

- se executa doar un proces la un timp

CURS 1

-  $10^9 \cdot 3$  instrucțiuni întă-o secundă pt. un procesor  
cu 3 GHz (succurent)

(lăsat și în următorul bloc de lucru, vă lăs *(ex: împărțește la 0)*)

- dacă apoi 3 instrucțiuni ilegale și nu sunt tratate  $\Rightarrow$   
 $\Rightarrow$  se repernătă CPU-ul

- La legea de date google ca să stergi datele din fizie  
îi le cifrează și anunță cheia.

CURS 2

- POSIX = UNIX API, folosit de Linux, macOS, etc

- ABI = convenție pe nivel de cod  
 $\hookrightarrow$  binary

- fd = file descriptor, dacă uităm să includem în cod, se include procesul, se ocolește din fd și dacă ajunge la 0 se include automat.

- dacă copiem date mai mult decât se pot scrie concurent, pt. c. poate să fie scris destul la noua adresa

ex: while (total\_bytes < target) {  
 current\_bytes = read();  
 // analog write();  
 total\_bytes += current\_bytes  
}

- Piecare syscall are un ID, care se poate schimba syscall-urile pot fi operate prin ID sau nume
  - int sys-write (proc \*p, void \*v, register\_t \*retval)
    - ↳ procedure
    - are place
    - arerea
    - abstractat

↓

args:

    - int fd
    - char \*buf
    - size\_t len

↳ return value

plus în eax

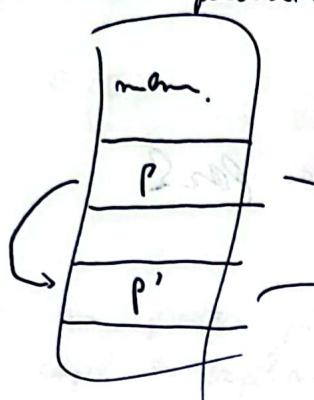
de schimbat

~~syn call org (int)~~ fd; today, ~~new items~~ = 16/1209 -  
~~syn call org (char \*)~~ buf; ~~list of names~~ = 109 -  
~~syn call org (size\_t)~~ len; ~~max size~~ =

Method } \*maple =  $\lambda$  notes' method setpoints obj - by -  
with by key / file access to memory objects so, be  
area. access to objects which is a simple  
pointer

- Prok  $\xrightarrow{\text{auf } [512]}$  Keras  $\xrightarrow{\text{auf } [512]}$  Mora Recurso  
 no é grande, existem 2 tipos  
 rule de prototípico  
 (Report & Obj. sent) divide: 2  
 copy in P  $\rightarrow$  K ( $P_{base} = obj$ ) - tronco  
 copy out K  $\rightarrow$  P ( $K_{base} = substantivo$ )  
 copy in P  $\rightarrow$  K ( $P_{base} = substantivo + adj$ ) - libert

- când facem alt proces, se face o copie identică a procesului parinte.



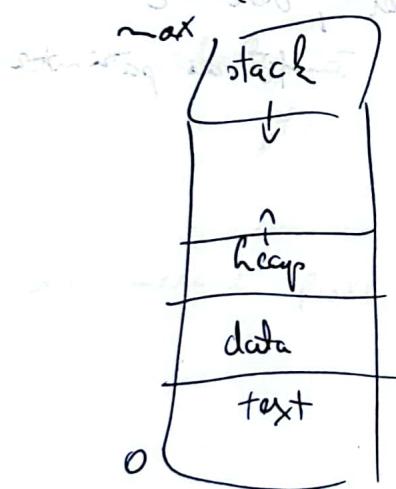
`POO();`

`creat-proces-nou();`

`bar();`

ambele au acces la  
bar dintr-o sarcină  
după pct. `creat-proces-nou()`  
(avă același instruction pointer)

### - atacuri buffer overflow



pt. să adăuga pe stivă, scădem, pt  
să trage adăugările.

`0x100 = big`

`0x100 = small`

`0x100`

- swap partition = pt. hibernare, se face o copie a /page file/virtual memory
- context switch = schimbare de le-așteptare a unui proces cu altul
- daemon de boot = services, se rulează la start, încearcă să înceapă să funcționeze

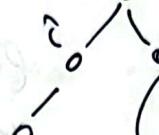
- ssh = protocol de securitate pentru conectarea de la distanță

- fork()

fork()

primul fork

pareinte



al doilea fork

fork()



(retinere în structură de date)

=> 2 <sup>nr. fork()</sup>

-1 se vor crea noi, dor <sup>nr. fork()</sup> cu

tot același parent

pid = fork()

if (pid == 0)

fork()

FORS &

- environment variables

- windows toate bibliotecile sunt împachetate în  
folderul aplicatiei > are nevoie de amv. var. pt  
a accesa resursele din cale relativă  
• prole: bibliotecile fol. pot fi duplicate

anex: bibliotecile sunt separate de aplicatiu, au  
folderul bin

- dacă un proces copil termină mai repede decât părintele să-l captureze cu wait()  $\Rightarrow$  procesul zâmboiu



$\rightarrow$  lărgește părțile când părintele ajunge la wait()

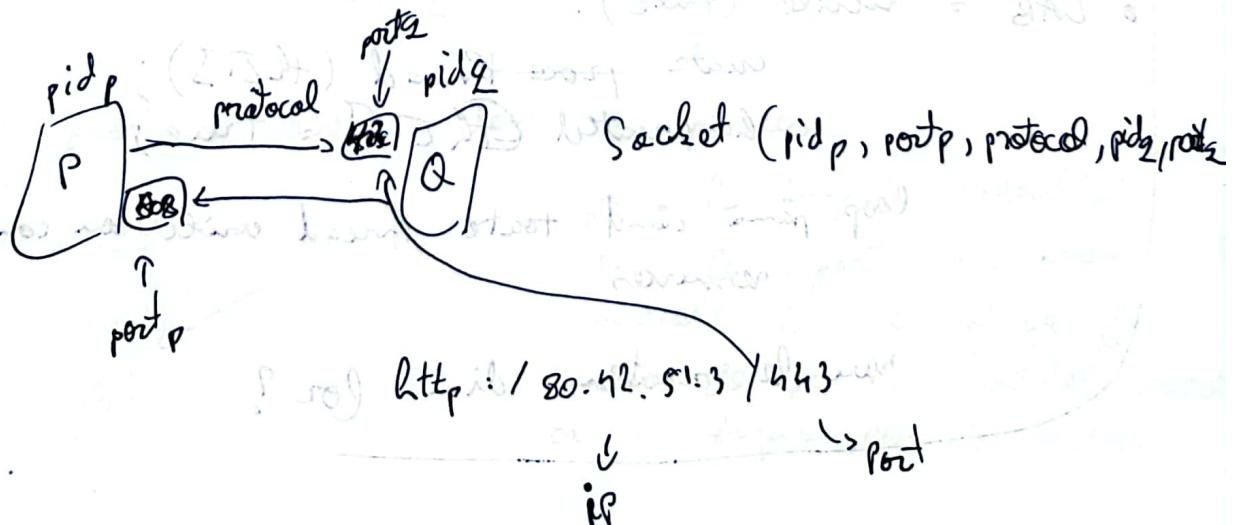
- dacă un proces părinte nu fol. wait()  $\Rightarrow$  procesul copil = orphan

în unele situații  
față de acestea  $\Rightarrow$  copilul poate fi legat de  
nimenic SAV  $\Rightarrow$  când proces părinte moare  
copilul își poate omora la fel ca și părintele

$\Rightarrow$  problema când copilul e legat de procesul root o  
ptc. nu îl mai poate omora nimenic atunci

- comunicare full duplex = ambele procese comunică în același timp.

- pt. lările mutex trebuie să nimice părțile când ore  
verse -> variabilitate de turn



- când strângem linie, nu le strigam, doar scapam de peisajul către el.

~~show.unlim2~~ = strâge legătura dintre painter și pînzer

- memoria este împărțită în bucăți egale



→ 4K

chiar dacă creem un byte  $\Rightarrow$  nu ne oferă 4K  
pt că afă înseși un bucătă  $\Rightarrow$  getPageSize()

~~\$ cat foo.txt | grep "bar" | grep -v "bar"~~

ia ce se află în foo.txt  $\Rightarrow$  îl dă ca input

în grep "bar"  $\Rightarrow$  scoate linile care contin

"bar"  $\Rightarrow$  îl dă ca input în grep -v "bar"  $\Rightarrow$

2) rămân linile care contin "bar" dar nu "bar"

o LAB = while (true):

create procThread (th[i]);

hasCreated (th[i]) = true;

loop pînă când toate threadurile au consumat  
resources

suntem creați dimidii for?

- dacă thread-urile unui proces crează → crează teste  
procese și ~~care~~ ~~care~~ thread-urile procesează și  
procesul în sine.

- multithread-un = poate procesa ca se întâlnesc, etc.  
un thread nu poate să aibă la  
memoria altor lărgi.

- un server = poate să proceseze aplicații  
diferite chiar dacă nu fac nimic  
în aşteptări clienti.

- concurrency:

Do single cores:  $T_1, T_2, T_3, T_4, \dots$

The diagram shows a horizontal timeline with arrows pointing to the right, labeled "Time". Above the timeline, there is a sequence of labels:  $T_1, T_2, T_3, T_4, \dots$ . This indicates that each thread  $T_i$  starts at a certain point in time and continues until it finishes, with the next thread  $T_{i+1}$  starting immediately after the previous one has finished.

- parallelism:

core 1:  $T_1, T_3, T_6$

core 2:  $T_2, T_4, T_5$

time

- e posibil să avem deasupra un proces

(CPU)  $\leftarrow P_0, P_1$

(I/O)

ptc. în timpul executiei  
lui  $P_0$ , poate să aibă  
necesitate de I/O și va fi  
nevoie de pe CPU. astfel dacă  
se face să pună CPU să fie  
deosebit de eficient.

- optimizarea accesului la memorie prin decouplarea de la CPU
  - $CPU_0 \leftarrow p_0, p_1, \dots$

- $CPU_1 \leftarrow p_2, p_3, \dots$
- în schimb, în loc să fie decouplat, memoria este încă în contact cu procesorul și urmărește programul să fie executat.

- $CPU_0$  și  $CPU_1$  au același strat de memorie
  - $CPU_0$  și  $CPU_1$  au același strat de memorie
  - memoria este încă în contact cu procesorul și urmărește programul să fie executat.

$p_0$  nu este în cache și este preluat din memoria.

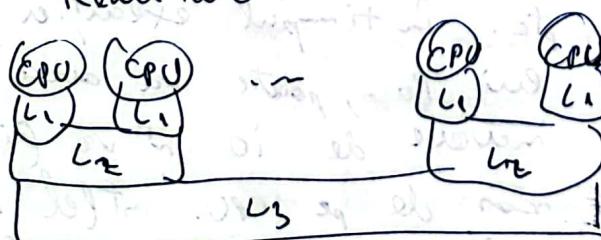
dacă  $p_0$  nu este pe  $CPU_0$  și după eșec, vom căuta tot pe  $CPU_0$ , căci este de la cache și rezultă în  $CPU_0$ .

>Selectie:



$L_1, L_2$  = nivele cache

Realitate:



$P_0$

- data parallelism = acelerare operatiile pe liniile cu dim date.

task parallelism = operatiile disponibile

- ~~sp~~ Amdahl's law:

$$\text{speedup} \leq \frac{1}{S + \frac{(1-S)}{M}}$$

$S =$  porțiune serializată  
 $M =$  nr. de core-uri

- în Python thread-unile sunt many-to-one  
din cauza interpretatorului  $\Rightarrow$  nu există paralelism  
la thread-unile

Soluție: facem procese, fiecare proces are interpretatorul său

CURS

- thread pool = punem 100 de thread-uri care nu  
fac nimic, când vine oare către ceva  
apelați un thread.

$\hookrightarrow$  nu folosește wait()

- OpenMP = punem paralelizare

- dacă avem procesuri la thread. și dăm porșcă  
din el  $\Rightarrow$  poate avea unul să se petreacă  
depinde de OS

există puncturi porșcă aleginte pt. a face asta

- SIGNAL = CTRL+C este un signal

SIGKILL = Închide un proces

CTRL+Z = Interrupe procesul curent, punând-l în background

Ctrl+D emisie → pt. a融化 conversie în shell  
se poate să fie în mid-process

SIGINT

• SIGWINCH

KILL (pid, nsignal)

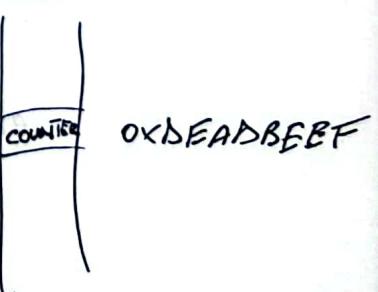
SIGNAL (nig, (\*poo)(1)) → pt. signalul ridicat apelarea pt. foo

- Thread cancellation:

- asynchronous - imediat
- deferred - checks periodically if it should be checked

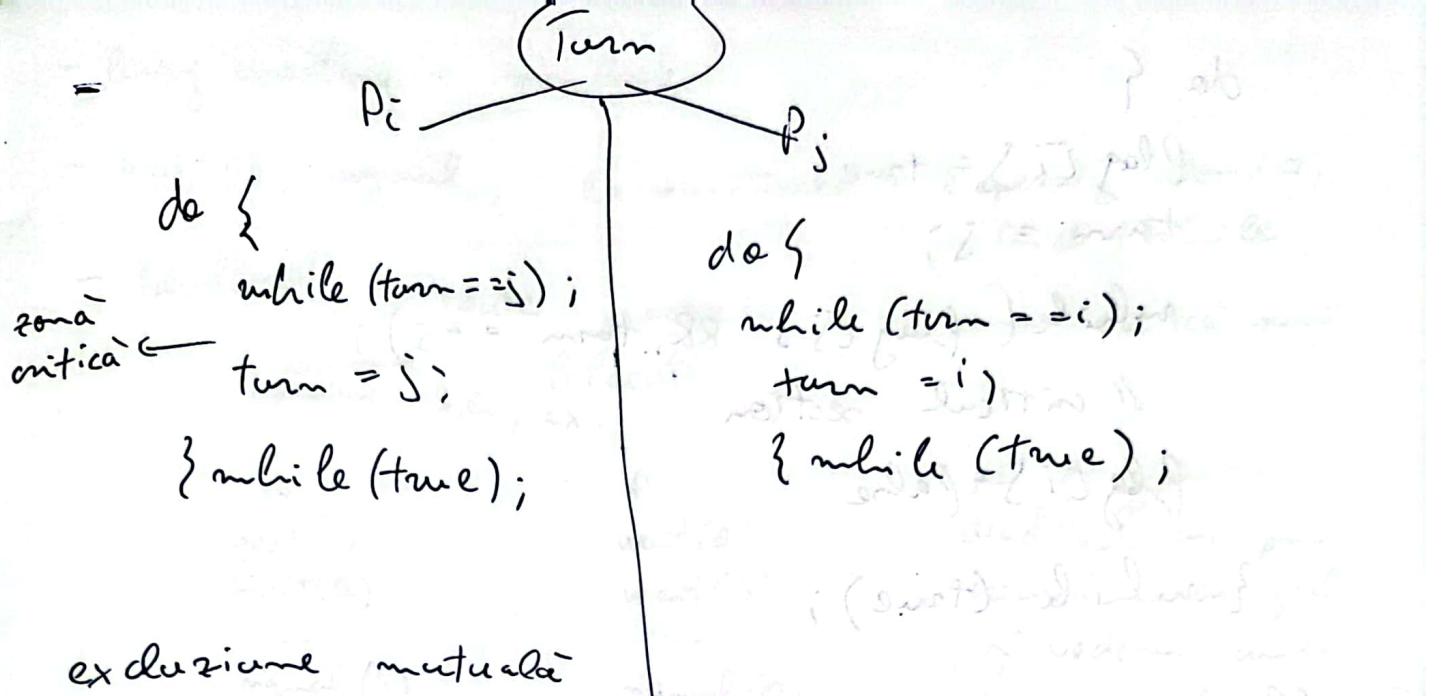
- counter ++

mov EAX, [0xDEADBEEF] ← număr



inc EAX

mov [0xDEADBEEF], EAX



exclusiune mutuală

! Dacă dispore un proces, celelalte pot rămâne blocați în limită

Totodată dacă sunt mai multe proceze e mai greu

- mutual exclusion  
procese

Round robin waiting

- un proces cu prioritate înaltă care se executa de mai multe ori poate bloca celelalte proceze care este aptă  $\Rightarrow$  se incalcă regulile de round robin waiting.

- preemptive  $\rightarrow$  poate fi intrupt

- non-preemptive  $\rightarrow$  dacă ajunge pe Kernel nu poate fi intrupt

- peterson's solution

```

do {
    flag[i] = true;
    turn = i;
    while (flag[j] && turn == s);
    // critical section
    flag[i] = false;
} while (true);

```

flag = reprezintă domnia procedură de care

execută

CURS

- lock test\_and\_set(\*lock){

w = \*lock;

\*lock = true;

return w;

}

do {

while (test\_and\_set(&lock));

// critical

lock = false;

// remainder;

} while (true)

→ nu se respectă  
termenul de  
acțiere

ex:

proc: P<sub>0</sub>, P<sub>1</sub>, ..., P<sub>n-1</sub>, P<sub>n</sub>  
pri; H H ... H L

⇒ n → ∞ nonstop -

- busy waiting = spinlock
- wait(), signal, la renapoare = 2 operații atomică, îndivizibile
- deadlock = toate procesele din zonă critică sunt 2 renapoare 2 procese S,Q, elocate, ex:

$P_0$	$P_1$ , poate $\Rightarrow$ fi gâtă
wait(S)	deadlock - un prim pas
wait(Q)	crearea unei grafuri de vede unde
...	avem acoperișuri
signal(S)	signal(Q)
signal(Q)	signal(S)

- starvation = procesul pe care nu îl poate scoce din renăzoruri inițiale

- priority inversion = lower-priority process holds a lock needed by higher-priority process

dacă un proces temporal este în zonă critică, atunci ia ceea cea mai mare prioritate dintre toate cele care o protează.

- problema: bounded-buffer readers and writers dining Philosophers

într-o reprezentare de lucru cu multe filozofi care să mănânce la un număr limitat de masațe.

CURS

1. Los ejemplos de los errores de la ejecución.

ultimul thread va fi degenerat ca sa  
eliminare celelalte thread-uri. Functionarea dar  
nu e bine.

- CPU scheduler:

- 2 tipuri de algoritmi: - preemptive  
- non-preemptive

- dispatch latency = cat timp rini ia sa pun alt proces pe CPU

- scheduling criteria:

- CPU UTILIZATION  $\rightarrow$  CPU nu fie Busy mult
- THROUGHPUT  $\rightarrow$  cate procese pot fi executate/time unit
- TURNAROUND TIME  $\rightarrow$  cat ri ia unui proces sa execute
- WAITING TIME  $\rightarrow$  cat timp asteptă în mait queue
- Response Time  $\rightarrow$  user time

- Alg. pt scheduling:

- FCFS = first come, first served

P <sub>1</sub>	P <sub>2</sub>   P <sub>3</sub>
0	24 27 30

waiting time: P<sub>1</sub> = 0

P<sub>2</sub> = 24\*

P<sub>3</sub> = 24+3

average = (0+24+27)/3 = 17

dacă le sortăm  $\Rightarrow$

<del>P<sub>1</sub></del>	<del>P<sub>2</sub></del>	<del>P<sub>3</sub></del>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>
0	3	6			30

waiting = P<sub>1</sub> = 0

P<sub>2</sub> = 3

P<sub>3</sub> = 24+6

average = 3

- SJF (shortest job first) : să luăm tot ce e mai scurt

- de unde să înțeleg că o nă. și ia unui proces înainte să fie executat?

(regresie liniară)

$T = \text{ce ghicesc}$

$t = \text{cât a stat}$

$$\Rightarrow \bar{T}_{m+1} = \alpha t_m + (1-\alpha) \bar{T}_m$$

$$0 \leq \alpha \leq 1$$

dacă  $\alpha = 1 \Rightarrow$  iau cătă a stat cel anterior  
 $\alpha = 0.5 \Rightarrow$  o medie liniară

de alici.  $\alpha = 0.5$

Cât e  $\bar{T}_0 = ?$

$$(1-\alpha) \text{ va fi } (1-\alpha)^{\infty} \text{ pt } \bar{T}_0 \quad \left| \begin{array}{l} \text{este } 0 \\ \text{stănd } 0 \end{array} \right. \Rightarrow \bar{T}_0 = 0$$

- SJF preemptive (shortest time remaining time first)  
 execut un proces și dacă nu este în așteptătură, procesul să se execute mai puțin, îl pun pe el

	Arrival	Time	Process
$P_1$	0	8	
$P_2$	1	0-4	= partition
$P_3$	2	4-9	
$P_4$	3	9-5	

$P_1$	$P_2$	$P_3$	$P_4$	$P_5$	$P_6$	$P_7$
0	1	5	10	14	26	

average waiting time =  $\frac{(10-1) + (1-1) + (17-2) + (5-3)}{4} = 6.5$

deafach nu, dar poate apărea starvation

soluția: aging (cum că stai mai mult în coadă și atunci ești prioritar)

### • round robin (preemptive)

fiecare proces primește un timp, dacă îl depășește și next.

probl: dacă quantum-ul



este prea mică → prea multe context

switch-uri

acum se face și dacă e prea mare

face context switch

→ nu am rezolvat

acum nevoia de a schimba din nou

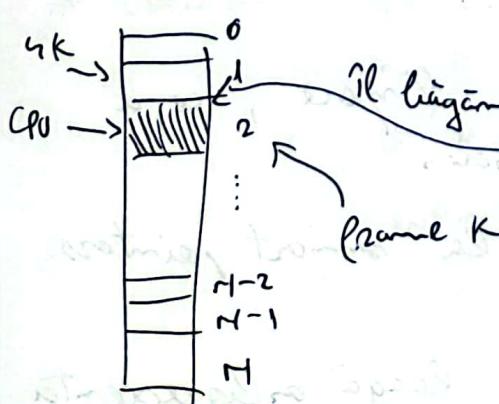
memorie

(Curs)

- lucratile în care e împărțită memoria = frame-unitate

mem

disk



il luăm în memorie



block

512B

4K

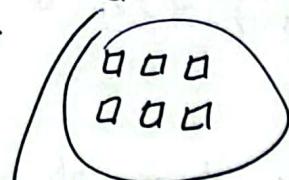
(C, H, S)

C = cylinder

H = head

S = sector

→ metodațe despre fișiere



→ aici se află foo.txt, dar doar continutul, nu stiu ce tip, nume, etc.

→ dacă pierdem fișierul astăzi am pierdut datele.

- dacă avem 2 fișiere la fel (duplicate)

putem sănătatea fișierul și să numărăm  
adăugăm -1, -2, -3 (mindown)

sau să hashăm și să punem numele cheie

- dacă primim acel același fișier din  
mai multe surse → nu vom avea duplicate  
(de dupicare) pt c. să aibă același hash,  
același nume deci nu are să fie copia

- Linux /proc → retine informații despre procese

foo.txt

foo.txt



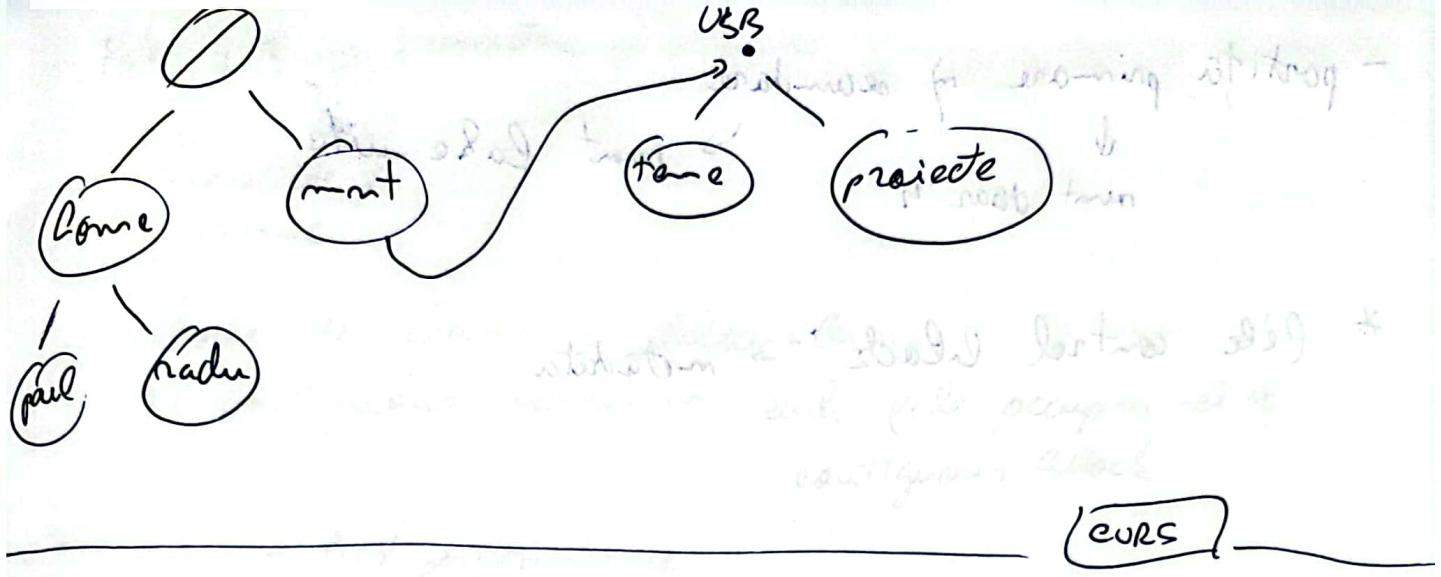
foo.txt

→ dacă folgem conexiunile  
(curl -L) nu mai putem accesa fișierul.

acea se face când ne șterg fișiere, cea  
pește acolo va fi suprascriasă.

funcționarea ca la smart pointers.

- operația de montare la USB: legătură rezervată  
USB-ului de root/mont



- orice program în orice OS are legături cu librării.
- Logical file system → traduce nume și path-uri în index (file descriptor)
- File systems = Windows: FAT, FAT32, NTFS  
Linux: UFS, FFS
- \* FAT e folosit pt stocarea pt că poate fi folosit pe orice OS, FAT e foarte basic și elementele sunt multe, multe și sunt ușor de accesat
- \* boot control block = informații necesare pt sistem pentru a boota OS-ul din acel volum
- \* volume control block (superblocks, master file table)
  - numărul de blocuri
  - numărul blocurilor libere
  - block size
  - free block

În final, sistemul va rula și va fi posibil să se pună și să se stearbeți.

- partitie primare și secundare

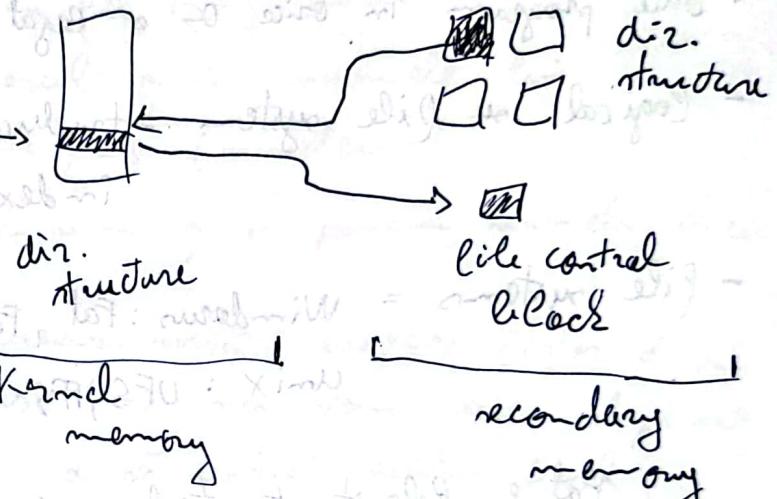
↓  
număr deacă

→ sunt false-nite

\* File control block → metadata

read(index)

open(filename)



- partitia deasemenea = nume (nu are fișier sistem, doar  
sequenție de blocuri)

- VF (Virtual file system - API)

- implementare Directory:

- listă liniară cu nume directory și pointer  
către locații de date (contințor și conținut)

- hash table (coliziuni)

foo.txt

bar.txt {merg}

minicu-jpg → dimensiune diferență, trebuie  
adăugat padding la  
celalalte două

- Fapt 8.3  $\rightarrow$  3 caractere pt extensie

1/  
3 caractere pt  
nume

- metode de alocare a blocurilor:

1) contiguous allocation: each file occupies set of contiguous blocks

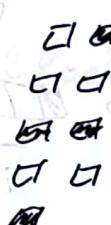
- best performance

- simple

- problema la generare de spatiu

ex: *baz.txt*

3 blocks



$\rightarrow$  neatenem spatiu baz

2) puternice defragmentare:

metoda care alocă și menține blocurile ocupate la început.

metoda costisitoare, trece pe fiecare pt. picătare  
lizier și nu modifică  
FCB-ul (file control block)

$\rightarrow$  și poate face online sau offline  
(downtime)

- ca nu are acces în memoria memorie:

logical address  $\leftarrow$  CA / 512  $\rightarrow$  cat = la loc

$\rightarrow$  rest = unde în la loc

2) extent = importanța fizicul în mai multe  
parti (extenții) contigune.

### 3) Circular list

- locurile se pot afla in locuri complete  
diferente  $\Rightarrow$  nu stiu
- dacă un la pointer la următoare se retrage  
nu mai putem parcurge totă lista.  
(nu poate întâmpla ca închiderea  
șertată / ameață a călcătării)
- putem face extazuri și aici  
(internal fragmentation)



înosit, dacă avem un extaz  
de 5 locuri era perfect

accesare memoria  
logical address  $\leftarrow$  offset / 511  $\rightarrow$  cat = locul din lista  
timon minte  $\rightarrow$  rest + 1 = displacement în  
locul respectiv  
în un pointer  
(continuă)

La mijloc proprietatile  
size = cat are fizicul  
on disk = cat ocupă pe disk (cu totul  
redondantă)

pointer de 8 biti  $\Rightarrow$  256 max la locuri

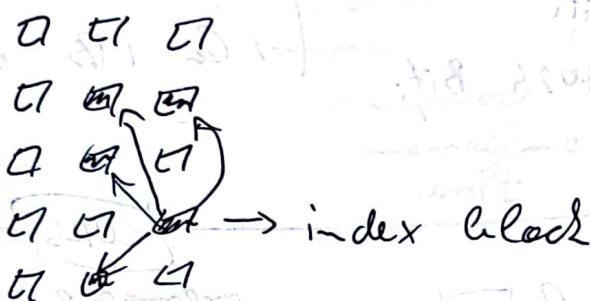
$\Rightarrow$  max file size =  $256 \cdot 511$  bytes

- bad  $\rightarrow$  când un block nu mai poate fi folosit  
nu este corect.

hard-unile noii de 1TB vin de la part cu 1,5-2TB  
în care se trimiț căsuțe de block-uri

#### 4) Indirect allocation:

- un block cu metadata care pointează către fizicele celelalte



- acceseaza memoria:

$$4A / (512 \cdot 512) \rightarrow \text{cat},$$

rest,

$$R_1 / 512 \rightarrow \text{cat}$$

rest

- UNIX VFS:

mode

owner

size

direct block  $\rightarrow$  data

single indirect  $\rightarrow$   $\square \rightarrow$  data

double ind.  $\rightarrow$   $\square \rightarrow \square \rightarrow$  data

triple ind.  $\rightarrow$   $\square \rightarrow \square \rightarrow \square \rightarrow$  data

→ intel iCore i7 3,46 GHz = 159.000 mips

Hdd = 250 io/s

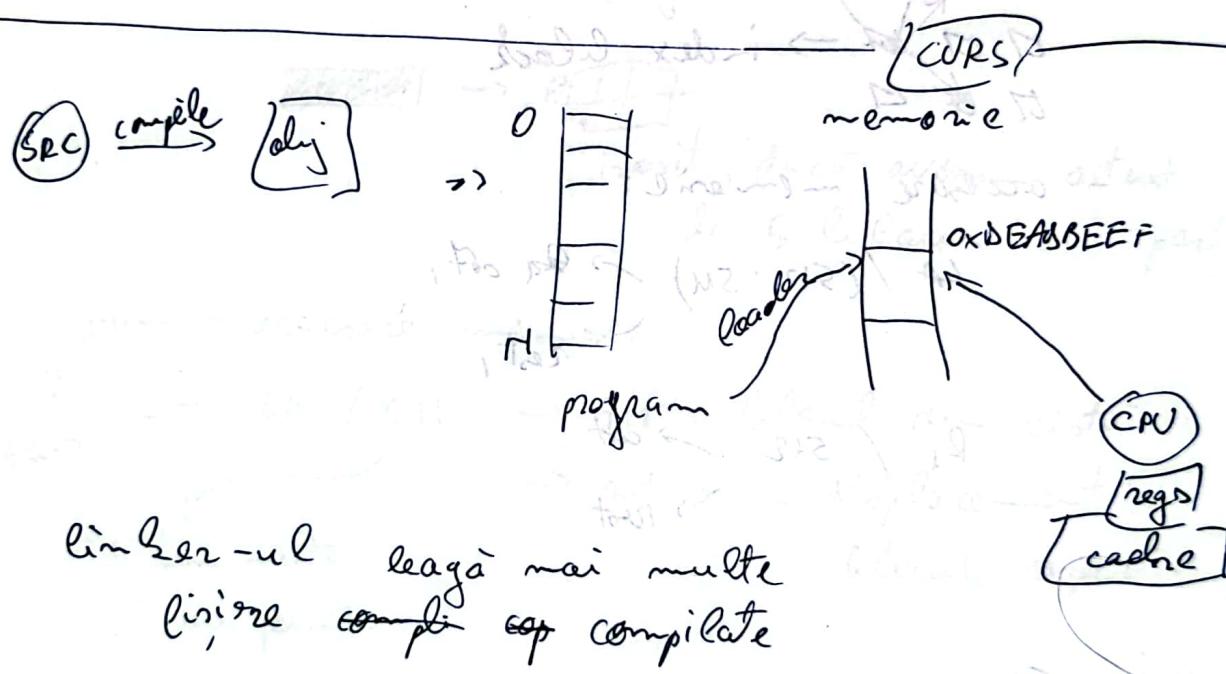
SSD = 60.000 io/s

$159.000 / 60.000 = 2.65$  mil. io/s rămasă ptă secundă

- HDD urmărește să încerce să scrie 1000 GB, rezultă

1k = 1000 biti

dacă 1k = 1024 Biti,  $\rightarrow$  1GB = 1TB se vede

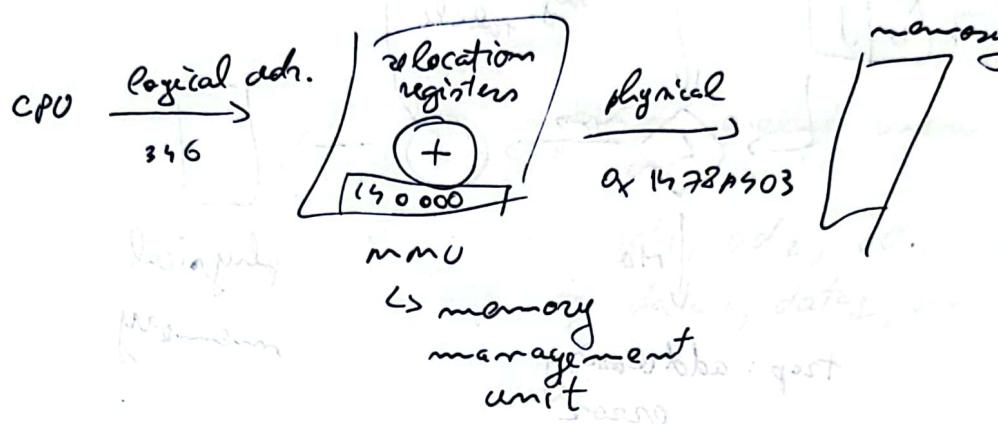


- compile time
- load time
- execution time

- .dll sau bibliotecă externă, putem presupune că există deja în memorie, dacă nu  $\Rightarrow$  noare.

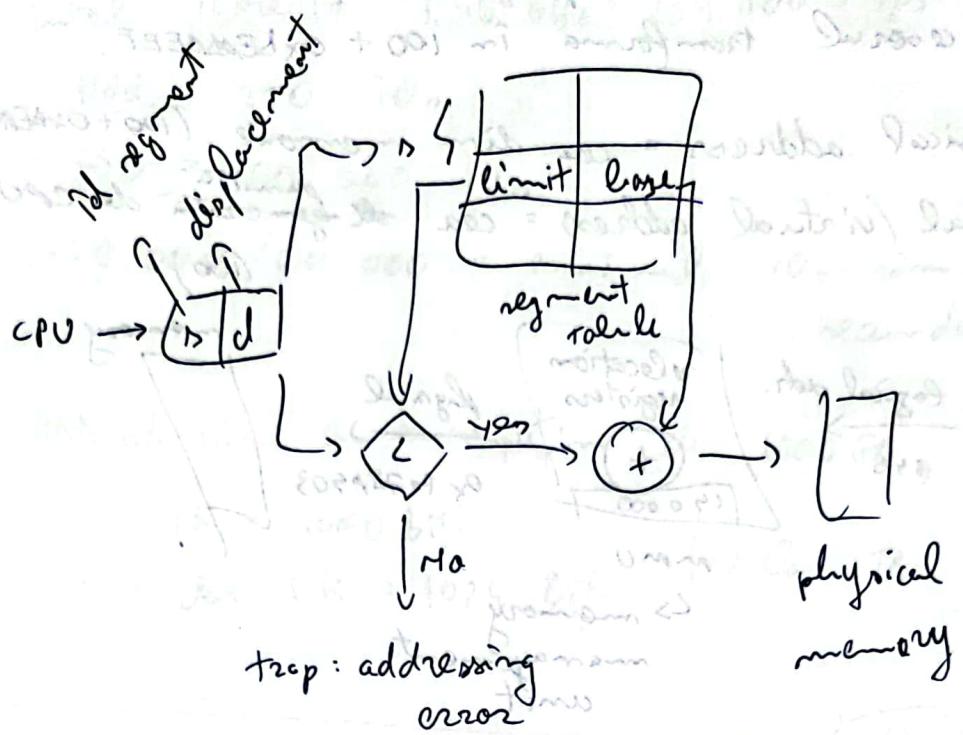
Pute să să le introducem în programul nostru

- dacă apelăm o pct. din prog. de la adresa 100 =>  
 => procesorul transformă în  $100 + 0x\text{DEADBEEF}$ .
- physical address = cea din memorie ( $100 + 0x\text{DEADBEEF}$ )
- logical / virtual address = cea de <sup>primă</sup>~~generată~~ de CPU  
 (100)



- static linking → biblioteca e pusă în program =>  
 => pot exista duplicate biblioteci
- dynamic linking → se luă biblioteca de program la runtime
- La static dări biblioteca are probleme =>  
 => trebuie recompilat tot programul vice versa cu linkare și linkare
- La dynamic nu
- mapping → un proces poate fi mapped temporar din memorie și după puțin înapoi.

- dynamic storage-allocation problem:
    - first-fit → prima bucată de memorie care e destul de mare
    - best-fit → cea mai mică bucată destul de mare
    - worst-fit → cea mai mare
- Facți lucruri mai ușoare și multe



- paginare = în memoria procedură (adresare reală)
- frame-uri în memoria fizică

- Exercițiu:

$(P|d)$  reprezintă

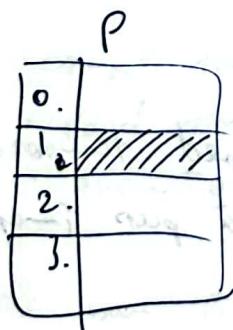
0	1	2	3
---	---	---	---

$\xrightarrow{\text{CPU}} (p|d)$

4 biti  
word size

$p = 2$  biti  $\Rightarrow 4$  pagini maxime

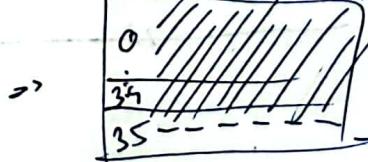
$d = 4$  biti  $\Rightarrow 2^4 = 16$  bytes



- page size  $= 1024$

$1024 = 2^{10}$

$\Rightarrow 35$  pages + 1024 bytes  $\Rightarrow$



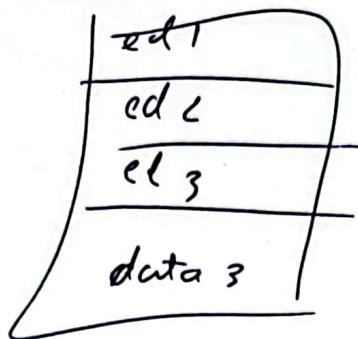
$\Rightarrow$  locația de aici nu va fi folosită  
 $\Rightarrow$  internal fragmentation



`proc1`



`proc2`



`proc3`

$\rightarrow$

0	<code>data1</code>
1	<code>data3</code>
2	<code>ed1</code>
3	<code>ed2</code>
4	<code>ed3</code>
5	
6	
7	<code>data2</code>

$\Rightarrow$  nu trebuie să parăm același nume de  
mai multe ori  $\Rightarrow$

$\Rightarrow$  reținem `ed1`, `ed2`, `ed3` deoarece  
dată și `data1`, `data2`, `data3`