12th, 13th and 14th Course:

Nucleu / Kernel

Sistem de operare = program care ruleaza pe un hardware dat -> care are doua componente (procesorul si memoria/hard drive-ul) care nu lipsesc niciodata

\*Hardware:

1.Gestiunea Intreruperilor

-toate deviceurile(mouse, tastatura etc) comunica cu sistemul de operare prin intreruperi

2.Gestiunea proceselor

- aka cine, cand si pe ce procesor se executa

- scheduler

3.Dispatcher procesoare

-parte din planificator. Da instructiuni procesorului

\*Gestiune de memorie

\*Input/Output la nivel fizic

\*Gestiune fisiere

\*Planificare lucrari si gestiuni si alocare resurse

-> gestioneaza cereri de resurse

-> se asigura ca nu se va bloca

\*Gestiune tehnica a SO

-> inregistrari a ceea ce contine calculatorul

-> ce drivere sunt instalate

\*Gestiune economica a SO

-> procese curente

-> cine are acces la sistem

Serviciu (user land)

Compilatoare, asambloarea, link editoare, loadere -> nivelul 1

Interpretoare (ex. Python), macroprocesoare, editoare de text, debuggere -> nivelul 2

Utilitare (programe ca si grep sed awk), biblioteci, baze de date, etc. -> nivelul 3

Medii de programare(IDE) -> inglobeaza 1,2 ,3

Bibliotecar (Archiver) -> utilitar care creeaza biblioteci

(G)UI -> nivelul 4

Utilizator -> nivelul 5

Starile unui proces

Prima stare: HOLD

- nu e vizibila in calculatoarele modern

READY

- nu e asa de vizibila pentru utilizator

- procesul este pregatit sa ruleze instructiunile, asteapta processor

RUN

- instructiune cu instructiune procesul se executa

- so-ul poate decide ca procesul sa fie trimis inapoi in READY

WAIT

- nu e asa de vizibila pentru utilizator

- nu are procesor, si nici nu poate fi considerat pt a primi procesor, fiindca asteapta ceva (asteapta dupa i/o)

- cand ceea ce asteapta este rezolvat, trece in READY

SWAP

- e mai vizibila, dar rara

- cand memoria e destul de incarcata

- extensie de memorie

- cand sunt programe in swap, memoria e foarte lenta

- cand se elibereaza memoria, procesul e readus in READY

FINISH

- elibereaza toate resursele, se curata memoria

Deadlock-uri

SO poate avea o imagine de ansamblu

Trebuie sa omori un process (in cele mai multe cazuri, pentru a evita livelock-urile)

Se poate si “livelock” (cand dam o masina in spate, dar revine in fata, si nimic nu se schimba)

Aspecte:

1. Iesirea

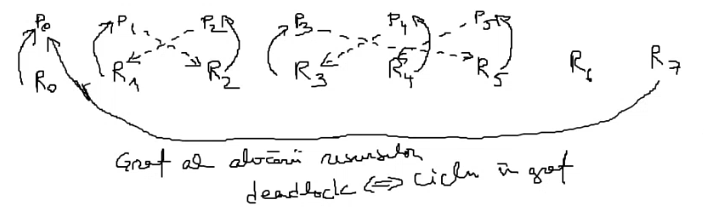
- omori un process

- daca aducem un proces intr-o stare anterioara, se poate intampla un livelock

1. Detectarea

- desen cu toate procesele, resursele, ce au blocat si ce vor sa blocheze = graf al alocarii resurselor

- orice deadlock = ciclu in graf



Deadlock intre P1 si P2;

Deadlock intre P3, P4 si P5

1. Evitarea

- o singura modalitate de evitare!

- ce face un deadlock posibil:

a) excludere mutuala (locking)

b) lock and wait

c) no preemption

d) circular wait

- evitarea deadlock-ului

a) nu se poate evita

b) nu poate fi eliminata

c) nu poate fi eliminata

d) stabilim o ordine (oricare) pentru resurse (mecanisme de sincronizare) si blocheaza-le intotdeauna in aceeasi ordine

Gestiunea proceselor

- planificarea executiei:

-> in ce ordine executam procesele primite?

1. FCFS (First Come First Served)
2. Shortest Job First (SJF)

- trebuie sa stim / estimam durata fiecarui proces

- risc de starvation (procesele mari nu primesc procesor)

1. Priorities
2. Deadline Scheduling

- so planifica job-uri astfel incat fiecare se termina la momentul “din contract” (daca nu se poate, cum planificam astfel inca delay-urile sa fie minime)



1. Round Robin

Adress translation

- pentru un program care contine mai multe surse a.c, b.c si c.c ->(prin compilare) a.obj, b.obj si c.obj ->(prin link editare) se face un executabil din toate a.out (executabil) ->(prin rulare) se ajunge la un process

- referire memorie (variabile) se face prin nume

- prin compilare aceste nume dispar, si se transforma in locatii in fisierul .o

- prin compilare se schimba in locatii in executabil

- prin rulare se schimba in locatii in RAM

1. Allocation methods (unde incarcam un program in memorie la pornire)
   1. Real allocation
      1. Single-user/Single-tasking (A1)
      2. Multi-user/Multitasking
         1. Fixed partitions
            1. Absolute (A2)
            2. Relocatable (A3)
         2. Variable partitions (A4)
   2. Virtual allocation
      1. Paged (A5)
      2. Segmented (A6)
      3. Paged-segmented (A7)
2. Loading methods (ce parti ale programului incarcam de la bun inceput in memorie)
3. Replacement methods (ce facem cand memoria e plina)
4. Ce facem la un apel malloc/free
5. Memory hierarchy
   * + 1. Allocation methods

A1 – Single-user/Single-tasking

* Un singur program deodata => primul

A2 – Fixed absolute partitions

* Pentru mai multe programe – impartim memoria in mai multe partitii predefinite => putem rula mai multe programe in partitii diferite
* Probleme: Nr limitat de programe

Nu poti rula 2 programe compilate pe aceeasi partitie

A3 – Fixed relocatable partitions

* Putem face calcule de memorie in timpul rularii programului
* Calculul de adresa la runtime = inceput partitie + offset din exectuabil (rezolva coliziunile pentru aceleasi partitii)
* Probleme: Nr limitat de programe

Limitare dpdv al dimensiunii procesului – nu poti rula un proces mai mare decat cea mai mare partitie

A4 – Variable partitions

* luam primul liber si punem programul acolo
* partitiile devin variabile, nu au dimensiuni fixe
* nu suntem limitati, doar de dimensiunea totala a memoriei
* Probleme: din cauza ca rulam procese doar in memorie contigua => !Fragmentarea memoriei (Solutie: Compactarea memoriei libere – e de durata, trebuie sa oprim un proces)

A5 – Virtual paged allocation

* Impartim RAM-ul in “pagini fizice” – 4 kb, iar programul in “pagini virtuale” – 4kb
* atunci cand rulam un program, nu il incarcam contiguu in memorie
* rezolvam fragmentarea prin fragmentare => de la bun inceput "fragmentam" programul si il incarcam in memorie in functie de unde este spatiu
* Adresa nu mai e un numar, ci o pereche (pagina virtuala si offset)
* Probleme: calcul mai complicat la adresa de memorie (din (0,64)-adresa virtuala -> cautare in tabelu de pagini a procesului => se transforma in (12,64)-pagina fizica -> care se transforma in 12\*4kb+64 – adresa fizica; alt exemplu pt pagina 3 cu offsetul 32: (3,32)->(22,32)->22\*4kb+32)

A6 – Virtual segmented allocation

* Segmentele, fara paginare, aduc aceleasi probleme ca partitiile variabile – vor conduce la fragmentare
* Permiteau separarea codului dupa rol – segment de cod, segment de date (SO proteja segmentul de cod)
* Grupare a codului si datelor
* Protejezi zona de memorie – se punea intr-un segment cod care era read only => protectia codului de modificari
* Offsetul segmentului + offsetul adresei => protectie a memoriei, iar segmentele care grupau cod puteau fi refolosite
* Segmentele permiteau sa folosesti mai multa memorie decat puteai accesa
* Probleme: Fragmentarea memoriei

A7 = A5 + A6

* Grupam codul in segmente, iar fiecare segment este paginat
* Nu exista fragmentarea memoriei, avem protectia memoriei, avem refosibilitate, si putem adresa mai multa memorie decat ne permite adresa
  + - 1. Loading methods

- (NOT GOOD) Load all program pages into RAM from the beginning – A: execution is fast; D: slow startup + occupy memory with data that is never used =>

- (NOT GOOD) Load first page and then load every page when needed – A: no wasted memory, D: slower execution =>

Locality principle: a process is likely to need soon the pages next to the page that was just loaded => always load a page and pre-fetch its neighbors’ pages

* + - 1. Replacement methods

We move pages to the SWAP, regardless of the process (which pages can we move so that the impact is minimal?) =>

We try to guess which pages will not be used in the future. => We need a uick and light approach

Approach one, not best – Fifo –> always eliminate the oldest page in memory

Better approach – Not recently used -> mark every page with 2 bits r and w – bits will be updated on whether the page is read or written -> set to 1 when a read/write is done on the page, and are periodically reset to 0 =>

Categories:

R W

0 = 0 0

1 = 0 1

2 = 1 0

3 = 1 1

Even better approach – Least recently used -> for a system with n pages, maintain an nxn matrix of bits -> when a paged is accessed, fill its line with 1 and its column with 0

For a system with 4 pages, with access to those pages 0 1 2 3 2 3:

-> -> -> -> We choose a victim page from those with a minimum line sum

* + - 1. What does an OS do when calling malloc/free?

We keep 2 linked lists – one of free spaces, one of occupied spaces

Problem – fragmentation (can we do something about it?) =>

First – Fit: Allocate the required amount in the first location where it fits

A: Fast; D: Doesn’t solve fragmentation

Best – Fit: Go through the whole list, choose a spot that fits the best

D: Slower + creates very small free slices = fragmentation

Worst – Fit: Choose largest available, so you’re left with larger chunks

A: leaves large free slices; D: Slower + can still lead to fragmentation

Buddy – allocation: Doesn’t allocate, for example, 13 bytes, but the closest to a power of 2

2n = 2k + 2k + 2k+1 + 2k+2 + … + 2n-1

* + - 1. Memory/storage hierarchy

Registers

L1

L2

L3 (maybe, if pc is not cheap)

RAM

SSH/HDD

Cache

* Where/how to place a page in the cache so that it can be found fast and improve performance?
* Given a cache of size N pages, where should we place physical page K?
  + Direct-cache organization: K -> K%N
    - Given a cache of size 16 pages, the page request sequence 1, 17, 33, 97, 1, 17, 1, 17 will cause cache collisions = cache thrashing
  + Set-cache organization: the free cache slot is searched sequentially through the entire cache (same for finding if a page is already in the cache)
    - Slow
    - No cache thrashing
  + Set-associative cache: organize the cache pages in groups and determine the group of a page through %, then search for the page or free slot iteratively
    - The way to go
    - Combines the 2 before

File structures: Symbolic links & Hard links, I-nodes

How things are organized on a hard drive

Creating a symbolic link: ~$ ln -s myFile.txt mySecondFileS.txt (or between directories)

Creating a hard link: Only the root can create hard links

~$ ln myFile.txt mySecondFileH.txt (or between directories)

Symbolic links are displayed as links, but we see hard links as regular files

If we remove myFile.txt, the symbolic link will be broken (if we try to access mySecondFileS.txt it will say there is no such file or directory); the hard link will not be broken, will still be accessible (with the data from myFile.txt).

. – the link to itself

.. – the link to the parent

Mounting process – how to have multiple drives in Linux:

For Linux – we have a single root

If we come with a USB stick (whose root is / ) – the root of that USB will become B (whatever B had before the mounting is still there, but is obscured, invisible)

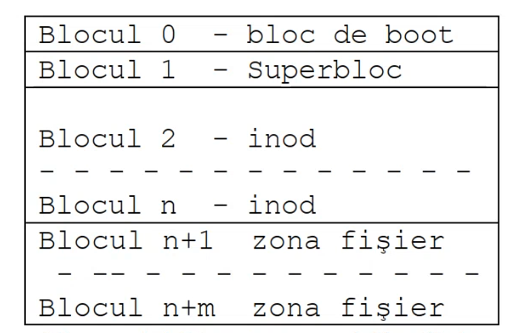
~$ df -h -> see free disk space

~$ cat /etc/fstab -> what volumes should be mounted when the system starts up

Formatting partitions

When we format a partition (whole drive / a piece of it) – it will be organized in blocks.

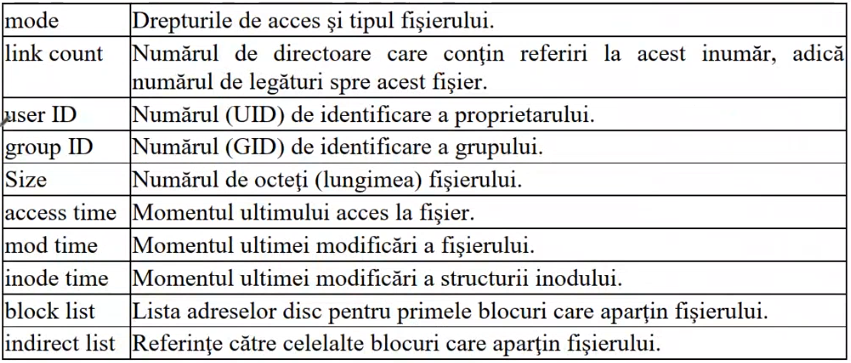
Blocks = the unit of data exchange between the hard drive and the rest of the computer (8kb or 4kb).



First block = Boot block – if not empty, contains a little program that knows how to load the OS; very small program = 4kb/8kb -> loaded into memory, executed and it knows how to load the rest of the OS

Second block = Superblock – like a header; it tells how large the other regions are, what’s the architecture of the i-node

I-node – entry point to the list of blocks of a file (kind of like the head of a linked list)

* It contains an identification area
  + Block list = Has direct access links (links to data blocks) – 10 blocks
  + Indirect lists:
    - Can have (simple indirect) links to addresses to other blocks – +128 blocks
    - Can have (double indirect) – link that goes to a block that has addresses of other blocks that point to data blocks – +1282 blocks
    - Can have (triple indirection) – +1283 blocks

Directories – in data blocks contain 2 things (so directories contain pairs of): file name and i-node (for that file).

Symbolic link => multiple files point to the same i-node

Hard link => multiple i-nodes that point to the same data

Files can be kernel structures (like pipe) or socket (that go to another OS).

