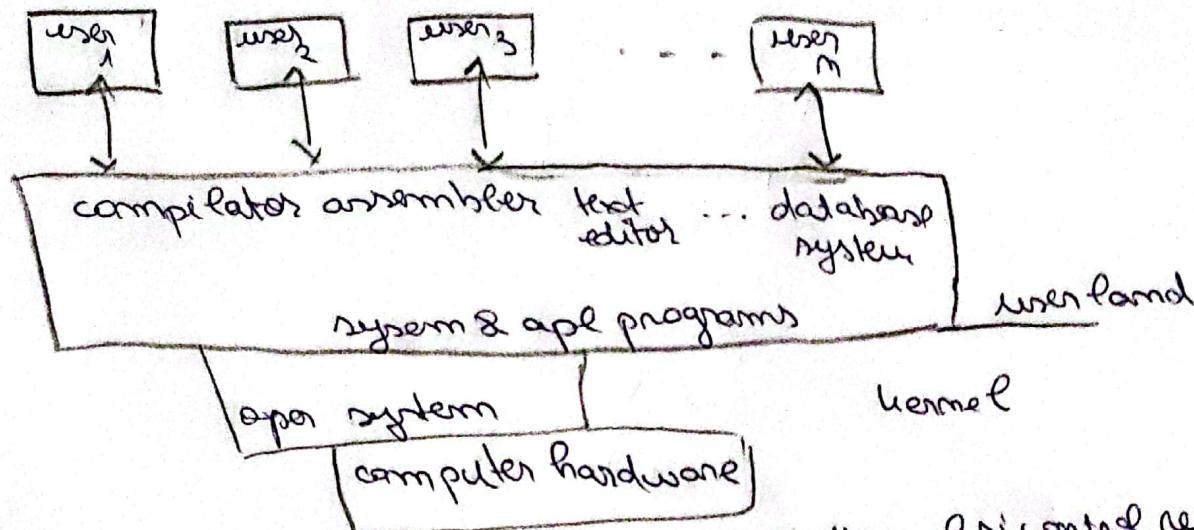
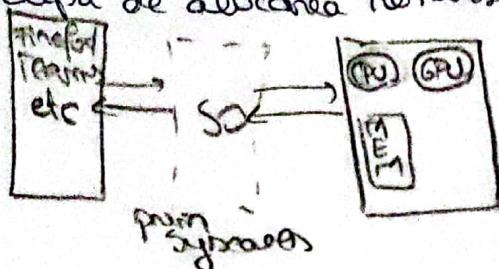


# Teorie SO

## 1) Introducere

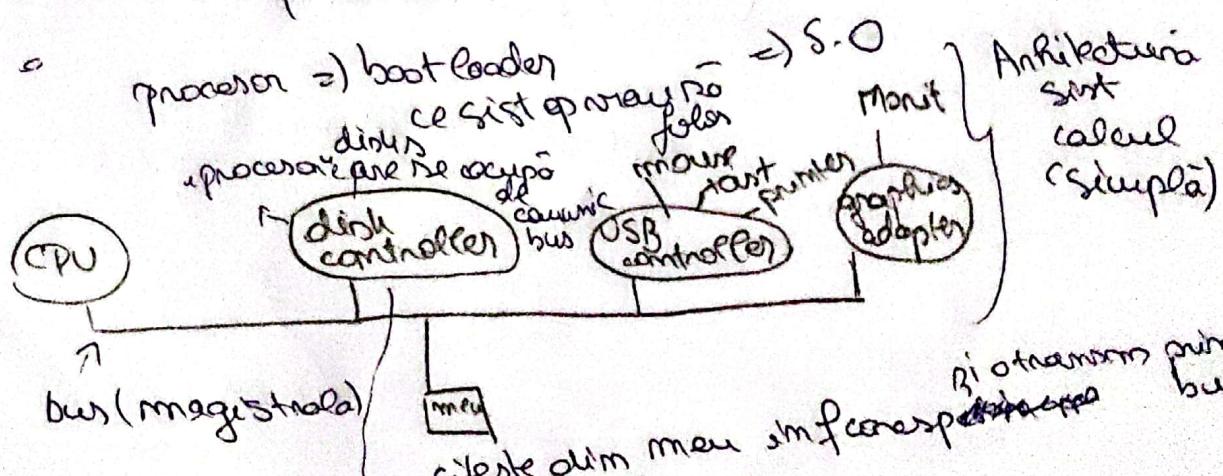
(e e un sistem operare)

- program interne între user - hardware
- se ocupă de alloarea resurselor + controlarea programmei



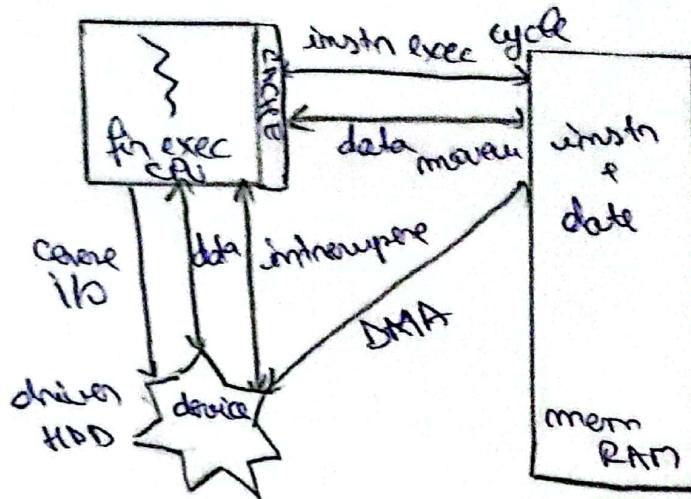
- Kernel = partea S.O. care realizează tot din punct de vedere hardware

- processor  $\Rightarrow$  boot loader  $\Rightarrow$  S.O.

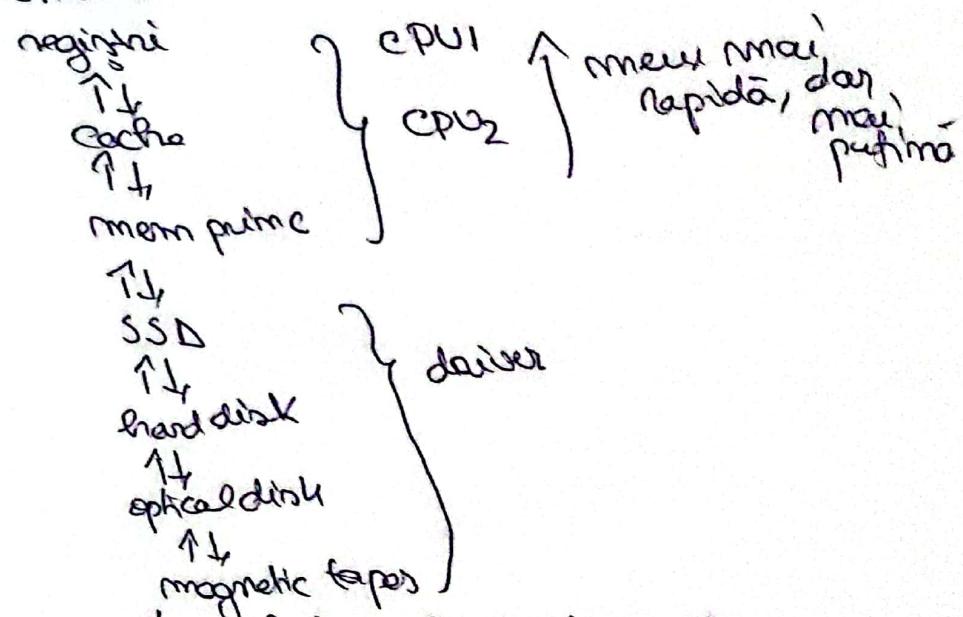


- driver - identifică bucată hardware
- comunică acest bucată cu CPU

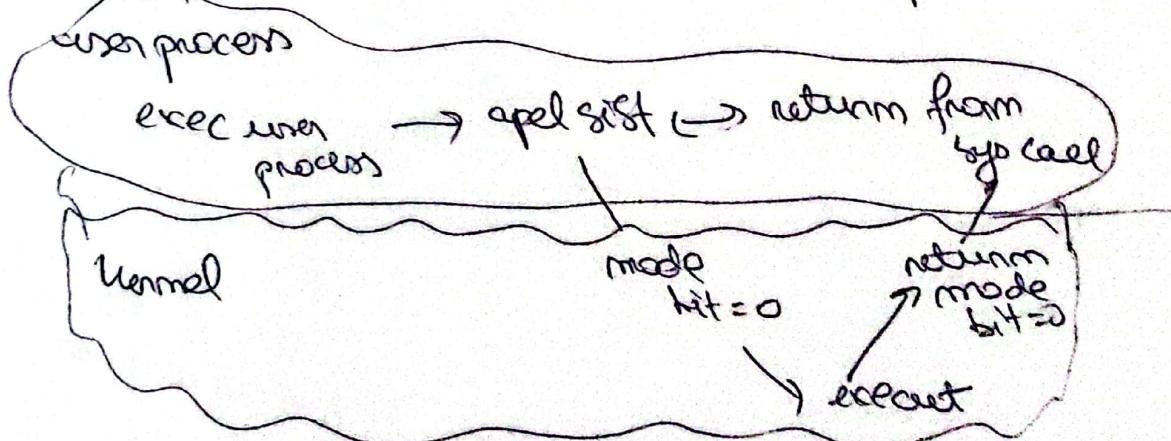
↳ fund calculator (Arch von Neumann)



↳ Hierarchia memoriei

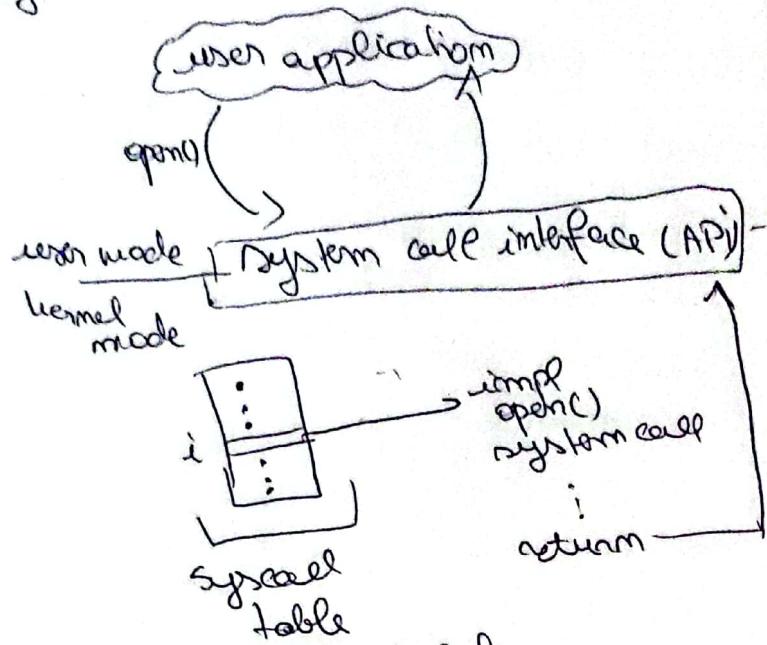


↳ transz. user  $\rightarrow$  kernel (dual-mode oper)



↳ funcții sisteme (se apelă în fot de un index)  
↳ se află într-un tabel

↳ API system call

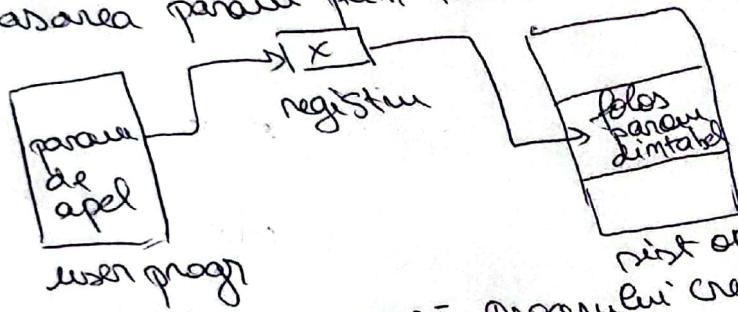


adding pe  
stg la  
param  
system  
callului

push arg 0 }  
push arg 1 }  
push arg N }

pop arg N } focus  
pop arg 1 } de  
syscall  
impl

↳ pararea param. prin tabel

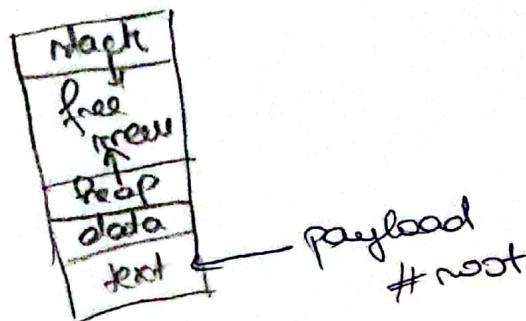


- ↳ sist de operare parcurge procesul creat o listă de impreună cu cele să aferze (unde să devină etc)
- ↳ procesele sunt luate în mem prim într-o coadă de priorități
- ↳ arbore pt proces → proces ⇒ proces copil
- ↳ introduc pt că pe o listă nu ne poate face binary search
- ↳ căutare a unui proces în timp log

## 2) Procese

- ↳ program im execute (cod executivă)
- ↳ spārți cod programului → text section  
activ curență → program counter, registrii
- stack (pt ce alocă în fundă a nōmas cu exec  
data section static)
- ↳ heap pt alocare dinamică  
(malloc)

↳ procese în mem:

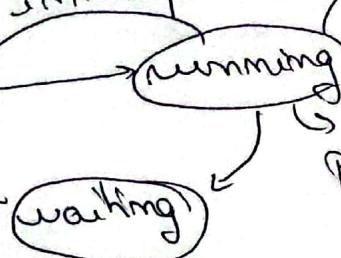


↳ starte unui proces  
fork, exec



dacă CPU are timp, poate exec ceea  
din program

↳ interrupt



↳ PCB (process control block)

acceptă cao imnēant  
op să fie realizat ⇒ ready

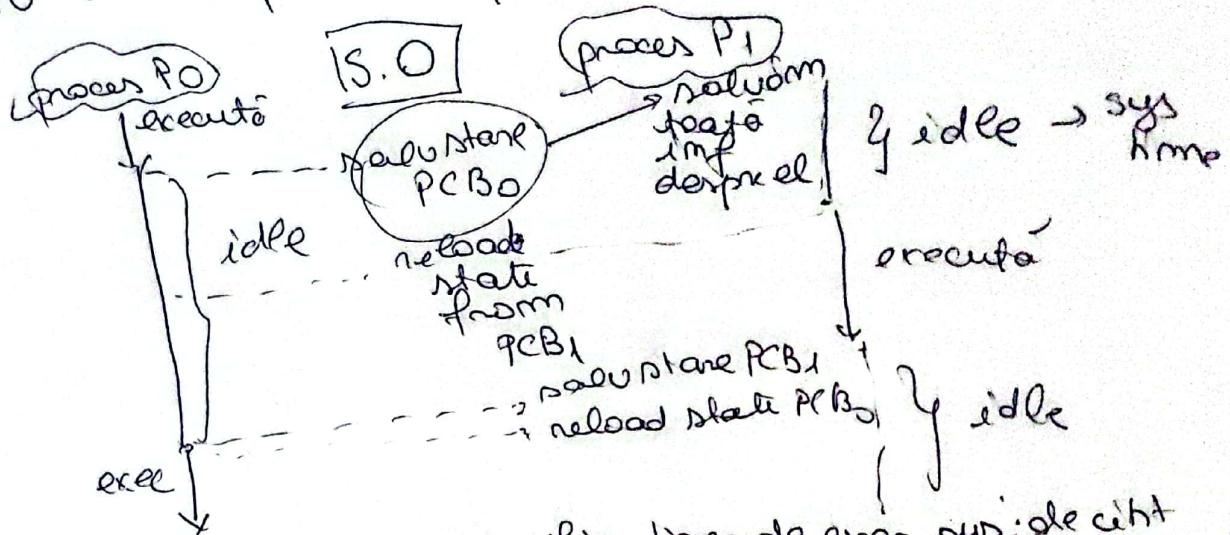
| C ca procesul să nu blocheze  
CPU-ul,

struct

- ↳
- process state; (new, ready, waiting -);
  - process number; (pid → cod de identificare);
  - program counter;
  - registri;
  - memory limits;
  - list of open files (stdin, stdout, stderr)

↳ PCB;

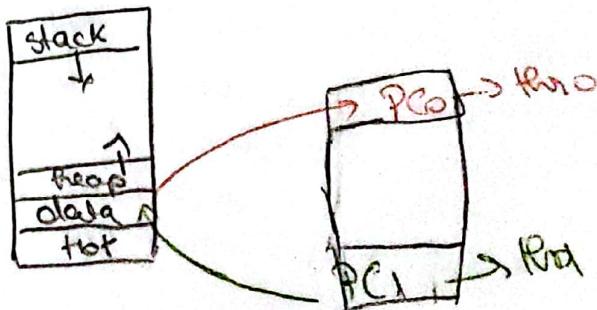
↳ CPU switch proces la proces



C time-ls pentru a verifica timp de exec sys: de cînd  
fisiere din tabela de file system

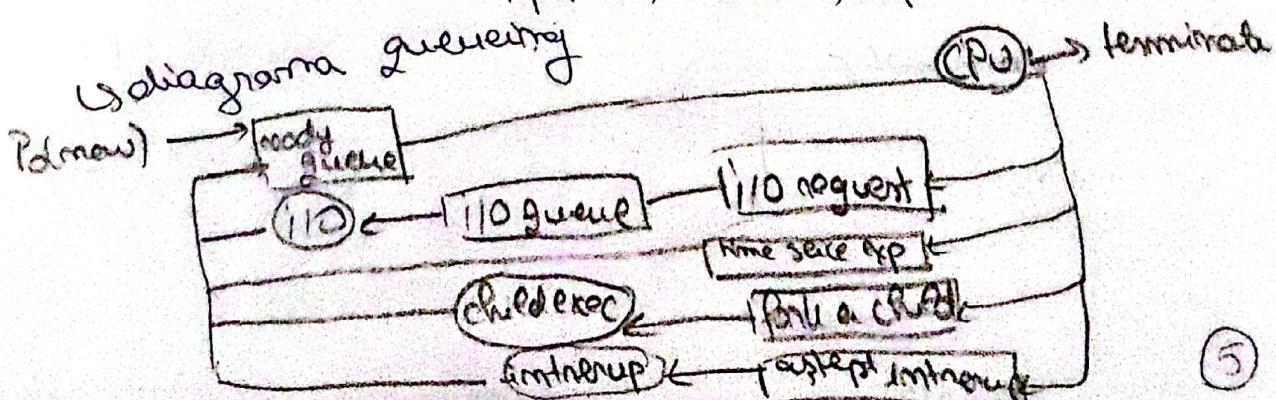
o timp real: user space + kernel space

- o **THREAD** → un thread poate fi folosit pentru un singur proces, un proces poate avea mai multe threads
- ↳ exec simultan mai multe zone de cod și date dintr-un program
- ↳ efect acum



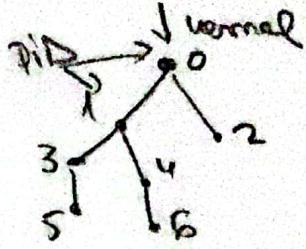
- o **Process Scheduling**, se dîmna procesele disponibile pe CPU
  - ↳ coada de procese (acersta înregistrată între cele 3 tipuri)
  - ↳ ready queue (procesele din memória gata să aștepte execuție)
  - ↳ job queue (blocurile de dimisii)

↳ diagrama queuing



5

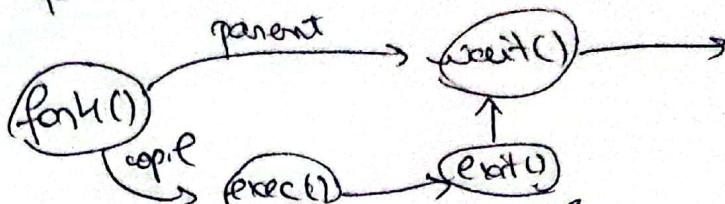
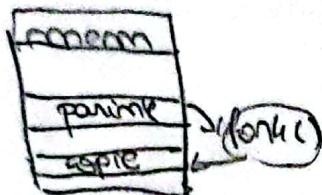
\* creare proces  
 - părinte  $\Rightarrow$  copiu  $\Rightarrow$  alți copii  
 arbore de procese



- fiecare proces are un pid (identifier)

- părintul și copii se pot exec număr

↳ părintele poate exec părintea cănd termină copilul



- fork() - copiază "mee părintelui" în meu copil  
 ↳ dăcă nu avem suf mem ram  $\Rightarrow$  eroare (dintră dedesubt)
- dacă pid < 0  $\Rightarrow$  nu s-a putut deschide procesul copil  
 ↳ pid > 0 bucată de cod se execută identic

### Terminarea unui proces

S. O eliberarea resurselor unei program

↳ exist()

↳ returnarea rezultatului de la părinte la copil  $\Rightarrow$  wait()

↳ părintele poate termina ex unui copil  $\Rightarrow$  assert()  
 (de ce depărțesc res. alocate, tacilul copilului  
 nu mai este necesar) părintele dă exit()

↳ procesul părinte așteaptă exec unui copil  $\Rightarrow$  wait()

o dacă unul părinte nu dă wait()  $\Rightarrow$  ZOMBIE

↳ procesul D-a încheiat (nu putem  
copil)

sterge PCB-ul  
dim siy

o dacă părintele a terminat exec și nu a făcut  
wait()  $\Rightarrow$  ORFAN

Exemplu:

ZOMBIE  
 $pid = fork();$   
 if ( $pid == 0$ ) {  
 || proc copil  
 exit(0); }  
 else  
 {  
 || proc părinte  
 sleep(30); }

ORFAN  
 $pid = fork();$   
 if ( $pid == 0$ ) {  
 || proc copil  
 sleep(30); }  
 else {  
 || proc părinte  
 exit(0); }

```

for i=0; i<5; i++)
    \ pid=fork();
    if (pid==0)
        break;
    \ if (pid!=0) → poate fi
        wait(NULL); → informații căptă de la un copil
                        → zombie: copil se exec mai repede

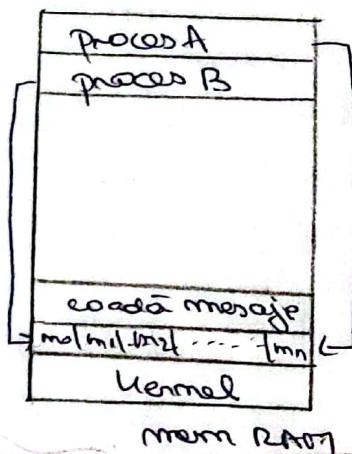
```

### • Multiprocess architecture - chrome

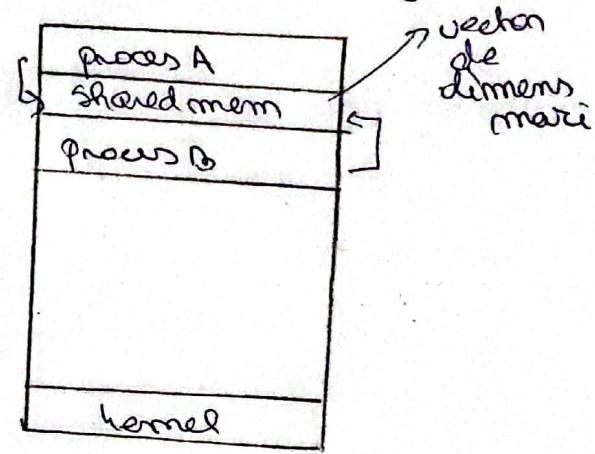
- ↳ 3 tipuri procese
  - browser: user interface, dule + network I/O
  - renderer: se ocupă de HTML, JS
  - plug-in

### ~ Comunicarea dintre procese (IPC)

#### 1) pasarea mesajelor



#### 2) Shared memory



**ex** problema producător-consumator

(ex: copiere fizie, cureau web)

↳ buffer

↳ pt web producător-consumator

Server - browser  
cu web

Shared memory (bounded-buffer)

#define BUFFER\_SIZE 10

typedef struct { int item; } item;

item buffer[BUFFER\_SIZE];

spunse

unde sunt

int in=0;

int out = 0;

pointer out



o produce item next-produced;

while (true)

↳ produce un item

while ((im < 1) & BUFFER-SIZE == out)

↳ free item  
buffer(im) = next-produced;  
if producer can't cons  
 $im = (im + 1) \% \text{BUFFER-SIZE}$

y

, am ajuns la producția maximă

item next-consumed

while (true)

while (im == out)

if consumer can't consume

next-consumed = buffer[ad]

out = (out + 1) %

BUFFER-SIZE

if consumer terminal

y

↳ Shared memory : controlul este la nivel de proces (decid unde și cât va fi mem partaj)

P<sub>1</sub>    P<sub>2</sub>

↳ message passing 1/2 fete

send(message)

receive(message)

Send(P<sub>1</sub>, mes) → transmit mes către P<sub>2</sub>

receive(P<sub>2</sub>, mes) → prim mes de la P<sub>2</sub>

full-duplex

half-duplex

2) communication link

Implementation

- fizic

shared mem  
bus  
retea

printn-un  
mem  
(part)

- logic

direct / indir  
memori extin

automat / explicit buffering

blocking (renderul este blocat pînă cînd mes este transmis)

## Exemple de IPC - POSIX

↳ shared memory

1) cream regim de mem.

shmfd = shm-open (name, 0\_CREAT | 0\_RDWR, 0666); (read + write)  

sh	g	o
rwx	rwx	rwx
110	110	110

 spune cîntă și write  
fiecare cifră ce urmărește în baza 8

→ permisiuni de mem  
shareuid

spune cîntă și write  
fiecare cifră ce urmărește în baza 8

2) stabilește dimensiuni obiectelor max 4 KB linii  
făcute (fdm-fd, 4096)

3) pfn = mmap(0, size, PROT-WRITE, MAP-SHARED, fd, 0)  
memory map ↘ cot se va lăsa din zona de memorie  
de unde se începe alocarea ↗ ce drepturi va avea lectura și scrierile (protection for the mapping)

4) dnm - unlink (name); → standard pt a sterge fiz

5) Socetuți: mod de comunicare în rețea

număr → destinație  
- ip - ip  
- port - port  
→ tuplu: 5 elemente ip:port → server  
ip + port → client → protocol  
o conexiune

6) Remote procedure call

↳ mecanism prin care un progr scrie într-un calce să pleagă o procedură sau funcție care rulează pe alt calce

↳ modelul client-server → TCP / IP

face o  
cere  
de  
exec  
a unei por la rețea

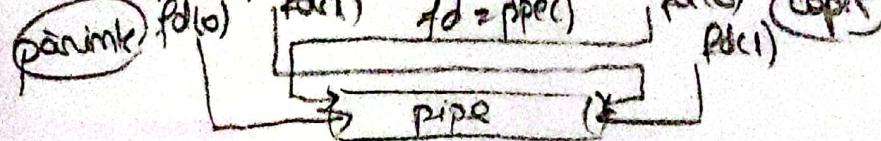
↳ folos un socket pentru a comunica în rețea

7) Riparii

↳ folos cînd între 2 progr există o rel (ex PROC părinte → copie)

↳ creare înainte de fork

↳ după fork() variabila pipe este copiată (de la părinte la copie)



9

### 3) THREAD

- Diff thread - process

proces independent ce rulează pe celișcăre  
space address, resurse + variabile  
(totul este separat între ele)

→ un proces poate avea mai multe threaduri, acestea  
partajează sp de adrese, resurse, dar fiecare are o altă  
set of register values

↳ procesele permit separarea programelor → pot lucra îndepl., pe  
când kernelul oferă o modalitate unui program de a  
executa mai multe taskuri simultan  
(ex: suma elem unei vectori → mai eficient thread!)

kernel → multi-thread

o Regula lui Amdahl: speedup maxim teoretic at cădăcă avem  
o secu de ~~cores~~ core reale / paralel

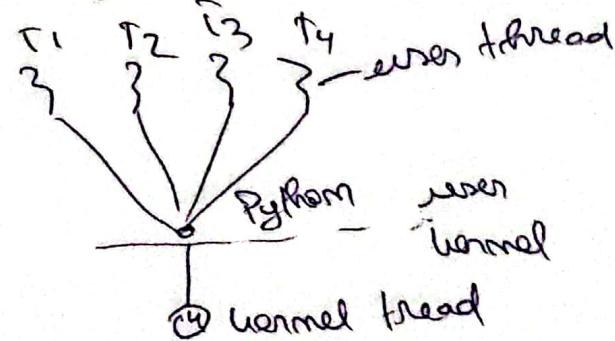
$$\text{Speedup} \leq \frac{1}{S_e(1-S)}$$

series:  $\text{for}(i=0; i < m; i++)$   
 $a[i] = a[i-1] * 2 + a[i-2]$

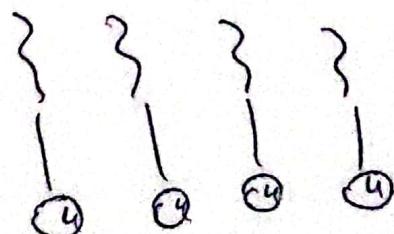
parallel:  $\text{for}(i=0; i < m; i++)$   
 $c[i] = a[i] + b[i]$

o modele multithreading (user-kernel)

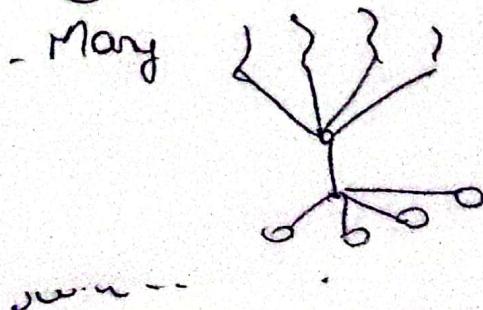
1) Many-to-One



2) One-to-One



3) Many-to-Many



## pthread

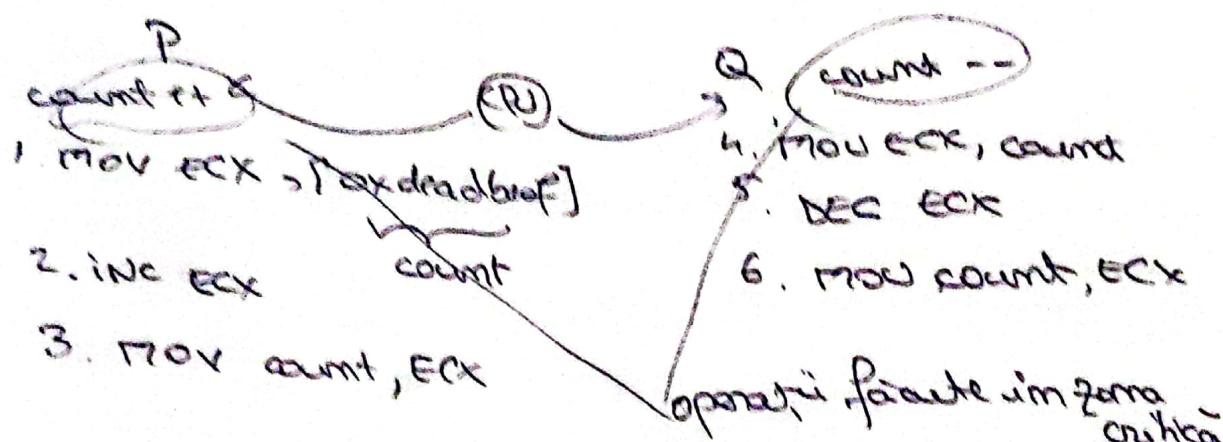
pthread - `attr init(& attr);`  
 pthread - `create (& tid, & attr, run, args[1]);`, ~ fork() la pro  
 pthread - `join (tid, NULL);` ~ wait()

pthread - `exit (0);` ~ exit()

simultan

## Sincronizarea proceselor

- procesele nu pot executa concurent
- exemplu (pb paral-consum) acest lucru la date → inconsistenta  
 counter (numără bufferele)



- zona critică a unui proces:

do

} entry section

critical section

exit section

↳ while (true) → remainder section

- Ex de alg pt procesele  $P_i, P_j$

$i:j:m = h:j:j \rightarrow$  resurse împărțite între procese

do

entry → while (turnm == i);  
critical section

exit → turnm = j;  
remainder section

↳ while (true);

do  
while (turnm == i)

critical section

turnm = i;  
remainder section

↳ while (true)

(11)

↪ sol  $\Rightarrow$  Locks

doh acquire lock  $\rightarrow$  media disponibilitate (acces)  
zona c  
release lock  
Y while (true)  
remainder next

↪ test-and-set

boolean test-and-set (booleam + target)

exec atomic

boolean no = + target,  $\rightarrow$  returnez o val  
+target = TRUE;  
return no;

releasing new val  
a paralel la true

putem shareni variabila lock, initializ cu false

↪ compare-and-swap

⇒ primește & arg o val, o val  
expected și new-val  
 $\text{if}(\text{+ value} == \text{expected})$   
 $\text{+ value} = \text{new-val};$   $\rightarrow$  dacă este o,  
nu mă î

while (compare-and-swap (& lock, 0, 1) != 0)  
atribuim true

[Propri zonă critică]

o mutual exclusion, dă un proces  $P_i$  exec zona sa  
ex vide privat (val + arg) / critică, niciun alt proces nu poate  
exec zona c.

↪ for (int i = 0; i < 10000) {  
increm var sem-wait (mutex); aj la mutual excl  $\Rightarrow$  lock  
count++; sem-post (mutex);  $\rightarrow$  așteptăm ca reac să se libereze  
folosim pt mutua sem-post (semunten),  $\rightarrow$  eliberarea sem  $\Rightarrow$   
inc set NULL; indică faptul că reac devine available

int main() { sem-init (& mutex, 0, 1)

pthread t1, t2;

pthread create + join pt exec

sem-destroy (& mutex);

(12)

- **progress**: dacă niciun proces nu are zona de critică și îl procesează vor să intre în zona de critică, ele se vor selecta în timp limitat
    - || putem urmări progresul proceselor folosind un array pt a reține starea free proces, după ce este completat procesul, array-ul este actualizat.
  - **bounded-waiting**: dacă și procesul în zona critică nu-a făcut o coadă
    - impune o limită sup. timpului pentru a accepta în coadă pt a accesa o res.
- Ex
- ```

void *thread_func(void *arg)
{
    int thread_id = *((int *)arg);
    sem_wait(&mutex);
    waiting_time[thread_id] = 0;
    while (waiting_time[thread_id] < Max_waiting_time)
    {
        printf("Thread %d: waiting time = %d\n",
               thread_id, waiting_time[thread_id]);
        sleep(1);
        if (sem_post(&mutex));
        else
            break;
    }
}
  
```
- + thread  
acceptă  
sem înainte  
de a se executa  
timpul de  
acceptare

- **executare kernel**
  - preemptive: permite preemptionul proceselor
    - ↳ procesul poate fi interupt de sistem / de un task cu prioritate mai mare
    - ↳ permite să se aloce resurse, să facă switch între taskuri și să se enândeze în funcție de prioritate
  - non-preemptive
    - ↳ procesul este gata să intre imediat în c
- **sem. Peterson**
  - folosește 2 variabile → tuam → al cărui numărul să fie impar
    - ↳ flag
    - array
      - care indice care urmărește dacă un proces e gata să intre în 2.c

(3)

( $\hookrightarrow$ ) Alg pt  $P_i$

do h

flag[ij] = true;

turn = j;

while (flag[ij]  $\neq$  & turn = = j)

|| zona critica

flag[ij] = false;

Def. normă

While (true);

|| ex  $\rightarrow$  bounded - waiting cu test and alt

do h

waiting[ij] = true;  $\rightarrow$  prx în aşteptare

key = true;

while (waiting[ij]  $\neq$  & key)

key = test\_and\_net (lock);

waiting[ij] = false;

j = (j+1)  $\mod M$

white(j) = 1  $\neq$  waiting[ij])

j = (j+1)  $\mod M$

if (j = = 1)

lock = false

else waiting[ij] = false;

While (true);

DEADLOCK & STARVATION

( $\hookrightarrow$  și sau mai mult) procese așteptă pt un even cauzat de

unul din procesele care aștepta

fiecare proces  
așteptă după  
aceleia să  
elibereze res

procesul nu e scos de pe coada

Seul în care e suspendat

un proces nu poate  
accesa resurse pt că  
sunt fără de astfel de proces

(nu va fi scos nici odată  
de pe lista de așteptare)

nu va elibera coada

pt proces ce priorită >

multă

14

## Exemplu

### Deadlock

```

void * f1(void * arg) {
    int id = (int) arg;
    while(true)
        ↳ // aici se resurses nu fie elibrate
        // asteptam accesul la res A
        sem_wait(&resource_a);
        printf(...);
        sem_wait(&mutex);
        sem_post(&resource_a);
        // elib resursa
        sem_post(&mutex);
}

```

### Starvation

```

void * f1(void * arg) {
    ↳ // acceptam accesul la res A
    sem_wait(&resource_a);
    // asteptam acces la res B
    sem_wait(&resource_b);
    sem_post(&resource_b);
    sem_post(&resource_b);
}

```

↳

- fctia este un semafon de resurse  
daca intră în zona critică protejată de sem mutex
- dacă un thread nu poate accesa resursa, va aștepta la mutex  $\Rightarrow$  starvation
- putem adăuga un alg de alocare a resurselor pt a ajusta dinamic disponibilitatea nem de resurse

↳

void \* f2(void \* arg)  
// imbec res acu res b

↳ acoste 2 threaduri pot începe să acceseze res A și B

$\Rightarrow$  deadlock (ambile așteaptă ca altul să elibereze res)

↳ ajustare dimensiune a securinței

## Semafane

- Sun nr întreg
- dă  $S > 0 \Rightarrow$  o resursă este disponibilă
- 2 metode  $wait(S)$

↳ while  $S <= 0$

    ↳  $S--;$

    ↳ compară res

    ↳ signal(S)

    ↳ Set;

    ↳ eliberează resurse

- semafon binar = mutex

(15)

o simple busy waiting (spin locks)

```
do {
    wait (mutex);
    // Zone critique
    Signal (mutex);
    // remainder action
} while (true);
```

o simple fair busy-waiting

```
wait (Semaphore *S) {
    S->value--;
    if (S->value < 0)
        // adding proc in S->list;
        block();
    Signal (Semaphore *S);
    S->value++;
    if (S->value <= 0)
        // remove proc P from S->list;
        wakeup (P);
}
```

typedef struct

int value;

struct process \*list;

Semaphore

↳ 7.6 Synchronization

↳ Bounded Buffer Problem ②

Reader & writer Problem ⑤

Dining - Philosophers Problem ③

② several m pos in buffer → max multi prod von n solchen elem im min. -en buffer  
- semaphore   
    < mutex pt access to buffer, init eu 1 limited  
    empty col pos next libere → m full pos occup. init 0  
do {
 wait (full); // de buffered este qd aceptado
 wait (mutex);
 // sacar elem del buffer
 signal (mutex)
 signal (empty)
} while (true); // consumo elem del

④

(4) CDU (chocul de măsură)

- (5) pb la scriere  $\rightarrow$  în același timp scriem în acel rând ceea ce self rescriem  
→ scrieră  $\left\{ \begin{array}{l} \text{mutex pt a sincroniza accesul la val} \\ \text{countreaders} \rightarrow 0 \\ \text{unt pt accesa la scriere} \\ \text{readers y scrie să le doară în schimba} \\ \text{inf disp} \end{array} \right.$

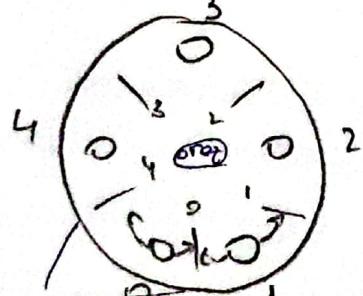
writer:

do h  
wait(unlock);  
// face scriere  
signal(unlock);  
} while(true);  
// scrierea este apărată pînă  
// când toată scrierea  
// este efectuată și că

do h  
wait(mutex);  
readCount++;  
if(readCount == 1)  
    wait(unlock);  
    signal(mutex);  
    // se actualizează  
    wait(mutex);  
    readCount--;  
    if(readCount == 0)  
        signal(unlock);  
        signal(mutex);

$\rightarrow$  dacă înf și actualizare  
 $\rightarrow$  prioritățile cititorilor, apăsa  $\Rightarrow$   
prioritățile scriitorilor

- (6) 5 filozofi  $\rightarrow$  5 bețigări (fiecăruia poate să mănânce de cîte 2  
(dacă un mânancă, ceilalți patru nu pot mânca)



ambele trebuiesc să fie obținute simultan  
(acces concomitant la o resursă limitată)

- (1) o sol scrieră cheaptă [§5]

do h  
wait(chopstick[i]);  
wait(chopstick[(i+1)%5]);  
// mânancă  
signal(chopstick[i]);  
signal(chopstick[(i+1)%5]);  
// găzduind  
} while(true);

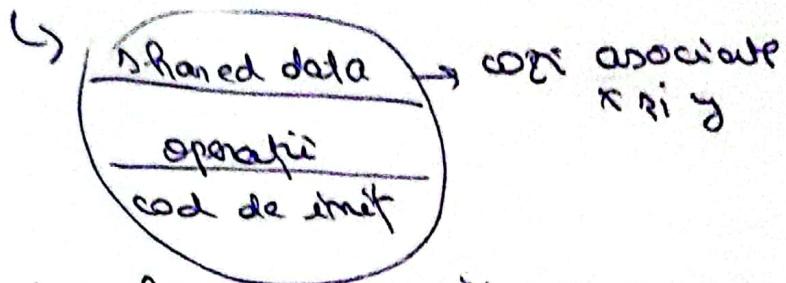
nu primește  
ce bețigă  
lucră  
4  
pot fi date  
patru dim  
stă  
deadlock



## ↳ See ⇒ folosire MONITOR

↳ op wait signal = interne ale monitorului

↳ protejează un bloc întreg ale felii  
dintr-oare unei de initializare



↳ folos var conditioale (acestia concre  
re exec wait (signal) → săl un alt  
proces)

↳ În plus folosiri → un monitor

enum THINKING, HUNGRY, {  
EATING, idlet(i); } -> var condit

↳ funcție - pickup(i) → ridicare bat

state  $i_j$  = HUNGRY

test( $i$ )

if ( $state[i] \neq EATING$ ) // dacă vecinul nu mânca  
self.wait( $i_j$ ); // și lui  $i$  este foame,  
atunci poate mâncă

~~putdown(i)~~

- putdown( $i$ )

state  $i_j$  = THINKING

test ( $(i+1)_j = 5$ )

// testă beginul  
din stările

test ( $(i+4)_j = 5$ )

// dreapta

- test :

if ( $state[(i+4)_j] \neq EATING$ )  $\Rightarrow$   
 $(state[i_j] = HUNGRY) \Rightarrow$

$(state[(i+1)_j] \neq EATING) \Rightarrow$   
 $state[i_j] = EATING;$

self[i].signal();

g g

↳ posibil ⇒ starvation

↳ deadlock nu

18

## 4) CPU Scheduling

- ↳ baza nist ale op ac multi programme
- ↳ short-term scheduler select din procesele din coda ready
- ↳ CPU scheduling decisions when
  - choice de la multe la waiting state
  - u - ready state
  - trase de la waiting la ready
- non preemptive
- restul preemptive
- terminata

↳ Dispatcher → se ocupă de managementul proc

- schimbarea contextului de exec

- trecere la user mode

- done la o scadă de unde se poate relua progr

### criterii

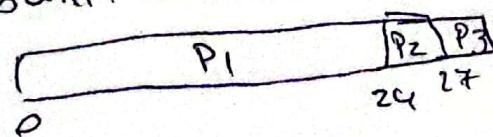
- MAX {
- CPU utilization: ocupare CPU pe cat posibil
  - throughput - nr de procese / unit de timp
- MIN {
- turnaround time: timpul de exec al unui proces
  - waiting time: timpul de asteptare al unui proces in coda
  - response time: timpul de raspuns la prima procesare cererii

o First come, first served

(FCFS)

|        | Proces         | Burst Time |
|--------|----------------|------------|
| utm    | P <sub>1</sub> | 24         |
| im     | P <sub>2</sub> | 3          |
| ac     | P <sub>3</sub> | 3          |
| ordine |                |            |

Gantt Chart:



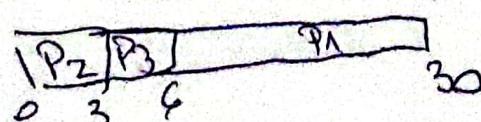
- timp așteptare  $P_1 = 0, P_2 = 24, P_3 = 27$

- timp median așteptare:

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

↳ optimizare ⇒ cele cu timp mai mic ⇒ prioritate

SHORTEST JOB FIRST



- timp așteptare  $P_2 = 0, P_3 = 3, P_1 = 6$

- timp median așteptare

$$(0 + 3 + 6) / 3 = 3 \text{ (17)}$$

↳ determin CPU burstului urm  
 $t_m$  = lungimea de-ai m laea CPU burst  
 $T_{next}$  = val. proiză a urmării CPU burst  
 $d, 0 \leq d \leq 1$

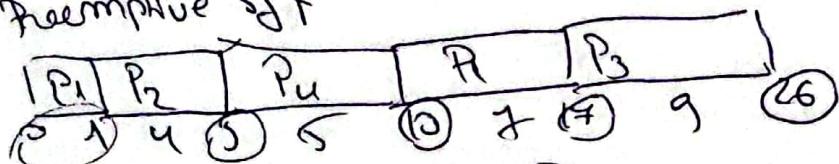
$$T_{next} = d + t_m + (k - d) t_m$$

$$\text{distribuția } d = \frac{1}{2}$$

o shortest remaining time first

| Process        | Time la care aj | Burst Time |
|----------------|-----------------|------------|
| P <sub>1</sub> | 0               | 8          |
| P <sub>2</sub> | 1               | 2          |
| P <sub>3</sub> | 2               | 9          |
| P <sub>4</sub> | 3               | 8 11       |

↳ preemptive SJF



vom aj P<sub>3</sub> și P<sub>4</sub> → multitudine

↳ preemptive SJF

$$\left\{ (10 - 1) + (1 - 1) + (17 - 2) + (5 - 3) \right\} / 4 = 26/4$$

$$\begin{matrix} P_1 & P_2 & P_3 & P_4 \\ 0 & 1 & 4 & 5 \end{matrix}$$

$$\begin{matrix} P_1 & P_2 & P_3 & P_4 \\ 1 & 2 & 7 & 14 \end{matrix}$$

$$\begin{matrix} P_1 & P_2 & P_3 & P_4 \\ 4 & 6 & 15 & 16 \end{matrix}$$

$$\begin{matrix} P_1 & P_2 & P_3 & P_4 \\ 5 & 7 & 17 & 26 \end{matrix}$$

→ probabilitatea = starvation

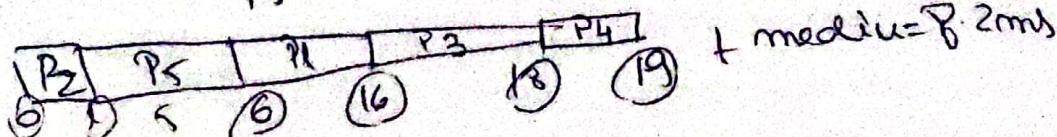
→ cu cat contextul mai multe, vei avea prioritati mai mici

→ cu cat contextul mai multe, vei avea prioritati mai mici

↳ nu se poate exec procesele

o Priority scheduling:

| Process        | Burst time | Priorit |
|----------------|------------|---------|
| P <sub>1</sub> | 10         | 3       |
| P <sub>2</sub> | 1          | 1       |
| P <sub>3</sub> | 2          | 4       |
| P <sub>4</sub> | 1          | 5       |
| P <sub>5</sub> | 5          | 2       |



+ mediu = 8.2ms

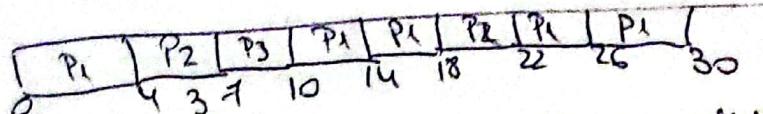
20

### Round Robin:

- ↪ einterzito: impun cuantă de timp de exec
- ↪ după maxim  $(m-1)q$  se exec urm proces

$oexec \rightarrow RR$  cu  $q = 4$

| Proces         | Burst time     |                |                |
|----------------|----------------|----------------|----------------|
|                | P <sub>1</sub> | P <sub>2</sub> | P <sub>3</sub> |
| P <sub>1</sub> | 24             |                |                |
| P <sub>2</sub> |                | 32             | 2              |
| P <sub>3</sub> |                |                | 32             |

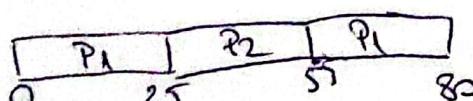


- ↪ alg de planificare care dă o porțiune de timp fixă unei task din coadă, acela se exec pînă bucată de timp, apoi este mutat la finalul cozii și trăiește următoarele

### Earliest Deadline First (EDF)

- Earliest Deadline First (EDF)
- ↪ priorități date de deadline (ce cat mai rapid, ce cat pînă e mai  $\rightarrow$ )

$$\begin{array}{ll} \text{ex) } P_1 & P_1 = 50 \quad t_1 = 25 \\ \text{u) } P_2 & \text{periodea} \\ & P_2 = 75 \quad t_2 = 30 \end{array}$$



$$\begin{aligned} (P_1, P_2) &= (100, 75) \\ \hookrightarrow P_2 &\text{ se exec pînă } QSS \\ (P_1, P_2) &= (100, 150) \end{aligned}$$

$\Rightarrow$  ne exec P<sub>1</sub>

$$(P_1, P_2) = (150, 150)$$

$\Rightarrow$  ne exec P<sub>2</sub>, apoi P<sub>1</sub>

|                | time exec | Deadline |   |
|----------------|-----------|----------|---|
| P <sub>1</sub> | 10        | 30       | ② |
| P <sub>2</sub> | 20        | 50       | 3 |
| P <sub>3</sub> | 15        | 25       | 1 |

$t=0 \rightarrow$  selectăm P<sub>3</sub> (cel mai early deadline) (25)

$t=15 \rightarrow$  a opus P<sub>3</sub> (mai devreme decât deadline 30)

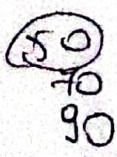
$\hookrightarrow$  selectăm P<sub>1</sub> ca urm deadline

$\rightarrow$  sel P<sub>2</sub> (50)

$t=25$  task 1 gata

$t=45$ , se term P<sub>2</sub>

$\hookrightarrow$  daca adaugăm și periode



$t=50 \rightarrow P_1$  se eliberaș și pînă pe coada de ready

21

- ↳  $t=t_0 \rightarrow$  elibereză și-l punem în coadă
- ↳ fiecare proces th să fie liber înainte de release-ul următor

## o Load-balancing

- ↳ mai mulți pasi
  - monitorizare = utilizarea resurselor (care sunt ocupate / libere)
  - request distribution
    - ↳ când se primește o cerere, load balancerul o distribuie către cea mai puțin ocupată re
  - redirectionare

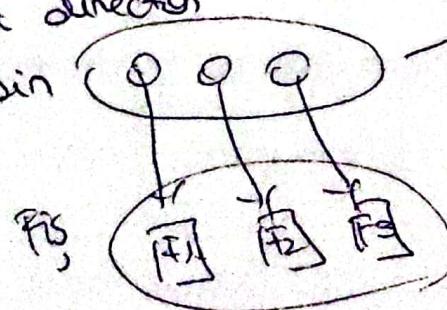
## 5) FILE SYSTEM

- conceptul de file
  - data
    - numeric
    - text
    - correct
    - binary
  - program
- text file, source file, exec file
- Atributele unor fiziere
  - nume
  - identificator
  - tip
  - locatie
  - dimensiune
  - protectie (cum poate citi, scrie, executa)
- inf despre fis → struct
  - directorului
  - ora, data + nume
  - identif
  - dirk

- operatiuni cu fis
  - create
  - deschide
  - citire
  - scrise
  - securizare
  - stergere
  - truncare
  - deschide (Fi)
  - + include (Fi)
- Open - file table
- file pointer catre ultima locatie unde s-a scris la cikt
- file open count: de cat ori e deschis un fisier

- o open file locking
  - similar cu reader-writer lock
  - < shared lock: reader lock mai multe concurent
  - exclusive lock → ca writer lock

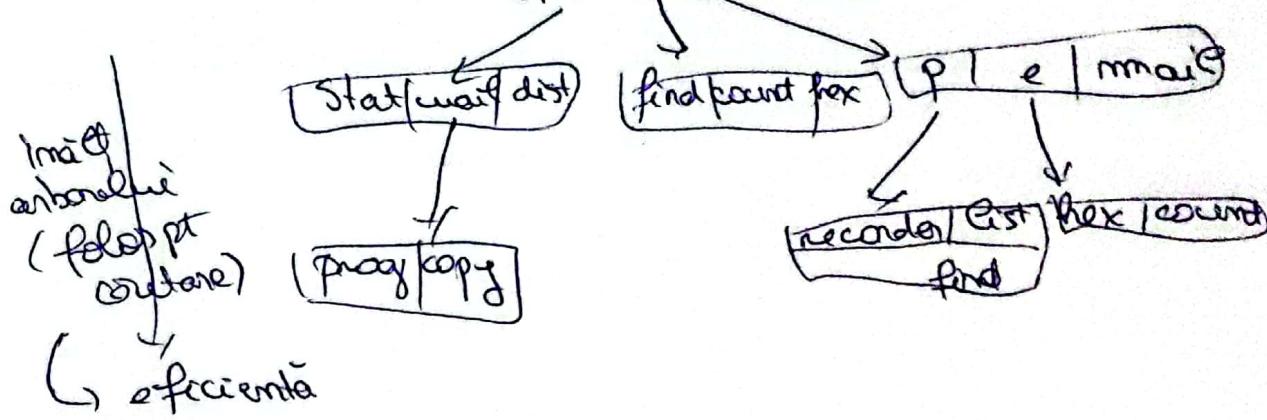
- o struct director



Modularizare inf despre fizice

↳ Structură arborescentă de directoare

root: (specify him) programs



↳ eficiență

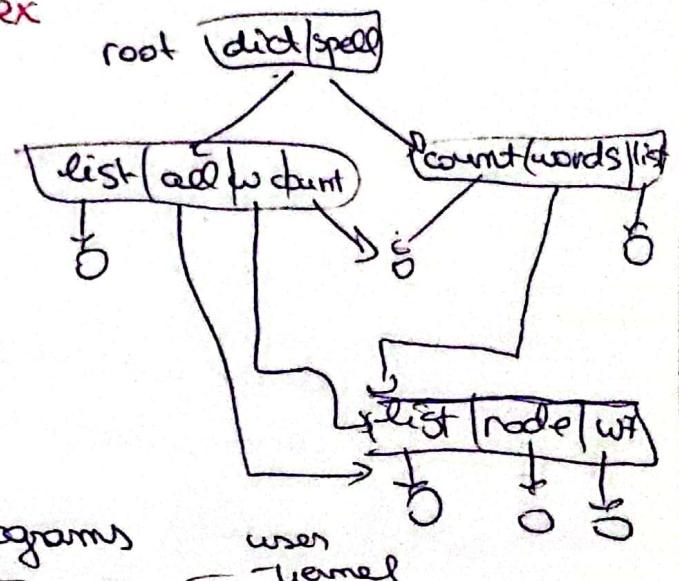
↳ pe bună grăsuță

↳ directoarele curente: cwd

↳ sferturile fizice: nom <file-name>

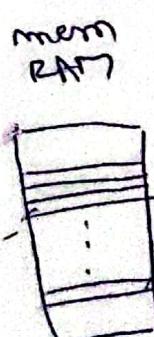
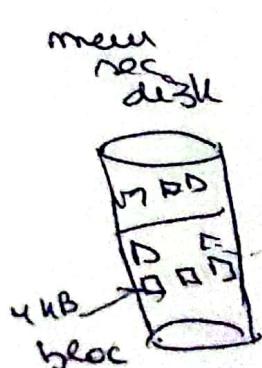
↳ crearea fizica: mldir <dir-name>  
Subdir

↳ graf acyclic de dir → ex



→ mod de organizare

application programs



logical file system: FCB, file, directory

file - organization module (nume fiz)

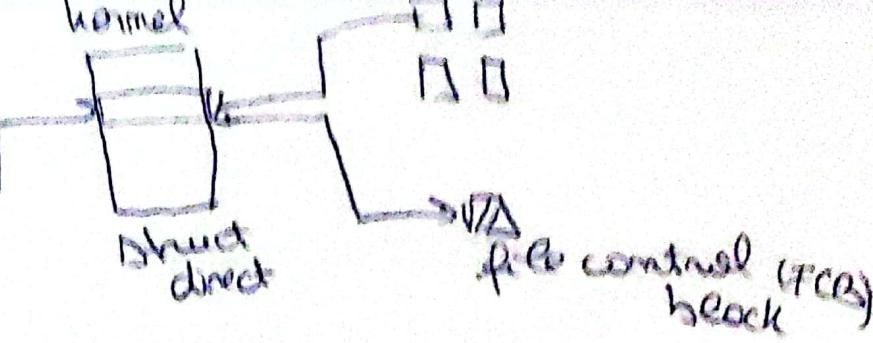
basic file system (refine idem blocuri de fis)

controlul unorare deschidere diverse

15B, disk

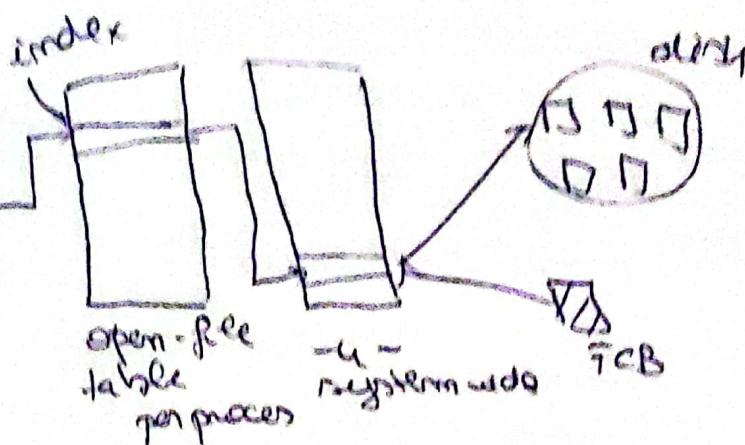
## Solar Ridge user

:index = open(file)  
 fd = token space/  
 file descriptor



## Scritire

read(index)



FCB - puncte între dintre dintre de fisiere și dintre de openare

↪ partea cea mai mult fizic, acția de pe device, mărimea, datele de creare + modificare, permisiuni + alte metadate

↪ implementa structura direct

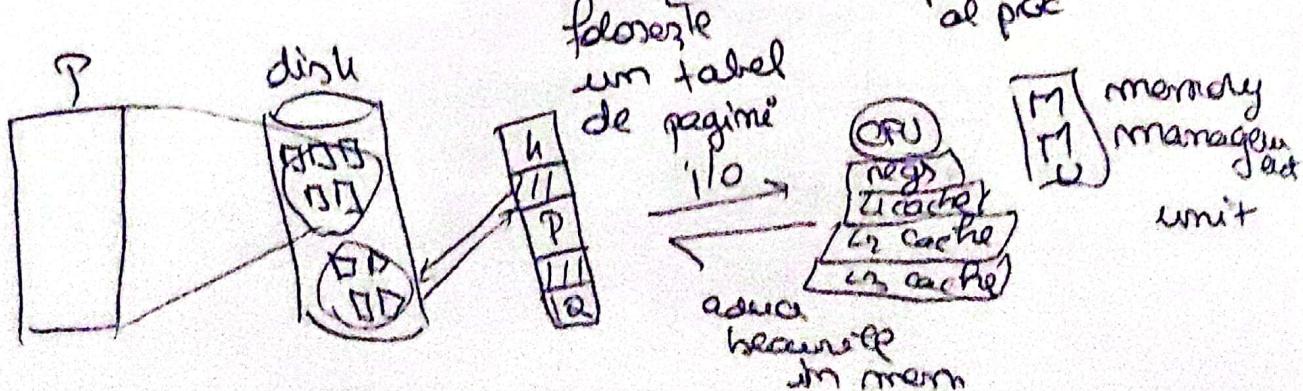
- lista circulară cu numele fizic + pointer către blocuri de date

- hash table - lista circulară + hash

(folositoare comod întrările sau eliminarea fixă sau folos chained - overflow method)

## Memorie

o maparea proceselor în memoria  $\Rightarrow$  SO : virtual memory address space al proc  $\Rightarrow$  fizic



PCB

↳ process control block

↳ reg

reg-base EMMU  
reg-limit

CE se intămpăță cu un progr?

1.

cc log.c → log.o  
coupl

- transf im

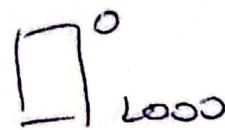
objekt

- instead, Sunf wird im Objektmappe

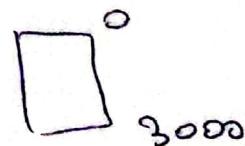


met.c → met.o

ui.c → ui.o



main.c → main.o

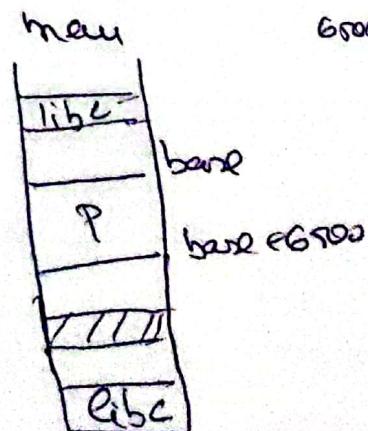


2. Linkers, editors

⇒ ELF, PE



3.

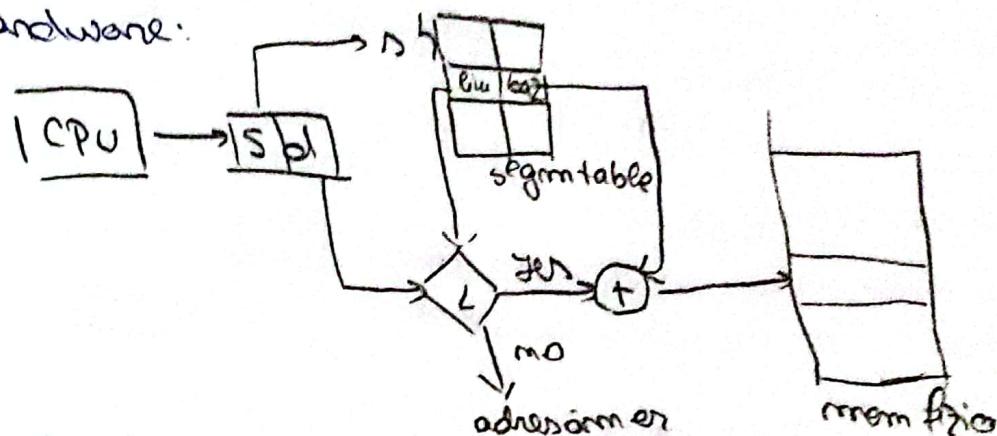


Symtable

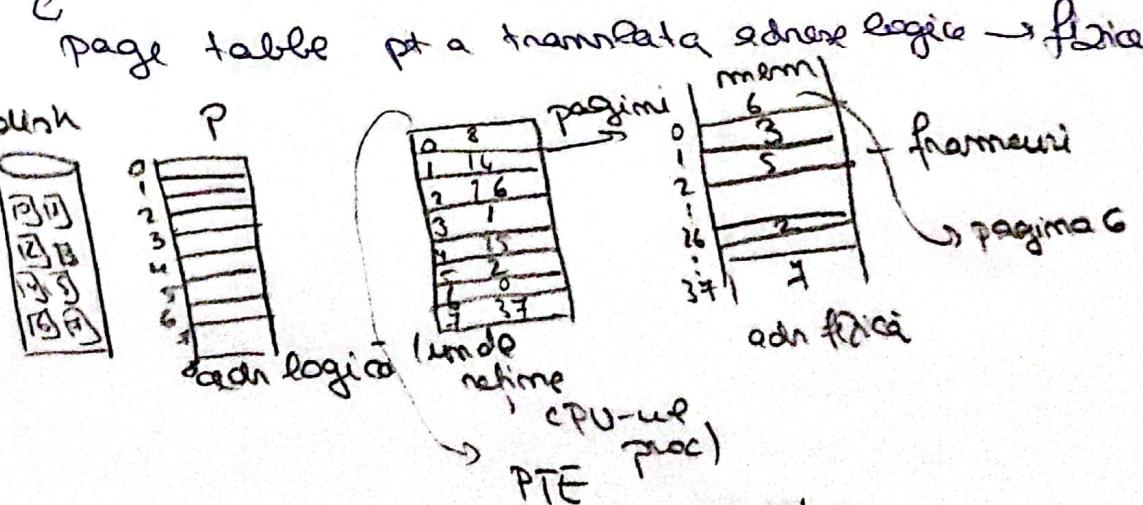
## Segmentare

- adresa logică = triplet <segment-number, offset>
- segment table: mapază adresa fixă
  - (SFRB) base register: locația tab.  
mem
  - base register: locația tab.  
mem
  - length register: nr. de byte din mem.

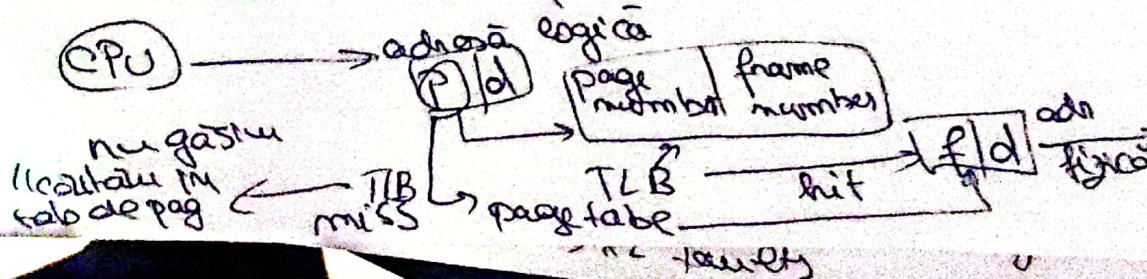
Hardware:



Paginare: împărțirea  
memoriei în blocuri  
logice și fizice



- TLB (Translation Look-aside buffers) → mem asociativă
  - pt accesul la cele 2 tipuri de memorie
  - hardware cache pentru tabelul de pagini



↳ bit valid / invalid în tab pagini

indica daca o mapare este valida si poate fi folosita pentru accesul la memorie

→ daca bitul = 0 ⇒ maparea nu este valida pt că

- pagina fizica ce corespunde celor din

a fost schimbată pe disk

- pagina virtuală nu i-a fost

asignată una fizică

page fault exception → rezolvare

maparea

+ setform

bitul la 1

↳ mecanism eficient

pt managementul mem

+ mem

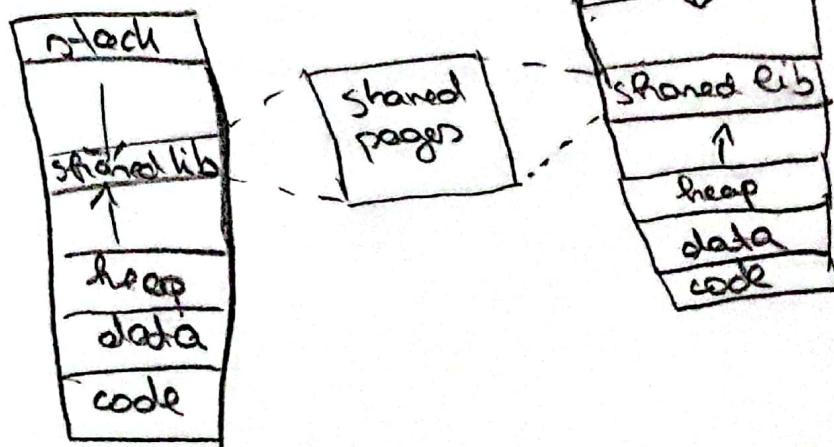
virtuală

### --- MEM VIRTUALĂ ---

↳ dep mem logice a utilizat de mem fizică

↳ permite sap de adrese să fie shareate între mai multe programe

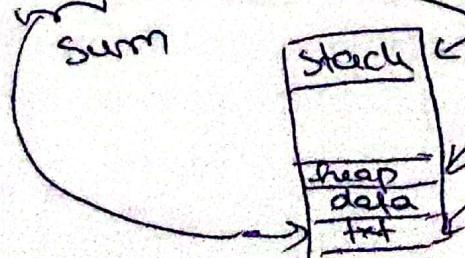
→ reutilizare adres space



ex: int sum=0;  
for( int i=0, i < 10; i++ )

    sum = sum +  $\text{V}[i]$ ;

ADD EAX, [ESI + OF]  $\rightarrow$   $i$



110

int main()

h v = malloc(10);

y

Accesăm la pag

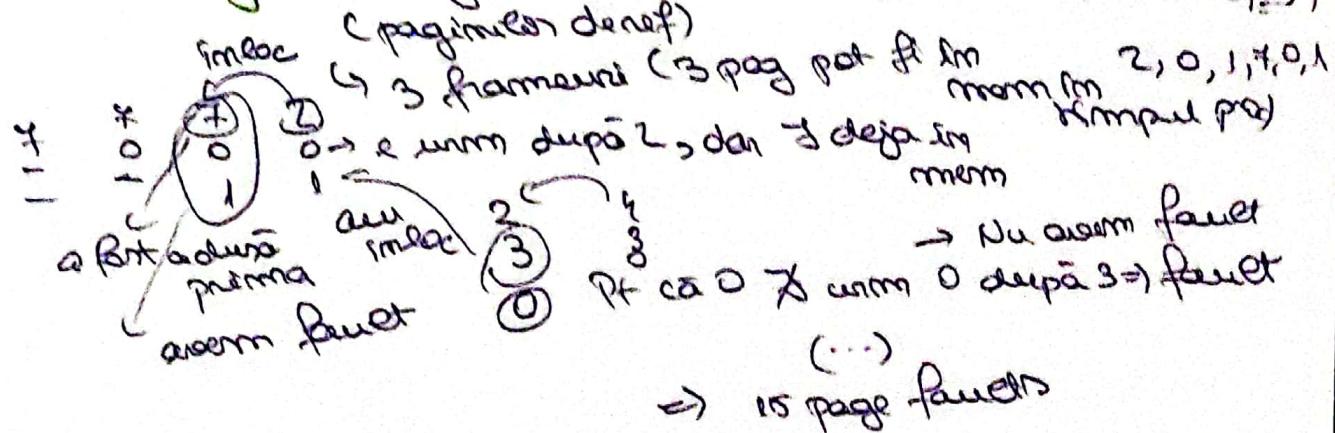
(2)

- ↳ Handling page faults
  - dacă  $\exists$  o pag în prima rlf  
de ac pag va fi  
primită în SO
  - S.O nu există sau nu este pt a  
ref invalidă  $\rightarrow$  abordare  
nu e în mem  
gasită în framele  
de la înaintea pag în frame  $\Rightarrow$  scheduleaza  
acestăm de la început pt a indica că  
pag nu e în mem  
față că pag nu e în mem
- ↳ Page replacement  $\rightarrow$  rep. între mem-hogresă și fizică

↳ Page replacement  $\rightarrow$  Rep. intre mem-locuri a' fizice

- ↳ gās locatia paginii dator pe dinh
  - ↳ gās un frame liber (nāc nectim frame)
  - ↳ ad pagina datoră em formulel eleh
  - ↳ restantiam iñost care a provocat trap-ue

FIFO alg., starting ref.: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1,



ALG OPTIM

ALG OPTIM  
↳ imloc pag care nu a fost folos pînă mai lungă perioadă

$\rightarrow$  Imloc pag cone nra a pos 4  
 $\rightarrow$   $\begin{pmatrix} 7 & 2 \\ 9 & 1 \end{pmatrix}$   $\begin{pmatrix} 2 \\ 3 \end{pmatrix}$   $\rightarrow$  a mai for folos  
 Bi va fi foros după 4

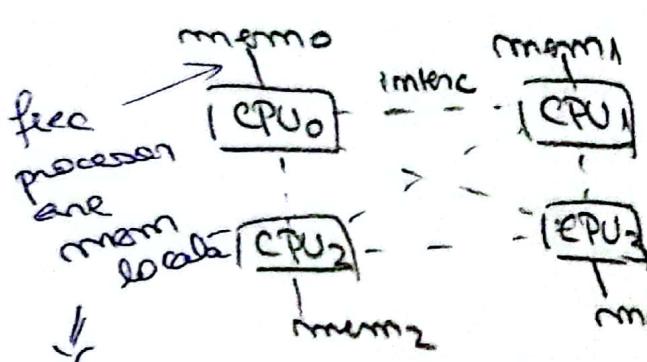
↓ two main folios per ref B 9 page facets  
↓ 1 comm file per page

LRU ALG → least recently used (accetta l'ultima data da cache o file)

↪ NUMA (Non-uniform memory access)

↪ CPU-mem interconnect

pim bus



↪ acces în memorie și proprietăți  
CPU-ului pe care threadul este  
executat

(-) imbinătatea dintre pim și  
multii procesori pim redusese  
tempului mediu de acces la  
mem  
accesarea mem locală a procesorului e mai rapidă  
decat accesarea mem  
compartită cu alte  
proc

↪ Thrashing : de un proces nu are destule pagini

Σ sunt dim mem fix

→ ce putem face în mem

↑  
niciun de page fault  
page fault pt că niciun  
memoriu framele existănt  
tb să avem și framele înloc  
- utiliză mai multe CPU  
- O.S → crește multiprogr  
- adaugă set proc în  
sist

O.S primește ceea ce încearcă  
pagina din mem ⇒ și rămâne în memoria  
pag din mem  
fixă

↪ pt prevenirea de page fault

↑ page replacement alg  
(ce pagini pătrânn în  
mem și ce pag putem  
schimba)