Operációs rendszerek 1. – 8. előadás
Multiprogramozott operációs rendszerek

Soós Sándor

Nyugat-magyarországi Egyetem
Simonyi Károly Műszaki, Faanyagtudományi és Művészeti Kar
Informatikai és Gazdasági Intézet

E-mail: soossandor@inf.nyme.hu


Tartalomjegyzék

1. Multiprogramozott operációs rendszerek
   - Multiprogramozás
   - Processzorütemezés
   - Tárkezelés
2. Befejezés
   - Emlékeztető kérdések


Hol tartunk?

1. Multiprogramozott operációs rendszerek
   - Multiprogramozás
   - Processzorütemezés
   - Tárkezelés
2. Befejezés
   - Emlékeztető kérdések


Multiprogramozás, I

A multiprogramozás fogalma:

- Mikor alakult ki?
- Miért alakult ki?
- Mi a lényege?
- Milyen feltételei vannak?
- Manapság használjuk?


Multiprogramozás, II

- A számítógépek 3. generációjának idején alakult ki az igény és a technikai lehetőség
- Használjuk ki a gyors processzort akkor is, amikor a lassú perifériás műveletek zajlanak
- Több program futtatása „egyszerre”
- Mit jelent az idézőjel?


A multiprogramozás működési elve

- Egyszerre több program futhat (virtuálisan)
- Egy processzor egy adott pillanatban azonban csak egy feladaton dolgozhat
- (Napjainkban a többmagú processzorok ebből a szempontból több processzornak számítanak)
- Egy folyamat addig fut a processzoron, amíg várakozásra nem kényszerül, vagy le nem telik a számára kiosztott időszelet (preemptív rendszer)
- Ilyenkor a folyamat állapota mentésre kerül
- Amelyik folyamatnak teljesül a továbbfutási feltétele, az bekerül a választhatók listájába
- A processzor kiválasztja valamelyik folyamatot a listából, visszaállítja az eltárolt adatait, és továbbindítja


Mit kell megvalósítani ehhez?

- Választani kell a folyamatok közül ⇒ **CPU-ütemezés**
- Több program osztozkodik a táron ⇒ **Tárgazdálkodás**
- Meg kell osztani a rendszer erőforrásait a folyamatok között, szükség esetén garantálni kell az időbeli korlátok betartását (kölcsönös kizárás, randevú, sorrend), kezelni kell a holtponthelyzeteket ⇒ **Erőforrás-gazdálkodás**
- Biztosítani kell, hogy a folyamatok ne zavarják egymást és a rendszer működését ⇒ **Védelem**
- El kell rejteni a rendszer fizikai részleteit, felhasználóbarát kezelhetőség ⇒ **Virtuális gép**
- Kommunikációs felület más számítógépek és programok felé ⇒ **Hálózatkezelés**

Mivel nem foglalkoztunk még ezek közül:

CPU-ütemezés

Hol tartunk?

1. Multiprogramozott operációs rendszerek
   - Multiprogramozás
   - Processzorütemezés
   - Tárkezelés
2. Befejezés
   - Emlékeztető kérdések


Folyamatkezelés (emlékeztető)

- Folyamatmodell:
  - Minden folyamatnak saját processzora és saját memóriája van
- A valós helyzet:
  - Egyetlen processzor és egyetlen memória van a számítógépben
- Az operációs rendszer feladata:
  - A valóságos környezet felhasználásával virtuális környezet biztosítása a folyamatok számára
  - logikai processzor
  - logikai memória


Logikai memória

- Amikor fut a folyamat, megkapja a fizikai memória egy részét
- Ebben helyezkedik el a kód és az adatok egy része, amennyi elfér
- A többi a háttértáron marad
- Amikor szükség van rá, akkor betöltődik a memóriába, de mivel ez sokáig tarthat, a folyamatnak várakoznia kell, közben más futhat helyette
- Kérdés:
  - Hogyan oldjuk meg ezeket a cseréket, mi kerüljön ki és mi töltődjön be?
  - Erről rendelkeznek a lapcserélési algoritmusok


Logikai processzor

- Párhuzamos futtatás
  - A fizikai processzor felváltva futtatja a logikai processzorokat
  - Ha elég gyakran és gyorsan történik a váltás, akkor úgy tűnik, mintha párhuzamosan működnének a logikai processzorok, és így futnának a folyamatok
- Megvalósítás:
  - A valós processzorhoz kapcsolunk egy várakozási sort
  - Amikor felszabadul a processzor, akkor ebből a sorból választjuk ki a következő futó folyamatot valamilyen algoritmus szerint
- Kérdések:
  - Meddig futhat egy folyamat?
  - Hogyan válasszuk ki a következőt?
  - Erről rendelkeznek a CPU-ütemező algoritmusok

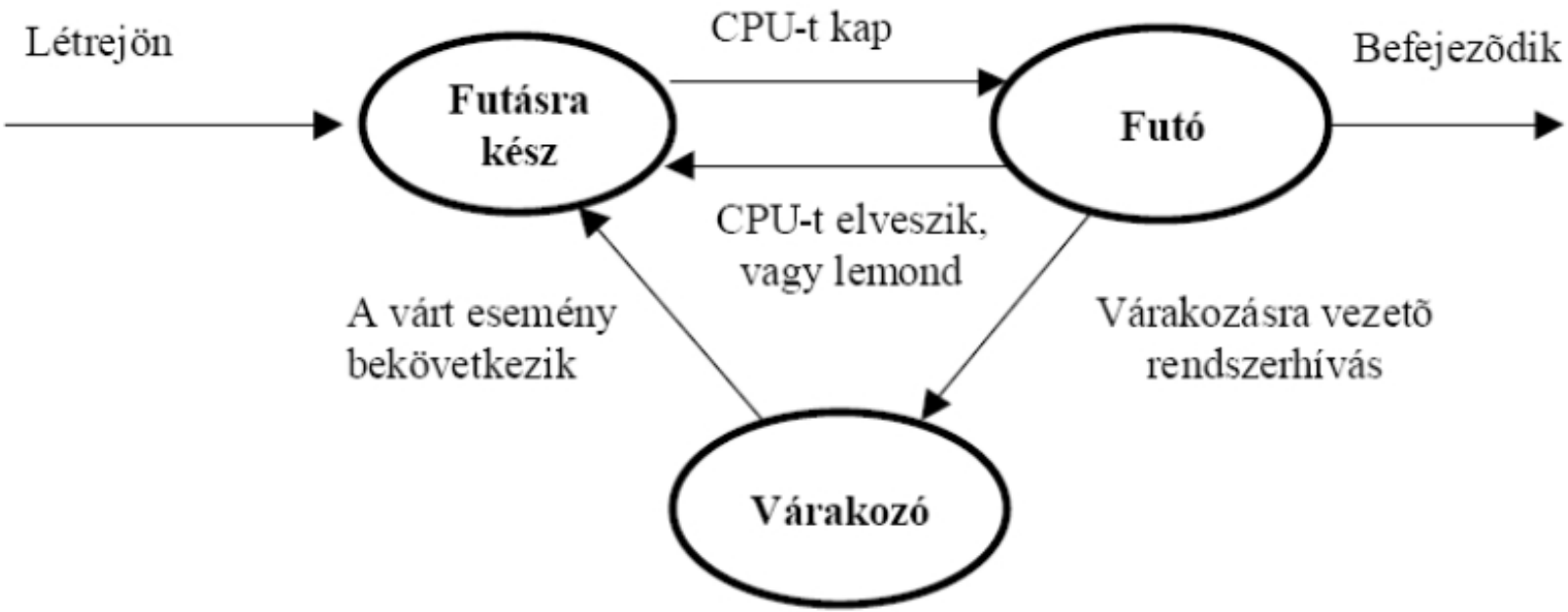

A folyamatok felépítése

- A folyamatok futása során kétféle tevékenység váltakozik
  1. **Processzorlöket (CPU-burst)**: processzor által végrehajtandó utasítássorozat
  2. **Be-/kiviteli löket (I/O-burst)**: a processzor nélkül, a perifériák által végrehajtandó utasítássorozat
- Amikor a folyamat I/O-lökethez ér, elindíthatunk egy másik folyamatot
- Az egyes folyamatok eltérő arányban tartalmazzák a löketeket, vannak CPU-intenzív és I/O intenzív folyamatok
- A rendszer eredő teljesítménye akkor a legjobb, ha a kétféle folyamat nagyjából egyelő arányban fordul elő
- Általában ezt nem tudjuk befolyásolni, de ha van rá mód, akkor érdemes figyelni erre

Állapotmodell

- Állapotmodell
  - A folyamatok minden pillanatban három különböző állapot valamelyikében lehetnek:
    1. Fut (mindig csak egy folyamat lehet ebben az állapotban)
    2. Futásra kész
    3. Várakozik
- Az állapotok meghatározott rend szerint válthatják egymást
- Ezt a rendet az állapot-átmenet diagrammal tudjuk ábrázolni
- Ezt az eszközt más esetekben is használhatjuk, pl. programozás közben


Állapot-átmenet diagram, I



Állapot-átmenet diagram, II

- Állapotátmenetek:
  - **Létrejön**
    - Elindul a folyamat, megkapja a logikai memóriát, betöltődik a kezdéshez szükséges kód és az adatok
    - A folyamat futásra kész állapotba kerül
  - **CPU-t kap**
    - Az ütemező kiválasztotta a folyamatot, megkezdődik a végrehajtás
    - A folyamat futó állapotba kerül
  - **CPU-t elveszik, vagy lemond**
    - Letelt a folyamat számára engedélyezett időszelet, vagy önként lemond a futásról, de bármikor folytatható a futása
    - A rendszer elmenti a folyamat állapotát
    - A folyamat futásra kész állapotba kerül

Állapot-átmenet diagram, III

- **Várakozásra vezető rendszerhívás**
  - A folyamat végrehajtott egy olyan rendszerhívást, ami várakozást eredményez
  - A rendszer elmenti a folyamat állapotát
  - A folyamat várakozó állapotba kerül
- **A várt esemény bekövetkezik**
  - Véget ért a várakozást kiváltó tevékenység
  - A folyamat futásra kész állapotba kerül
- **Befejeződik**
  - A folyamat befejeződött
  - Végrehajtja az ekkor szükséges rendszerhívást

- Ez egy alapmodell, ennél bonyolultabb rendszerek is léteznek, több állapottal és átmenettel

Processzorütemezés, I

- A kernel része
- Állandóan a memóriában van
- Nagyon gyorsnak kell lennie, mert nagyon gyakran futnia kell
- Kétféle ütemező létezik:
  - **Preemptív**: Megszakíthat egy éppen futó folyamatot
  - **Nem preemptív**: Futó folyamatot nem szakíthat meg, a folyamat csak maga függesztheti fel magát


Processzorütemezés, II

- Mikor történik ütemezés?
  - Az éppen futó folyamat befejeződik
  - Az éppen futó folyamat várakozó állapotba kerül
  - Egy folyamat futásra kész állapotba kerül
  - A futó folyamat újraütemezést kér, vagy preemptív ütemezés esetén elveszti a futási jogot


Processzorütemezés, III

- Milyen elvárásaink vannak az ütemezővel szemben?
  - Kezeljen prioritásokat
  - Ne legyen éhezés
  - Bármekkora terhelés esetén garantáljon felső korlátokat bizonyos paraméterekre, pl. várakozási idő, átfutási
  - Engedje előre a nem használt erőforrásokat igénylő folyamatokat
  - Minél kevesebb adminisztrációra legyen szükség
  - Elegáns visszaesés (graceful degradation): a terhelés növekedésével legfeljebb lineárisan csökkenjen a teljesítmény, ne omoljon össze a rendszer
- Ezek a követelmények ellentmondanak egymásnak, nem lehet egyszerre kielégíteni ezeket
- Ezért van sok különböző ütemezési algoritmus a piacon


Processzorütemezés, IV

- Teljesítményelemzés, milyen szempontok szerint minősítjük az ütemező algoritmusokat?
  - Processzor kihasználtság:

$$\frac{hasznos\_ido}{osszes\_ido} \times 100\ [\%]$$

    - *hasznos_ido*: A teljes idő és a tétlenül töltött idő (adminisztráció és egyéb járulékos tevékenységek, overhead) különbsége
    - Ma tipikusan $40 - 90\%$ között van
  - Átbocsátóképesség:

$$\frac{elvegzett\_munkak\_szama}{ido} \quad \left[\frac{1}{s}, \frac{1}{min}, \frac{1}{h}\right]$$

    - Értéke tág határok között mozog


Processzorütemezés, V

- Körülfordulási (átfutási) idő:

$$vegrehajtasi\_ido + varakozasi\_ido$$

- Várakozási idő:

$$\begin{gathered} utemezesek\_ideje + \\ felfuggesztes\_ideje + \\ futasra\_kesz\_allapot\_ideje + \\ varakozas\_ideje + \\ egyeb\_nem\_hasznos\_idok \end{gathered}$$

  - Mivel a végrehajtás ideje egy adott rendszerben állandó, ezért elég ezt figyelni

- Válaszidő:
  - Időosztásos rendszerekben egy kezelői parancs kiadásától a válasz megérkezéséig eltelt idő

Egyszerű ütemezések, I

- Legrégebben várakozó (**FCFS - First Come First Served**)
  - Nem preemptív
  - Nagyon egyszerűen implementálható
  - Nagy lehet az átlagos várakozási idő
    - Konvoj hatás: egy hosszú folyamatot végig kell várni a többieknek a processzornál és a perifériáknál is
    - Példa: csekkbefizetés a postán
- Körbeforgó (**RR - Round Robin**)
  - Preemptív
  - A folyamat legfeljebb egy adott ideig futhat, utána a sor végére állítjuk
  - Kritikus kérdés az időszelet hosszának beállítása
    - Ha túl hosszú: gyakorlatilag FCFS-sé válik
    - Ha túl rövid: túl sok környezetváltás

Egyszerű ütemezések, II

- Gyakorlati ökölszabály: a folyamatok 80%-a legyen rövidebb az időszeletnél

Egyszerű ütemezések, III

- **Prioritásos** ütemezések
  - A várakozók közül a legfontosabb (legmagasabb prioritású) folyamatot választjuk
  - A prioritás lehet:
    - **külső** (a kezelő adja), vagy **belső** (a rendszer adja)
    - **statikus** (időben állandó), vagy **dinamikus** (időben változhat)
  - A következő prioritásos algoritmusok esetében a processzorlöket hossza lesz a belső prioritási szempont
  - Legrövidebb löketidejű (**SJF - Shortest Job First**)
    - Nem preemptív
    - A legrövidebb becsült löketidejű folyamatot választja a várakozók közül
    - Becslésen alapszik a korábbi adatok, vagy a felhasználó állítása alapján
    - Kiküszöböli a konvojhatást, az átlagos várakozási idő és a körülfordulási idő is optimális


Egyszerű ütemezések, IV

- **Prioritásos** ütemezések (folytatás)
  - Legrövidebb hátralévő idejű (**SRTF - Shortest Remaining Time First**)
    - Preemptív SJF
    - Mikor egy folyamat felébred, újraütemez, és a legrövidebbet választja (figyelembevéve a környezetváltás plusz időigényét)
  - Legjobb válaszarány (**HRR - Highest Response Ratio**)
    - A prioritás meghatározása:

$$\frac{loketido + k \times varakozasi\_ido}{loketido}$$

k egy alkalmas konstans

    - Öregíti a régebb óta várakozó folyamatokat, így elkerülve az éhezést

Többszintű ütemezések, I

- A különböző prioritási szintekhez külön-külön sorokat definiálhatunk
- Az egyes sorok más-más stratégiával működhetnek

- Statikus többszintű sorok (**SMQ - Static Multilevel Queues**)
  - A folyamatok induláskor bekerülnek valamelyik várakozási sorba, és végig ott is maradnak
  - Gond az éhezés!


Többszintű ütemezések, II

- Visszacsatolt többszintű sorok (**MFQ - Multilevel Feedback Queues**)
  - Az éhezés kiküszöbölésére dinamikus prioritásokat használ, így a folyamatok mozoghatnak a sorok között
  - A magasabb prioritású sorok egyre kisebb időszeletű RR (robin Round) ütemezést használnak, a legkisebb pedig FCFS-t
  - Az új folyamatok a legnagyobb prioritású sorba kerülnek, de ha átlépik az időszeletet, akkor egy szinttel lejjebb kerülnek
  - Később feljebb is kerülhetnek pl. az átlagos löketidő alapján
  - A régóta bentlévő folyamatok prioritása növelhető


Többprocesszoros ütemezések

- Két különböző eset: heterogén, vagy homogén processzorok
  - Heterogén (eltérő felépítésű) processzorok
    - Minden folyamat csak a neki megfelelő processzoron futhat, másikon nem
    - Ilyenkor egymás mellett futó egyprocesszoros rendszerekről van szó
  - Homogén (azonos) processzorok
    - Bármelyik processzoron futhatnak a folyamatok
    - Közös várakozási sor kezeli az összes processzort és folyamatot
    - Hogyan kezeljük ezt a közös sort?
      Aszimmetrikus rendszerekben van egy kitüntetett processzor, amelyik futtatja az ütemezést, szétosztja a feladatokat
      Szimmetrikus rendszerekben mindegyik processzoron fut ütemezés, a várakozási sort kölcsönös kizárással kezelik


Hol tartunk?

1. Multiprogramozott operációs rendszerek
   - Multiprogramozás
   - Processzorütemezés
   - Tárkezelés
2. Befejezés
   - Emlékeztető kérdések


Tárkezelés, I

- Eddig a processzorokkal foglalkoztunk
- Ahhoz, hogy a processzor futtatni tudjon egy programot, annak kódját és a hozzá tartozó adatokat a memóriában kell tartani
- Multiprogramozás esetén több programnak is a memóriában kell lennie egyidőben
- Problémák:
  1. A memória túl kicsi ahhoz, hogy mindig minden szükséges adat elférjen benne
  2. A leggyorsabb háttértár (merevlemez) is nagyságrendekkel lassabb, mint a memória

Tárkezelés, II

- Megoldás:
  - *„Dolgozzunk a processzor keze alá”*
  - Oldjuk meg azt, hogy mindig a szükséges adatok legyenek a memóriában, a már/még nem szükségeseket töröljük ki onnan
  - Mindezt úgy oldjuk meg, hogy a processzornak ne kelljen várakoznia
- Milyen részfeladatokat kell megoldanunk ehhez
  - Társzervezés
    - Az a mód, ahogyan a memóriát megosztjuk a felhasználók között
  - Tárkialakítási módszerek
    - Hogyan szervezzük a memóriát?
  - Virtuális tárkezelés
    - A felhasználó elől elrejtjük, hogy mekkora a fizikai memória


Társzervezés, I

- A társzervezés az a mód, ahogyan a memóriát megosztjuk a felhasználók között
- A következő kérdésekre keresünk választ:
  - Hány felhasználó férhet hozzá, egy vagy több?
  - Több felhasználó esetén egyforma, vagy különböző méretű részeket (partíciókat) kapnak?
  - A partíció mérete futás közben változhat-e?
  - A terület egybefüggő, vagy darabokból állhat?


Társzervezés, II

- Egypartíciós rendszer
  - Csak egy felhasználói folyamat lehet a memóriában
  - A folyamat az első szabad címtől kezdődően helyezkedhet el
  - Az operációs rendszer védelme
    - Egy regiszter mutatja a program által használható memória határát, ennek értéke csak rendszermódban változtatható
  - Folyamatok váltása
    - Tárcserével (swapping)
  - Probléma:
    - Nem hatékony a multiprogramozás megvalósítása


Társzervezés, III

- Többpartíciós rendszer
  - Rögzített méretű partíciók
    - Különböző, de rögzített méretű partíciók vannak a rendszerben
    - Minden partíciót csak egy folyamat birtokolhat
    - Csak olyan partícióba kerülhet bele a folyamat, amiben elfér
    - A multiprogramozás fokát a partíciók száma korlátozza
    - A folyamatok a nekik megfelelő méretű partíciókhoz rendelt várakozási sorokban várakoznak
    - Védelem: alsó és felső határregiszter
    - Probléma: belső tördelődés (nem használt terület a partíciók végén)


Társzervezés, IV

- Változó méretű partíciók
  - Nincsenek előre rögzített partíciók
  - Amikor betöltünk egy folyamatot, akkor kap egy megfelelő méretű partíciót, nem nagyobbat, így nincsen belső tördelődés
  - Probléma: külső tördelődés (a partíciók közötti nem használt terület)
  - Megoldás: szemétgyűjtés (garbage collection), azaz a szabad területek egyesítése (Lásd PROG2: Java)
  - Probléma: a szemétgyűjtés nagyon idő- és erőforrásigényes
  - Másik megoldás: ügyes tárfoglalási stratégiák


Társzervezés, V

Tárfoglalási stratégiák

- A feladat:
  - Egy adott pillanatban hogy néz ki a memória?
  - Foglalt és szabad területek váltakoznak
  - Mindegyik eltérő hosszúságú
  - Ekkor kell helyet találni a következő folyamat számára
  - Különböző szempontok alapján választhatunk


Társzervezés, VI

- Első megfelelő (**First Fit**)
  - Gyors, de átlagosan a memória 30%-a kihasználatlan marad
- Következő megfelelő (**Next Fit**)
  - Hasonló hatásfokú, mint a First Fit
- Legjobban megfelelő (**Best Fit**)
  - A lyukakat próbálja minimalizálni, de lassabb
- Legkevésbé megfelelő (**Worst Fit**)
  - A fennmaradó szabad területet maximalizálja
  - A legrosszabb eredményt adja, az összmemória kb. fele kihasználatlan marad


Tárcsere, I

- Adatmozgatás a memória és a háttértár között (swapping)
- Egy folyamat teljes memóriaterületét kiírjuk a háttértárra, így helyet biztosítunk mások számára
- Vagy betöltjük a háttértárról egy szabad területre
- Időigényes, mert sok adatot kell mozgatni a gyors memória és a lassú háttértár között
- Optimalizálni kell a műveletet, minimalizálni kell a lapcserék számát


Tárcsere, II

- Optimalizálási lehetőségek:
  - Olyan folyamatot válasszunk a futásra készek közül, amelyik éppen a memóriában van
  - Ha olyan folyamatot írunk ki a háttértárra, aminek a memóriaképe nem változott a háttértárhoz képest, akkor nem kell másolni, elég törölni. Ehhez folyamatosan adminisztrálni kell a módosításokat
  - Átlapolt lapcsere:
    - A lapcserével ne várjuk meg, amíg véget ér az előző folyamat
    - Miközben még fut a folyamat, írjunk ki egy másikat a háttértárra és olvassunk be egy másikat


Tárkialakítási módszerek, I

1. Szegmensszervezés
   - A program különböző logikai részeit (szegmenseit) egy-egy blokknak feleltetjük meg (kódszegmens(ek), adatszegmens(ek), veremszegmens, stb.
   - A szegmensek különböző méretűek lehetnek, ezért nincs belső tördelődés
   - Blokktábla tárolja az egyes blokkok adatait:
     - a blokk mérete
     - egy bit mutatja, hogy a memóriában van-e
     - hozzáférési információk, ki írhatja, olvashatja, hajthatja végre

Tárkialakítási módszerek, II

2. Lapszervezés
   - Azonos méretű blokkokra (lap) osztjuk a memóriát
   - Nincsen külső tördelődés, viszont van belső, átlagosan fél lap / folyamat
   - A folyamatokat egy vagy több lapba töltjük a méretétől függően
   - A laptáblában tartjuk nyilván a lapok adatait:
     - melyik folyamathoz tartozik
     - memóriában van-e
     - hozzáférési információk
   - Mekkora legyen a lap mérete?
     - Ha nagy, akkor nagy a belső veszteség
     - Ha kicsi, akkor csökken a veszteség, de nagyobb a laptábla, lassabb a címzés, és lassabb az adatmozgatás (mert kisebb blokkokban történik)
     - A lap mérete mindig 2 hatvány, 512 bájt és 16 kilobájt között

Tárkialakítási módszerek, III

3. Kombinált szervezés
   - Szegmensszervezés és lapszervezés együtt
   - A szegmensekhez lapokat rendelünk
   - A szegmenstábla laptáblák címeit tartalmazza
   - A laptábla tartalmazza a lapok címeit
   - Nincsen külső tördelődés
   - Belső tördelődés szegmensenként van

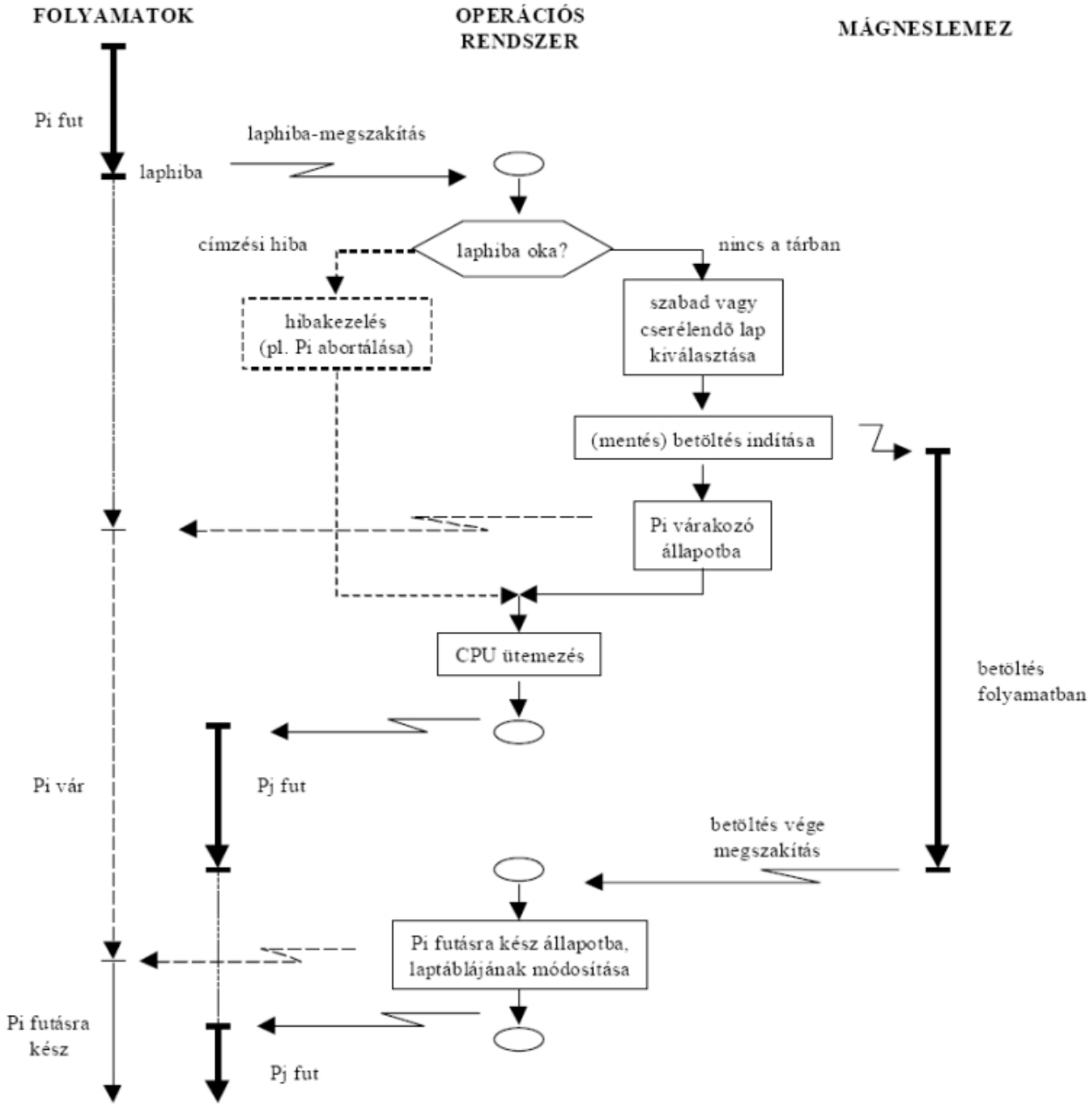

Virtuális tárkezelés, I

- Az operációs rendszer gondoskodik róla, hogy a folyamatoknak csak az éppen szükséges része tartózkodjon a memóriában, mégis lehessen hivatkozni az egészre
- Az operációs rendszer megoldja, hogy ilyenkor automatikusan betöltésre kerüljenek a szükséges dolgok
- Miért van lehetőség erre?
  - A programoknak vannak ritkán, vagy soha nem futó részeik
  - A programozó gyakran a szükségesnél nagyobb memóriaterületet foglal
  - Az egyes programrészek más-más időpontban futnak, nincs rájuk egyszerre szükség
  - Lokalitási elv:
    - Egy rövidebb időintervallumban általában egymáshoz közeli utasításokra és adatokra van szükség


Virtuális tárkezelés, II

- Miért jó a virtuális tárkezelés?
  - A fizikai memóriánál nagyobb programok is futtathatók
  - Nő a multiprogramozás foka, ezzel nő a rendszer teljesítménye
  - Gyorsabb lehet a rendszer, mert egyszerre kevesebb adatot kell mozgatni
- Hogyan működik a virtuális tárkezelés?
  - Ha olyan címre hivatkozik a folyamat, ami nincs a memóriában, akkor hibamegszakítás történik (hardvertámogatás)
  - A rendszer felfüggeszti a folyamatot, elmenti az állapotát
  - Ha a virtuális memória okozta a megszakítást, akkor elindítja a szükséges terület betöltését (ha kell, előtte helyet csinál)
  - Újraütemez
  - A betöltés végén ismét megszakítás történik
  - Preemptív rendszer esetén azonnal, egyébként később újraütemez


A laphiba kezelésének folyamata



A virtuális tárkezelés működése, I

- Eldöntendő kérdések
  - Mikor melyik lapot hozzuk be?
  - Melyik lap helyére hozzuk be?
  - Hány lapot kapjon egy folyamat?


A virtuális tárkezelés működése, II

- Mikor melyik lapot hozzuk be?
  - **Igény szerinti lapozás**: csak a kért lapot hozzuk be
    - egyszerű
    - gyors
    - nem hoz be felesleges lapot
    - gyakoribb a laphiba
  - **Előretekintő lapozás**: több lapot hoz be jóslás alapján
    - bonyolultabb
    - több lapot mozgat
    - felesleges lapokat is behozhat
    - ritkább lehet a laphiba


A virtuális tárkezelés működése, III

- Melyik lapot vigyük ki?
  - Amelyik a legkevésbé fog kelleni
  - Amelyikre a legkésőbb lesz újra szükség
  - Ötletek:
    - Ha olyan lapot viszünk ki, akkor nem kell írni, elég törölni. Ezt jelzi a Modified bit
    - Ha egy lap nemrég használatban volt, akkor valószínű, hogy újra szükség lesz rá. Ezt jelzi a Referenced bit. Ezt bizonyos esetekben törölni kell (egy idő után)


Lapcsere algoritmusok, I

- **Optimális algoritmus**
  - A ténylegesen legkésőbb szükséges lapot választja ki
  - Így lenne legkisebb a laphibák száma
  - A gyakorlatban megvalósíthatatlan, mert előre végre kellene hajtani a programot
  - Összehasonlítási alapnak jó, ehhez hasonlíthatjuk a gyakorlati algoritmusokat
- **Legrégebbi lap algoritmus** (FIFO - First In First Out)
  - A legrégebben behozott lapot viszi ki
  - ⊕ egyszerű
  - ⊖ a gyakran használt lapokat is kiviszi

Lapcsere algoritmusok, II

- **Újabb esély algoritmus** (SC - Second Chance)
  - FIFO elven működik, de figyeli a Referenced bitet is
    - Ha 1 (használták a lapot), akkor a lista végére teszi (újabb esély) és törli a bitet
    - Ha 0, akkor kiválasztja a lapot
  - Így a gyakran használt lapok soha nem kerülnek ki
- **Legrégebben nem használt algoritmus** (LRU - Least Recently Used)
  - Azt választja ki, amelyet a bentlévők közül a legrégebben használtak
  - ⊖ hardvertámogatás kell hozzá
  - ⊖ bonyolult
  - ⊕ ez közelíti legjobban az optimális algoritmust

Lapcsere algoritmusok, III

- **Legritkábban használt algoritmus** (LFU - Least Frequently Used)
  - Alapelv: a nemrég használt lapokra várhatóan megint szükség lesz
  - Minden laphoz rendelünk egy számlálót
  - Időnként végignézzük a lapokat és a számlálóhoz hozzáadjuk a Referenced bitet, és töröljük a bitet
  - Mindig a legkisebb számlálójú lapot választjuk
  - ⊖ A gyakori használatot még sokáig megjegyzi, ezen segíthet az öregítés (időközönként csökkentjük a számlálót, ha nem használtuk a lapot)
  - ⊖ Az új lapok számlálójának értéke kicsi

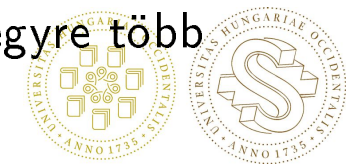
Lapcsere algoritmusok, IV

- **Utóbbi időben nem használt algoritmus** (NRU - Not Recently Used)
  - Prioritási kategóriákat képez az R (Referenced) és az M (Modified) bit alapján
  - A legkisebb prioritású csoportból választ véletlenszerűen

Prioritás	R	M	
3	1	1	hivatkozott, módosított
2	1	0	hivatkozott
1	0	1	módosított
0	0	0	érintetlen


Virtuális tárkezelés, I

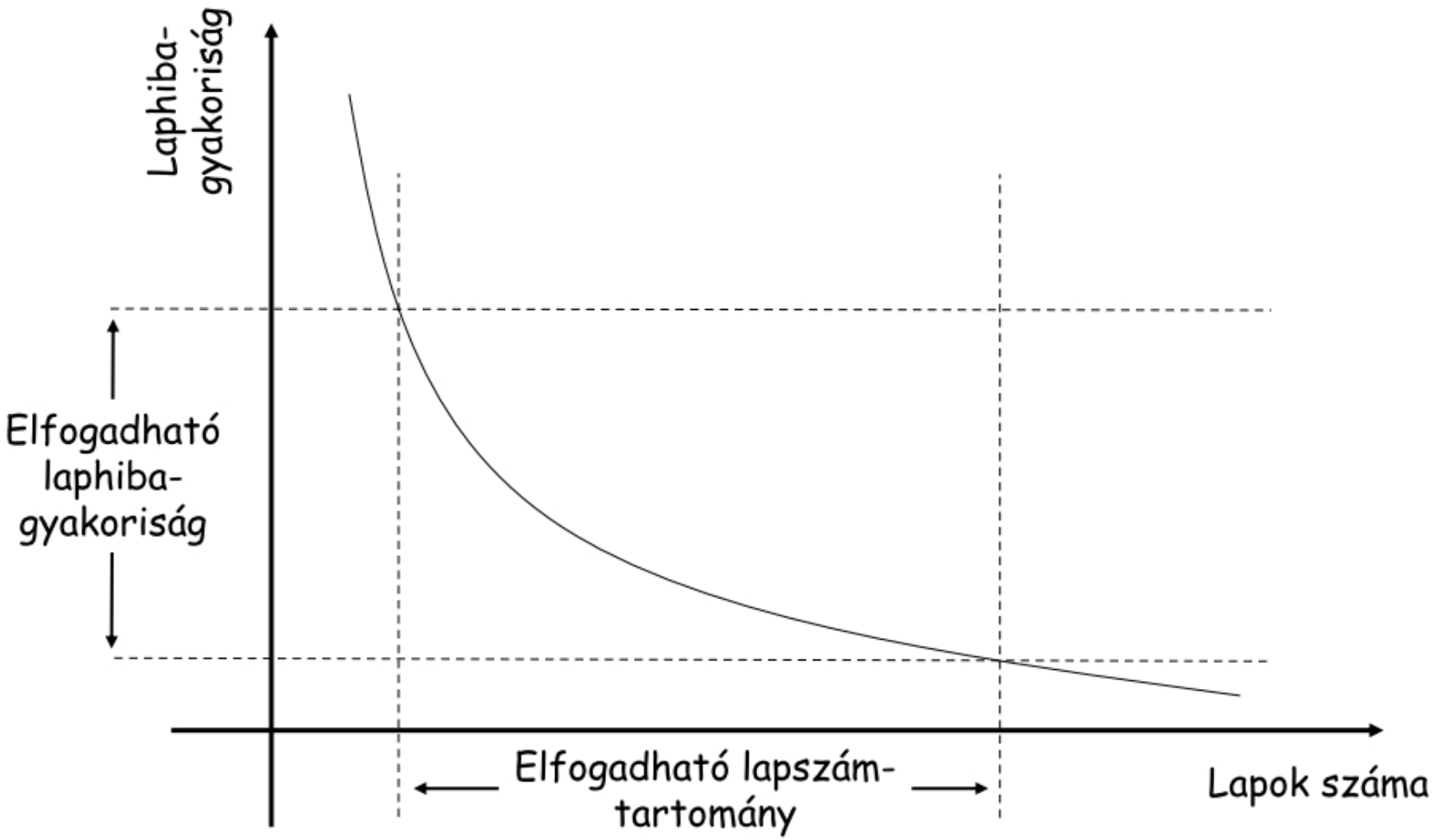
- Mennyi lapot adjunk egy folyamatnak?
  - Minél többet
    - A folyamatok szempontjából jó, így kevesebb laphiba történik, gyorsabban futhatnak
    - A rendszer szempontjából nem jó, mert kevesebb folyamat futhat, így nagyobb az esélye, hogy minden folyamat várakozik (erőforrásokra, vagy egymásra) és tétlen a processzor
  - Minél kevesebbet
    - A rendszer szempontjából jó, nagyobb a processzor kihasználtsága
    - A folyamatok szempontjából rossz, mert nő a laphibák gyakorisága
    - Folyamatossá válhatnak a lapcserék
    - Ekkor a processzor tétlenné válik, ezért az ütemező egyre több folyamatot indít el, . . . , „ördögi kör”
    - Ezt nevezzük vergődésnek (thrashing)



Virtuális tárkezelés, II

- A CPU-kihasználtságnak van egy optimuma a multiprogramozás foka függvényében, ezt szeretnénk megközelíteni
- Ehhez a laphiba gyakoriságát kell mérni és csökkenteni
- A lokalitási elv miatt érdemes a memóriában tartani a hivatkozott lap környezetében lévő lapokat is
- **Előrelapozás** (prepaging):
  - Amikor lapcsere történik, akkor nem csak a hivatkozott lapot töltjük be, hanem a környezetében lévőket is
  - Ezáltal nagy valószínűséggel megelőzzük a laphibákat egy időre
- Ha egy folyamat túl sok laphibát okoz, akkor adunk neki újabb lapokat
- Ha pedig bővében van a lapoknak, akkor elveszünk tőle, hogy más folyamatoknak tudjuk odaadni

A rendszeregyensúly biztosítása



Hol tartunk?

1 Multiprogramozott operációs rendszerek
- Multiprogramozás
- Processzorütemezés
- Tárkezelés

2 Befejezés
- Emlékeztető kérdések


Emlékeztető kérdések, I

1. A multiprogramozás fogalma, működési elve
2. Hogyan kezeli az operációs rendszer a folyamatokat?
3. Mit jelent a logikai memória?
4. Mit jelent a logikai processzor?
5. Milyen tevékenységekből épülnek fel a folyamatok?
6. Milyen állapotokat vehet fel a folyamat a futása során?
7. Mit jelent és mire használható az állapot-átmenet diagram?
8. Hasonlítsa össze a preemptív és a nem preemptív ütemezőket!
9. Milyen elvárásokat támasztunk az ütemező algoritmusokkal szemben?
10. Milyen mérőszámokkal jellemezhetjük az ütemező algoritmusokat?


Emlékeztető kérdések, II

11. Mutassa be a különböző ütemező algoritmusokat! (egyszerű, prioritásos, többszintű és többprocesszoros ütemezések)
12. Mit értünk társzervezés alatt?
13. Milyen társzervezési módszereket ismerünk?
14. Milyen tárfoglalási stratégiákat használhatnak az operációs rendszerek?
15. Milyen tárkialakítási módszereket ismerünk?
16. Mit jelent a virtuális tárkezelés? Miért jó? Hogyan működik?
17. Mit jelent a laphiba? Hogyan kezelhető?
18. Milyen lapcsere algoritmusokat ismerünk?
19. Hogyan gazdálkodjunk a memórialapokkal? Mennyi lapot adjunk egy folyamatnak?
20. Hogyan biztosíthatjuk a rendszer egyensúlyát?


Befejezés

Köszönöm a figyelmet!