Curs 10 (6a)

* diferența între fișiere text și fișiere binare
  ↓
  imagini, pdf, exe

⇒ \0

metadate : → type, location, size, protection, time, date
↳ salvate în fișiere de metadate, într-o tabelă separată.

→ Cu cât avem fișiere mai multe ⇒ cu atât mai mare fișierul de metadate.

HD e fragmentat în blocuri.
↳ at. când creezi un fișier ⇒
⇒ rezervi un bloc (chiar dc. nu îl folosești în întregime)

Open-file table : tracks open files
File-open count : numără de câte ori e deschis un fișier
→ dc. sunt deschise des → accesul către ele e mai facil
→ se face o predicție, caching că vei deschide fișierul.

---

* pipelining, branch-prediction, out-of-order execution.

Open-File Locks : shared lock ⇔ reader lock
exclusive lock ⇔ write lock

un alt program nu poate să altereze un fișier deschis deja de alt fișier → lock

excel face așa; notepad++ te lasă să deschizi., te înfurie ~~instant~~ de mesaj.

seek() → pointer în fișier ← rewind
reposition to a începutul

→ mai bine să accesăm secvențial un fișier deși avem acces aleator → să e mai eficient
posibilit. de a facem lucrurile secvențial.

! Procesarea eficientă se face citind tot conținutul fișierului; excepție dc. fis. e prea mare (element cu element)
↓
pt. a fi încărcat în ram

Directoarele = metadate despre folder, nu F fizic pe HD
- nume
- ce conține
- securitate, permisiuni
- autor, etc.

→ structură arborescentă.

. → calea la dir curent
.. → folderul anterior
} ⇒ var. importante linux

windows: cd.. ⇒ du-te în folderul anterior.

linux: ./main
windows: nu e necesar, verif. dc. exe. e în fold. curent.
dos

→ doar te plimbi cu un pointer prin arborele fis.

calea absolută → de la root
relativă → din folderul actual

Când în Linux ai comunicare cu un folder de altundeva: linkeri
windows: shortcut

backpointer → mergi în pointerul precedent.

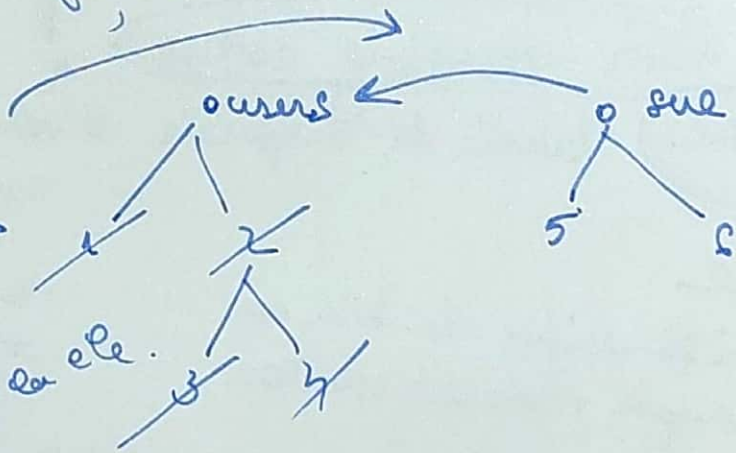
Acyclic-Graph Directories

→ graful de directoare trb. să fie aciclic

File System Mounting

montarea sistemelor de fișiere

mount /sue



- rămân în metadate
- am înlăturat accesul la ele
- nu mai am comandă până la ele.

când bagi un stick → se citesc metadatele + se creează arborele + este montat la my computer

Punctele de mount.

rm → șterge doar metainformațiile
delete

→ datele rămân în sectoarele hard discului

COMPUTER FORENSICS → în cele mai multe cazuri conținutul HD poate fi recuperat.

Network File systems

File Sharing

windows : samba. (CIFS) ↓ standard windows protocol.

NFS : UNIX : client-server file sharing protocol.

window Active Directory ⇒ tool, acces la anumite foldere ?

↓ giant

structură de date împărțită ~~mai multor user~~

windows nt tehnologia de servere de la microsoft.

---

error-correcting codes: !

CRC tabele de dispersie → verifică corectitudinea conținutului.

↓

de biții s-au schimbat.

+ eventuala corecție

→ ECC

știe să verifice de. biții au fost corupți + corecția erorilor.

Sistemul de acces : 7 7 7

usr | public

grup

RWX
1 1 1 = 7

RWX
1 1 0 = 6

RWX
0 0 1 = 1

Microsoft Azure.

Active Directory nu mai există de când cu Windows 10.

Implementarea unui sistem de fișiere

write block 42
read block 10

↳ mereu scriem blocuri întregi

FCB = file control block = ~~metadate~~ datele efective sunt blocurile de date în HDD.

open = doar metadate
read = datele adevărate.

Curs 11 (So)

arbitrary code execution

→ arbori balansați ; arbori roșu-negru

AVL tree → memory-based solutions

B tree → disk-based solutions ; (databases)

↳ diferența constă în alg. de restructurare în momentul adăugării

Metode de alocare

1 → Contiguous Allocation → blocurile trebuie să fie consecutive unde se alocă

* cât mai rapid decât copy → face doar schimb pathul la copy copiez fizic la altă locație

2 Linked

listă înlănțuită

↓

ai în nod adresa blocului următor

→ controllerul de HD citește mai multe blocuri

caching ← de. citești consec. sistemul este eficient.

↳ la citiri non-contiuă acel HD-ului trebuie relocat fizic → timp.

↓

pre-fetching

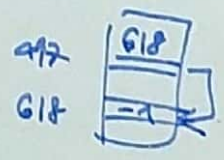
exact ca logica de la memorie și cache

FAT

file allocation table ⇒ cu liste înlănțuite

↓

sist. fol. întâlnite de MTFS

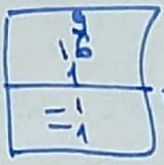


→ sisteme de fișiere implementat de toate SO sist. chiar de. nefol.

→ e prez. pe stickuri

③ Indexed Allocation → un nod are toate locurile ocupate

ultima adresă → pointer către un potențial alt bloc mereu



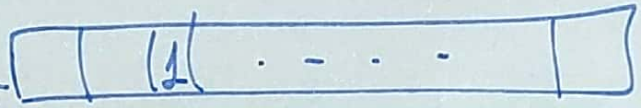
10^100 nr. part. ale fișierului

pt. a verifica dc. a fost citit întreg file-ul.

nr. negative → în complement față de 2 (se pierde un bit)

Free Space Management → bitmap

nr. total de blocuri
↑
n-1



popcnt → returnează biții activi ai reg.
bsr → bit scan reverse → primul bit activ. cel mai din stânga.

→ linked free space list

ocupă mai multe spații

Curs 12 (SO)

Planificarea proceselor:

Threadripper (AMD)

exec. simultan un nr. f. mare de proc. și threaduri.

CPU burst
I/O burst ⇒ wait for I/O

reg < ram < hd < retea ~~extrie~~ (viteză extire)

Cât timp un pr. necesită I/O ⇒ alt pr. să fie pe CPU că e liber.

Putine cazuri în care un pr. stă pe CPU „> 8 milisec"

~~1. waiting~~

1. running → waiting (i/o event)
2. running → ready (round-robin, cuantă)
3. waiting → ready.
4. terminates.

non-preemptive.
↓
1, 4
(decide programul)

preemptive
↓
decide CPU SCHEDULING

Dispatcher → schimbarea contextului
CPU SCHEDULE → switching user mode/kernel mode
→ să continui munca de unde ai lăsat-o
↓
cu instruction pointer.
* se resetează memoria cache → sari în memorie (recopierea datelor în cache)

Dispatch Latency → timpul în care înghețăm și dezgheț un proces.

Sch. Criteria

1. CPU utilization (MAX)
2. throughput = # procese exec. în unit. tip. (MAX)
2. turnaround time = timp exec. un proces (MIN)
3. waiting time = cât a așteptat un proces în ready queue (MIN)
4. response time = prima reacție a ps., un resp. request status

---

(MIN)

wallclock time = CPU time / exec + cât a așteptat în coadă pt. a primi acces la CPU

CPU time.

Δt = primul răspuns este produs.
răspuns parțial.

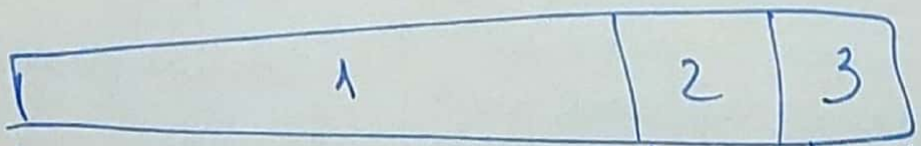
UI = user interface
UX = user experience.

Xerox = a inventat interfața cu ferestre și mouse

FCFS = first come first served.

The Gantt Chart

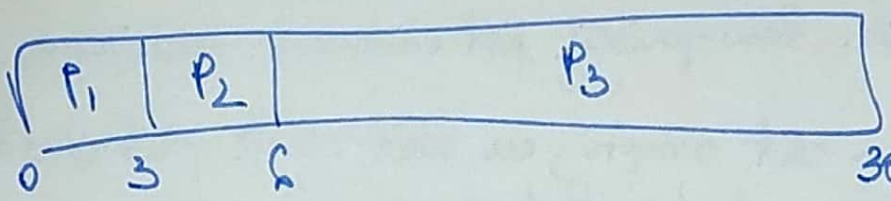
P	Burst time
1	24
2	3
3	3



24 3 3

Timp mediu: (0+24+27)/3 = 17 → MARE (X)

(0 = cât a așteptat fiecare proces.)



$(0+3+6)/3 = 3$ — MAI BINE ÎN MEDIE, (NU PT. P_1 de ex.)

Convoy effect → procese scurte înaintea celor lungi.

⟹ **Shortest-Job First** SJF

dar nu știm burst time a priori → preemptive

)

factor de uitare $\alpha^3 t_{n-2}$

↓

scade pe măsură ce înaintăm

$\alpha \cdot t_n + (1-\alpha)\tau_n$

↓

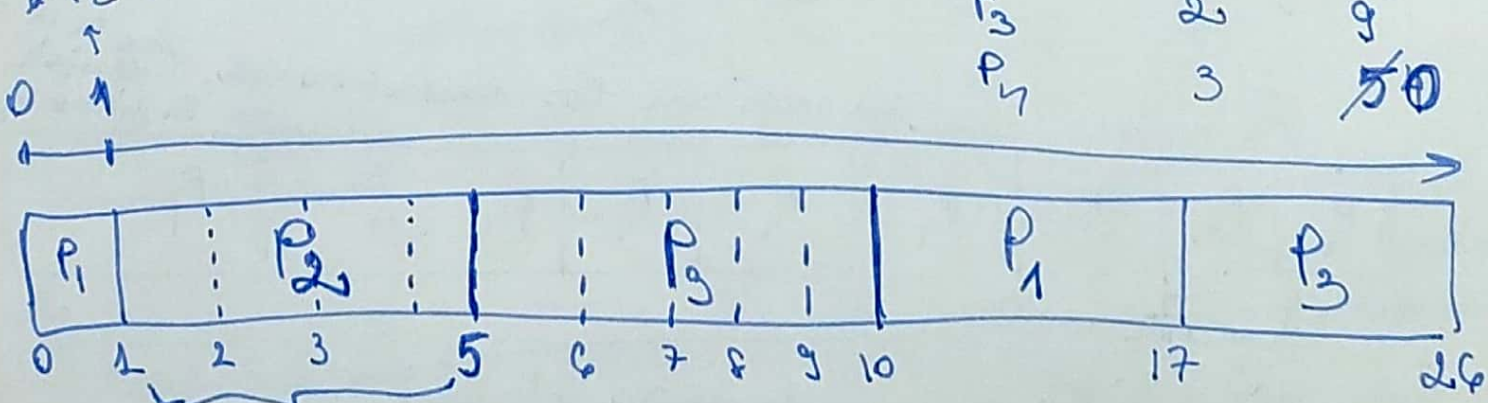
predicția sau ce prevede din predicția inițială

⟹ **Shortest-remaining-time-first**.

Preemptive SJF Gant chart

P	Arrival	Burst.
P_1	0	~~8~~ (7)
P_2	1	~~4~~ ~~3~~ 2 0
P_3	2	9
P_4	3	~~5~~ 0

P_2 ajunge și rămâne



4

timp de așteptare $\frac{(10-1)+(1-1)+(17-2)+(5-3)}{4} = 6,5$.

Starvation ⇒ proc. low-priorit. pot să nu se mai exec.

soluție ⟹ ~~Aging~~ ⇒ cu cât aștepți, cu atât crește prioritatea proc.

Priority Scheduling ⇒ pun și simplu sortezi după prioritate.

Round Robin (RR)

→ fiec. proces primește o **cuantă** de timp
= fixat
↗ pt că

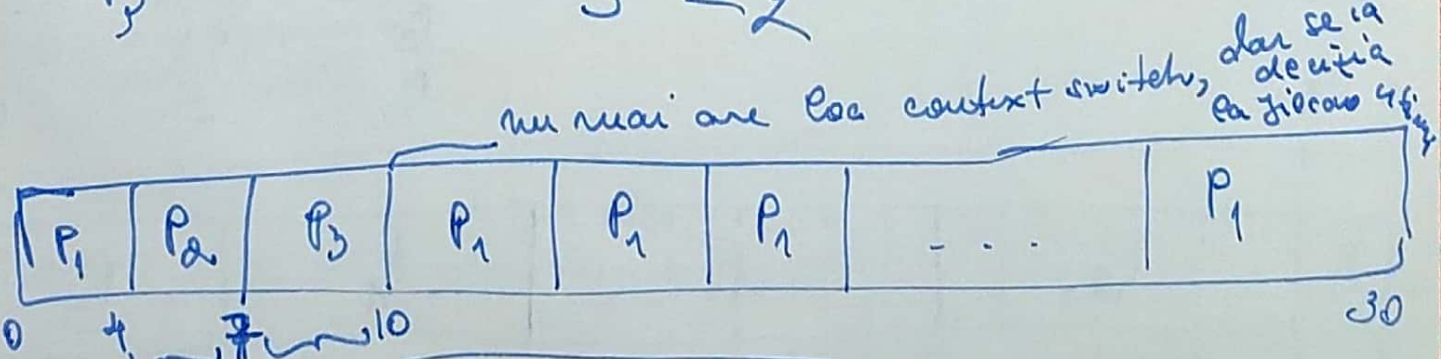
coadă FIFO

→ throughput negativ ⇒ se pot realiza în mod ineficient context switches.

dar ne dă cu răspuns mai rapid.
un rezultat!

Time Quantum = 4

P	Burst time
P_1	24
P_2	3 < 4
P_3	3 < 4



$q \to \infty$ ⇒ RR = FIFO

$q >$ context switch time

$q \cong$ 10 ms - 100 ms ; cont. sw. < 10 usec

80% CPU burst < q

<=> first job scheduling, dar fără starvation.

Procese → foreground (interactive) → vrean RR
backgound (batch) → FCFS

Există totuși o ierarhie a tipurilor de procese.

→ Mai multe cozi:

Q_0 → $q = 8$
Q_1 → $q = 16$ ↓ dc. nu a reușit
Q_2 → FCFS

Thread Scheduling

PCS → process-s-scope ⇒ în funcție de procesul părinte
SCS → system-contention-scope ⇒ per total, la nivel de kernel.

---

sistem → omogene, simetrice, asimetrice { multiprocesor } CPU GPU, OPEN CL, MPI

soft affinity
hard affinity

îi spui pr. pe care core să ruleze
încărcat = affinity.

---

load balancing: echilibru al muncii
push migration: alt CPU mai liber →
pull migration.

NUMA
non uniform mem. acces.
fiecare CPU → are o memorie
↓
accesul dintre CPU e lent
mem. diferite.

Curs 13 (fo)

Main Memory

- se verifică baza și limita cui pr. în RAM
  → trap ⇒ segfault

Exec.

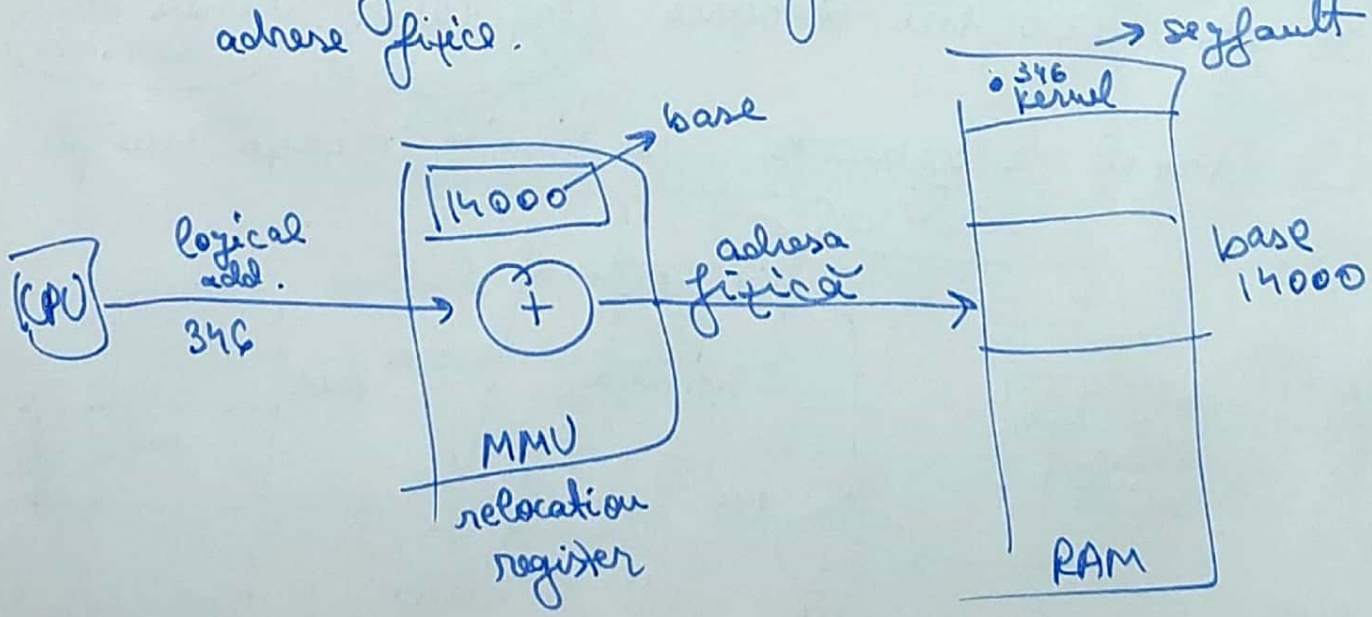
ELF → linux , PE = portable executable windows

biblioteci încărcate dinamic : . so (linux)
. dll (windows)

Adrese logice → generate de CPU, adrese virtuale
Adrese fizice. → adresarea RAMULUI

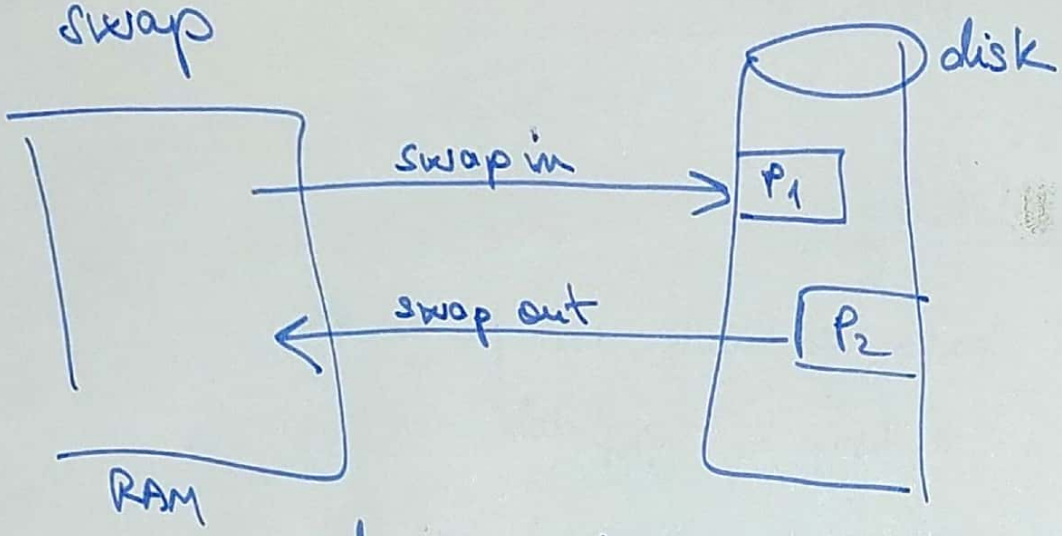
MMU ⇒ memory management unit.

transformă adresele logice emise de CPU în adrese fizice.



acțiune I/O → de durată

) swap



(init-un fs swap)

de. nu mai aveau RAM → swap, pe disk de procesele

Fragmentation Architecture

→ External Frag ⇒ memorie există, dar nu e continuă ~~e frag~~.

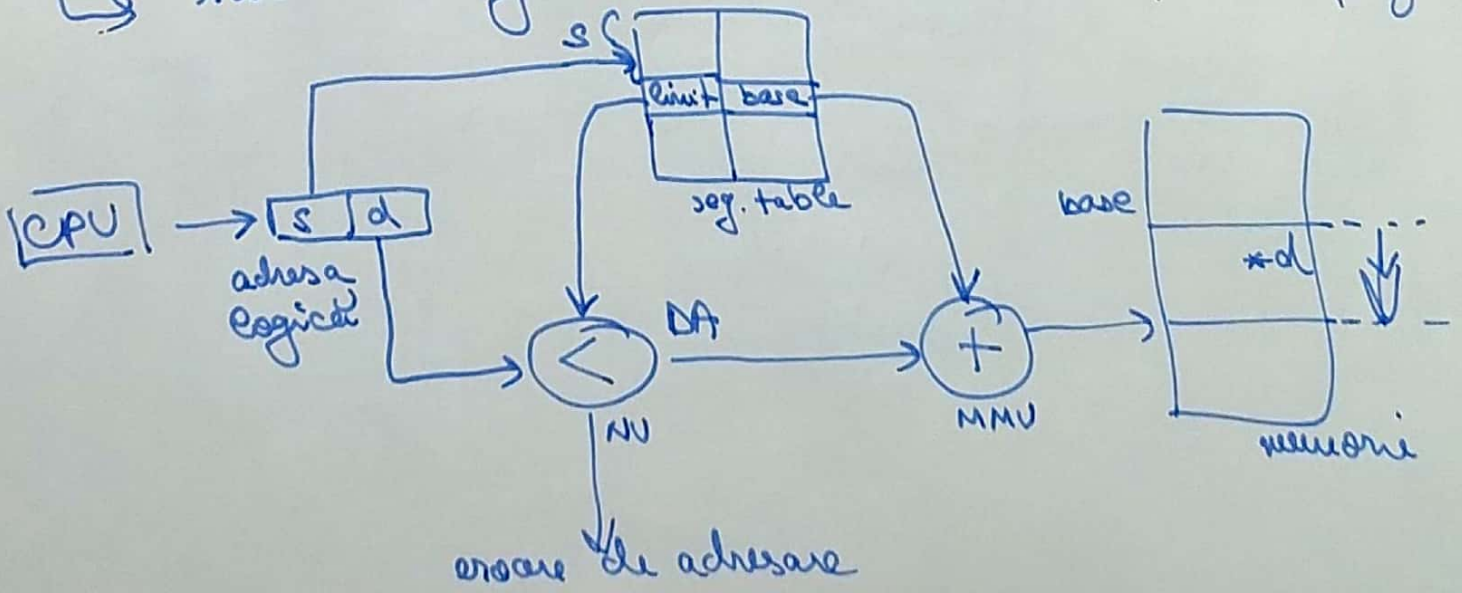
~~→ Internal Frag → prin adăugarea proc. fragmentării pagin. în interiorul mai face memoria.~~

Fragmentării unui proces → pagini ?

~~Fragmentării~~ Segmentării ex pr. în mai multe ~~proces~~ segmente, pt. a putea fi încărcat în RAM ↓ proces

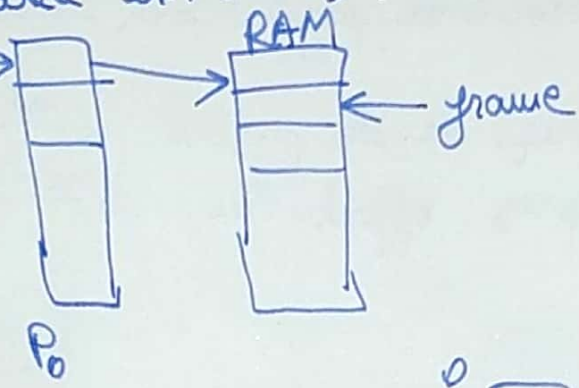
→ ~~dacă~~ în funcție de permisiuni (r, w, e) și dacă seg. e valid (nu e setat la 0) CPU-ul îl procesează sau întoarce seg fault. eroare de adresare

↳ tabel de segmente (cu limita + baza unui progr.)



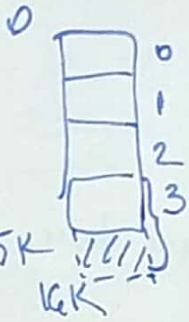
Paging

→ evită fragmentarea externă ⇒ totul e împărțit în mod egal

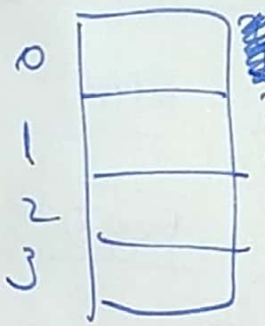


multiple de 4K

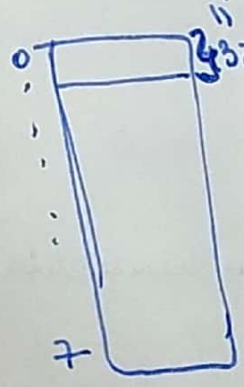
pagină în proces → frame în memorie



fragmentare internă



Ni se dau { nr. de pagini / dimensiunea unei pagini ⇒ spațiul de adresare logic



page size = 2^n

logical address space = 2^m

$\Rightarrow 2^{m-n}$ = ~~date~~ nr. de pagini ~~în memorie~~

Din nou : Alg.

Cum împărțim memoria unui prog.

→ nr. de pagini → p ⟶ n

→ offsetul unei pagini = d ⟶ m − n

→ aflăm repr. nr. de pagini de $\underline{n}$ → extragem n

→ aflăm repr. offsetului de $\underline{m-n}$ → extragem m−n

aflăm m

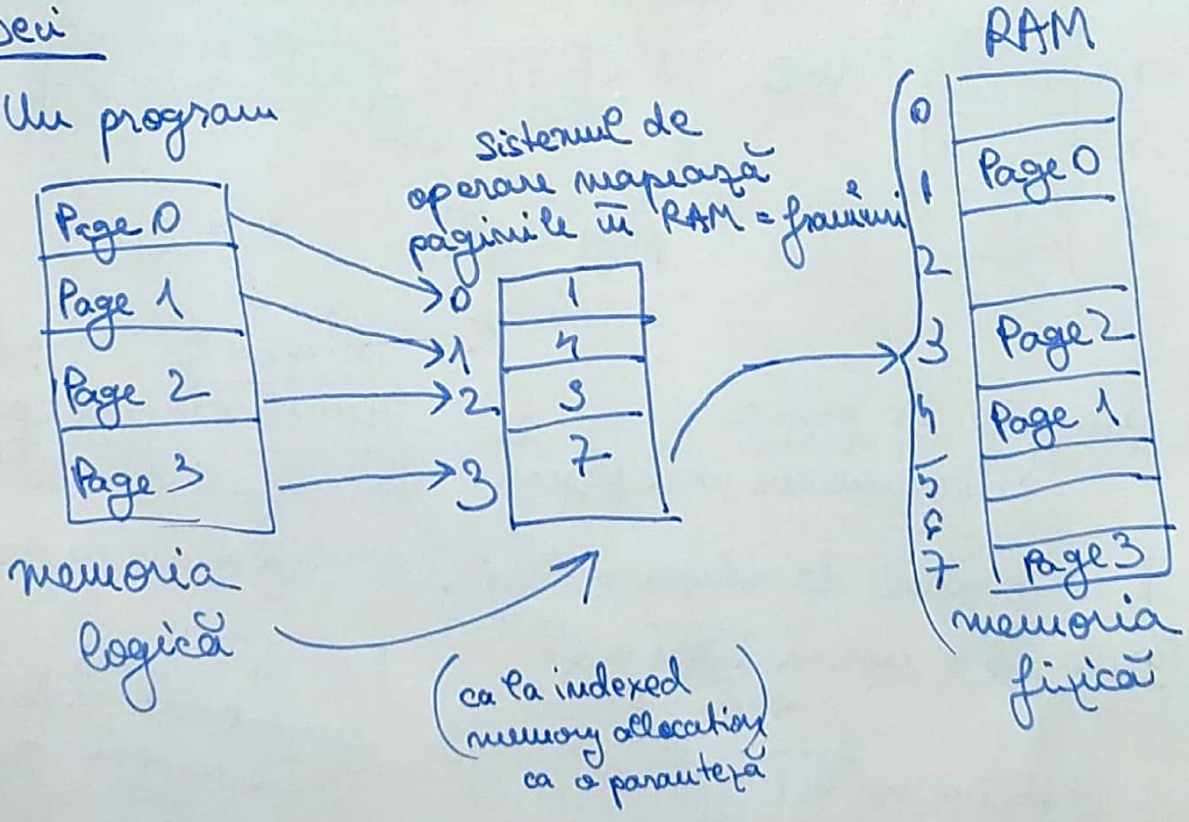
↓

2^m = memoria totală a unui prog

* o pagină are dim. unui frame.

tabelă de pagini per proces.

Deci



Facem din pagini în frameuri și păstrăm deplasamentul din cadrul paginii / frameului.

Implementare

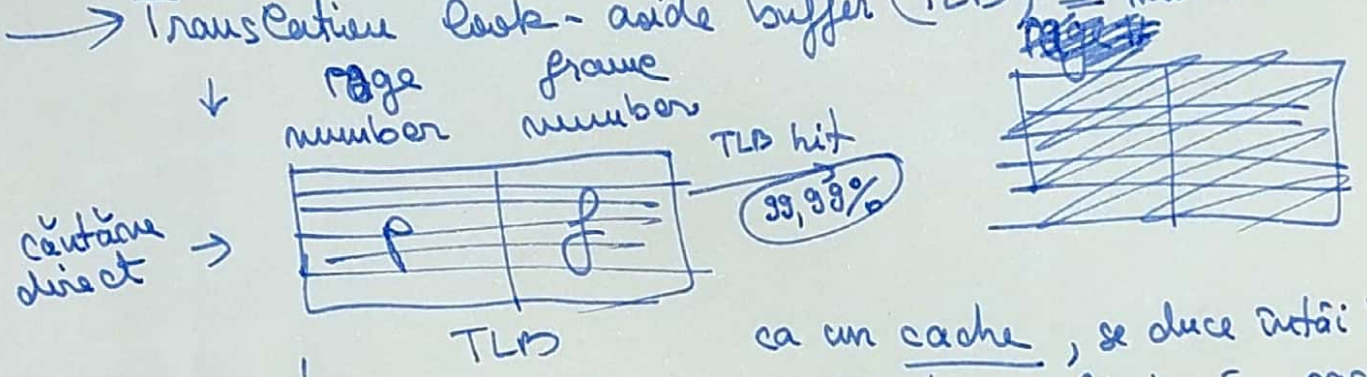
Page-table base register (PTBR) => unde începe tabela de pagină a procesului

Page-table length register (PTLR) => lungimea page tabelului

↓

în MMU

→ Translation look-aside buffer (TLB) = memorie asociativă



ca un cache, se duce întâi în TLB
dc. negăsește → liniar în page table.

↓

TLB miss.
0,01 %

instrucțiuni
↓
zona de text

date
↓
zona de date, stack, heap

departe => 2 TLB pt. fiecare.
(MIPS ex. bun)

Protecția memoriei: bitul de validitate din page table

→ V → OK
invalid → seg. fault
↓
pointează către o zonă invalidă din RAM
inaccesibilă

Shared Pages

pagini identice → partajate → 1 frame pe RAM, chiar dc. avem mai multe procese apelante.

2 nivele de paginare → un tabel cu tabele

→ sol. pt. (nr. mare de pagini) → economicesc spatiu

→ pt. sisteme pe 32

la sistemele de 64 ⇒ 4 nivele de indirectare.