**Chapter 7**

Memóriakezelés – 10. előadás

Alapvetések

* A programnak a storageból a memóriába kell kerülnie, hogy le tudjuk futtatni
* A CPU csak a regisztereket éri el
* A memory unit csak címek sokaságát, illetve olvasás írás requestet lát
* Cache a memória és a CPU között van
* Memóriát védeni kell, hogy megfelelően tudjon működni
* Base és limit regiszterek meghatározzák a memóriacímek tartományát, ami elérhető a usernek
* A CPU-nak meg kell néznie, hogy a user módban generált memória eléréshez joga van-e a usernek

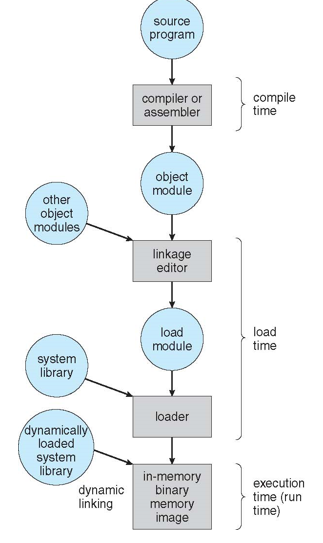
Hardveres védelem

* Ha rossz memóriacímre hivatkozik a program 🡪 Kilövi a programot

Cache szükségessége

* CPU gyorsabb, mint a memória 🡪 Stall 🡪 CPU vár az adatra a memóriából
* Ezért hozták létre a Cachet
  + Nagyon gyors, és nagyon drága tárolóegység kis mérettel
  + Régen alaplapon, most a processzoron
  + Megpróbálja kiküszöbölni a Stalling-ot

Memória Címek

* Utasításoknak és adatnak memóriacím adás, memória allokálás három időben történhet
  + Forditás ideji cím
    - Fordítás előtt tudjuk hol az adat
    - Abszolút memória cím/Direkt memória cím
  + Betöltés ideji cím
    - Olyan kód kerül előállításra, ami nem direkt beégetett kód, memória cím, csak annyit mond kell neki 4 byte memória
      * Megmondja melyik memória címtől kezdődik az ő helye
      * Relatív memória cím
  + Futási idejű cím
    - Nincs megadva betöltéskor az allokált memória cím
    - Mondod a gépnek mennyi memória kell 🡪 Ad

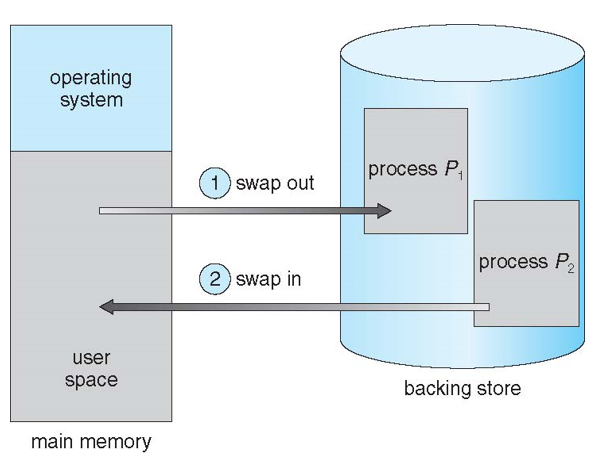
Logikai vagy fizikai memóriacím

* Logikai/Virtuális memóriacím
  + CPU által generált
  + Lassabb, de megbízhatóbb
* Fizikai memóriacím
  + Memory unit által látott
* Adress space🡪 Egy program által generált összes Logika/Fizikai memóriacím
* Memory Management Unit – MMU
  + Logika címhez fizikai címet sorsoló hardver eszköz
  + Dinamikus cím
  + Felhasználói programok soha nem foglalkoznak fizikai memóriacímmel, csak logikaival

Linkelés

* Statikus linkelés
  + #include header.h 🡪 Beleépíti a programba a headert lényegében
    - Egyszerű címzés
    - Több memória kell hozzá, nem olyan hatékony
* Dinamikus linkelés
  + Stub/Csonk 🡪 Megnézi az adott függvénykönyvtár be van-e már töltve
    - Csökken az induló memóriahasználat 🡪 Idővel ahogy használjuk felkúszhat ugyanoda
  + Dll 🡪 dynamic link library 🡪 Könyvtárak max 1 példányban a memóriában

Swapping

* Alapötlet 🡪 Ha kevés a memória/RAM 🡪 Told bele más tárolóba a folyamatot
  + Ez a tároló általában a háttértár
  + Nagyön költséges feature
* Ha az a folyamat kell éppen, ami a háttértáron van 🡪Swap
  + Kitol RAM-vól valamit, ami az ütemező szerint nem kell és betolja a RAM-ba, ami kell a háttértárról
  + Úristen very lassú

Többpartíciós allokálás

* Első oprendszerek 🡪 Fix memóriaterület adva egy folyamatnak
  + We no likey, that much
* Változó méretű partíció/memóriaterület allokálás
  + Minden folyamatnak annyi memóriát ad, amennyit kér
    - Annak a folyamatnak majd deallokálni is kell a mennyiséget, amit allokált, itt lehetnek problémák
  + Összefüggő memóriaterület kell legyen 🡪 Ha nincs el sem indul, vagy vár amig lesz
  + Itt a probléma, a lyukak kezelése, miként, hova allokálunk memóriát?

Dinamikus Memóriaallokálás – Típusok

* First fit
  + Megyünk végig a memórián, és az első összefüggő üres helyre, ahova tudjuk „berakjuk”, ott allokálunk neki memóriát
* Best fit
  + Oda rakjuk, ahol a legkisebb a lyuk, ami még elég a programnak
    - Legkevesebb a pazarlás
* Worst fit
  + A lehető legnagyobb összefüggő memóriaterületen allokáljunk a programnak memóriát
* First-fit és Best-fit vált használttá
  + Ez viszont feldarabolja a szabad helyet 🡪 Külső töredezettség
* Külső töredezettség
  + Van elég memória, de nem összefüggő terület
  + Kell egy „töredezettségmentesítő”
  + Egy gondolat a megoldásra 🡪 Compaction/Tömörítés
    - Rendezzük újra az összes memória blokkot, hogy egy nagy üres memóriaterület maradjon
    - Csak gondolat maradt, mert ha egy program ugyanarra a memóriacímre hivatkozik, mint tömörítés előtt, nem fog működni, illetve az I/O műveletek sem szeretnék
    - We no likey összességében

Megoldások töredezettségre

* Szegmentálás
  + Daraboljuk fel a programot 🡪 A memóriában több kisebb darab kell nem egy nagy összefüggő memóriaterület
  + Hagyományos, elavult
  + Természetes következménye a Paging/Lapozás
  + A logikai memóriacím több részre bomlik
    - Szegment sorszám/number
    - Szegmenten belüli cím
  + Szegment tábla
    - Base
      * Megmondja melyik fizikai címen kezdődik egy szegmens
    - Limit
      * Megadja a szegmens hosszát
    - Megvan a szegmens eleje és vége
  + Ezzel nincs belső töredezettség, és nagyban csökken a külső töredezettség
* Lapozás/Paging
  + Vissza a gyökerekhez 🡪 Felosztjuk a memóriát fix méretű keretekre
    - 512 byte – 16 Mbyte
  + Ezzel kapunk n darab lapot/paget
  + Nyilvántartás kell a foglalt és üres keretekről
  + Page Table 🡪 Logikai címet lefordítja a gépnek fizikai címre
  + Nincs külső töredezettség, de bejön a belső töredezettség
  + 80-as években tökéletes, de bajos ahogy nőtt a memória méret
    - Sokáig tart 10000000 lap között megkeresni, amit akarunk
    - Megoldás 🡪 Több szintű lapozás
* Több szintű lapozás
  + Ha van 10000000 lapunk, felosztjuk 100000 lapos blokkokra, és felindexeljük ezeket a blokkokat 🡪 Tudjuk, hogy a 3. blokkban van valami, nem kell az összesen végigmenni 🡪 Relatív gyors
* Osztott Lapozás
  + Bent vannak a lapokon a függvénykönyvtárak 🡪 Nem kell többször betölteni a memóriába, nem foglal feleslegesen memóriát
  + Stub-ok használata
  + Külső töredezettség nincs