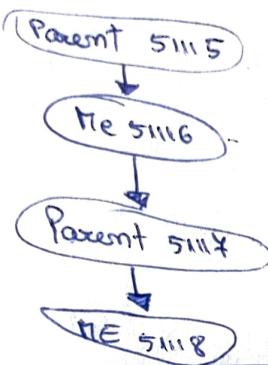


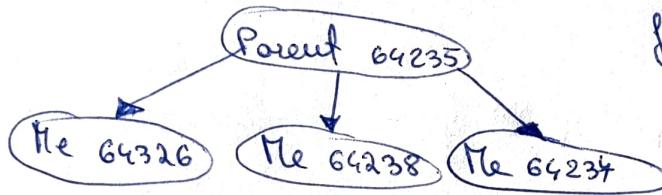
`Fork()` - creaza proces mai → returnează nr < 0 → nu execută proces (fail)
 \Rightarrow = 0 → child proces numai creat
 \Rightarrow > 0 → id of the process id of that particular child care să fie creat în mem rgs.

```
if (!fork()) {
    if (!fork())
        fork();
```



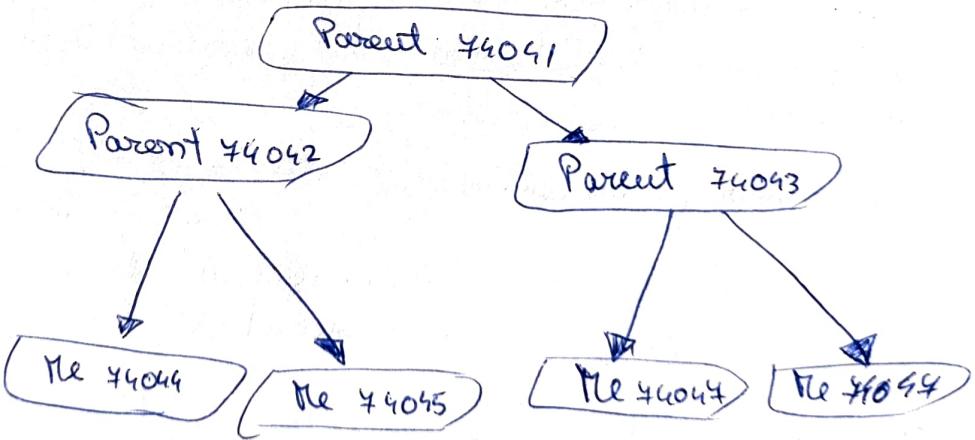
`!fork()` = continuă dacă suntem în copie
 $\text{id} = \text{fork}() \Rightarrow \text{id} == 0$

```
if (fork())
    if (fork())
        fork();
```

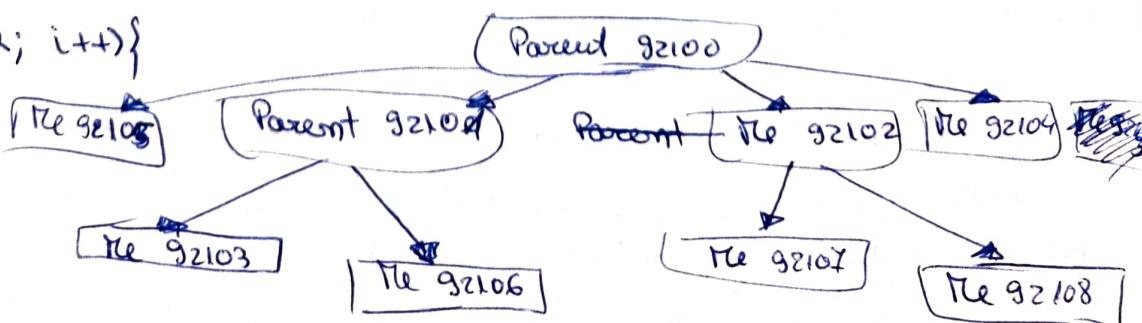


`fork()` = continuă dacă suntem în patruncă
 $\text{id} = \text{fork}() \Rightarrow \text{id} == 1$

```
if (!fork())
    if (fork())
        fork();
    else
        if (!fork())
            if (fork())
                fork();
```



```
for (int i=0; i<2; i++)
    if (fork())
        fork();
```



Proces zombie = copilul își termină execuția și părintele nu are timp să ajungă la exit status/să termine

```
int main() {
    pid_t pid;
    int status;
    if ((pid = fork()) < 0) { perror ("fork"); exit (1); }
    if (pid == 0) { exit (0); } // child
    sleep (100);
    wait (NULL);
}
```

Proces orfan = părintele termină execuția, iar copilul încă merge

```
int main() {
    if (!fork ()) { sleep (30); }
    return 0;
}
```

HOW DOES THE OS HANDLE THIS?

- cascade termination = moare părintele, mai și copiii
- copilul este reassigned to the init process sau kernel (pid=1)
(oică altc.)

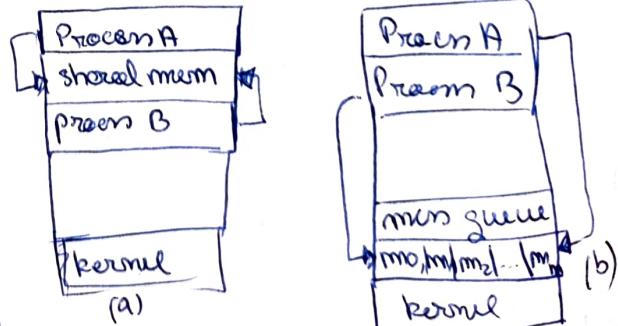
IPC = inter-process communication \Rightarrow cooperating processes affect one another
so they must communicate

Proces independent = nu share-uește niciun date cu celelalte executind procese

Proces cooperation = affects / is affected by the other processes — in the system

- Există 2 moduri \rightarrow shared memory (a)
 \rightarrow message passing (b)

Regimul de shared-memory se află în
spațiul de adrese al procesului care creașă
segmentele de memorie partajată, iar fiecare
alt proces care dorește să comunice trebuie să o ataseze la proprietățile de
acest spațiu.



Fiecare proces are propriul său spațiu de memorie virtuală, iar memoria
partajată trebuie împodă ca să fie accesibile.

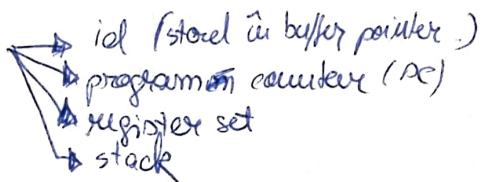
Shm-open() = creates / opens a shared memory obj.

mmap() = maps shared mem. into the process's address space

Shm-unlink() = removes the shared memory object

THREADS

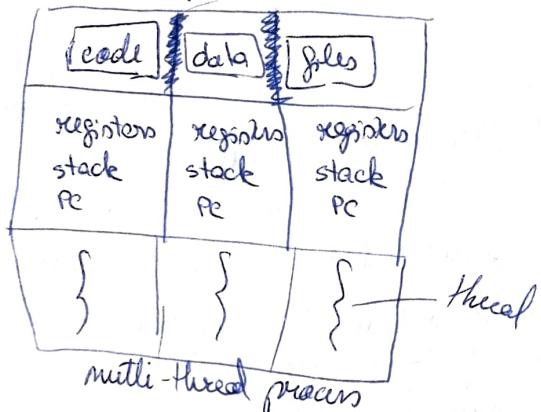
= a basic unit of CPU utilization from a dim



= unită de execuție dintr-un proces

Linux, android, creates several kernel threads performing a specific task

In multicore system, threads will not run in parallel, the system assigning a thread to each core



pthread-create = creare un thread în proces

start_routine() = începe execuția thread-ului

attr points to pthread_attr_t, if null => default

pthread-join = așteptă ca thread-ul să termine, cum termină și se termină

PROCESS SYNCHRONIZATION BACKGROUND

Caând shared-data is accessed not controlled => corrupt data values may appear

Fenomenul și numește race condition (acea în concurrent system)

↳ 2 sau mai multe fizici execută același / modif o resursă
partajată în același timp și ordinea lor depinde (nu e
controlată) pt că rezultatul final depinde de ordinea lor.

PRODUCER-CONSUMER PROBLEM

Procesul producător produce înf pe care le consumă și consumă proces.

E nevoie de un buffer care să fie apărat de producător și să fie de consumă.

↳ regimul de memorie shared

Shared memory approach => producer waits deoarece buffer e full
consumer waits -> e gol

THE CRITICAL-SECTION PROBLEM

No two processes are executing in their critical sections at the same time.

Each process must request permission to enter its critical section.

Kernel = componentă centrală a unui SO care gestionează resursele sistemului
și permite comunicarea între hardware & software
= intermediar între aplicații și hardware

↳ gestionează procesele, accesul la aplicații

↳ gestionează memoria (alocare & eliberare RAM)

↳ - , - dispozitive

↳ securitate & permisiuni și comunicare între procese

Linux: kernel monolithic & modular

↳ toate funcțiile sistemului sunt incluse într-un singur cod de memorie

Multicore system = multiple cores on a single processing chip, with each core operating as a separate CPU to the SO

Concurrent system = supports more than one task by allowing all the tasks to execute ^{proiect} ~~simultaneously~~

Parallel system = can perform more than one task simultaneously

O soluție pt. critical-section problem trebuie să satisfacă urm 3 reguli

1) mutual exclusion: if P_i se execută în secț. excls., atunci niciun alt proces nu are voie acolo

2) progress: dacă niciun proces nu se execută în secția excls. \rightarrow dar sunt procese care vor să intre acolo, atunci unul va fi sigur selectat

3) bounded-waiting: \exists un bound / limită on the number of times that other processes are allowed to enter in a sec. critica depășește un proces a dat request să intre și următoarea cerere să fie granted (permis?).

MUTEX LOCKS

mutex = short for mutual exclusion

= we tool care resolve the cs problem

↳ mechanism di sincronizzare

previne race-condition in proteggere cs.

Un process must acquire the lock before entering a cs. and release the lock when it exits the cs.

while (true) {

 | acquire lock

 cs.

 | release lock

 } remainder section

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);  
// create un obiect de tipul mutex  
pthread_mutex_lock(&mtx); // caid nu mai aveai acces  
pthread_mutex_unlock(&mtx);  
//CS.  
pthread_mutex_destroy(&mtx);
```

SEMAFOARE

Un semafor este un integer variabil care, începând din initializarea, este accesat numai prin intermediul a 2 operații standard atomică:

wait() → signal()

Un mutex este un caz particular al semaforului S=1

int sem_init(sem_t *sem, int pshared, unsigned int value);

sem_wait() → decrementarea S cu o unitate, dacă S>0, închide funcția acceptată ca val. să urce înainte, blocând threadul.

sem_post() → incrementarea S - 1, și dacă \exists blocked - n, eliberează pe cel care a acceptat cel mai mult în cadrul

CPU → executază instrucțiuni
↳ gestionează fluxul de date
↳ coahorte operații

Functie din - unitatea de control (CU)
- arit. & logică (ALU)
- registri
- cache

DISPATCHER LATENCY

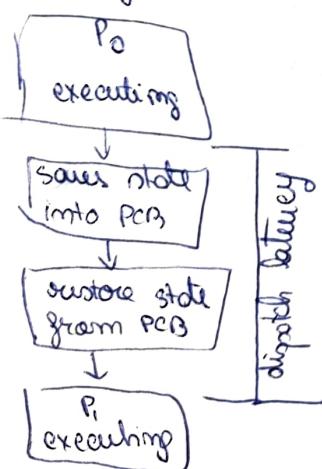
↳ un modul that gives control of the CPU's core to the process selected by CPU scheduler.

Timpul căruia îl ia dispatcherul să se găsească un proces și să înceapă actual și numele dispatcher latency.

Pt fiecare CONTEXT SWITCH, d.l. este invocat,

When is a process switched?

- no wait queue
- time slice expired
- child termination wait queue
- interrupt wait queue.



context switch = switching CPU from the context of one process to the context of another.

DEADLOCK

deadlock = the implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can only be caused by one of the waiting processes

ex : job \rightarrow experienced \Rightarrow experienced \rightarrow job

P₀

wait(S);
wait(Q);

:

signal(S);
signal(Q);

wait(Q); // S in Q semaphore cu val=1

wait(S);

signal(Q);
signal(S);

/

P₁

holds R₁ needs

P₁

P₂

needs

R₂ holds

Pt. că somefcore > 0, atunci după wait() trebuie să aibă signal().

Pt. că signal() nu se poate executa, atunci P₀ și P₁ sunt deadlocked.

THE BOUNDED-BUFFER PROBLEM

// Producer-consumer data structures

```
int m; // pool of m buffers
sem mutex=1; // mutex=binary sem
sem empty = m; // buffer has m empty positions
sem full = 0; // buffer has 0 full positions
```

// producer process

```
while (true) {
    wait(empty); // wait item so
    // take an empty
    spot
    wait(mutex); // viewer in cs
    signal(mutex); // place item in cs
    signal(full); // added an item
```

// consumer process

```
while (true) {
    wait(full); // if possible sem so, / consume position
    wait(mutex); // viewer in cs
    signal(mutex); // place item in cs
    signal(empty); // one empty position is added to
    // the semaphore.
```

REAL TIME SCHEDULING

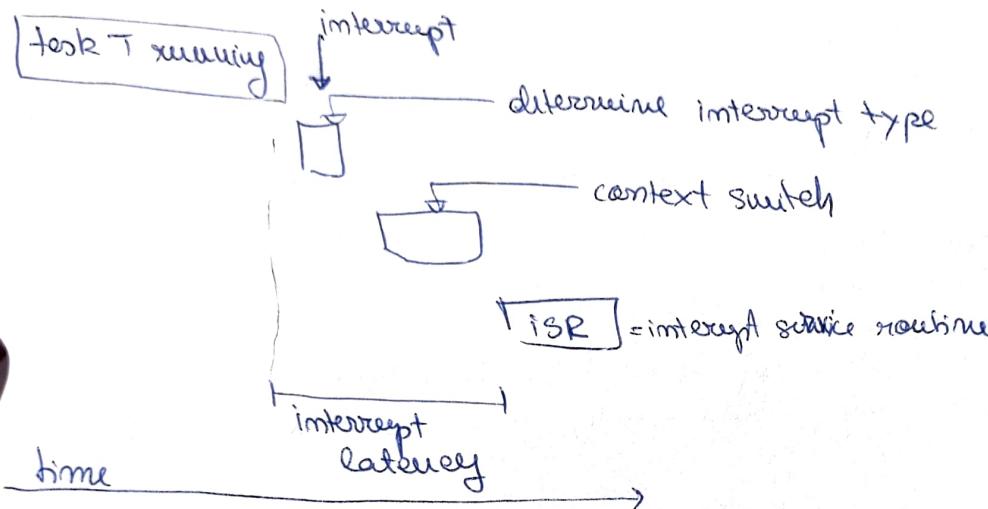
- priority based scheduling

Time Systems: → Soft real-time = it will only guarantee that the process will be given preference over non-critical processes

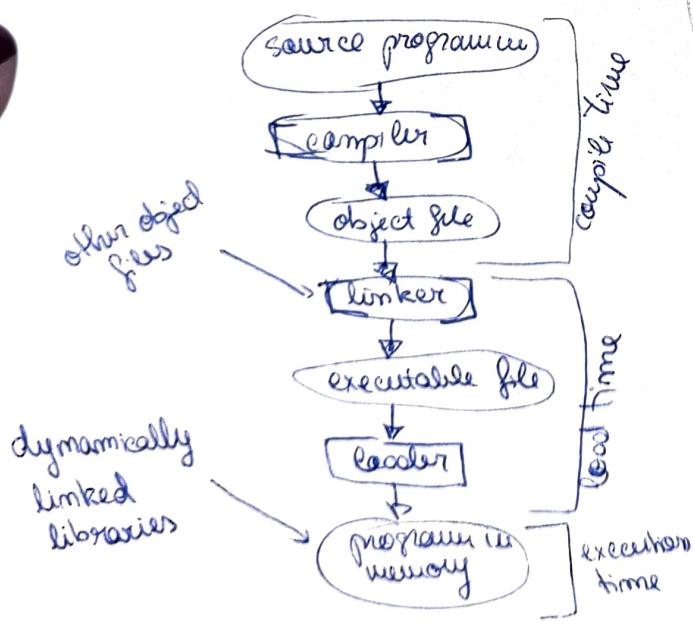
→ Hard (?) real-time = there is a deadline for the process, service offered after the deadline has expired is the same as no service at all

Event latency = amount of time elapsed from when an event occurs to when it is serviced

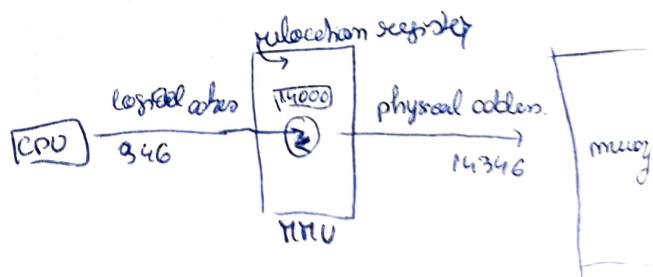
Interrupt latency = period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt



Address binding



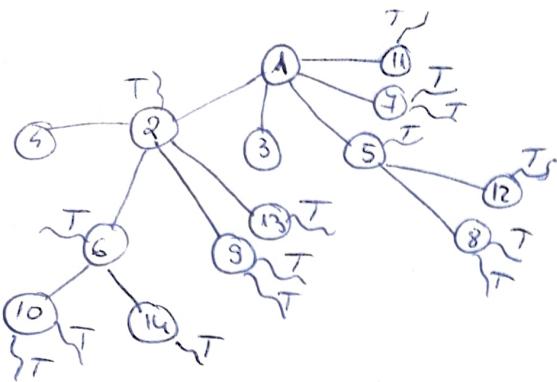
Logical vs Physical address space



PCB = process control block = data structure used by OS to manage info about a process

cluster = spatial in memory liber

EXAMEN 2024 RESTANȚĂ



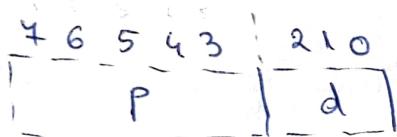
14 procese & 16 threaduri

```

for k();
if (fork()) {
    fork();
} else if (!fork()) {
    pthread-create();
}
else {
    fork();
}
pthread-create();
}

```

2) Fie un procesor pe 8 biți (1 byte/octet) cu paginare ce emite adrese logice de tip:



• Care e nr. total de pagini? Care e dimensiunea unei pagini? Care e dimensiunea totală a memoriei virtuale?

5 biți pt. a codifica nr. de pag. (p dim diagraamă) $\Rightarrow 2^5 = 32$ pagini

3 biți pt. a - - - dimensiunea (d dim - - -) $\Rightarrow 2^3 = 8$ bytes

Dimensiunea totală e $2^8 = 256$ bytes

• Fie că pe acest sistem suntem să scriem în vectorul $v[8]$ de int de căte frame-uri e nevoie să lăsăm tot vectorul în memoria fizică?

Apare fragmentarea și de ce tip dacă da? Scrieți tabula de pagini.

Unde se află $v[7]$ și cum arată adresa log. pt acces?

suntem int = 4 bytes

avem 9 elemente $\Rightarrow v[9] = 36$ bytes

dimensiunea unei frame = dimensiunea unei pag = 8 bytes \Rightarrow înegă 2 elem pe pag

Averem număr de $\frac{36}{8} = 5$ frame-uri (notăm fără să nu înegă tot vect.)

Apare fragmentare internă pe ultima pag (ocupă 4 bytes din totalul de 8)

Pag	Elemente
0	$v[0], v[1]$
1	$v[2], v[3]$
2	$v[4], v[5]$
3	$v[6], v[7]$
4	$v[8]$

$v[4]$ se află pe pag 3

offset-ul în pagina este 4

codificarea este = 00011 100 (addr. log.)

② Câte copii ale lui v-ai putea fișe maxim în memorie? Cum să modif
paginarea pentru a stoca mai multe? Cât copii ar începe din nou?
Unde să se află v[5] și cum ar arăta adr. log. pt. acces?

$$5 \text{ pag pt } v \Rightarrow 32 \text{ pag. pt. } 6 \text{ copii}$$

O abordare e să minimizăm fragmentarea internă, deci reducem dimensiunea paginii pt. mai mult control.

Fie noua dimensiune a pag = 4 bytes (un element / pag) $\rightarrow 64 \text{ pag. și } 9 \text{ pag mesaj/vedetă}$

$$64/9 = 7 \text{ copii în memorie}$$

$$\text{Codificare } v[5] = 00010100$$

③ Fie următoarea coadă de antetare a pag: 4, 2, 3, 2, 1, 0, 1, 5, 6, 5, 1, 7, 4, 0.
În coadă se cărează nr. reprezentând identificatorul unei pag. ce trebuie adusă în memoria principală.

fol. alg. LRU, care e nr. min. de frame-uri necopiate pt. ea după căcerarea inițială în memorie, să nu se producă niciun page fault

niciun page fault \Rightarrow toate hărurile aduse în memorie \Rightarrow numărul elem. distincte = 8 \Rightarrow 8 frame-uri

$$m = 8 \Rightarrow m-3 = 5$$

④ Fie m acest nr., ilustrați cum arată duplicarea alg. pt. m-3 frameuri
Nr. din paranteză = ultimul pas la care nr. a fost folosit.
Cu cât ~~se~~ mai mărește, cu atât a fost la un pas mai vechi
al cărui nr. cel mai mic e eliminat.

Pas	Frame 1	Frame 2	Frame 3	Frame 4	Frame 5
1	4(1)				
2	4(1)	2(2)			
3	4(1)	2(2)	3(3)		
4	4(1)	2(4)	3(3)		
5	4(1)	2(4)	3(3)	1(5)	
6	4(1)	2(4)	3(3)	1(5)	
7	5(7)	2(4)	3(3)	1(5)	0(6)
8	5(7)	2(4)	6(8)	1(5)	0(6)
9	5(9)	2(4)	6(8)	1(5)	0(6)

10	5(9)	2(4)	6(8)	1(10)	0(6)
11	5(9)	7(11)	6(8)	1(10)	0(6)
12	5(9)	7(11)	6(8)	1(10)	4(12)
13	5(9)	7(11)	0(13)	1(10)	4(12)

nr de pasi = cătă nr sunt în coada de așteptare (cu tot cu dulluri ele.)

① Fie un disk cu 2024 cilindri și urm. coadă de cereri 110 în ordinea:

1984, 2005, 42, 320, 1001, 512, 31, 400

Fie oare imprezintă un cilindru, iar capul de citire al discului se află în poz. 1848 și a fost înainte de poz. 1900.

② Începând de la poz. curață, care e ordinea & dist. totală parcursă de cap pt. a satisface toate cererile din coadă fol. alg. SCAN?

Cas. că se deplasează de sus în jos ($1900 \rightarrow 1848$), deci procesarea cererile vor fi spre 0 și < 1848 . Sortare cer. disponib.: 2005, 1984, 1001, 400, 512, 320, 42, 31

Conform SCAN, capul cilindrelui le proceză:

$1848 \rightarrow 1001 \rightarrow 400 \rightarrow 512 \rightarrow 320 \rightarrow 42 \rightarrow 31 \rightarrow 0$

Ajuns la 0, capul cilindrelui acum se întoarce & proce. de jos în sus, din ordinea este 1984, 2005

$1848 \rightarrow 1001 \rightarrow 400 \rightarrow 512 \rightarrow 320 \rightarrow 42 \rightarrow 31 \rightarrow 0 \rightarrow 1984 \rightarrow 2005$

Calculăm în modul dist. la fiecare pas \Rightarrow totalul = 3853

② Dacă fol. alg. FCFS?

La FCFS, cererile sunt procesate în exact ordinea în care apar în coadă, fără să existe criterii de altă natură.

$1848 \rightarrow 1984 \rightarrow 2005 \rightarrow 42 \rightarrow 320 \rightarrow 1001 \rightarrow 512 \rightarrow 31 \rightarrow 400$

În total = 9418

③ Fie un procesor cu instrucțiuni de tipul: instr. memop, memop, memop (ex: add [ax][bx] <{0xb105}). Care e nr. de framei max? Discutăți toate situațiile care pot apărea în practică + exemplu

În funcție de poz. instr. în mem. putem avea după cas.

- 1 frame \Rightarrow cădă toate instr. sunt pe același pag.

- 2 framei \rightarrow 2 diu inst. sunt pe o pg \Rightarrow a 3-a pe una altă
- 3 - a \Rightarrow 3 col inst pe pag altă