos tervezés. Rendszerhívás során gyakran felügyelt (kernel) módba kerül a CPU. Paraméterek a regiszterekben, valamint a csapdázás (trap) is jellemző.
- ***Rétegelt szerkezet***:

6. Gépkezelő
5. Felhasználói programok
4. Bemeneti / Kimenet kezelése
3. Gépkezelő-folyamat
2. Memória és dobkezelés
1. Processzorhozzárendelés és multiprogramozás

3. Előadás

Mágnesszalagok: sorrendi, lineáris felépítéssel rendelkezik, keretek rekordokba szerveződnek. Rekordok között rekord elválasztó (record gap), fájlok között fájl elválasztó (file gap) található. Gigabyte-ra levetítve a legolcsóbb tárolási módszer. Biztonsági mentésekre, nagy mennyiségű adat tárolására használatos.

FDD (Floppy Disk Drive) és **HDD** (Hard Disk Drive): Az FDD (általában) egy, a HDD (általában) több kör alakú lemezből áll. Ezek sávokra (tracks), a sávok blokkokra (blocks), a blokkok szektorokra (sectors) vannak felosztva. A több lemezen elhelyezkedő, egymás alatt lévő (azaz azonos sugarú) sávok összeségét cilindernek (cylinder) nevezzük (ld. 1. ábra). A HDD rendelkezik egy író/olvasó fejjel is, mely e lemezek fölött ill. között mozog, e segítségével lehet a különféle műveleteket végrehajtani.



1. ábra. HDD részei

Logikai tároló (logical disk): Olyan tároló, mely az operációs rendszer számára egybefüggő memóriaterületnek tűnik. A „logikai" szó arra utal, hogy gyakran ezek nem ténylegesen így tárolódnak, hanem akár több lemezen keresztül is, azonban a HDD firmware programja gondoskodik arról, hogy számunkra a teljes terület egybefüggőnek tűnjön.

Optikai tárolók: Fényvisszaverődés alapján működnek, ilyen a CD, DVD, blue-ray, stb.

Eszközmeghajtó (Device driver): Az a program, amely a közvetlen kommunikációt végzi az eszközzel. Ez a kernelnek (operációs rendszer magja) a része. Lemezek írása, olvasása során DMA-t használnak. (DMA: 6. EA-ban van kifejtve), réteges felépítés.

Mágneslemez formázása: sávos-szektoros rendszer kialakítása.

- Quick format – Normal format: normal format hibás szektorokat is keres.
- Alacsony szintű formázás: szektorok kialakítása (ez gyártóknál elérhető).
- Logikai formázás: a partíciók kialakítása – max 4 logikai rész alakítható ki.

0. Szektor – MBR (Master Boot Record): Partíciós szektor a merevlemez legelső szektorának (azaz az első lemezfelület első sávjának első szektorának) elnevezése. Csak a particionált merevlemezeknek van MBR-jük. A MBR a merevlemez legelején, az első partíció előtt található meg. Gyakorlatilag a merevlemez partíciók elhelyezkedési adatait tárolja.

Boot folyamat: ROM-BIOS megvizsgálja, lehet-e operációs rendszert betölteni, ha igen, betölti a lemez

MBR (lásd. fent) programját a `7c00h` címre (konvenció szerint ez mindig erre a címre történik). Ez után az MBR programja vizsgálja meg mi az elsődleges partíció, majd azt betölti a memóriába.

Sorrendi ütemezés (FCFS – First Come First Service): Ahogy jönnek a kérések, úgy sorban szolgáljuk ki azokat. Biztosan minden kérés kiszolgálásra kerül, de nem törődik a fej aktuális helyzetével, kicsi az adatátviteli sebesség, és ezért nem igazán hatékony. Átlagos kiszolgálási idő, kis szórással.

„Leghamarabb először" ütemezés (SSTF – Shortest Seek Time First): a legkisebb fejmozgást részesíti előnyben, átlagos várakozási idő kicsi, átviteli sávszélesség nagy, fennáll a kiéheztetés veszélye (azaz előfordulhat, hogy egy kérést sose teljesít, mert minden más kérés teljesítéséhez kevesebbet kéne mozogia a fejnek).

Pásztázó ütemezés (SCAN) módszer: a fej állandó mozgásban van, és a mozgás újtába eső kéréseket kielégíti. A fej mozgás megfordul ha a mozgás irányában nincs kérés, vagy a fej szélső pozíciót ér el. Rossz ütemben érkező kérések kiszolgálása csak oda – vissza mozgás után kerül kiszolgálásra. Középső sávok elérésének szórása kicsi.

Egyirányú pásztázás (C-SCAN): csak egyirányú mozgás, ezért gyorsabb a fejmozgás, nagyobb sávszélesség, átlagos várakozási idő hasonló, mint a SCAN esetén, viszont a szórás kicsi. Nem igazán fordulhat elő rossz ütemű kérés.

Ütemezés javítások: FCFS esetében ha az aktuális sorrendi kérés kiszolgálás helyén van egy másik kérés is akkor szolgáljuk ki azt is (Pick up).

Lemezek megbízhatósága: A lemezek meg tudnak sérülni, mely adatvesztéshez vezet. Erre egy megoldás lehet, ha az adatokat redundánsan tároljuk úgy, hogy egy lemez sérülése esetén se történjen adatvesztés.

Ütemezés javítása memória használattal:

- DMA maga is memória (6. EA)
- Memória puffer:
  - Olvasás: ütemező feltölti, felhasználói program kiüríti.
  - Írás: felhasználói folyamat tölti, ütemező kiüríti.
- Disc cache használatával

Dinamikus kötet: Több lemezre helyez egy logikai meghajtót. Méret összeadódik.

RAID (régebben Redundant Array of Inexpensive Disks, mostani időkben inkább Redundant Array of Independent Disks):
Ha oprendszer nyújta akkor SoftRaid-nek is nevezzük, ha külső vezérlőegység akkor Hardver Raid.
A RAID egy tárolási elv, mely az adatok biztonságos tárolását írja le. Az alapelve az, hogy egyszerre több lemezről ír és olvas (ezért redundáns). Az operációs rendszer számára ez a több disk egynek tűnik (egy tömbnek, angolul array). Ez az elv régen azzal a szlogennel élt, hogy „an array of inexpensive drives could beat the performance of the top disk drives of the time" (ezért olcsó, angolul inexpensive), de ez a mai árak szerint drágább, így az inexpensive szót leváltották independent-re.
A RAID-nek 7 szintje vagy verziója létezik:

- RAID 0: több lemez logikai összefűzésével egy meghajtót kapunk, ezek összege adja az új meghajtó kapacitását. A logikai meghajtó blokkjait szétrakja a lemezekre, ezáltal egy fájl írása több lemezre kerül. Gyorsabb I/O műveletek (azaz messze ez a leggyorsabb RAID), de nincs meghibásodás elleni védelem. Nevével szemben nincs benne redundáns adattárolás.
- RAID 1: Két független lemezből készít egy logikai egységet, minden adatot párhuzamosan kiír mindkét lemezre. Tárolókapacitás a felére csökken, drága megoldás, csak mindkettő lemez egyszerre történő meghibásodása esetén okoz adatvesztést.
- RAID 2: Több lemezen is tárol adatot, és valamennyi külön erre dedikált lemezen csak ezen adatok hibavizsgáláshoz és helyreállításhoz szükséges információkat ment le. Ennek nincsen semmilyen előnye a RAID 3-hoz képest, és már nem is használják.

– RAID 3: Egyetlen disc-en tárol paritás információkat (melyek alapján eldönthető, hogy sérült-e a fájl). Mivel egyszerre csak egy kérést tud teljesíteni, hosszú beolvasások és kiíratások esetén nagyon hatékony, gyakori rövid műveletek esetén a legrosszabb.

– RAID 4: A RAID 0 kiegészítése paritásdiszkkel. Ennek köszönhetően egyszerre több olvasást is végre tud hajtani. Semmilyen előnye nincs a RAID 5-tel szemben.

– RAID 5: Nincs paritásdiszk, az oda tartozó információ el van osztva az összes disc-re. Adatok is elosztva tárolódnak. Intenzív CPU igény, két lemez egyidejű meghibásodása esetén okoz adatvesztést. Azaz +1 diszket igényel.

– RAID 6: A RAID 5-höz hasonlóan tárolja a paritásinformációkat, de ezek mellett még hibajavító kódokat is tárol. Ez közel kétszer annyi területet foglal backup-ra mint a RAID 5, azonban két disc egyszeri meghibásodása se jár adatvesztéssel. Meglehetősen drága.

– Megjegyzés: leggyakrabban az 1, 5 verziókat használják, a 6-os vezérlők az utóbbi 1-2 évben jelentek meg.

Fájl: Adatok egy logikai csoportja, névvel, paraméterrel ellátva.

Könyvtár: Fájlok logikai csoportosítása.

Fájlrendszer: Módszer a fizikai lemezünkön, kötetünkön a fájlok és könyvtárak elhelyezés rendszerének kialakítására.

Elhelyezkedési stratégiák:

– Folytonos tárkiosztás
– Láncolt elhelyezkedés
– Indextáblás elhelyezkedés: katalógus tartalmazza a fájlhoz tartozó kis tábla (un. *inode*) címét. E tábla segítségével érhetőek el azok a blokkok, melyek a fájlt tartalmazzák (ugyanis egy fájl gyakran több blokkban van).
Az inode tábla 15 rekeszből áll, melyből az első 12 a fájl blokkjaira mutat. Ha ez kevés a 13. rekesz egy újabb inode-ra mutat, mellyel +15 rekesz érhető el. Amennyiben ez is kevés, a 14. rekeszbe újabb inode kerülhet, és így tovább.

Naplózott fájlrendszer (journaling file system): A fájlrendszer nyilvántartást vezet a szándékozott változtatásokról (pl. fájl törlése). Sérülés, áramszünet, stb. esetén ha hiba következik be az ezáltal könnyen helyreállítható. Nagyobb erőforrás idényű de jobb a megbízhatósága.

Fájlrendszerek:

– **FAT** (File Allocation Table): a FAT tábla a lemez foglalási térképe, annyi eleme van ahány blokk a lemezen. Lévén egy fájl jó eséllyel több blokkon helyezkedik el (akár nem is szekvenciálisan), így a FAT a katalógusában a fájl adatok (név stb) mellett csak a fájl első blokk sorszámát tárolja el. A FAT blokk azonosító mutatja a fájlhoz tartozó következő blokk címét, ha nincs ilyen akkor az értéke `FFF`. A fájl utolsó módosítási idejét is tárolja.

  Töredezettségmentesítés szükséges (azaz újra kell rendezni a blokkokat), amennyiben a fájlok blokkjai nagyon szétszórva helyezkednek el a disc-en. Ennek hiányában jelentős lelassulnak az I/O műveletek.

– **NTFS** (New Technology File System): A FAT-tel szemben számos újdonsággal/javulással rendelkezik: kifinomult biztonsági beállítások, POSIX támogatás, fájlok és mappák tömörítése, felhasználói kvóta kezelés (azaz egyes felhasználók csak a disc bizonyos részeihez férhessenek hozzá), ezen felül az NTFS csak klasztereket (más néven blokkokat) tart nyilván, szektort nem.

  Az NTFS partíció a Master File Table (MFT) táblázattal kezdődik, mely (hasonlóan a FAT-hez) az egy adott fájlhoz tartozó klasztereket tárolja. Egy adott fájlhoz 16 attribútumot rendel (pl. név). Egy attribútum max 1 kb lehet, ha ez nem elég, akkor egy attribtum mutat a folytatásra. Amennyiben a fájl maga kisebb mint 1 kb, akkor belefér egy attribútumba, ezáltal közvetlenül elérhető lesz

az MFT-ből. A biztonság kedvéért két MFT-t is vezet az operációs rendszer, amennyiben az egyik tönkremegy, a másikat tudja használni. Töredezettségmentesítés szükséges. (ld. 2. ábra)

– **UNIX könyvtárszerkezet**: Indextáblás megoldás, boot blokk (erről majd később bővebben) után a partíció un. szuperblokkja következik (mely leírja a rendszer tulajdonságiat, pl. a méretét, mekkora legyen egy blokk mérete, és még sok egyéb), ezt követi a szabad terület leíró rész (i-node tábla, majd gyökérkönyvtár bejegyzéssel). Moduláris elhelyezés, gyorsan elérhető az információ, sok kicsi táblázat, ez alkotja a katalógust. Egy fájlt egy i-node ír le.

2. ábra. NTFS partíció felépítése.

4. Előadás

Valódi-e a Multi Task? - A Multi Task az, amikor több process párhuzamosan fut. Ezt egy darab processzormag nem tudja megvalósítani, ennek csak a látszatát tudja kelteni a processzek közötti kapcsolgatással. Egy időben csak egy folyamat aktív.

Egy feladat-végrehajtáshoz egy processzorra, külön rendszer memóriára és egy I/O eszközre van szükség.

Környezetváltásos rendszer: Csak az előtérben lévő alkalmazás fut

Kooperatív rendszer: Az aktuális process bizonyos időközönként, vagy időkritikus műveletnél önként lemond a CPUról (Win3.1).

Preemptív rendszer: Az aktuális process-től a kernel bizonyos idő után elveszi a vezérlést, és a következő várakozó folyamatnak adja. (Ma tipikusan ilyen rendszereket használunk.)

Real time rendszer: Igazából ez is preemtív rendszer (különbségek később).

Folyamatok létrehozásának okai: Például boot, folyamatot eredményező rendszerhívás(fork, execve), felhasználói kérés (parancs`&`), nagy rendszerek kötegelt feladata.

Folyamatok kapcsolata: A folyamatok között szülő-gyerek kapcsolat lép fel.
Az így felírható folyamatfában

– egy folyamatnak egy szülője van.
– egy folyamatnak több gyereke is lehet.

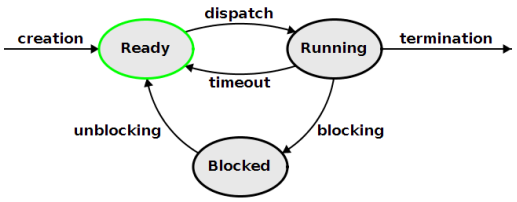
Linuxon, az első folyamat amelyik elindul az az `Init`, ez rendelkezik az 1. számú ID-vel. (C-ben egy process ID-jét lekérdezhetjük a `getpid()` (Get Process ID) függvénnyel)

Reinkarnációs szerver: Meghajtó programok, kiszolgálók elindítója. Ha elhal az egyik, akkor azt újraszüli, reinkarnálja.

Folyamatok befejezése
Két oka lehet:

- Önkéntes befejezés: Ilyen a szabályos kilépés (exit, return), korai terminálás a program által feldezett hiba miatt, stb.
- Önkéntelen befejezés: Akkor következik be, ha a folyamat pl. illegális utasítást hajt végre (túlcímez), végzetes hibát vét (nullával való osztás). A befejezés végbemehet külső segítséggel, vagy a felhasználó által.

Folyamat: önálló programegység, utasításszámlálóval, veremmel stb. Általában nem függetlenek, más folyamatok eredményétől függ a tevékenységük. Három állapotban lehet: futó, futásra kész, vagy blokkolt. (ld. 3. ábra)



3. ábra. Folyamatok állapotai

Folyamat megvalósítása: 1 aktív folyamat 1x-re. Mindent meg kell őrizni(regiszter, utasítás számláló, nyitott file infó). Globálisan maszkolható, a nem maszkolható le van kötve. Attól lesz egyszerű, hogy ezen folyamatokból válogatva valósul meg, valamint a taszk szegmens regiszterekkel mutat a táblákra, így a proci közvetlen támogatást ad a processzéhez.

Folyamatok váltása: időzítő, megszakítás, esemény, rendszerhívás kezdeményezés. CASH nem menthető. Szál: egy folyamaton belül több egymástól „független" végrehajtási sor. A folyamatnak önálló címtartománya van, szálnak viszont nincs.

Szálak (threads): Általában egy folyamat egy utasítássorozatból áll. Azonban néha szükség lehet egyszerre több, egymástól független utasítássorozatra is egy folyamaton belül. Ezeket az utasítássorozatokat szálaknak, thread-eknek vagy lightweight process-nek hívjuk.
Minden folyamat egy szál, de nem minden szál folyamat. Csak egy folyamat rendelkezik címtartománnyal, globális változókkal, megnyitott file leírókkal, gyermek folyamatokkal. Mind a folyamatok, mind a szálak rendelkeznek utasításszámlálóval, regiszterrel, veremmel.

Folyamatleíró táblázat (Process Control Block, PCB): A rendszer inicializálásakor jön létre. Már indításakor is tartalmaz 1 elemet, a rendszerleíró már bent a rendszer indulásakor bekerül. Tömbszerű szerkezete van (PID alapján). Egy sorában egy összetett processzus adatokat tartalmazó struktúra található. Egy folyamat fontosabb adatai: azonosítója, neve, tulajdonosa, csoportja stb.

Szálproblémák:

- Fork: gyerekben kell több szál, ha a szülőben több van.
- Filekezelés: egy szál lezár egy másik által használt file-t.
- Hibakezelés: globális hibajelző (`errno` C-ben egy globális változó, ha több szál akarja használni, könnyen galiba lehet).
- Memóriakezelés
- A rendszerhívásoknak kezelni kell tudni a szálakat (azokat hívásokat, melyek ezt tudják teljesíteni, thread-safe hívásoknak is nevezzük).

IPC (Inter Process Comunication): Programozási interfészek egy halmaza, melyek lehetővé teszik a processzek kommunikációját. Lévén gyakran szükség van egy program futtatásakor sok másik programra is, az ezek közötti kommunikációt is biztosítani kell. Példa.: pipe, szemafor.

Versenyhelyzet: két vagy több folyamat közös memóriát ír vagy olvas.

Kritikus programterület: Az a rész, ahol több process vagy szál ugyanazt az erőforrást (memóriát) használja.

Kölcsönös kizárás: (szemléletes ábra 4. EA 21.dia) A jó kölcsönös kizárás az alábbi feltételeknek felel meg:

- Nincs két folyamat egyszerre a kritikus szekciójában.
- Nincs sebesség, CPU paraméterfüggőség.
- Egyetlen kritikus szekción kívül levő folyamat sem blokkolhat másik folyamatot.
- Egy folyamat sem vár örökké, hogy a kritikus szekcióba tudjon belépni.

Megvalósítás:

- Megszakítások tiltása: Belépéskor az összes megszakítás letiltásra kerül, kilépéskor engedélyeződnek. A kernel pl. ezt (is) használja, azonban nem épp szerencsés megoldás.
- Osztott, un. zárolós változó használata: Egy több folyamat által is látható változó értéke adja meg, hogy a kritikus szekció foglalt-e. Pl. ha ez a változó 0, a kritikus szekcióban egyetlen folyamat se fut és 1, amennyiben egy program fut benne. Ennek az a hátránya, hogy ha két folyamat egyszerre akar belépni, akkor lehetséges hogy mindketten befférnek, mielőtt az a változó 1-re vált.
- Szigorú váltogatás: Több folyamatra is általánosítható. A kölcsönös kizárás feltételeit teljesíti, azonban felléphet az, hogy kritikus szekción kívül lévő folyamat blokkol egy másikat. Például, tekintsük az alábbi kódrészleteket, ahol `next` mind a két folyamat számára látható egész változó:

```
//process 0
while(1)
{
    while(next != 0) {}
    critical_section();
    next = 1;
    non_critical_section();
}
```

```
//process 1
while(1)
{
    while(next != 1) {}
    critical_section();
    next = 0;
    non_critical_section();
}
```

  Amennyiben `process 1` lassú, nem kritikus szekcióban van, és a `process 0` gyorsan belép a kritikus szekcióba, majd befejezi a nem kritikus szekciót is, akkor a végtelen ciklus következő futásakor nem tud újra belépni a kritikus szekcióba, még akkor sem, ha `process 1` még mindig a nem kritikus szekcióban van. Ezáltal `process 0` saját magát blokkolja.

G.L. Peterson javítása (a szigorú kizáráson): A kritikus szekció előtt minden folyamat meghívja a belépés, majd utána kilépés fv-t.

```
int turn;
int wants_to_enter[2]; // Most csak két processre mutatja be az algoritmust.

// Legyen ez process 0.
while(1)
{
```

```
    enter_critical(0);
    critical_section();
    exit_critical(0);
    non_critical_section();
}

// process 1 hasonlóan
```

Az `enter_critial` és `exit_critical` implementációja:

```
void enter_critical(int entering_process_id)
{
    // Megállapítjuk, hogy ez process 0 vagy process 1-e.
    int other_process = 1 - entering_process_id;

    // Jelezzük, hogy a process be akar lépni a kritikus szekcióba
    wants_to_enter[entering_process_id] = 1;

    // Jelezzük, hogy a soron következő process a belépni kívánó process legyen
    turn = entering_process_id;

    while(turn == entering_process_id && wants_to_enter[other_process])
    {
        // Aktív várakozás.
        // Amíg a másik process is be akar lépni, és még nem jelezte hogy őlesz
        // a soron következő, vagy pedig a másik process nem fejezte be a futását
        // (a wants_to_enter[other_process_id] 0-ra állításával) addig ez a process
        // várakozik.
    }
}

void exit_critical(int exiting_process_id)
{
    // Jelezzük, hogy ez a process már nem akar belépni a kritikus szekcióba.
    wants_to_enter[exiting_process_id] = 0;
}
```

Megfigyelhető, hogy ha két sort felcserélünk az `enter_critical`, már hibás működéshez vezethez az algoritmus:

```
void enter_critical(int entering_process_id)
{
    int other_process = 1 - entering_process_id;
    turn = entering_process_id; // ez a sor
    wants_to_enter[entering_process_id] = 1; // és ez a sor meg van cserélve!
    while(turn == entering_process_id && wants_to_enter[other_process]) {}
}
```

Tegyük fel, hogy ismét `process 0` és `process 1` próbál belépni a kritikus szekcióba. Az ütemező `process 0` utasításait hajtja először végre, és eljut vele az `enter_critical` függvényen belül a `turn = entering_process_id;` sorhoz, így annak értékét 0-ra állítja.
Ismét tegyük fel, hogy az ütemező ekkor átvált `process 1`-re, mely átállítja `turn` értékét 1-re, nem várakozik (hisz `wants_to_enter[0]` értéke még mindig 0), és belép a kritikus szekcióba.
Ismételten tegyük fel, hogy ütemező átvált `process 0`-ra. A ciklusfeltétel itt sem teljesül, hisz `turn` át lett állítva 1-re. Ennek hatására mind `process 0`, mind `process 1` bekerült a kritikus szekcióba.

TSL (Test and Set Lock): megszakíthatatlan atomi művelet, pl. Peterson 1x-bb változata

Alvás-ébredés: Peterson megoldásában a folyamatok végtelen ciklusban várakoznak, mely pazarolja a processzor időt. Erre megoldás lehet, ha blokkoljuk a várakozó folyamatot, és felébresztjük amikor futhat. Ezt hívjuk alvás-ébredésnek.

Szemafor: egyfajta „kritikus szakasz védelem". A szemafor maga egy egész típusú változó, mely „tilosat mutat", ha értéke 0, és 0nál nagyobb értéket, ha a folyamat beléphet a kritikus szekcióba. A szemafor változó módosítása csakis atomi (megszakíthatatlan) műveletekkel történhet, pl. rendszerhívással. Ez implikálja, hogy kizárólag felhasználói szinten nem lehet megvalósítani. (Mi a „baj" a szemaforokkal? - könnyen el lehet rontani a kódolás során).

Elemi művelet: Megszakíthatatlan művelet, mellyel megakadályozható a versenyhelyzet kialakulása. Pl. ilyennek kell lennie a szemafor változó ellenőrzésének, módosításának.

5. Előadás

Monitor: Hasonló a szemaforhoz, de itt eljárások, adatszerkezetek lehetnek. Egy időben csak egy folyamat lehet aktív a monitoron belül. Megvalósítása mutex (lásd: köv. fogalom) segítségével történik. Apró gond: mi van ha egy folyamat nem tud továbbmenni a monitoron belül? Erre jók az állapot változók (condition). Rajtuk két művelet végezhető – wait vagy signal. A monitoros megoldás egy vagy több VPU esetén is jó, de csak egy közös memória használatánál. Ha már önálló saját memóriájuk van a CPU-knak (dedikált memória) akkor ez a megoldás nem az igazi.

Mutex: A mutex egyik jelentése az angol mutual exclusion (kölcsönös kizárás) szóból ered. Programozástechnológiában párhuzamos folyamatok használatakor előfordulhat, hogy két folyamat ugyanazt az erőforrást (resource) egyszerre akarja használni. Ekkor jellemzően felléphet *versengés*. Ennek kiküszöbölésére a gyorsabb folyamat egy, az erőforráshoz tartozó mutexet zárol (ún. lock-ol). Amíg a mutex zárolva van (ezt csak a zároló folyamat tudja feloldani - kivéve speciális eseteket), addig más folyamat nem férhet hozzá a zárolt erőforráshoz. Így az biztonságosan használható. (Például nem lenne szerencsés, ha DVD-írónkat egyszerre két folyamat használná.)
Röviden *a mutex egy bináris szemafor.*

A szemafor és a mutex közti különbség: Az a különbség, hogy míg utóbbi csak kölcsönös kizárást tesz lehetővé, azaz egyszerre mindig pontosan csakis egyetlen feladat számára biztosít hozzáférést az osztott erőforráshoz, addig a szemafort olyan esetekben használják, ahol egynél több - de korlátos számú - feladat számára engedélyezett a párhuzamos hozzáférés.

Üzenetküldés: A folyamatok jellemzően két primitívet használnak: send (célfolyamat, üzenet) és receive(forrás, üzenet) – a forrás tetszőleges is lehet.

Nyugtázó üzenet: Ha a küldő és a fogadó nem azonos gépen van akkor szükséges egy úgynevezett nyugtázó üzenet. Ha ezt a küldő nem kapja meg (azaz úgy tűnik, hogy az üzenet nem érkezett meg), akkor ismét elküldi az üzenetet, ha a nyugta veszik el a küldő újra küld. Ismételt üzenetek megkülönböztetésére sorszámot használ.

Randevú stratégia: Üzenetküldésnél ideiglenes tárolók (más szóval temporális változók) is jönnek létre mindkét helyen (levelesláda). Ezt el lehet hagyni, ekkor a send előtt van receive, a küldő blokkolódik illetve fordítva. - ez a randevú stratégia.

Ütemező (scheduler): Ez az a program, mely eldönti, hogy melyik folyamat fusson (mikor jussanak processzoridőhöz, erőforrásokhoz) egy ütemezési algoritmus alapján.

Folyamat tevékenységei: Egy folyamat jellemzően két tevékenységet szokott végezni: vagy I/O igényt jelent be, azaz írni olvasni akar adott perifériára, vagy számításokat végez. Ez alapján megkülönböztetünk:

- Számításigényes feladat: hosszan dolgozik, keveset vár I/O-ra
- I/O igényes feladat: rövideket dolgozik, hosszan vár I/O-ra

Az ideális ütemező tulajdonságai: Pártatlan, minden folyamat hozzáfér a CPU-hoz, minden folyamatra ugyanazok az elvek érvényesek, minden processzormag azonos terhelést kap. Az ütemezőket továb-

bá 3 kategóriákba sorolhatjuk, minden kategória az előbb említetteken kívül különböző tulajdonságokat próbál megvalósítani:

- Kötegelt rendszerek, ahol további szempont az áteresztőképesség, áthaladási idő és CPU kihasználtság.
- Interaktív rendszerek, ahol további szempont a válaszidő, megfelelés a felhasználói igényeknek.
- Valós idejű rendszerek, ahol további szempont a határidők betartása (erről majd később), adatvesztés, minőségromlás elkerülése.

Kötegelt rendszerek ütemezése (áteresztőképesség, áthaladási idő, CPU kihasználtság):

- **Sorrendi ütemezés** (First Come First Served, FCFS): Egy folyamat addig fut, amíg nem végez vagy nem blokkolódik. Egy pártatlan, egyszerű láncolt listában tartjuk a folyamatokat, ha egy folyamat blokkolódik, akkor a sor végére kerül.
  Nem szakítja meg a már futó, nem blokkolt folyamatokat.
- **Legrövidebb feladat először** (Shortest Job Next, SJN, Illés hibásan SJB-re rövidíti): Kell előre ismerni a futási időket, akkor optimális ha a kezdetben minden folyamat elérhető.
  Nem szakítja meg a már futó, nem blokkolt folyamatokat.
- **Legrövidebb maradék futási idejű következzen**: Minden alkalommal, amikor egy újabb folyamat kerül be a sorba, újrarendezi a sort a hátralevő futási idő szerint. Amennyiben egy új process hátralévő futási ideje a legrövidebb a sorban, a sor elejére kerül.
  Megszakíthatja a már futó folyamatokat is.
- **Háromszintű ütemezés:**
  - Bebocsátó ütemező: a feladatokat válogatva engedi be a memóriába.
  - Lemez ütemező: ha a bebocsátó sok folyamatot enged be és elfogy a memória, akkor lemezre kell írni valamennyit, meg vissza. - ez ritkán fut.
  - CPU ütemező: a korábban említett algoritmusok közül választhatunk.

Interaktív rendszerek ütemezése (válaszidő, megfelelés a felhasználói igényeknek):

- **Körben járó ütemezés** (Round Robin, RR): Minden folyamatnak ad egy időszeletet (valamekkora processzoridőt), aminek a végén, vagy blokkolás esetén jön a következő folyamat. Időszelet végén a lista végére kerül az aktuális folyamat, ami pártatlan és egyszerű. Gyakorlatilag egy listában tároljuk a folyamatokat és ezen megyünk körbe-körbe. A legnagyobb kérdés hogy mekkora legyen egy időszelet? Mivel a process átkapcsolás időigényes, ezért ha kicsi az időszelet sok CPU megy el a kapcsolgatásra, ha túl nagy akkor esetleg az interaktív felhasználóknak lassúnak tűnhet pl. a billentyűkezelés.
- **Prioritásos ütemezés:** Fontosság, prioritás bevezetése, a legmagasabb prioritású futhat. Prioritási osztályokat használ, egy osztályon belül az előbb említett Round Robin fut. Minden 100 időszeletnél újraértékeli a prioritásokat, jellemzően a magas prioritású folyamatok alacsonyabb prioritásra kerülnek, és viszont. Ennek hiányában nagy lenne a kiéheztetés veszélye.
- **Többszörös sorok:** Szintén prioritásos és Round Robinnal működik. A legmagasabb szinten minden folyamat 1 időszeletet kap, az alatta levő 2-t, az alatti 4-et, 16-et, 32-et, 64-et. Ha elhasználta a legmagasabb szintű folyamat az idejét egy szinttel lejjebb kerül.
- **Legrövidebb folyamat előbb:** Hasonlóan működik mint ahogy az a kötegelt rendszernél le volt írva, csak nem tudjuk előre a futási időt, így azt megbecsüljük, és az így kapott eredménnyel dolgozik az algoritmus.
- **Garantált ütemezés:** minden aktív folyamat arányos CPU időt kap, nyilván kell tartani, hogy egy folyamat már mennyi időt kapott, ha valaki arányosan kevesebbet akkor az kerül előre.
- **Sorsjáték ütemezés:** Mint az előző, csak a folyamatok között „sorsjegyeket” osztunk szét, az kapja a vezérlést akinél a kihúzott jegy van.

– **Arányos ütemezés:** Mint a garantált, csak felhasználókra vonatkoztatva.

Valós idejű rendszerek: (határidők betartása, adatvesztés,minőségromlás elkerülése)
Az idő a kulcsszereplő, garantálni kell adott határidőre a tevékenység, válasz megoldását. A programokat kisebb folyamatokra bontják.

– Hard Real Time (szigorú) abszolút nem módosítható határidők.
– Soft Real Time (toleráns) léteznek határidők, de ezek kis méretű elmulasztása tolerálható.

Szálütemezés:

– **Felhasználói szintű szálak:** A kernel nem tud róluk. A folyamat kap egy időszeletet, ezen belül a szálütemező döni el, melyik szál fusson. Gyorsan vált a szálak között. Lehetőség van alkalmazásfüggő szálütemezésre.
– **Kernel szintű szálak:** Kernel ismeri a szálakat, kernel dönt melyik folyamat szála következzen. Lassú váltás, két szál váltása között teljes környezetátkapcsolás kell.

6. Előadás

I/O eszközök:

– **Blokkos eszközök:** Adott méretű blokkokban tárolják az információt, egymástól függetlenül írhatók vagy olvashatók, illetve blokkonként címezhető. Ilyen pl. HDD, és a szalagos egység.
– **Karakteres eszközök:** Nem címezhető, karakterek (bájtok) egybefüggő sorozata.
– **Időzítő:** kivétel, nem blokkos és nem is karakteres.

I/O eszközök és a megszakítások: A megszakítások erősen kötődnek az I/O műveletekhez. Ha egy I/O eszköz „adatközlésre kész”, ezt egy megszakításkéréssel jelzi. Alternatívaként, állapotfigyeléssel meg lehet ezt oldani.

– Állapotbittel: Általában az eszközöknek van állapotbitjük, jelezve, hogy az adat készen van. Ez nem egy hatékony megoldás. Tevékeny várakozás ez is, nem hatékony, ritkán használt.
– Megszakítás kezeléssel:
  • A hardver eszköz jelzi a megszakítás igényt az INTR hardver interrupt-al. Ez egy maszkolható interrupt (azaz figyelmen kívül hagyható).
  • CPU egy következő utasítás végrehajtás előtt, a tevékenységét megszakítja! (Precíz, imprecíz)
  • A kért sorszámú kiszolgáló végrehajtása. A kívánt adat beolvasása, a szorosan hozzátartozó tevékenység elvégzése.
  • Visszatérés a megszakítás előtti állapothoz.

Közvetlen memória elérés (DMA): Tartalmaz: memória cím regisztert, átviteli irány jelzésére, mennyiségre regisztert. Ezeket szabályos I/O portokon lehet elérni.

- Működésének lépései:

1. CPU beállítja a DMA vezérlőt (regisztereket). Ezután a CPU más számításokat végez, nem várja ki a teljes műveletet.
2. A DMA a lemezvezérlőt kéri a megadott műveletre.
3. Miután a lemezvezérlő beolvasta a pufferébe, a rendszersínen keresztül a memóriába(ból) írja (olvassa) az adatot.
4. Lemezvezérlő nyugtázza, hogy kész a kérés teljesítése.
5. DMA megszakítással jelzi, befejezte a műveletet.

I/O szoftverrendszer felépítése: Tipikusan 4 réteggel rendelkezik:

1. Megszakítást kezelő réteg: legalsó kernel szinten kezelt, szemafor blokkolással védve.
2. Eszközmeghajtó programok
3. Eszköz független operációs rendszer program

4. Felhasználói I/O eszközt használó program

Eszközmeghajtó programok (driver): Egy olyan eszközspecifikus kód, mely pontosan ismeri az eszköz jellemzőit, feladata a felette lévő szintről érkező absztrakt kérések kiszolgálása. Kezeli az eszközt I/O portokon, megszakítás kezelésen keresztül.

Holtpont (deadlock): Egy folyamatokból álló halmaz holtpontban van, ha minden folyamat olyan másik eseményre vár, amit csak a halmaz egy másik folyamata okozhat. (Nem csak I/O eszközöknél, hanem jellemző pl. párhuzamos rendszereknél, adatbázisoknál stb.)

Holtpont feltételek: Coffman E.G. szerint 4 feltétel szükséges a holtpont kialakulásához:

1. Kölcsönös kizárás feltétel: minden erőforrás hozzá van rendelve 1 folyamathoz vagy szabad.
2. Birtoklás és várakozás feltétel: Korábban kapott erőforrást birtokló folyamat kérhet újabbat.
3. Megszakíthatatlanság feltétel: Nem lehet egy folyamattól elvenni az erőforrást, csak a folyamat engedheti el.
4. Ciklikus várakozás feltétel: Két vagy több folyamatlánc kialakulása, amiben minden folyamat olyan erőforrásra vár, amit egy másik tart fogva.

Holtpont stratégiák:

1. **A probléma figyelmen kívül hagyása:** Ezt a módszert gyakran strucc algoritmus néven is ismerjük. Kérdés, mit is jelent ez, és milyen gyakori probléma? Vizsgálatok szerint a holtpont probléma és az egyéb (fordító, oprendszer, hardver, szoftver) összeomlások aránya 1:250. A Unix, Windows világ is ezt a „módszert" használja.

2. **Felismerés és helyreállítás:** Engedjük a holtpontot megjelenni (kör), ezt észrevesszük és cselekszünk. Folyamatosan figyeljük az erőforrás igényeket, elengedéseket. Kezeljük az erőforrás gráfot folyamatosan. Ha kör keletkezik, akkor egy körbeli folyamatot megszüntetünk. Másik módszer, nem foglalkozunk az erőforrás gráffal, ha egy bizonyos ideje blokkolt egy folyamat, egyszerűen megszüntetjük.

3. **Megelőzés:** A 4 szükséges feltétel egyikének meghiúsítása. A Coffmanféle 4 feltétel valamelyikére mindig él egy megszorítás.

   – Kölcsönös kizárás. Ha egyetlen erőforrás soha nincs kizárólag 1 folyamathoz rendelve, akkor nincs holtpont se! De ez nehézkes, míg pl. nyomtató használatnál a nyomtató démon megoldja a problémát, de ugyanitt a nyomtató puffer egy lemezterület, itt már kialakulhat holtpont.

   – Ha nem lehet olyan helyzet, hogy erőforrásokat birtokló folyamat további erőforrásra várjon, akkor szintén nincs holtpont. Ezt kétféle módon érhetjük el. Előre kell tudni egy folyamat összes erőforrásigényét. Ha erőforrást akar egy folyamat, először engedje el az összes birtokoltat.

   – A Coffman féle harmadik feltétel a megszakíthatatlanság. Ennek elkerülése eléggé nehéz (pl. nyomtatás közben nem szerencsés a nyomtatót másnak adni).

   – Negyedik feltétel a ciklikus várakozás már könnyebben megszüntethető. Egy egyszerű módszer erre, ha minden folyamat egyszerre csak 1 erőforrást birtokolhat. Egy másik módszer: Sorszámozzuk az erőforrásokat, és a folyamatok csak ezen sorrendben kérhetik az erőforrásokat.
     Ez jó elkerülési mód, csak megfelelő sorrend nincs!

4. **Dinamikus elkerülés:** Erőforrások foglalása csak „óvatosan". Van olyan módszer amivel elkerülhetjük a holtpontot? Igen, ha bizonyos info (erőforrás) előre ismert. Bankár algoritmus (Dijkstra,1965) Mint a kisvárosi bankár hitelezési gyakorlata. Biztonságos állapotok, olyan helyzetek, melyekből létezik olyan kezdődő állapotsorozat, melynek eredményeként mindegyik folyamat megkapja a kívánt erőforrásokat és befejeződik.

Bankár algoritmus több erőforrás típus esetén:

Az 1 erőforrás elvet alkalmazzuk. Jelölések:

- $F(i,j)$ az i. folyamat j. erőforrás aktuális foglalása
- $M(i,j)$ az i. folyamat j. erőforrásra még fennálló igénye
- $E(j)$ a rendelkezésre álló összes erőforrás.
- $S(j)$ a rendelkezésre álló szabad erőforrás.

Az algoritmus:

1. Keressünk i. sort, hogy $M(i,j) \leq S(j)$, ha nincs ilyen akkor holtpont van, mert egy folyamat se tud végigfutni.
2. Az i. folyamat megkap mindent, lefut, majd az erőforrás foglalásait adjuk $S(j)$-hez
3. Ismételjük 1,2 pontokat míg vagy befejeződnek, vagy holtpontra jutnak.

7. Előadás

Alapvető memóriakezelés: Kétféle algoritmus csoport létezik: Swap (szükséges a folyamatok mozgatása, cseréje a memória és a lemez között, amennyiben nincs elég memria, éa lemezt is igénybe kell venni) vagy nincs szükség, ha elegendő a memória)

Monoprogramozás: Egyszerre egy program fut. Nincs szükség progrmakat ütemező algoritmusra. Pl. bash-ben is ez történik.

Multi programozás: Párhuzamosan több program.A memóriát valahogy meg kell osztani a folyamatok között. Ez megvalósítható

- **Rögzített memória szeletekkel:** Osszuk fel a memóriát n (nem egyenlő) szeletre (Fix szeletek). Ezt például rendszerindításnál meg lehet tenni. Vagy van egy közös várakozási sor, mely leosztja hogy melyik feladat melyik memóriacímre jusson, vagy minden szeletre külön-külön sor adott. Kötegelt rendszerek tipikus megoldása.
- **Memória csere használattal:** időosztályos. Nincs rögzített memória partíció, mindegyik dinamikusan változik, ahogy az op. Rendszer oda-vissza rakosgatja a folyamatokat. Dinamikus, jobb memória kihasználtságú lesz a rendszer, de a sok csere lyukakat hoz létre! Memória tömörítést kell végezni! (Sok esetben ez az időveszteség nem megengedhető!).
  Grafikus felület esetén nem a legjobb megoldás.
- **Virtuális memória használatával**
- **Szegmentálással**

Dinamikus memória foglalás: Általában nem ismert, hogy egy programnak mennyi dinamikus adatra, veremterületre van szüksége. A program „kód" része fix szeletet kap, míg az adat és verem (stack) része változót. Ezek tudnak nőni (csökkenni). Ha elfogy a memória, akkor a folyamat leáll, vár a folytatásra, vagy kikerül a lemezre, hogy a többi még futó folyamat memóriához jusson. Ha van a memóriában már várakozó folyamat, az is cserére kerülhet.

Dinamikus memória nyilvántartása: Allokációs egység definiálunk először: ennek mérete kérdéses: ha kicsi akkor kevésbé lyukasodik a memória, viszont nagy a nyilvántartási „erőforrás (memória) igény". Ha nagy, akkor túl sok lesz az egységen belüli maradékokból adódó memória veszteség. A nyilvántartás megvalósítása megtörténhet bittérkép használattal vagy láncolt listával.

Memóriafoglalási stratégiák:

- First Fit (első helyre, ahova befér, leggyorsabb, legegyszerűbb)
- Next Fit (nem az elejéről, hanem az előző befejezési pontjából indul a keresés, kevésbé hatékony mint a first fit)
- Best Fit (lassú, sok kis lyukat produkál)
- Worst Fit (nem lesz sok kis lyuk, de nem hatékony)
- Quick Fit (méretek szerinti lyuklista, a lyukak összevonása költséges)

Virtuális memória: Egy program használhat több memóriát mint a rendelkezésre álló fizikai méret. Az operációs rendszer csak a „szükséges részt” tartja a fizikai memóriában. (A definíció erősen kötődik a következőhöz (MMU)!)

Sajnos innentől az anyag nincs tovább javítva, mert a dia teljesen érthetetlen, és baromi sok munka lenne azt a részt feldolgozni nagy valószínűséggel plusz 1 vagy 2 pontért:(

MMU (Memória Menedzsment Unit): virtuális címtér lapokra osztva (tábla). Jelenlét/hiány bit. Figyeli, hogy minden lap a memóriában van-e, ha nem akkor laphiba, majd operációs rendszer lapkeretet kitesz és behozza a lapot. Pl.16bit-4kb lap (lap=2^n, 16 bites virtuális címből 4 a lapszám, a többi offset; 1 jelenlét/hiány bit jelzésére; kimenő 15 fizikai címsínre.) 32 bit esetén 12bit(4kb) lapméret, és 20 bit a laptábla mérete (1MB=1 millió elem) minden folyamathoz saját címtér, laptábla 64 bit esetén ilyen méretű laptábla megvalósíthatatlan 2 szintű laptábla: 10 bit felső szintű és 10 bit második szintű, lapon belüli offset 12 bit Táblabejegyzés szerkezete: lapkeret száma, jelenlét/hiány bit, védelmi bit (=0:RW, =1:R), dirty bit(módosítás, =1: módosult a lapkeret memória, ki kell írni), Hivatkozás bit =1:hivatkoznak a lapra, Gyorsító tár tiltás bit: (fizikai memó=I/O eszköz adatterület)

TLB (asszociatív memória): MMU-ba kicsi HW egység, kevés bejegyzéssel, szoftveres TLB kezelés, 64 elem miatt a HW megoldás kispórolható

Invertált laptáblák: valós memória méret, laptáblában fizikai memóriából keletkező számú elem, TLB használata, ha hiba, akkor invertált laptáblában keresünk, TLB-be rakjuk.

Lapcserélési algoritmusok: ha nincs virtuális című lap a memóriában, egy lapot kidobni, másikat berakni. Optimális: címkézés, ahány CPU utasítás hajtódik végre hivatkozás előtt, legkisebb számú lap kidobni (megvalósíthatatlan)

- NRU: Modify and Reference bit használata, modify időnként 0-ra, 0-3.osztály: (nem,nem; nem,igen; igen,nem; igen,igen) megfelelő eredmény, nem hatékony
- FIFO: legrégibb eldob, ha új kell (előre jön, végéről megy). Javítása a Második lehetőség: ha hiv.bit 1->sor eleje bit=0
- Óra: Második Lehetőséghez hasonló, mutatóval járjunk körbe, legrégebbi lapra mutat, ha hiv.bit=1->0, továbblépünk
- LRU: legrégibb algoritmus. HW vagy SW megvalósítás.
- NFU: lapokhoz számláló, ehhez referencia bitet adunk, óramegszakításkor, legkisebb értékűt dobjuk el, nem felejt!

Munkahalmaz modell: előlapozás, használtak a fizikai memóriában tartva, lapnyilvántartás (Óra javítása: WSClock)

Lokális, globális helyfoglalás: laphibánál hogyan vizsgáljuk, méret szerinti lap-dinamikus, PFF alg. laphiba/mp (terhelosztás)

Helyes lapméret meghatározása: kicsi (lapveszteség kicsi, nagy laptábla), nagy (fordítva) n*512 bájt a lapméret

Szegmentálás: virtuális memó (1D címtér, 0-tól maxig), több progi dinamikus területtel, egymástól független címtér (szegmens), cím 2 része (szegmens szám, ezen belüli cím), egyszerű osztott könyvtárak, logikailag tagolható (adat és kód), védelmi szint egy szegmensre, fix lapméret, változó szegmensméret

Pentium processzor virtuális címkezelése: sok szegmens: 2^{32} bite (4 GB), 16000 LDT (folyamatonként) GDT(Globbal Descriptor Table 1db) fizikai cím: szelektor+offset művelet szegmens elérés: 16 bites szegmens szelektor Lineáris cím TLB a gyors lapkeret eléréshez védelmi szintek: 0-3 (Kernel,Rendszerhívások, Osztott könyvtárak, Felhasználói programok) alacsonyabbról adatelérés engedélyezett, fordítva tiltott eljárás hívása ellenőrzött módon felhasználói programok osztott könyvtárak adatait elérhetik, de nem módosíthatják. Fontosak még: