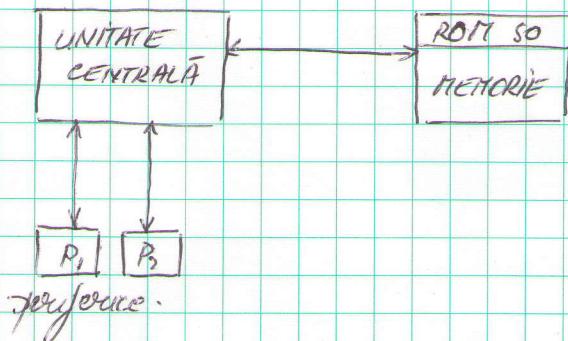


NIVELE DE ORGANIZARE A CALCULATORULUI

- ① FIZIC
- ② MICROPROGRAMAT
- ③ LIMBAJ DE ASAMBLARE
- ④ SISTEM DE OPERARE
- ⑤ APlicații

Nivelul fizic este organizat după schema lui von Neumann.



Regești: SP, PC ..

Salturi JMP, CALL, RET

CALL adr;
adr : c₁
c₂.
...
RET

Orice procesor poate să trateze în nivale de intrerupere 0, 1, ..., n-1

Fiecare intrerupere are asociată o rutină handler.

INTRERUPERI

Orică procesor tratează în nivele de Interrupere 0, ..., n-1
 Interruperea de nivel 0 are cea mai mare prioritate
 În timpul unei interruperi de nivel k, nu se ia în
 considerare interrupările de nivel k+...

MOV ... *intreruperea scârba corespondentă locației de*
lifii *memorie*
 MOV ...

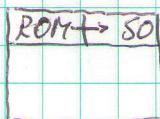
D_i

MOV ... *Interrupările nu sunt luate în considerație.*
 MOV ...

E_i

Pending.

SISTEMUL DE OPERARE



FUNCȚIONALITĂȚILE SISTEMULUI DE OPERARE

1. INTERFAȚA CU UTILIZATORUL

A. LINIE DE COMANDĂ (SHELL)

B. GRAFICĂ TIP WINDOWS

2. GESTIUNEA PERIFERICELOR

3. GESTIUNEA FIȘIERELOR

4. GESTIUNEA PROCESELOR (MULTITASKING)

5. GESTIUNEA UTILIZATORULUI și SECURITATE (MULTIUSER)

6. GESTIUNEA și PROTECȚIA MEMORIEI

7. FACILITĂȚI DE LUCRU ÎN REȚEA

8. APELURI 8787EM

GESTIUNEA PROCESELOR

CARACTERISTICI

1. CONTEXT

Circuite care joacă rolul de periferice: ceasuri

Ceasurile pot fi programate: -periodic 

-one shot 

2. IDENTIFICATOR UNIC

a) IDENTIFICATOR PROCES PĂRINTE (UO - Unix only) PID

3. STAREA PROCESULUII - stare Ready

-stare Running (User mode, Kernel mode)

-stare Sleeping

-Stepped UO

-Zombie UO

4. PROPRIETAR

5. DIRECTORUL CURENT

6. VARIABILE DE MEDIU și VALOREA LOR.

APELURI SISTEM UNIX

- se împart în două categorii :

- care returnează int (în caz de eroare, returnează -1)
- care returnează pointer (în caz de eroare returnează null)

int errno : variabilă globală folosită pt. a depista erorile int.

errno = 0 \Rightarrow succes
 $\neq 0 \Rightarrow$ eroare

void perror (char *c) : afisează mesajul asociat variabilei eroare

INCEPEREA PROCESELOR

- după d.v. al unui programator, procesul începe cu funcția main () .

- main : funcție apelată de modulul start

int main (int argc, char **argv, char **env)

env : environment

- în UNIX main poate avea 3 argumente.

env : lista de seturi de caractere de formă var = val.

ultimul pointer este null.

TERMINAREA PROCESELOR

- orice proces emite un cod de return (intre 0 și 255)

- emisarea unui cod de return se face prin apelul funcției:

void exit (int k);

exit (0); s-a terminat normal

exit (1);

dacă nu se face un exit(0) explicit, funcția care apelează pe main face automat un exit(0)

main apelează pe f, f pe g, g pe h, h pe main
 return h din main = exit(h) decat daca main este pe primul
 nivel de iteratie.

(1 - ~~descriere, scriere si run~~) ~~introducere~~

CREAREA PROCESELOR

int fork() : singurul mod in care programul poate crea procese copii.

a = b;

fork();

c = d;

: creează un nou proces f. asemănător cu procesul deja creat.

procesul execută imediat c = d

executia copilului incepe după fork
 codul de return al lui fork:

1 - eroare (nu s-a creat proces copil)

> 0 procesată, pid copil

= 0 proces copil.

int pid;

pid = fork();

if (pid < 0) {

}

/* err */

if (pid > 0) {

}

/* tata */

else {

}

(pid = 0)

}

int wait (int *st)

- asteapta terminarea procesului copil

- în codul de return pid

- 1 : err (Z proces copil)

ERERO ECHILIBRARE

TERMINARE

Dacă s-a terminat procesul copil, iar procesul tata nu a făcut față, procesul intră în stare zocuire.

st : parametru de ieșire, unde funcția wait scrie felul în care s-a terminat procesul copil.

int *st

wait (st)

} gresit \Rightarrow tb. initializat st.

int *st;

st = malloc (sizeof(int));

wait (st);

} ok, dar daca apelam malloc, se

face free st

\Rightarrow malloc - fundație costisitoare.

int st;
wait (&st); } ok.

wait (NULL); // ok, cind nu ne intereseaza codul de return

int st;
wait (&st);
WIFEXITED (st); // daca programul s-a terminat cu exit
WEXITSTATUS (st); // - codul de return
if (WIFEXITED (st))
 re = WEXITSTATUS (st);
WIFSIGNALED (st) // macro care testeaza daca programul
// s-a terminat in urma unui semnal
WTERMSIG (st) // cu ce semnal s-a terminat programul

wait : dezavantaje :

- apel blocat, sistemul ramane in sleeping
- daca se creeaza mai multe procese copii, iar noua ne trebuu anume.

int waitpid (int pid, int *st, int opt)
 ↓
 pid -ul asteptat

pid > 0 ⇒ astept dupa un pid anume

pid = -1 ⇒ astept dupa orice proces copil ; exact ca la wait

st : - returneaza pid
- se interpreteaza aproape la fel ca la wait

opt 0 = nicio optiune

opt WNOHANG - apel neblocant
- nu sta sa astepte, se scade automat catre cu
cod de return -1, iar urmatoare va barea EAGAIN

opt WUNTRACED → cod de return > 0 ⇒ pt. procesele stopate

Marea stopped : indiferent de comanda se opreste /imprejurata comanda

WIFSTOPPED (st) - daca a fost stopat

WSTOPSIG (st) - in urma carui semnal a fost stopat.

FAMILIA DE APELURI exec

- are ca prim argument un char * numele fișierului.

- ① int exec (char *exe, char *arg0, char *arg1, ... char *argn, NULL)

ex: ls -l

exec (" /bin /ls ", "ls ", "-l ", NULL);

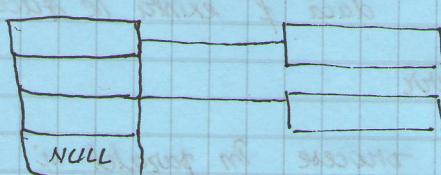
- ② int execvp (char *exe, char *arg0, char *arg1, ..., char *argm, NULL)

Obs: nu mai trebuie să dăruim exe (executabilul) cu cale completă pt. că time cout de variabilele de mediu PATH - lista de directoare cu cale completă separată prin : în UNIX (; pe windows)

PATH = . : \$PATH - în PATH se adaugă directorul curent
export PATH

- ③ int execle (char *exe, char *arg0, ... char *argn, NULL, char ** env)

env - lista de adrese către gruuri de caractere, de cele patru trebuie să arătă sintaxa variab = valoare
ultimul pointer = NULL



- ④ int execv (char *exe, char **argv)

↓ → ultimul pointer NULL

cale completă (cale care nu se termină cu "\0").

- ⑤ int execvp (char *exe, char ** argv)

- ⑥ int execve (char *exe, char ** argv, char ** env)

COMUNICARE INTRE PROCESE

write(a);

read(a);

write(b);

read(b);

1. FISIERE DE TIP SPECIAL

2. IPC SYSTEM V

3. SEMNALE

1. FISIERE DE TIP SPECIAL

- se folosesc două tipuri de fiziere: pipe, socket

- modul de operare exclude automat trei fiziere pe terminal

• stdin → tastatură

• stdout → ecran

• stderr

com < f - stdin nu mai e pe terminal, e pe fisierul f (citește din fisierul f)

com > f - redirectionare stdout (scrise în fisierul f)

com >> f - redirectionare stdout

diferența la > f dacă f există

la >> f dacă f există ne face append

com 2>f - redirectionarea stderr

c1 | c2 | ... | cm - se lansează cele m procese în paralel și

stdout c1 = stdin c2 ...

pe ecran ? lui cm ↓

1 pipe

- comunicări filtru

- cum lucat ? n-1 fiziere de tip special pipe

- pt. comunicare unidirectională între procese



Socket

- pt. comunicare bidirectională

- pt. comunicarea între procese aflate pe margini diferențiate

2. IPC SYSTEM V

- cozi de mesaje (msg), vectori de semafoare (sem), zone de memorie (shm)

- Caracteristici:

- identificatorul extern (*) internum

(*) key_t ftok (char *f, int nr)

figier, care nu să existe

- nu este apel sistem

- rezultaice numele unui identificator extern

msg

Pt fiecare IPC trebuie să avem un apel sistem care să recuperereze identif internum folind de la cel extern

int msgget (key_t cheie, int opt)

-1 în caz de insucces

> 0 în caz de succes : identificatorul internum al cozii de msg.

cheie ftok, IPC_PRIVATE

opt IPC_CREAT | IPC_EXCL | drepturi de acces

int msgsnd (int id, void *p, int lg, int opt)

adresa din memorie unde am construit msg

lungime msg

typedef struct myMsg {

unsigned long tip;

}

opt = IPC_NOWAIT

int msgrev (int id, void *p, int lg, long tip, int opt)

- citeste mesajul pînă la adresa p - mesaj se distruge
(cel mai vîndus mesaj de tipul tip)

tip > 0 cel mai mult mesaj de tipul respectiv

= 0 -- dim coada indiferent de tip

< 0 -- de tipul ? tip

opt : IPC_NOWAIT, MSG_NOERROR

↳ transmită mesaj la lg dacă e prea mare

int msgctl (int id, int oper, struct msqid_cbs *p)

oper - tipul operatiei dorite

IPC_RMID sterge IPC complet din sistem.

prin urmare faptul

este ca

TAIWAN - SAN = TAI


```
struct sembuf {
    unsigned short sem-mnum;
    short sem-op;
```

// descrie operatiu asupra unui
semajon atomic.
// index-ul unui reu atomic in vect.
de semajonare.

// descrie tipul operatiei: HIGH/LOW
(de mai sus).

sem-op = pozitiv \Rightarrow re ador. cu nr.
de valori = sem-op.

• negativ \Rightarrow decrementare cu un
nr. de valori

(se blocheaza daca ajunge la 0)
(si asteapta)

$\circ = 0 \Rightarrow$ asteapta ca sem.

să ajungă pe 0

short sem-flg;

// optiuni ale operatiei'

IPC_NOWAIT

SEM_UNDO \Rightarrow la terminarea
procesului, efectul nu
fie reversat.

Apelul operatiei este dupa urmatoarele

int semop (int id, struct sembuf *tab, int nr)

cel membrat
prin semget

nr. de elem
din tab.

- se poate opera pe mai multe sem. din vect deodata.
- apelul poate rămașe blocat, daca cel putin una din oper. duce in stare blocață.
- returneaza -1, daca id-ul e invalid
- cod de return: 0 : toate op. au avut succes.

- apel la nivelul interrupților.

(stop tru, stop tru, stop tru) tempor tri : librat

stop. stop. stop.
stop. stop. stop.
stop. stop. stop.

deosebit de
lipsa stoping
(stop. stop.)

lipsa stoping la initial si -1 - OK dupa stop librat

. stop. stop. + -

Apelul

int semctl (int id, int oper, ...)

Operatiile:

IPC-STAT → încarcă obiecturi în structură

IPC-SET → instalează obiecte în structură.

IPC-RMID

GETVAL → arg 3 tb. nu fie iată și nr. numărator.

în caz de succes returnează val. numerică a scui.

SETVAL → modifică val. numărator; arg 3 → indicele sau.

arg 4 → noua val a scui

GETALL → arg 3 : vector de int ; ret. val tuturor scui

SETALL → modif val. tuturor scui ; arg 3 : vect. de int cu noiile valori.

• utilă, pt. initializarea semajonului.

→ nu se folosește pt. arbitrarea proceselor concurente (vezi semop)

ex: tip de date FILE dim stdio.h e definit printr-un struct,

dar se folosește ca atare.

FILE : struct {

(fbo * fiov) fiovptr fmi ;

structs sbo sboops ; sbo ;

char * sbo ; } ;

(q * 2b -> fiovptr tbuffer, fbo fmi, sbo fmi) fiovptr fmi ;

TAT3_391

TAT2_391

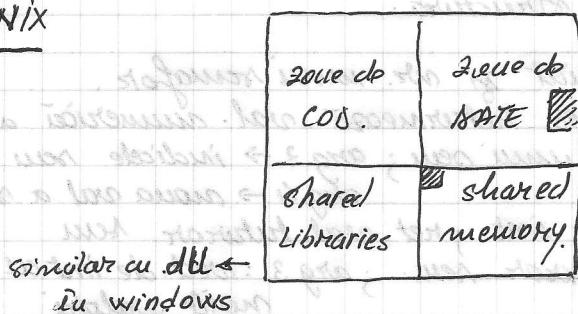
ATMA_391

scrierile este să comunică între ele informații și să interacționeze

3. ZONE DE MEMORIE PARTAJATA

- în zona unui proces nu poate scrie niciun alt proces.

HP-UNIX



pe lângă z.-de date datei procesului de S.O. ne mai poate folosi ap`o zonă în cadrul procesului care nu mai are certitudinea că un alt proces nu o mai folosește.

Folosinu apelul :

- `int shmget (key_t cheie, int nr, int opt)`
→ recuperază id-ul.

- `void *shmat (int id, void *adr, int opt)`
→ operatie prin care se atrelgează.
→ exec : met NULL.

→ optiuni : SHM_RDONLY

se punte NULL, pt. a lăsa S.O. să aleagă adresa.

- `int shmdt (void *adr)`
→ operatia de detasare.
→ adr': adresa recuperată prin shmat

- `int shmctl (int id, int oper, struct shmid_ds *p)`

IPC_STAT

IPC_SET

IPC_RMID.

Concluzie

⇒ instrumente puternice pt. comunicarea între proceze.

SEMNALE

- procesele utilizatorilor normali nu pot trimit semnale decat catre procesele lansate tot de ei.

signal.h

NSIG

#define SIGINT 4 // fiecare semnal e identif. printr-un nr.

- pot exista si semnale reportabile altre sisteme.
- la primirea unei semnale procesul se comporta ca la primirea unei interruperi.

handler

- (1) implicite // fiecare semnal are handlerul lui
 - (2) scris de programator // exista un apel de sistem care ne permite sa instalam handlerul nostru
- majoritatea handlerului nostru se term cu exit();
 - (2) 1-ar termina prim return

shell

kill pid // trimite catre sistem SIGTERM

kill -nr pid

kill -INT pid

numele generic al semnalului din signal.h
far de prefixul SIG, care e obligatoriu

apelul sistem

int kill (int pid, int sig)

- se folosesc numele generice ale semnalelor.
- exec : -nu exista semnalul -
- nu am obiectua altora procesului.

* Semnale din partea sistemului de operare

■ de la tastatura de control

* SIGINT

la apăsarea tastei Intr (de obicei are val. ^C
Ctrl-C)

* comportament implicit de term procesului:

SIGQUIT

la apăsarea ^\

* SIGSUSP

la apăsarea tastei SUSP, cel mai ușor: ^@

- pună procesul în stare stopată.

* SIGCONT

- are efect de la tast.

- dacă procesul era stopat, îl continuă / trezeste.

SIGKILL

- tot nemodificabil

- sfondul este exit()

(kill - kill sau kill -9)

■ în urma greselilor de programare

SIGSEGV

- încercare de scriere unde nu avem dreptul.

- comportament implicit: corte și exit.

SIGILL

- când programul computer-ului ia o valoare ilegală.

SIGBUS

- pe lângă nu se prevede

- eroare de magistrată.

Paranteza:

Pointeri către funcții

P. normali → constante
P. normali → variabile

P. către funcții

ex: int f (...) {
 ...
}

f (...);

f; // adresa din zona de cod unde se află fct. f.

int (*p)(); // p reprez. un pointer către f.
în acest caz neinitializat.

(*p)(...) // se primește SIGILL dacă p nu e initializat
în cursa programării.

int g(...);

...
}

if (a < b)
 p = f;
else
 p = g;

ex: int qsort(void *tab, int size_element, int nr_elem,
 int (*comp)(void *a, void *b))

// arg 4: pointer către fct, care primește ca arg 2 pointeri

Vectori de pointeri către funcții

ex: int (*tab)()[100] = {f0,..fi,...} // ? sintaxă
 (*tab[i])(...)

↑
oper.

SEMNALE (continuare)

Semnale din partea sistemului de operare (continuare)

SIGBUS

- emis de sistemul de operare în urma unei hotărîri a utilizatorului și reprezintă o eroare de magistrală
 - nu se primește pe toate sistemele de operare (ci în funcție de hardware, de ex. pe procesoarele Intel nu are loc)

Avem, de exemplu, instructiunile:

MOV EAX, addr;	//descarcă un registru pe 32 de biți la o adresă
MOV AX, addr;	//descarcă un registru pe 16 de biți la o adresă
MOV AL, addr;	//descarcă un registru pe 8 de biți la o adresă

Pe procesoarele Intel durata executiei celei de-a doua instructiuni diferă în functie de valoarea `addr`:

- dacă este pară, se execută într-un timp;
 - dacă este impară, se execută în doi timpi;

Similar, avem pentru prima instructiune:

- dacă este multiplu de 4, se execută într-un timp;
 - dacă este pară, dar nu multiplu de 4, se execută în doi timpi;
 - dacă este impară se execută în 4 timpi:

OBSERVATIE:

Compilatoarele aşază variabilele la cea mai „bună” adresă posibilă; Nu avem control asupra alegerii adresei la care se aşază în memorie.

ex.: int i,j; // nu putem face o afirmație privind modul în care este așezat j față de i
int t[2]; // variabilele tabloului sunt asezațe una după alta

- privitor la structuri;

Ayem: struct{

char x;

int y;

1

Ca și mai sus, nu putem face o afirmație privind modul în care sunt așezate în memorie variabilele din interiorul structurii. În plus, ca urmare a alinierii diferite după caz în memorie, nu putem face, în general presupunerea că:

`sizeof(struct s) = sizeof(char) + sizeof(int)`

OBSERVAȚIE:

Anumite procesoare nu acceptă descărcarea de registrii decât la adrese bine aliniate.
(Pe Intel merge, dar mai lent).

Avem, următoarea situație, în care pe anumite procesoare am primi semnalul SIGBUS, semnificând o eroare de magistrală:

```
char buf[100];
char *p;
int x;
x=*(int*)(buf+1);      /* buf se incrementează cu 1 ca adresă, iar apoi se descarcă 4 octeți de la
                        adresa respectivă (care nu mai este multiplu de 4); moment în care se
                        primește SIGBUS */
```

De aceea, atenție în general la casting.

SIGFPE

- semnalează floating point exception;

OBSERVAȚIE:

Reamintim, că implicit comportamentul semnalelor este terminarea procesului.

APELURI SISTEM PENTRU LUCRUL CU SEMNALE

Blocarea semnalelor

Reamintim că procesele se comportă la primirea unui semnal, ca în cazul întreruperilor.

Dacă primim un semnal blocat, acesta devine **pending** (agățat), până în momentul deblocării, când este livrat procesului. În cazul în care mai multe semnale se află în stare pending, în momentul deblocării, se livrează doar ultimul.

OBSERVAȚIE:

Cele trei semnale nemodificabile (SIGSUSP, SIGCONT, SIGKILL) nu pot fi blocate.

În header-ul „**signal.h**” este definit tipul de date **sigset_t**. Acesta reprezintă o mulțime de semnale. Nil putem imagina că pe un vector de n biți, unde biții 0 semnifică neapartenența semnalului la mulțime.

Avem următoarele primitive (nu sunt apele sisteme):

- **int sigemptyset (sigset_t * p)**

- inițializează **p** cu mulțimea vidă de semnale;
- cod de return: întotdeauna 0;

- **int sigfillset (sigset_t * p)**

- inițializează **p** cu mulțimea totală (toți biții sunt 1);

- **int sigaddset (sigset_t * p, int sig)**

- **sig** este identificatorul unui semnal;
- adaugă semnalul **sig** la mulțimea **p**;
- dacă bitul corespunzător semnalului respectiv este deja 1, nu se modifică nimic;

- **int sigdelset (sigset_t * p, int sig)**

- **sig** este identificatorul unui semnal;
- elimină semnalul **sig** din mulțimea **p**;
- dacă bitul corespunzător semnalului respectiv este deja 0, nu se modifică nimic;

- **int sigismember (sigset_t * p, int sig)**

- **sig** este identificatorul unui semnal;
- returnează: 1, dacă semnalul **sig** aparține mulțimii **p** sau
0, în caz contrar;

Pentru blocarea/deblocarea semnalelor utilizăm apelul de sistem:

```
int sigprocmask ( int op, sigset_t * p_nou, sigset_t * p_vechi)
```

- **op** indică operația pe care o realizăm, astfel poate avea valorile:

- **SIG_SETMASK** : caz în care **p_nou** va conține doar lista de semnale blocate, fără a ține cont de ce s-a întâmplat anterior;

- **SIG_BLOCK** : caz în care la lista anterioară se adaugă semnalele blocate din **p_nou**;
 - **SIG_UNBLOCK** : semnalele din **p_nou** vor fi deblocate;

- **p_nou** reprezintă mulțimea desemnată pentru a fi blocată/deblocată;
- **p_vechi** reprezintă mulțimea semnalelor deja blocate;
- ambiii pointeri trebuie fie să fie adrese valide, fie NULL;

se disting două cazuri:

- **p_nou** este NULL, caz în care vrem să aflăm lista semnalelor blocate și nu modificăm nimic;
- **p_vechi** este NULL, caz în care nu ne interesează decât să facem modificarea;

Instalarea handlerelor scrise de noi

- în acest caz se intră în usermode când se execută funcția noastră;
- funcția pentru tratarea semnalelor este de forma:

```
void hand (int sig); /*trebuie sa aibă parametrul int care pasează indicatorul
semnalului care a dus la acest apel. */
```

- există două moduri de instalare a unui handler scris de noi:

1. modul simplu, prin apelul:

```
void (* signal ( int sig, void (* hand ( int ))))(int)
```

- **sig** indică semnalul;
- al doilea parametru este funcția pentru tratarea semnalelor de mai sus;
- codul de return este un pointer către o funcție de tip **void**; (această funcție era handlerul de semnal de până acum)

OBSERVATIE:

La prima întâlnire a semnalului se folosește handlerul nostru, însă apoi se restaurează handlerul implicit. Deci programul moare. De aceea, avem:

```
void hand (int sig){
    signal (sig, hand); //comportament permanent
    .....
}
```

- precum:
- **hand** poate avea valoarea unor pointeri valizi sau a unor constante de tip pointer,
 - **SIG_DFL** - se restaurează comportamentul implicit
 - **SIG_IGN** - un simplu **return**, nu face nimic;

2. modul complicat, prin apelul:

```
int sigaction ( int sig, struct sigaction * p_nou, struct sigaction * p_vechi)
```

- **sig** indică semnalul;
- **p_nou** indică noul comportament, iar
- **p_vechi** este utilizat pentru a recupera situația anterioară;
- **p_nou** și **p_vechi** pot fi și NULL, ca și la **sigprocmask**;
- tipul **struct sigaction**:

```
struct sigaction{
    void ( * sa_handler) (int); //pointer către funcț void
    sigset_t * sa_mask; /* lista de semnale ce vreau să fie blocate pe timpul execuției handlerului.*/
    int sa_flags; // 0 sau SA_RESETHAND
}
```

- prin acest apel se instaurează un comportament definitiv;
- **SA_RESETHAND** face ca acest comportament să fie valabil doar pentru prima întâlnire a semnalului;

Discutăm următoarea problemă: ne dorim să sărim dintr-un punct al programului într-o altă funcție.

Headerul **setjmp.h** conține tipul de date **sigjmp_buf**, care memorează un context de proces într-un anumit punct.

Pentru a-l inițializa, folosim apelul:

```
int sigsetjmp ( sigjmp_buf buf, int opt )
```

- adaugă în **buf** contextul procesului în care s-a făcut apelul.
- **opt** – pentru a reține și masca de semnale blocate;

În handler, folosim apelul:

```
int siglongjmp ( sigjmp_buf buf , int val )
```

- se reîntoarce în punctul în care s-a făcut **sigsetjmp**
- apelul nu poate fi făcut decât într-o funcție care e apelată direct sau indirect (descendentă) de **sigsetjmp**;
- **val** pasează o valoare, care trebuie să fie diferită de **0**, pentru a semnala că venim din **siglongjmp**; putem pasa valori care să ne indice în urma cărui semnal s-a făcut saltul;

```
ex.:    int x;  
.....  
x=sigsetjmp(buf,0);  
//instructiuni  
switch (x) {  
    case 0: //normal  
    case 1: //un anumit semnal, ex. SIGSEF  
    .....  
}
```

PROBLEME CLASICE DE CONCURENTĂ

(cazul general, nu doar UNIX)

Problema Producator – Consumator

(d.p.d.v. al proceselor)

Presupunem că procesul producător pune în memoria partajată, iar cel consumator ia /citește de acolo. Deducem că avem nevoie de următoarele date partajate:

- un buffer, care trebuie să fie circular;
- adresa la care va scrie procesul producător;
- adresa de la care va citi procesul consumator.

Situările în care procesele așteaptă unul după altul, după cum bufferul este plin sau gol, trebuie să se semaforizeze, pentru a nu opri procesele.

Exemplu de implementare al acestui procedeu cu semafoare (tip Dijkstra):

```
#define N 100
typedef int semaphore; //acesta este de fapt un obiect sistem nu are tipul int
semaphore mutex = 1; //impiedică un proces să intre în zona critică, când alt proces este acolo
semaphore empty = N; //memorează numarul de locuri libere din buffer
semaphore full = 0; //memorează numărul de locuri ocupate din buffer

void producer (){
    int item;
    while (TRUE){
        item=produce_item; //produce următorul element;
        down (&empty); //decrementează numărul de locuri ocupate;
                        //dacă empty=0, rezultă că bufferul e plin, deci
                        //aștept./*
        down (&mutex); //intru în zona critică
        insert_item (item);
        up (&mutex); //ies din zona critică
        up (&full); //incrementează numărul de locuri ocupate
    }
}

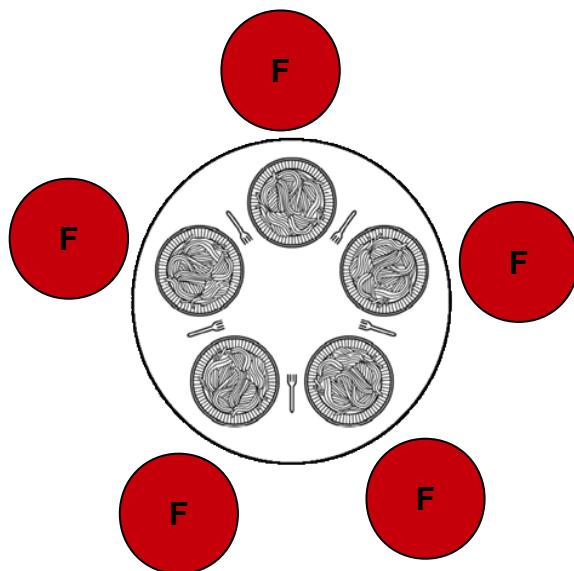
void consumer(){
    int item;
    while (TRUE){
        down (&full); //decrementează numărul de locuri ocupate;
                        //dacă bufferul e plin, atunci aătept */
        down (&mutex);
        item=remove_item();
        up (&mutex);
        up (&empty); //incrementează numarul de locuri libere
        consume_item(item);
    }
}
```

OBSERVAȚIE: În UNIX, putem utiliza message queues

Punerea unor monitoare pe obiecte

*Java (final de semestru)

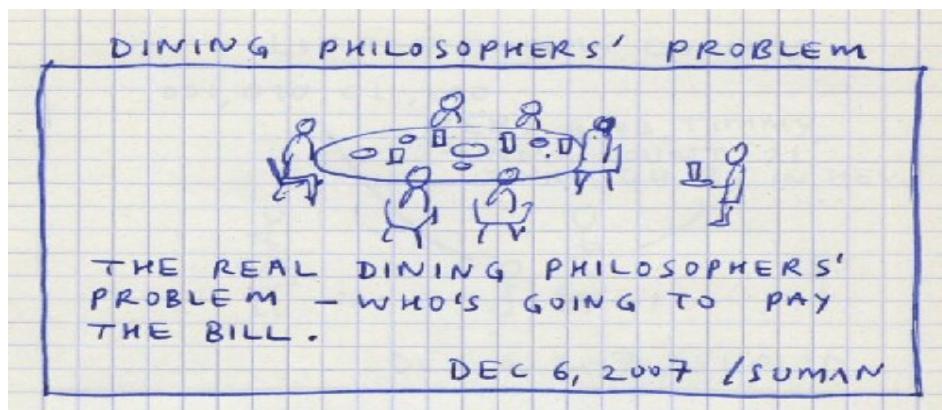
Problema celor 5 filosofi (n)



La o masă se află 5 filosofi, care doresc să mănânce. Pe masă se alfă farfurii și tacâmuri ca în imagine. Fiecare filosof are nevoie (evident) de două tacâmuri pentru a mâncă.

Filosofii reprezintă procese, iar tacâmurile resurse utilizate de procese.

Se dorește evitarea situației de **deadlock**.



Punerea unor monitoare pe obiecte

Alte sisteme de operare folosesc pentru excluderea zonei critice noțiunea de **monitor**.

Un exemplu de monitor in Java, folosind cuvântul cheie **synchronized**:

```
synchronized (ob) {  
    ...  
}
```

sau

```
public synchronized met(...){  
    ...  
}
```

Observatie: Atat obiectul **ob**, cat si functia **met** trebuie sa fie statice.

Threaduri (Fire de execuție)

- în cadrul unui proces se pot executa mai multe threaduri, care ruleaza în paralel.

Într-un sistem de operare, procesele pot fi:
- monothread (UNIX)
- multithread (Windows)

În UNIX s-a implementat o bibliotecă, numită **pthread**, care permite folosirea mai multor threaduri într-un proces.

Threadurile partajează variabilele globale și fac parte din același proces.

Problema celor 5 (N) filosofi

Noțiunea de deadlock

Acesta se produce, atunci când anumite obiecte pot lua anumite resurse pentru a le utiliza în mod exclusiv. Resursele respective sunt nepartajabile (trebuie eliberate de procesul care le detine, pentru a putea fi folosite mai departe de un alt proces).

ex.: $P_1 \rightarrow r_1$ (a se citi: „procesul P_1 achiziționează resursa r_1)
 $P_2 \rightarrow r_2$
 $P_1 \rightarrow r_2$ (P_1 va aștepta după P_2)
 $P_2 \rightarrow r_1$ (P_2 va aștepta după P_1)

și astfel s-a ajuns într-o situație de deadlock.

Sistemul de operare detectează astfel de situații și va alege unul din proceze ca *victimă* (apelul sistem al acestui proces ieșe afară cu cod de eroare, semnificând că nu a reușit să achiziționeze resursa respectivă).

În UNIX codul de return va fi **-1**, iar **errno** ia valoarea **EDEADLOCK**.

Observatie: Exista situații de așteptare (diferite de deadlock). În acestea sistemul de operare nu intervine.
De ex.: cu semafoare (care nu sunt privite ca resurse).

Un deadlock poate fi:

- **direct** (ca în exemplu);
- **indirect** (circular: P_1 așteaptă după P_2 , P_2 după P_3 , ...; în această situație, un proces este ales victimă și astfel se „sperate” cercul);

Observatie: Pentru evitarea deadlockurilor se dorește eliberarea resurselor folosite cât mai repede posibil.

Criteriile alegerii procesului victimă de către sistemul de operare:

- în funcție de resursele consumate de fiecare proces în parte. (cel cu mai puține resurse va fi ales);
- ia în considerare și timpul CPU consumat (procesul victimă va fi cel cu durată de timp mai scăzută);

Problema celor 5 (N) filosofi

```
#define N 5
void philosopher(int i){
    while(TRUE){
        think();
        take_fork();           /* apeluri blocante, va rămâne în așteptare dacă nu e liberă*/
        take_fork((i+1)%N);
        eat();
        put_fork(i);
        put_fork((i+1)%N);
    }
}
```

NU

```
#define N 5
#define LEFT (i+N-1)%N
#define RIGHT (i+1)%N
#define THINKING 0
#define HUNGRY 1
#define EATING 2

typedef int semaphore;
int state[N];
semaphore mutex=1;
semaphore s[N];

void philosopher(int i){
    while(TRUE){
        think();
        take_forks();          /*se iau ambele furculițe, dacă nu se trece mai departe*/
        eat();
        put_forks();
    }
}

void take_forks(int i){
    down(&mutex);
    state[i]=HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);            /*semaforul rămâne pe 0, dacă nu a reușit să le ia*/
}

void put_forks(int i){
    down(&mutex);
    state[i]=THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}
```

DA

```
}
```

Problema scriitorii-cititori

Primul proces	Al doilea proces
write(a); write(b);	read(a); read(b);

- zone critice (vezi cursul 4)

Problema frizerului somnorus

Într-o frizerie se află unul sau mai mulți frizeri și clienți, simbolizând două tipuri de procese.

Regulamentul: - dacă nu e niciun client, frizerul doarme (dacă nu doarme, tunde);

- clienții dacă nu găsesc frizer liber (dormind), se aşază să aștepte pe scaun;
- avem **n** clienți și **m** scaune;
- clienții dacă nu găsesc scaune libere, ies în afara frizeriei.

```
#define CHAIRS 5
typedef int semaphore;
semaphore customers=0;
semaphore barbers=0;
semaphore mutex=1;
int waiting=0;           /*variabila indică numărul de clienți aflați în așteptare pe scaun*/

void barber(){
    while(TRUE){
        down(&customers);
        down(&mutex);
        waiting--;
        up(&barbers);
        up(&mutex);
        cut_hair();
    }
}

void customer(){
    down(&mutex);
    if(waiting<CHAIRS){
        waiting++;
        up(&customers);
        up(&mutex);
        down(&barbers);
        get_haircut();
    }
    else
        up(&mutex);
}
```

Planificarea proceselor

Având o listă de procese, atunci când un proces *running* este întrerupt, urmează să continuăm cu un alt proces.

Criteriile de alegere a următorului proces:

- (vechi) FIFO;
- (modern) **prioritățile** proceselor;

Prioritatea este specificată de utilizator și ajustată în timp de sistemul de operare.

UNIX: (comenzi din shell)

\$com	- procesul va fi lansat cu prioritate standard
\$nice com	- procesul va fi lansat cu prioritate mai scăzută
\$nice -nr com	- în plus, se specifică în nr numărul de unități cu care este scăzută prioritatea (altfel se scade un număr standard de unități). Din root se poate specifica o valoare negativă, pentru a lansa un proces cu prioritate mai ridicată decât cea standard.

Pe parcursul desfășurării procesului, în funcție de timpul CPU consumat și resursele folosite, sistemul de operare poate hotărî modificarea (ajustarea) priorității.

- În UNIX se folosește în acest scop algoritmul Round Robin : lista de procese este partionată în funcție de prioritate.
- Alte sisteme de operare folosesc *tehnici de prognoză* (prognoza se face pe baza execuțiilor anterioare ale procesului)

Gestiunea & securitatea memoriei

I. Securitatea memoriei

Funcționalitatea ei este indispensabilă într-un sistem de operare multitasking;

Un proces este compus dintr-un executabil și din date. Cele două componente sunt pласate în zone diferite de memorie: prima este RO (ReadOnly), a doua RW.

Amintim de semnalul SIGSEF primit în urma încercărilor unui proces de a scrie într-o zonă în care nu are acces.

Zona de date a unui proces este compusă din trei subzone:

- A. Zona de date statice
- B. Zona de heap
- C. Zona de stivă (stack)

A. Zona de date statice

Executabilul plasează variabilele globale și pe cele locale statice în această zonă, la o adresă dată de compilator. (Variabilele sunt inițializate automat cu 0)

B. Zona de heap

Aici sunt pласate variabilele alocate dinamic cu funcțiile **malloc**, **calloc** și **realloc**.

(vezi mai jos *Alocare dinamică în C*)

C. Zona de stivă

În momentul apariției unei funcții, *contextul dinamic* al acesteia este pus pe stivă.

Contextul dinamic constă în:

- adresa de return;
- cod de return;
- parametrii;
- spațiu alocat pe stivă pentru variabilele locale automate.

II. Gestiunea memoriei

Pentru ca un proces să intre în stare **running**, întreg contextul său trebuie să fie încărcat în memorie.

Problema insuficienței spațiului: → apare fenomenul de sort

memorie → pagini → segmente

Contextul oricărui proces trebuie încărcat în totalitate în memorie și ocupă un număr întreg de pagini.

ex.: P1 → memorie
P2 → memorie
...
Pn → memorie

Se ajunge astfel ca la un moment x toată memoria să fie încărcată.

Pn+1 vrea k pagini contigüe (i.e. Așezate una după alta)

Atunci, sistemul de operare sacrifică unul sau mai multe procese contigüe, care însumate ocupă cel puțin k pagini. Se salvează conținutul zonelor de memorie ale acestora pe disc și se alocă k pagini nouului proces. Apoi, procesele salvate se vor reîncărca de pe disc în memorie și.a.m.d.

Acesta este fenomenul de sort, sau memoria virtuală.

Scopurile urmărite în momentul alegerii proceselor ce vor fi sacrificeate

1. Minimizarea numărului de accese la disc. (Deoarece discul este un periferic, accesele la el duc la o scădere a performanței).
2. Întârzierea pe cât posibil a fragmentării memoriei. (Dacă nu se găsește cel puțin un proces cu zonă continuă de k pagini, se caută pentru cel puțin k pagini. Se obține astfel o zonă mică rămasă goală. În timp apare o multitudine de astfel de zone mici libere și nu pot fi folosite ca atare, deci apare o fragmentare a memoriei. Sistemul de operare face din când în când o defragmentare a memoriei și se obține o zonă continuă liberă.)

Strategii posibile

1. FIFO (nu se aplică)
2. Best fit (Cea mai bună potrivire)
 - vreau să eliberez cel puțin k pagini, iau toate combinațiile posibile și caut cel mai apropiat număr mai mare decât k de pagini. Astfel obținem cea mai bună potrivire.
 - Dezavantaj: conduce rapid la fragmentare.
3. LRU (Last Recent User)
 - se iau în considerare și prioritățile proceselor și experiența cu acele procese și sacrifică procesele, cărora li se dau controlul mai rar (cele mai vechi pagini utilizate).

Alocare dinamică în C

Apelurile sunt definite în biblioteca standard C (stdlib.h). (Observăm că ele nu sunt apeluri sistem, dar în interiorul lor se pot face apeluri sistem.)

1. **void* malloc (int dim)**

- **dim** este dat cu ajutorul operatorului **sizeof()** și reprezintă dimensiunea în bytes a zonei de memorie pe care dorim să o rezervăm
- se va realiza cea mai bună aliniere posibilă în memorie
- returnează un pointer către zonă alocată sau NULL în caz de eroare (aceasta se produce dacă nu există suficient spațiu liber)

2. **void* calloc(int size, int dim)**

- alocare contiguă a **dim** obiecte de dimensiunea **size**
- **size** este dat cu ajutorul operatorului **sizeof();**
- returnează un pointer către zona alocată sau NULL în caz de eroare (aceasta se produce dacă nu există suficient spațiu liber)
- se observă că apelul *calloc(x,y)* este echivalent cu apelul *malloc(x*y)*. Diferența constă în faptul că funcția *calloc* initializează memoria cu 0;

3. **void* realloc(void* p, int dim)**

- realizează o alocare dinamică;
- **dim** reprezintă dimensiunea anterioară + un increment (ex.: 1000+2)
- pointerul **p** pasat trebuie să fie NULL sau un pointer obținut prin *malloc* sau *calloc*
- modul de funcționare: se pune întâi problema spațiului liber, dacă acesta este găsit, atunci se alocă memoria și se returnează **p**; dacă, însă, *increment x octeți* sunt ocupati se caută o altă zonă de memorie de dimensiunea *dim_anterioră+increment*; dacă o găsește, o alocă, copiază conținutul și eliberează zona de memorie indicată de **p**, și returnează un nou **p**.

4. **void free(void*p)**

- eliberează zona de memorie
- pointerul **p** trebuie să fie NULL (nu are niciun efect) sau un pointer obținut prin *malloc*, *calloc* sau *realloc*.

Gestiunea perifericelor

Tipuri de clasificări:

- **periferice de intrare:** tastatura
- **periferice de ieșire:** ecranul
- **periferice bidirectionale:** extensiile de memorare, HDD, Floppy

În funcție de cui i se adresează

- **human readable** (ieșire și intrare de la operator uman: tastatură, ecran)
- **machine readable** (HDD, modem)

Din punct de vedere al funcționalitatății

- **de tip caracter** (intrare + ieșire: octet cu octet; tastatură, imprimantă, ecran)
- **de tip bloc** (informația se citește în blocuri cu o dimensiune dată de fiecare sistem: discul)

I. Gestiunea perifericelor de tip caracter

Toate perifericele posedă un port de date care este de tip *in* sau *out* și un port de stare numai *in* – putem citi starea perifericului.

ROL: La perifericele de intrare în informația care cuprinde starea perifericului avem o informație de tip *boolean*:

ex.: RXRDY = 1 (este ceva pe „teavă”, are sens să citeșc portul de date)

La perifericele de ieșire: TXRDY

- stare normală = 1 (perifericul e gata să primească date)

Filosofii de a scrie drivere pe periferice

1. Polling (veche)

Intrare:

1. RXRDY?
2. NU – mergi la 1
3. Citește caracter

Ieșire:

1. TXRDY?
2. NU – mergi la 1
3. Trimite caracter

2. Pe intreruperi (modernă)

Fiecare periferic poate să fie legat (hardware) la un nivel de intrerupere - se produce când RXRDY (respectiv TXRDY) trece din 0 în 1 (era ocupat și nu mai este).

Intrare: (Programul principal – consumatorul – și intreruperea comunică printr-un buffer circular)

1. Buffer gol?
2. NU – returnează caracter
3. DA – return

Intreruperea (producătorul):

1. Citește caracter
2. Depune în buffer

Ieșire:

1. Buffer plin?
2. NU – depune în buffer
3. DA - return

Ieșire:

1. TXRDY?
2. DA – transmite caracter, return
3. Buffer plin?
4. NU – depune în buffer
5. DA - return

Inreruperea:

1. Buffer gol – return
2. Citește caracter din buffer
3. Trimite caracter

La ieșire programul principal este producătorul iar intreruperea consumatorul.
Principiile saltului de IO: intreruperea se realizează la trecerea din 0 în 1.

II. Gestiunea perifericelor de tip bloc

Perifericele de tip bloc corespund în general perifericelor care reprezintă extensia memoriei (HDD).

Observatie: în UNIX blocurile sunt de 512 octeți; marcați în memoria cache

SCOP: minimizarea numărului de accese la disc

fflush (FILE *fp) forțează golirea cache-ului chiar dacă nu era plin

Există trei modalități de golire a bufferului:

- a. Bloc clasic (implicit)
- b. Golire la newline
- c. Autoflush (cum am scris un caracter, cum este pus pe disc)

Observatie: În UNIX perifericele sunt cazuri particulare de fișiere.

Discurile sunt un periferic de tip special alcătuit din cilindrii, piste și sectoare.

Operații:

1. Poziționare cap R/W (penalizantă, deoarece este mecanică, deci trebuie minimizat numărul de mișcări al capului de R/W);
2. Rotire
3. Citire/Scriere propriu-zisă

Politici posibile:

1. *FIFO* (nu este bună, dar anumite periferice (precum imprimanta) nu pot fi tratate decât aşa)
2. *Cea mai apropiată cerere față de poziția curentă* (nu este bună, deoarece este inechitabilă)
3. *Tehnica ascensorului*

date de intrare: poziția curentă și direcția de deplasare (0 – jos, 1 – sus)
următoarea cerere care va fi servită este cea mai aproape de poziția curentă, dar în direcția de deplasare; dacă pe direcția de parcurgere nu mai găsește nicio cerere, schimbă direcția.

Gestiunea fișierelor

a

Un fișier este teoretic o informație (un sir mare de octeți) care este alcătuită din înregistrări, iar înregistrările sunt alcătuite din câmpuri.

O colecție de fișiere formează o bază de date: flat file.

Din punct de vedere al **structurii**, fișierele pot avea:

- **format fix:** fiecare înregistrare același număr de octeți, fiecare câmp același număr de octeți.
- **format variabil:** înregistrările nu au aceeași lungime; nu au același număr de înregistrări.
ex.: fișier text: înregistrări = linii; câmpuri = cuvintele de pe linii;

Notiuni generale despre fișiere

Operații asupra fișierelor

1. deschidere
2. exploatare
3. închidere

Mai amănunțit:

1. deschidere
2. închidere
3. citire
4. modificare (suprascriere)
5. scriere la sfârșit
6. ștergere informație
7. ștergere fișier
8. redenumire fișier
9. copiere fișier
10. modificare caracteristici

Caracteristici

1. Proprietării
2. Drepturi

La deschiderea fișierului există o informație: „poziție curentă”. (la deschidere: poziția curentă este pe primul octet; citirile și scrierile modifică poziția curentă spre sfârșitul fișierului)

Modalități de acces

1. **secvențial** (dacă vreau să citesc octetul *n* trebuie să citesc *n-1* octeți dinainte)
2. **aleator** (permite să modific poziția curentă fără să fac operații de citire/scriere și în aval și în amonte)

Observatie: În C : cu ajutorul apelului **fseek**

3. **secvențial indexat** (nu este build-in în sistemul de operare, depind de anumite softuri, biblioteci)

Gestiunea fisierelor

Modalități de acces

Accesul secvențial-indexat

Fisierele care permit accesul secvențial-indexat sunt cele cu format fix.

Ca fapt divers, Cobol are implementat accesul secvențial-indexat.

Un exemplu de bibliotecă de funcții ce permite accesul secvențial-indexat a fost dezvoltată de Informix (IBM Informix acum) este C-ISAM.

NOTA: Următoarele apeluri nu trebuie să fie scrisă în mod direct în programul de utilizator. Ele trebuie să fie scrisă în mod indirect, folosind o subrutină sau o procedură. Pentru mai multe informații, consultați documentația C-ISAM.

- **isbuild**
 - printre parametrii numele fișierului și lungimea înregistrării
 - creează fișierele xxx.dat și xxx.idx
 - **isaddindex**
 - adaugă un index în fișier
 - câmpurile c₁,...,c_n specificate nu trebuie să fie în ordine; creează indecsi pentru fiecare înregistrare
 - creează un arbore B (de căutare, dar nu binar)
 - **isdelindex**
 - ștergere index
 - **isopen**
 - deschidere fișier
 - **isread**
 - se specifică indexul și valoarea. (valoarea poate fi ISNEXT, caz în care se referă la următoarea înregistrare)
 - **iswrite**
 - scriere la sfârșit
 - **isrewrite**
 - modificare înregistrare curentă
 - **isdelete**
 - ștergere fișier

Ideeia este de a avea cât mai mulți indecsi posibil pentru a putea căuta rapid după orice criteriu. Pentru n câmpuri putem aveam maxim n! Indecsi. Numărul mare (maxim) de indecsi dezavantajează operațiile de scriere și ștergere (care provoacă actualizarea arborilor B). Suprascrierile nu sunt penalizante dacă câmpurile respective nu fac parte din nicio cheie.

Noțiunea de director (catalog)

Fișierele sunt grupate în directoare/cataloage. În majoritatea sistemelor de operare structura fișierelor/directoarelor este arborescentă.

Organizarea discului

În Windows:

- FAT (File Allocation System) (cu tabele de legături)
- NTFS (NT File System)

O diferență între UNIX și Windows ar fi că în UNIX există o singură arborescență de directoare, pe când în Windows fiecare partitie are propria ei structură arborescentă.

În UNIX există un disc0, de dimensiune mică, unde se află root.

Cu comanda **mount** se creează un director nou frunză și se declară că toate subdirectoarele sale vor fi pe discul nou creat, de ex. disc1. Cu **mount** se poate declara ca toată arborescența unui director să fie (de) pe altă mașină. (NFS – Network File System)

Fiecare disc posedă o tabelă de inoduri. La crearea unui fișier, informația se pune pe disc și apare o nouă intrare în tabela de inoduri.

Informatiile conținute de inoduri

- **pointer către zona de date**, adică, către adresa fizică de pe disc unde se află informația (de fapt este vorba de o tabelă de 8 pointeri, primul către începutul zonei de date, apoi în funcție de lungimea informației sunt folosite și următorii 6; dacă mai este nevoie de încă un pointer atunci al 8-lea pointer din tabel reprezintă adresa unui alt astfel de tabel; etc.)
- **număr de inod**, acesta este dat la creare și este egal cu numărul de disc + numărul de inod. (cum fiecare disc are propria tabelă)
- **tipul fișierului**
 - obișnuit (regular file)
 - directoare
 - periferice de tip caracter
 - periferice de tip bloc
 - link simbolic
 - pipeuri
 - socketuri (fișiere de comunicare bidirectională)
- **numărul de legături fizice** (în câte directoare e declarat)

Ex.: `ln x y` - în director apare o intrare y care corespunde același număr de inod al lui x (are două legături fizice din momentul acesta); pot fi declarate cu același nume, dar atunci în directoare diferite; practic comanda **ln** incrementă numărul de legături fizice, la polul opus se află comanda **rm** care îl decrementează atunci când șterge o intrare dintr-un director. Observăm că un fișier se consideră șters atunci când numărul de legături fizice este 0 și niciun proces nu-l mai are deschis.

Fișierele speciale de tip director

- fiecare înregistrare conține două câmpuri:
 - o numele intrării pe director (unic)
 - o numărul de disc + numărul de inod
 - se creează prin comanda **mkdir**
 - orice director conține la creare două intrări :
 - o . – numărul de inod propriu
 - o .. – numărul de inod al părintelui
 - aceste două intrări sunt obligatorii și nu pot fi șterse
 - intrările cu . sunt ascunse la comanda **ls** ; pentru a le vedea folosim **ls -a**
 - ștergerea se face cu comanda **rmdir** ; aceasta nu reușește decât dacă directorul este gol (directorul se consideră gol dacă tabela nu conține decât intrările . și ..)
 - comanda **rm -r** șterge recursiv directorul chiar dacă nu este gol
 - **proprietarul** (un utilizator, care nu trebuie să aparțină neapărat grupului proprietar)
 - **grupul proprietar** (observăm că grupurile nu sunt neapărat disjuncte)
 - **drepturi de acces**
 - pot fi modificate de proprietar
 - 9 biți împărțiți în trei secvențe de forma rwx (read,write,execute) pentru proprietar, grupul proprietar și ceilalți utilizatori.
Ex. : rwxrwxrwx – toată lumea are toate drepturile
 - absența unuia dintre drepturi se marchează prin –
Ex. : rwxr-xrw – grupul proprietar nu are drept de scriere
 - mai există 3 biți speciali : **set-uid**, **set-gid**, **sticky bit**; primele două pot fi **s** sau **S** (cu sau fără drept de execuție) pe bitul 3, respectiv bitul 6; sticky bit aplicat fișierelor de tip executabil le menține în zona de memorie chiar și după terminarea programului.
Ex. (pentru set-uid) : Știm că parolele sunt reținute în /etc/passwd , director al căruia proprietar este root și care în drepturile de acces are sigur setate : r-s--x--x ; atunci când un utilizator dorește să-și schimbe parola, proprietarul real nu mai coincide cu proprietarul efectiv, cum proprietarul procesului este root și proprietarul real devine proprietar al procesului, ceea ce îi permite să-și schimbe propria parolă.
- Remarcăm de asemenea, că **x** are efect asupra executabilelor, fișierelor text de tip scripturi shell, pentru directoare semnifică dreptul de a face change directory.

S-a mai amintit de comenzi :

```
df
chown
rm -r /*
rm -rf /*
```

Întrebare : Cum se șterge **rm** ?

Informatiile conținute de i-noduri (continuare)

- **data creării**
- **data ultimei deschideri**
- **data ultimei modificări a i-nodului**
- **data ultimei modificări a conținutului**
- **numărul de octeți conținuți în fișier**

Observație: fișierele **makefile** sunt fișiere text; conțin numele obiectelor de construit și lista dependințelor, precum și comanda de construcție.

Fișierele speciale de tip link simbolic

Ne amintim că o legătură fizică se creează cu ajutorul unei comenzi de forma:

In sursă destinație

unde **sursă** reprezintă un fișier existent, iar **destinație** este directorul (și noul nume al legăturii) în care apare o nouă intrare care are același număr de i-nor precum **sursă**.

Un link simbolic se creează cu o comandă de forma:

In -s sursă destinație

Diferența este că această comandă creează un i-nod nou care nu are informație proprie; informația din **destinație** reprezintă o trimiteră la fișierul **sursă**.

Linkurile simbolice, spre deosebire de legăturile fizice, pot fi folosite și pentru directoare. Un dezavantaj al linkurilor simbolice ar fi că în cazul ștergerii fișierului **sursă**, **destinație** nu este șters. Dacă se încearcă deschiderea fișierului **destinație** vom primi eroarea 'File Not Found' pentru **destinație**, nu **sursă**.

Structura sistemului de fișiere în memorie

Un atribut al proceselor este lista fișierelor deschise. Aceasta este reținută într-o *tabelă de descriptori proprii fiecărui proces*. Procesul are acces doar la numărul intrării în tabela de descriptori. Fiecare proces are la creare următoarele fișiere deschise: **stdin(0)**; **stdout(1)**; **stderr(2)**. Fiecare descriptor pointează către o intrare într-o altă tabelă, cea de fișiere deschise în memorie. În această *tabelă de descriptori a fișierelor deschise în memorie* se memorează poziția curentă în fișier.

Observație: Se reține o intrare nouă și un descriptor nou pentru fiecare deschidere de fișier, chiar dacă este vorba de același fișier.

O a treia tabelă este *tabela de i-noduri în memorie*. Fiecare fișier deschis are o singură intrare în această tabelă. Fiecare intrare din tabela de descriptori a fișierelor deschise în memorie pointează către o intrare din tabela de i-noduri în memorie. Mai departe, fiecărui i-nod îi corespunde o tabelă numită *tabela de blocaj asupra i-nodului*.

Apeluri sistem pentru gestiunea fișierelor

struct stat

Unul din câmpurile acestei structuri ce reține atributele unui i-nod este **int st_mode**. În acesta se memorează pe 4 biți tipul fișierului, pe 9 biți drepturile de acces și pe 3 biți drepturile speciale (set_uid, set_gid, sticky bit); există constante speciale cu care se poate testa conținutul.

```
ex.: if (s.st_mode & _S_IFDIR)
     { ...este director... }
     if (_S_ISDIR (s.st_mode))
     { ...este director... }
```

Asemenea constante există și pentru drepturile de acces și biții speciali.

- **int stat (char * nume, struct stat * s)**
 - încarcă în structura desemnată de **s** atributele i-nodului fișierului **nume**
 - dă eroare, dacă nu există fișierul **nume**
 - o problemă a acestui apel este că dacă **nume** este un fișier de tip link simbolic, atunci în **s** se trec atributele fișierului **sursă**

- **int Istat (char * nume, struct stat * s)**
 - funcționează la fel ca **stat**, dar dacă **nume** este un fișier de tip link simbolic, atunci în **s** sunt trecute atributele acestui fișier, nu ale fișierului sursă.

Exercițiu: Simulați comanda **ls -l**

Apeluri sistem pentru exploatarea fișierelor

Există două metode pentru exploatarea fișierelor în C sub UNIX: utilizând biblioteca standard C sau (și – mai rar) utilizând apeluri sistem de nivel jos.

Biblioteca sistem UNIX pentru exploatarea fișierelor

- **int open (char * nume , int mod , int dr)**
 - apelul returnează -1 în caz de eroare, iar în caz de succes întoarce indicele în tabela de descriptori;
 - **nume** reprezintă numele fișierului
 - **mod** reprezintă accesul la fișier și este construit prin disjuncție pe biți din constantele:
 - una și numai una din: **O_RDONLY**, **O_WRONLY**, **O_RDWR**
 - se pot alege optional și: - **O_CREAT** (dacă nu există fișierul nu dă eroare și îl creează)
 - **O_EXCR** (se folosește numai în prezența lui **O_CREAT**; dă eroare dacă fișierul deja există)
 - **O_TRUNC** (trunchiere de fișier, dacă fișierul există, conținutul este șters)
 - **O_APPEND** (toate scrierile se fac la sfârșit)
 - **O_SYNC** (scriere sincronizată, altfel se scrie întâi în cache)
 - **O_NONBLOCK**, **O_NDELAY** (eventualele operații de

citire/scriere care ar duce la un apel blocant, să nu fie operații blocante)

- **dr** reprezintă drepturile de acces și are efect doar când **open** creează fișier nou, dacă lipsește, se consideră drepturile implicate; este construit prin disjuncție pe biți din constante din structura **stat**

Observatie: Nu se poate anticipa că prin apelul **open** se va obține cel mai mic descriptor liber.

Observatie: **dr** poate lipsi, deci apelul poate avea 2 sau 3 parametrii

- **int close (int fd)**
 - **fd** este un descriptor de fișier valid (are același rol cu un file pointer)
 - apelul eșuează doar când **fd** este invalid
- **int read (int fd, void * p, int dim)**
 - **fd** este un descriptor de fișier valid
 - **p** este un pointer alocat unde se vor pune datele citite
 - **dim** reprezintă numărul maxim de octeți ce vor fi citiți
 - returnează -1 în caz de eroare sau numărul de octeți citiți
- **int write (int fd, void * p, int dim)**
 - scrie **dim** octeți de la adresa **p** în fișierul cu descriptoul **fd**
 - returnează numărul de octeți scriși efectiv

Un dezavantaj al bibliotecii sistem UNIX pentru exploatarea fișierelor este că nu avem la dispoziție scrieri/citiri formatare. De aceea există metode de trecere dintr-o bibliotecă în alta prin apelurile:

- **FILE * fdopen (int fd , char * mod)**
- **int fileno (FILE * fp)**

Duplicarea descriptorilor

- **int dup (int fd)**
 - **fd** este un descriptor valid
 - în caz de succes creează o nouă intrare în tabela de descriptori a procesului, care va pointa către aceeași intrare în tabela de descriptori a fișierelor deschise în memorie (prin urmare partajează poziția curentă)

Observatie: Duplicarea se face în cel mai mic descriptor liber.

Aplicație: Redirectarea fișierelor standard

ex.: < stdin | > stdout | >> stdout

\$ > titi	este echivalentă cu	<pre>int fd; fd = open("titi" , O_WRONLY O_CREAT O_TRUNC); close (1); dup (fd); close (fd);</pre>
-----------	---------------------	---

- **int lseek (int fd, int pozitie, int orig)** corespunde lui *fseek* din biblioteca standard C;
 - modificarea pozitiei curente este realizata fara operatii de citire, scriere;
 - **pozitie** poate fi una dintre: **SEEK_SET, SEEK_CUR, SEEK_END**;
 - **orig** reprezinta pozitia curenta (cu cati octeti se deplaseaza) in raport cu parametrul **pozitie**; **orig** poate fi si o valoare negativa;

Blocarea si deblocarea fisierelor

Ne amintim de tabelele de blocaj, care corespund fiecarei intrari din tabela de i-noduri.

Blocajele se pun cu ajutorul apelului:

- **int fcntl (int fd, int com, ...)**
 - apelul realizeaza operatiile de blocare si deblocare a fisierelor
 - **com:** **F_GETFD**
In acest caz apelul are doar doi parametrii si returneaza **int** (masca) care contine informatie pe biti despre fisiere
F_SETFD
In acest caz apelul are trei parametrii, ultimul fiind **int mask** si reprezinta aceleasi informatii pe biti despre fisiere
 - constanta **FD_CLOEXEC** (true/false) semnifica inchiderea (sau nu) a descriptorilor
ex.:

```
int attr;
attr = fcntl(fd, F_GETFD);           //citirea atributelor existente
attr |= FD_CLOEXEC;
fcntl(fd,F_SETFD,attr);
```
 - **com:** **F_GETFL**
F_SETFL
Obs.: Nu toate atributele pot fi modificate in runtime (O_APPEND, O_NONBLOCK, O_NDELAY, O_SIG)
 - **com:** (In aceste situatii apelul are trei parametrii, al treilea fiind de tip ***struct flock**, vedeti descrierea blocajelor si a acestei structuri de pe pagina 2)

F_SETLCK	:	pune un blocaj de tip consultativ
F_SETLKW	:	pune un blocaj de tip imperativ
F_GETLCK	:	in structura <i>flock</i> se testeaza existenta unui blocaj incompatibil cu blocajul nostru (in acest caz campurile structurii sunt complete cu informatiile blocajului existent → devine folositor campul <i>pid</i> ; altfel in <i>l_type</i> vom avea <i>F_UNLCK</i>)

Obs.: Pentru ca blocajele imperative sa actioneze efectiv trebuie ca fisierul pe care operam sa aiba indicatorul *set_gid* setat fara drept de executie.

Blocaje

Blocajele sunt proprietatea exclusiva a procesului care le pune. (*root* poate da *kill*, iar la terminarea procesului este ridicat blocajul).

Blocajele pot fi **exclusive** sau **partajate**.

ex.: daca cele doua blocaje din schema de mai jos sunt partajate, atunci ele pot coexista, daca sunt exclusive insa nu pot avea zona comună



Din punct de vedere al modului de actionare al blocajului, blocajele pot fi clasificate ca fiind:

- **consultative** (doar o incercare, nu duc la blocarea operatiilor de citire/scriere ale altor procese)
- **imperative** (duc la blocarea efectiva a zonelor de memorie)

Urmatoarea structura descrie blocajele sau tentativele de blocaj:

```
struct flock{
    short l_type;      /* Valorile pe care le poate lua sunt:
                        * F_RDLCK : blocaj partajat
                        * F_WRLCK : blocaj exclusiv
                        * F_UNLCK : ridicarea blocajului
                        */
    short l_mask;       /* Descrie pozitionarea blocajului
                        * (SEEK_SET,SEEK_CUR,SEEK_END)
                        */
    int l_start;        /* Descrie pozitia primului octet al zonei blocate relativ la l_mask */
    int l_len;          /* Descrie lungimea zonei blocate */
    int pid;            /* pid-ul procesului care a pus blocajul */
}
```

Operatii cu fisiere de tip director

Header-ul **dirent.h** defineste tipul **DIR**

- **DIR * opendir (char * nume)**
 - returneaza NULL daca fisierul *nume* nu exista sau exista, dar nu este de tip director
 - **struct dirent * readdir (DIR * d)**
 - utilizat pentru a citi intrarile dintr-un director : pentru fiecare intrare returneaza un pointer catre o structura *dirent*; dupa ce au fost parcurse toate intrarile, returneaza NULL;
- Obs.:** In struct *dirent* avem campul **char * d_name**, care reprezinta numele intrarii
- **void closedir (DIR * d)**
 - realizeaza inchiderea directorului
 - **void rewinddir (DIR * d)**
 - se revine la prima intrare...

COMUNICAREA INTRE PROCESE

(continuare)

Pipes (Fisiere de tip tub)

Sunt fisiere de tip special.

Se caracterizeaza prin:

- nu au descriptori de citire/scriere
 - citirea este destruktiva
 - citirea si scrierea sunt blocante (exceptie: O_NONBLOCK, O_NDELAY)

Pot fi de două tipuri: 1) cu nume
2) fără nume

1) Fisiere de tip tub cu nume

Acstea au cel putin o legatura fizica pe disc.
Din shell pot fi create folosind comanda:

\$ mkfifo nume

care creeaza un fisier cu numele *nume* de tipul .p (pipe)

Deschiderea, citirea, scrierea si inchiderea se realizeaza cu apelurile uzuale (open cu WRONLY, read, write, close).

Obs.: Deschiderile in scriere reusesc intotdeauna. Deschiderile in citire esueaza daca nu e deschis un descriptor in scriere, astfel observam ca pipe-urile reprezinta o modalitate de sincronizare a doua procese.

2) Fisiere de tip tub fara nume

c₁ | **c₂** | ... | **c_n** → n-1 pipe-uri fara nume

Acestea nu au nicio legatura fizica pe disc, exista doar ca i-noduri in memorie (deci la inchiderea tuturor descriptorilor va disparea si pipe-ul).

- **int pipe (int * p)**
 - **p** este un vector alocat de minim doi intregi
 - apelul creeaza un i-nod in memorie pentru un fisier de tip *pipe* si pune in **p[0]** descriptorul in citire si in **p[1]** descriptorul in scriere
 - pipe-ul este disponibil doar procesului creator si descendantilor sai

Exemplu:

ls -1a | wc -l

- numara fisierile al caror nume nu incepe cu ‘.’

```
int p[2];
pipe (p);
if (fork () > 0){
    close (1);
    dup (p[1]);
    close (p[0]);
    close(p[1]);
    execvp("ls","ls",-1a",NULL);
}
else{
    close(0);
    dup(p[0]);
    close(p[0]);
    close(p[1]);
    execvp("wc","wc",-l", NULL);
}
```

1. Facilitati multimedia

- extinderea perifericelor cu unele mai performante

Obs.: UNIX nu sist.... multimedia

2. Sistem multiprocessor

In sistem sunt mai multe procesoare care comunica printr-o magistrala cu memorie comuna. Sistemul de operare trebuie sa poata lucra cu mai multe procesoare.

Clasificare

- **arhitectura master-slave:** un procesor este master, iar celelalte n-1 procesoare sunt slave ; procesorul master distribuie pentru executie sarcinile celorlalte n-1 procesoare.
- **procesoare cu functii dedicate:** ex. : un anumit procesor este dedicat sa serveasca perifericele.

Lucrul cu reteaua

Transmisia de informatii intre calculatoarele din retea este intre masini eterogene.
Pentru proiectarea aplicatiilor in retea s-a introdus norma OSI care are 7 nivele :

7) nivelul Aplicatie

- pregateste informatia si o paseaza nivelului inferior

6) nivelul de Prezentare

- imbraca mesajul cu informatiile specifice
 - pune informatia intr-un format standard
- Obs. : protocolul cel mai cunoscut este XDR

5) Sesiune

4) Transport

- protocolele cele mai utilizate sunt TCP, UDP
- TCP : read/write pe octeti
- UDP : ca la cozi, R/W intregul mesaj
- primul nivel in care programatorii au acces (4 sau 3)

3) nivelul de Retea

- protocolul de retea este IP (Internet Protocol)
- vede destinatia, calculeaza drumul sursa-destinatie

2) nivelul de Legaturi de date

- protocolul este OHCP
- imbraca mesajul cu informatii de control : suma de control

1) nivelul Fizic

- transportul electronic de biti de la sursa la destinatie
- legaturi cu fir, wireless

Modul de realizare al acestor aplicatii :

RPC : remove procedure call
- apelarea unei proceduri pe o alta masina

in Java avem pachetul: Java.rmi → remote method interface

Arhitectura client-server

Pe o masina se executa o aplicatie server. Aplicatia server asteapta ca o aplicatie client sa se conecteze. Clientul emite cererea, serverul analizeaza cererea si transmite rezultatul.

In arhitectura client-server se utilizeaza pentru comunicare socket-uri.

In Java socket-ul este implementat astfel incat sa fie usor de folosit. In *java.net* sunt doua clase : ServerSocket si Socket.

Aplicatia server va face :

```
ServerSocket SS = new ServerSocket (int port) ;  
1 : Socket S = SS.accept(); /*metoda accept este blocanta ⇔ asteapta pana cand un client  
se va conecta la aplicatie*/  
// tratare cerere  
goto 1 ;
```

Aplicatia client :

```
Socket S = new Socket (String host, int port) ;  
// host reprezinta adresa unde se afla aplicatia server
```

Aplicatii clasice

DNS – face corespondenta intre adresa IP si nume

Telnet, SSH – se pot lansa terminale pe alte masini

FTP – File Transfer Protocol

SMTP – posta electronica

Baze de date relationale :ORACLE, SYBASE (MS SQL), DB2, INFORMIX, INGRES, GUPTA

Limbaj SQL : Standard Query Language

- are 4 instructiuni : select, update, insert, delete
- nu are facilitati procedurale

Aplicatii Web

Cozi de msaje

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>

int main() {

    struct msg{
        long mtype;
        char text[6];
        int x;
    };

    int key = ftok(".", 10);
    int qid = msgget(key, 0666|IPC_CREAT);

    int pid = fork();

    if(pid == 0) {
        struct msg send;
        send.mtype = 1;
        strcpy(send.text, "hello");
        send.x = 99;
        if(msgsnd(qid, &send, sizeof(send) - sizeof(long), 0)<0) {
            printf("Error child: ");
        }
    }
    else{
        struct msg recieve;
        if(msgrcv(qid, &recieve, sizeof(recieve) - sizeof(long), 1, 0)<0) {
            perror("Error parent: ");
        };
        printf("text: %s\nnumber: %d\nmtype:%ld", recieve.text, recieve.x,
        recieve.mtype);
    }

    return 0;
}
```

```

Procese:
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
int main(){

    int pid = fork();

    if(pid < 0) {
        return 1;
    }

    if(pid == 0){
        int i = 1;
        while(i<=2) {
            sleep(1);
            printf("PROCES 1\n");
            i++;
        }
    }
    if(pid > 0){

        int pid2 = fork();

        if(pid2 == 0){
            int i = 1;
            while(i<3){
                sleep(1);
                printf("PROCES 2\n");
                i++;
            }
        }

        if(pid2 > 0){
            int status;
            int pid;
            while(-1 != waitpid(-1, NULL, 0)){
                printf("S-a temrinat un proces!\n");
            }
        }
    }

    return 0;
}

```

Fisiere binare

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <stdlib.h>

void myFunct(char *fis, int m, int *n) {
    int f;
    if((f = open(fis, O_RDONLY)) < 0) {
        *n = -1;
        return;
    }

    if(lseek(f, m, SEEK_SET) + *n - 1 >= lseek(f, 0, SEEK_END)) {
        *n= -1;
        return;
    }

    if(lseek(f, m, SEEK_SET) == -1 ) {
        *n = -1;
        return;
    }

    for(int i=0; i<*n; i++) {
        char x;
        read(f, &x, 1);
        printf("%c\n", x);
    }
    *n=0;
}

int main() {
    int n = 2;
    myFunct("abc", 4, &n);
//    int f;
//    int buffer;
//
//    if((f = open("abc",O_WRONLY|O_CREAT|O_TRUNC)) < 0) {
//        return -1;
//    }
//
//    int* x = (int*)malloc(100*sizeof(int));
//    for(int i=0; i<100; i++) {
```

```

//      x[i] = i;
//    }
//
//    write(f, x, sizeof(int)*100);
//
//    close(f);
//
//    if((f = open("abc", O_RDONLY)) < 0) {
//        return -1;
//    }
//
//    int* v = (int*)malloc(100*sizeof(int));
//
//    read(f, v, sizeof(int)*100);
//
//    for(int i = 0; i<100; i++){
//        printf("%d ", v[i]);
//    }
//
//    close(f);
//
//    printf("\n");
}

return 0;
}

```

Semafoare

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void semcall(int sid,int op){
    struct sembuf buf;
    buf.sem_num=0;
    buf.sem_op=op;
    buf.sem_flg=0;
    if(semop(sid,&buf,1)<0) printf("eroare");
}

```

```

void p(int sid)
{semcall(sid,-1);}

void v(int sid)
{semcall(sid,1);}

int main(){
    int semid;
    semid = semget(1999,1,0660|IPC_CREAT);

    v(semid); //linia de sters
    int pid1, pid2;

    pid1 = fork();

    if(pid1 == 0){
        printf("inainte de zona critica process 1\n");
        p(semid);
        printf("in zona critica proces 1\n");
        sleep(3);
        printf("gata zona critica proces 1\n");
        v(semid);
        printf("dupa zona critica process 1\n");
    }
    else{
        pid2 = fork();
        if(pid2 == 0){
            printf("inainte de zona critica process 2\n");
            p(semid);
            printf("in zona critica proces 2\n");
            sleep(3);
            printf("gata zona critica proces 2\n");
            v(semid);
            printf("dupa zona critica process 2\n");
        }
        else{
            int status;
            while(wait(&status)!=-1);
            semctl(semid, 0, IPC_RMID, 0);
        }
    }
    return 0;
}

```

Semnale

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>

void handler1(int sig){
    if(sig == SIGINT){
        printf("\nAm primit SIGINT!\n");
        exit(0);
    }
}

void handler2(int sig){
    printf("Apasa ctr+c. Nu va merge. Dupa 3 secunde semnalul SIGINT nu va
mai fi blocat.\n");
    sleep(3);
    //printf("%d:Am primit semnalul %d\n", getpid(), sig);
}

int main(){
    if (signal(SIGSTOP, handler1) == SIG_ERR) printf("\ncan't catch
SIGSTOP\n");
    //signal(SIGINT, handler1);
    //signal(SIGUSR1, handler2);

    struct sigaction act;
    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    act.sa_mask = set;
    act.sa_handler = handler2;
    act.sa_flags = 0;
    //act.sa_flags = SA_RESETHAND;

    struct sigaction act2;
    sigset_t set2;
    sigemptyset(&set2);
    act2.sa_handler = handler1;
    act2.sa_mask = set2;
    act2.sa_flags = 0;

    sigaction(SIGINT, &act2, NULL);
```

```

sigaction(SIGUSR1, &act, NULL);

int pid = fork();

if(pid > 0){
    int status;
    pause();
}
else{
    printf("Fiul %d trimite semnal la parinte!\n", getpid());
    kill(getppid(), SIGUSR1);
}

return 0;
}

```

Shared memory

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main(){
    int key = ftok(".", 10);
    int id = shmget(key, 100, IPC_CREAT|0666);
    char* shm;
    if((shm = shmat(id, NULL, 0)) == (void*) -1){
        printf("Eroare");
        exit(1);
    }

    int pid = fork();

    if(pid > 0){
        char s[50] = "Ma auzi?";
        strcpy(shm, s);
        printf("PARINTE: mesaj trimis in shm - %s\n", s);
        int status;
        wait(&status);
        printf("PARINTE: mesaj primit - %s\n", shm+1);
    }
}

```

```
else{
    char recieve[100];
    int i=0;
    while(shm[i]!='\0') {
        recieve[i] = shm[i];
        i++;
    }
    recieve[i] = '\0';
    printf("FIU: mesaj primit - %s\n", recieve);
    strcpy(shm+1, "Da, te aud!");
    printf("FIU: mesaj trimis in shm - Da, te aud!\n");
}

}
```

F18. (2 puncte) Scrieti un program care sorteaza caracterele dintr-un fisier
al carui specificator este dat ca argument in linia de comanda. Nu se vor folosi fisiere auxiliare.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int f[256];

int main(int argc, char** argv)
{
    if(argc<2)
    {
        printf("Argumente insuficiente\n");
        return 0;
    }
    char *nume;
    nume=malloc(sizeof(char) * strlen(argv[1]));
    strcpy(nume, argv[1]);
    FILE *fp;
    fp=fopen(nume, "r+");
    if(fp==NULL)
    {
        printf("Eroare la deschiderea fisierului\n");
        return 0;
    }
    int size;
    fseek(fp, 0, SEEK_END);
    size=f.tell(fp);
    rewind(fp);
    for(int i=0; i<size; i++)
    {
        char aux;
        aux=fgetc(fp);
        if(aux!='\n')
            f[aux]++;
    }
    fclose(fp);
    fp=fopen(nume, "w");
    for(int i=0; i<256; i++)
    {
        while(f[i]!=0)
```

```

    {
        fputc(i, fp);
        f[i]--;
    }
}
fclose(fp);
return 0;
}

```

A2. (3 puncte) Scrieti un program "sortint" care se va lansa sub forma:

sortint f

unde f este un fisier continand reprezentari interne de intregi si sorteaza intregii din acest fisier. Se vor folosi doar functiile de la nivelul inferior de prelucrare a fisierelor si nu se vor folosi fisiere auxiliare.

```

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

//Verificarea corectitudinii argumentelor
int test_args(int argc, char *argv[])
{
    int fd;
    if(argc != 2)
        return -1;

    if((fd = open(argv[1], O_RDWR)) < 0)
    {
        perror(argv[1]);
        exit(1);
    }
    close(fd);
    return 0;
}

```

```

//Baga un intreg (el) intr-un vector (buf)
void add_to_int_buf(int **buf, int *buf_len, int el)
{
    int i;
    int *new_buf = malloc( (sizeof(int) * (*buf_len)) + 1);
    for(i=0; i<*buf_len; i++)
    {
        new_buf[i] = (*buf)[i];
    }
    new_buf[*buf_len] = el;
    free(*buf);
    *buf = new_buf;
    *buf_len = *buf_len + 1;
}

//Citeste un intreg char cu char si-l compune intr-un int.
int read_an_int(int fd)
{
    char buf;
    int bytes_rd;
    int sgn = 1;
    int result = 0;
    char prev_ch = 0;
    while((bytes_rd = read(fd, &buf, 1)) >= 0)
    {
        if(buf == '-')
        {
            sgn = -1;
        }
        if(buf <= '9' && buf >= '0')
        {
            result = (result * 10) + (buf - 48);
        }
        if((buf == ' ' || buf == '\n' || buf == 10 ||
            bytes_rd == 0) && (prev_ch >= '0' && prev_ch <= '9'))
        {
            return result * sgn;
        }
        if(buf == 0 || bytes_rd == 0)
        {
            errno = 127;
            return -1;
        }
        prev_ch = buf;
    }
}

```

```

}

//"Comparitorul" pentru functia qsort (folosita in main).
int compare(const void *a, const void *b)
{
    return *((int *) a) - *((int *) b);
}

//Scrie un array intr-un fisier.
void write_int_array_to_file(int *buf, int buf_len, int fd)
{
    off_t seek_st;
    if( (seek_st = lseek(fd, 0, SEEK_SET)) < 0 )
    {
        perror("error");
        exit(1);
    }
    else
    {
        char *num;
        int i;
        num = malloc(sizeof(char) * 10);
        for(i=0; i<buf_len; i++)
        {
            sprintf(num, "%d", buf[i]);
            write(fd, num, strlen(num));
            if(i < buf_len-1)
                write(fd, " ", 1);
        }
        free(num);
    }
}

int main(int argc, char *argv[])
{
    if(test_args(argc, argv) < 0)
    {
        printf("Usage: %s file\n", argv[0]);
        return 0;
    }
    else
    {
        int fd = open(argv[1], O_RDWR);
        int an_int;
        int *buf = NULL;

```

```

        int buf_len = 0;
        while( ((an_int = read_an_int(fd)) != -1) || errno != 127)
        {
            add_to_int_buf(&buf, &buf_len, an_int);
        }
        qsort(buf, buf_len, sizeof(int), compare);
        write_int_array_to_file(buf, buf_len, fd);
    }
    return 0;
}

A6. (2 puncte) Scrieti un program care primeste ca argument in linia de
comanda un fisier si inlocuieste in el orice succesiune de caractere albe
(blank, tab, cap de linie) cu un singur blank. Se vor folosi doar functii
de
    la nivelul inferior de prelucrare a fisierelor, fara lseek si nu se vor
folosi
    fisiere auxiliare. Indicatie: fisierul va fi accesat prin doi
descriptori,
    unul in citire, altul in scriere.

```

```

#include <stdio.h>
#include <stdlib.h>

int is_space(char t){
    if(t==' ' || t=='\n' || t=='\t' || t=='\n' || t=='\v' || t=='\r' ||
t=='\f')
        return 1;
    return 0;
}

int main(int argc, char const *argv[])
{
    FILE *fin, *fout;
    char c;
    int count=0;

    if(argc!=2){
        printf("%s\n", "One argument required");
        exit(0);
    }

    if( (fin = fopen(argv[1], "r")) == NULL || (fout = fopen(argv[1], "r+"))
== NULL ){
        perror("Problem opening file, exiting...\n");
        exit(1);
    }
}

```

```

}

rewind(fout);

while( ( c = (char)fgetc(fin) ) != EOF ){
    if(is_space(c)){
        count++;
        if( count == 1 )
            fprintf(fout, "%c", ' ');
    }
    else{
        count = 0;
        fprintf(fout, "%c", c);
    }
}

if (ftruncate(fileno(fout), ftell(fout) ) != 0){
    printf("%s\n", "I couldn't truncate!");
    exit(2);
}

fclose(fin);
fclose(fout);
return 0;

```

A7. (2 puncte) Scrieti niste programe care arata daca fii obtinuti cu fork ai unui proces "p" mostenesc redirectarile intrari/iesirii standard care au

loc daca lansam "p" in cadrul unui filtru:
 ... | p | ...

```

#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <fcntl.h>

int main()
{
    char str[100];

    int fin;
    fin = open("date.in", O_RDONLY);
    close(0);
    dup(fin);
  
```

```

int fout;
fout = open("date.out", O_WRONLY | O_CREAT | O_TRUNC, 644);
close(1);
dup(fout);

pid_t pid;

pid = fork();
if(pid == -1)
{
    perror("fork");
    exit(1);
}
if (pid == 0)
//ChildProcess
{
    fgets(str,100,stdin);
    printf("procesul copil a primit: %s\n",str);
    return 0;
}
else
//ParentProcess()
{
    fgets(str,100,stdin);
    printf("procesul tata a primit: %s\n",str);
    fflush(stdin);
    wait(&pid);
    return 0;
}
}

```

A8. (1.5 puncte) Scrieti un program care genereaza un fiu cu fork iar acesta trimite tatalui un numar aleator de semnale SIGUSR1; fiul numara cate a trimis, tatal numara cate a primit, apoi fiecare afisaza numarul respectiv.

Se va asigura protectia la pierderea unor semnale.

```

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

int main(int argc, char **argv)

```

```

{
/*FILE *fp;
f=fopen("fisier.txt", "w+");
if(argc<2)
{
    printf("Argumente insuficiente\n");
    return 0;
}
char *nume;
nume=malloc(sizeof(char) * strlen(argv[1]));
strcpy(nume, argv[1]);
FILE *fp;
fp=fopen(nume, "r+");
if(fp==NULL)
{
    printf("Eroare la deschiderea fisierului\n");
    return 0;
}
int i;
for(i=0; i<=133; i++)
{
    errno=i;
    if(i!=41 && i!=58)
        fprintf(fp, "%s %d \n", strerror(errno), errno);
}
return 0;
}

```

A9. (2 puncte) Scrieti un program care arata ce se intampla daca mai multe procese care se ruleaza in paralel in foreground incearca in acelasi timp sa citeasca de la tastatura un caracter (fiecare il va citi, sau doar unul aleator il va citi, etc.); procesele respective vor fi fii cu fork ai programului initial. Nu se vor folosi decat functii de la nivelui inferior de prelucrare a fisierelor.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
```

```

int main(int argc, char **argv)
{
    const int N = 10;
    int i, number = 0;

    for (i = 0; i < N; ++i)
    {
        pid_t pid = fork();
        if (pid < 0)
        {
            perror("Fork failed:");
            exit(EXIT_FAILURE);
        }
        else if (pid == 0)
        {
            scanf("%d", &number);
            printf("Your number from child %d is: %d\n", i + 1, number);
            exit(EXIT_SUCCESS);
        }
    }

    pid_t p;
    while ((p = wait(NULL)) > 0);

    exit(EXIT_SUCCESS);
}

```

A10. (6 puncte) Scrieti un program care sa sorteze prin interclasare un fisier de caractere in maniera descrisa mai jos.

Sortarea prin interclasare presupune impartirea sirului in doua jumatati, sortarea fiecarei parti prin aceeasi metoda (deci recursiv), apoi interclasarea celor doua parti (care sunt acum sortate).

Programul va lucra astfel: se imparte sirul in doua, se genereaza doua procese fiu (fork) care sorteaza cele doua jumatati in paralel, tatal ii asteapta sa se termine (wait sau waitpid), apoi interclaseaza jumatatile. Nu se vor folosi fisiere auxiliare.

```

#include <sys/ipc.h>
#include <sys/types.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/syscall.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <unistd.h>

#include <stdlib.h>

```

```

#include <stdio.h>
#include <string.h>
#include <errno.h>

int test_args(int argc, char *argv[])
{
    int fd;
    if(argc != 2)
        return -1;

    if((fd = open(argv[1], O_RDWR)) < 0)
    {
        perror(argv[1]);
        exit(1);
    }
    close(fd);
    return 0;
}

int get_file_contents(int fd, int *fc_len)
{
    char *str;
    struct stat file_prop;
    int shmid;

    if(fstat(fd, &file_prop) < 0)
    {
        perror("");
        exit(1);
    }
    else
    {
        *fc_len = file_prop.st_size;

        //Aloc memorie partajata doar intre toate procesele ce se
        //forkuesc din procesul initial (IPC_PRIVATE) de marime *fc_len
        //(marimea fisierului).

        //Pentru info suplimentare vedeti "man shmget"
        if((shmid = shmget(IPC_PRIVATE, *fc_len, IPC_CREAT | 0666)) < 0)
        {
            perror("shmget");
            exit(1);
        }
        else

```

```

{
    str = shmat(shmid, NULL, 0); //Montez memoria partajata
}

int i;
char buf;
for(i=0; i<*fc_len; i++)
{
    if(read(fd, &buf, 1) < 0)
    {
        perror("");
        shmdt(str); //Demontez memoria partajata
        shmctl(shmid, IPC_RMID, NULL); //Dezaloc (eliberez)
memoria partajata
        exit(1);
    }
    else
    {
        str[i] = buf;
    }
}
shmdt(str);
return shmid;
}

void merge(int shmid, int start1, int end1, int start2, int end2)
{
    int i, j, temp_count=0;
    int temp_size = sizeof(char) * (end1-start1) + (end2-start2) + 2;
    char *temp = (char *) malloc(temp_size);
    char *str;
    i = start1;
    j = start2;

    str = shmat(shmid, NULL, 0);

    while(i <= end1 || j <= end2)
    {
        if(i <= end1 && j <= end2)
        {
            if(str[i] <= str[j])
            {
                temp[temp_count++] = str[i];
                i++;
            }
        }
    }
}
```

```

        }
    else
    {
        temp[temp_count++] = str[j];
        j++;
    }
}
else
{
    if(i <= end1)
    {
        temp[temp_count++] = str[i];
        i++;
    }
    else
    {
        temp[temp_count++] = str[j];
        j++;
    }
}
}

for(i=0; i<temp_count; i++)
{
    str[start1+i] = temp[i];
}
free(temp);
shmdt(str);
exit(0);
}

void merge_sort(int shmid, int left, int right)
{
    if(right - left > 0)
    {
        int m = (left+right) / 2;

        if(!fork())
            merge_sort(shmid, left, m);

        if(!fork())
            merge_sort(shmid, m+1, right);

        wait(NULL);
        wait(NULL);
    }
}

```

```

        merge(shmid, left, m, m+1, right);
        exit(0);
    }
}

int main(int argc, char *argv[])
{
    if(test_args(argc, argv) == -1)
    {
        printf("Usage: A10 file\n");
        return 0;
    }

    int fd = open(argv[1], O_RDWR);
    char *file_contents;
    int fc_len;
    int shmid;
    int i;

    shmid = get_file_contents(fd, &fc_len);

    file_contents = shmat(shmid, NULL, 0);
    printf("Unsorted:");
    for(i=0; i<fc_len; i++)
    {
        if(!(i % 16))
            printf("\n");
        printf("%02X ", file_contents[i]);
    }
    printf("\n");
    shmdt(file_contents);

    if(!fork())
        merge_sort(shmid, 0, fc_len-1);
    wait(NULL);

    file_contents = shmat(shmid, NULL, 0);
    printf("\nSorted:");
    for(i=0; i<fc_len; i++)
    {
        if(!(i % 16))
            printf("\n");
        printf("%02X ", file_contents[i]);
    }
}

```

```

        printf("\nTotal size: %d\n", fc_len);
        shmctl(shmid, IPC_RMID, NULL);
        shmdt(file_contents);

        return 0;
}

```

A16. (1 punct) Scrieti un program care afisaza valoarea variabilei de environment TERM, apoi o asigneaza la valoarea "vt52", apoi genereaza un proces fiu (cu fork) care afisaza valoarea lui TERM mostenita.

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    const char *name = "TERM";
    const char *value = "vt52";

    printf("%s\n", getenv(name));
    setenv(name, value, 1);

    pid_t pid = fork();
    if (pid < 0)
    {
        perror("Fork failed:");
        exit(EXIT_FAILURE);
    }
    if (pid == 0)
        printf("%s\n", getenv(name));

    exit(EXIT_SUCCESS);
}

```

A17. (1 punct) Scrieti un program care sa verifice daca schimbandu-si valoarea variabilei de environment PWD, functia getcwd returneaza acelasi director curent.

```

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
    char cwd[256], nwd[256], t[256], var[256];

```

```

getcwd(cwd, sizeof(cwd));
printf("Valoare returnata de getcwd: %s\n", cwd);
printf("PWD=%s\n", (char *)getenv("PWD"));
printf("Valoarea cu care se inlocuieste PWD: ");
fgets(t, 255, stdin); //gets(t);
strcpy(var, "PWD=");
strcat(var, t);
putenv(var);
printf("PWD=%s\n", (char *)getenv("PWD"));
getcwd(nwd, sizeof(nwd));
printf("Valoare getcwd dupa modificarea PWD: %s\n", nwd);
if(strcmp(nwd, cwd) == 0) printf("Valoarea returnata nu se
modifica\n");
else printf("Valoarea returnata se modifica\n");
return 0;
}

```

A21. (4 puncte) Scrieti un program "filtru" care se lanseaza sub forma:

filtru f1 ... fn
 unde f1, ..., fn sunt fisiere executabile de pe disc si lanseaza procesul:
 f1 | ... | fn

```

#include <stdio.h>
#include <stdlib.h>

#include <signal.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <unistd.h>

int ORIGINAL_STDIN;

void start_forking(int argc, char *argv[], char *env[], int carg)
{
    int pipe_fd[2];
    pid_t child;
    char *newargv[] = {NULL, NULL};

    if(carg > 0)
    {
        if(pipe(pipe_fd) < 0)
        {
            perror("pipe");
            exit(EXIT_FAILURE);
        }
    }

```

```

    if(dup2(pipe_fd[0], STDIN_FILENO) < 0)
    {
        perror("dup2");
        close(pipe_fd[0]);
        close(pipe_fd[1]);
        exit(EXIT_FAILURE);
    }

    child = fork();

    if(child < 0)
    {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if(child == 0)
    {
        if(dup2(pipe_fd[1], STDOUT_FILENO) < 0)
        {
            perror("dup2");
            close(pipe_fd[0]);
            close(pipe_fd[1]);
            exit(EXIT_FAILURE);
        }

        start_forking(argc, argv, env, carg-1);
    }
    close(pipe_fd[1]);
    wait(NULL);
}

else
{
    if(dup2(ORIGINAL_STDIN, STDIN_FILENO) < 0)
    {
        perror("dup2");
        exit(EXIT_FAILURE);
    }
}

newargv[0] = argv[carg];
execve(argv[carg], newargv, env);
perror("execve");
}

```

```

int main(int argc, char *argv[], char *env[])
{
    if(argc < 2)
    {
        printf("Usage: %s f1 .. fn\n", argv[0]);
        exit(EXIT_SUCCESS);
    }

    ORIGINAL_STDIN = dup(STDIN_FILENO);
    start_forking(argc, argv, env, argc-1);
    return EXIT_FAILURE;
}

```

A22. (4 puncte) Implementati tipul arbore binar cu varfuri numere intregi alocat in lantuit (cu pointeri). Scripti o functie care primeste ca parametru un arbore (pointer la radacina sa) si afisaza varfurile sale parcurgandu-l pe nivele. In general algoritmul de parcurgere pe nivele foloseste o coada in care initial se incarca radacina, apoi intr-un ciclu care tine cat timp coada este nevida se extrage un varf, se afisaza, apoi se introduc in coada fii lui (pointeri la radacinile lor). Pe post de coada functia va folosi un fisier tub fara nume (care va exista doar pe perioada apelului). Scripti un program ilustrativ pentru aceasta functie.

```

#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

const long _NULL = 0;

struct node
{
    int info;
    struct node *left;
    struct node *right;
};

void insert(struct node **root, int key)
{
    if( (*root) == NULL)
    {
        *root = (struct node *) malloc(sizeof(struct node));
        (*root)->info = key;
        (*root)->left = NULL;
        (*root)->right = NULL;
    }
    else

```

```

{
    if( (*root)->info < key)
    {
        insert( &((*root)->left), key);
    }
    else
    {
        insert( &((*root)->right), key);
    }
}

void print(int pipe_fd[2], int level)
{
    struct node *current;
    size_t nread;
    int nodes_found = 0;

    while( (nread = read(pipe_fd[0], &current, sizeof(struct node *))) )
    {
        nodes_found++;
        if(nread < 0)
            goto READFAILURE;

        if(current)
        {
            if(nodes_found == 1)
                printf("\nLevel %d: %d ", level, current->info);
            else
                printf("%d ", current->info);

            if(current->left)
                if(write(pipe_fd[1], &current->left, sizeof(struct node *)) < 0)
                    goto WRITEFAILURE;

            if(current->right)
                if(write(pipe_fd[1], &current->right, sizeof(struct node *)) < 0)
                    goto WRITEFAILURE;
        }
        else
        {
            write(pipe_fd[1], &_NULL, sizeof(struct node *));
            if(nodes_found != 1)

```

```

                print(pipe_fd, level+1);
                break;
            }
        }

    return;

WRITEFAILURE:
    perror("write");
    goto FAILURE;

READFAILURE:
    perror("read");
    goto FAILURE;

FAILURE:
    close(pipe_fd[0]);
    close(pipe_fd[1]);
    exit(EXIT_FAILURE);

}

int main(int argc, char *argv[])
{
    int pipe_fd[2];
    if( pipe(pipe_fd) < 0)
    {
        perror("pipe");
        return EXIT_FAILURE;
    }

    struct node *root = NULL;

    insert(&root, 30);
    insert(&root, 20);
    insert(&root, 40);
    insert(&root, 45);
    insert(&root, 15);

    if( write(pipe_fd[1], &root, sizeof(struct node *)) < 0)
        goto WRITEFAILURE;
    if( write(pipe_fd[1], &_NULL, sizeof(struct node *)) < 0)
        goto WRITEFAILURE;

    print(pipe_fd, 0);
}

```

```

printf("\n\n");

close(pipe_fd[0]);
close(pipe_fd[1]);

return EXIT_SUCCESS;

WRITEFAILURE:
perror("write");
close(pipe_fd[0]);
close(pipe_fd[1]);
return EXIT_FAILURE;
}

```

D1. (4 puncte) Program care primeste ca argument in linia de comanda un director si afisaza arborescenta de directoare si fisiere cu originea in el (similar comenzii tree /f din DOS).

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <dirent.h>
void parcurgere(char *desc, int nivel)
{
    DIR *dir_actual = NULL;
    if( (dir_actual = opendir(desc)) == NULL)
        if(!(errno == ENOTDIR || errno == ENOENT || errno == EACCES)){
            perror(desc); exit(1);
        }
    else;
    else
    {
        struct dirent *data_dir_actual = NULL;
        while( (data_dir_actual = readdir(dir_actual)) != NULL)
        {
            if(strcmp(data_dir_actual->d_name, ".") == 0 ||
            strcmp(data_dir_actual->d_name, "..") == 0)
                continue;

            int i;
            for(i=0; i<nivel; i++)
                printf("\x1B[31m|  ");
            if(data_dir_actual->d_type == DT_DIR)
                printf("\x1B[34m%s\x1B[37m", data_dir_actual->d_name);
            else

```

```

        printf("\x1B[32m%s\x1B[37m", data_dir_actual->d_name);
        int dim_dir = strlen(data_dir_actual->d_name);
        char *desc_nou = malloc(sizeof(char) * (strlen(desc) + dim_dir
+ 2));

        strcpy(desc_nou, desc);
        if(strcmp(desc, "/"))
            strcat(desc_nou, "/");
        strcat(desc_nou, data_dir_actual->d_name);

        parcurgere(desc_nou, nivel+1);
        free(desc_nou);
    }
    closedir(dir_actual);
}
}

int main(int argc, char *argv[])
{
    if(argc != 2) { printf("Argument invalid\n"); return -1; }
    char nume[256];
    strcpy(nume, argv[1]);
    parcurgere(nume, 0);
    return 0;
}

```

D5. (4 puncte) Program care primește ca argument în linia de comandă un director și sterge (recursiv) toata arborescentă cu originea în el.

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <dirent.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>

void delete_dir(char *path)
{
    DIR *dir;
    struct dirent *d;

    if ((dir = opendir(path)) == NULL)
    {
        int e = errno;

```

```

        printf("Cannot open %s: %s\n", path, strerror(e));
        return;
    }

    while ((d = readdir(dir)) != NULL)
    {
        if (!strcmp(d->d_name, ".") || !strcmp(d->d_name, ".."))
            continue;

        char *new_path;
        new_path = (char *)malloc(strlen(path) + strlen(d->d_name) + 2);
        strcpy(new_path, path);
        strcat(new_path, d->d_name);

        if (d->d_type == DT_DIR)
        {
            strcat(new_path, "/");
            delete_dir(new_path);
        }
        else
        {
            int r = unlink(new_path);
            if (r == -1)
            {
                int e = errno;
                printf("Cannot delete %s: %s\n", new_path, strerror(e));
            }
        }

        free(new_path);
    }

    closedir(dir);
    int r = rmdir(path);
    if (r == -1)
    {
        int e = errno;
        printf("Cannot delete %s: %s\n", path, strerror(e));
    }
}

int main(int argc, char **argv)
{
    if (argc < 2)
    {

```

```

    printf("Please specify a directory!\n");
    exit(EXIT_FAILURE);
}
char *path = (char *)malloc(strlen(argv[1]) + 2);
strcpy(path, argv[1]);
if (path[strlen(path) - 1] != '/')
    strcat(path, "/");
delete_dir(path);

exit(EXIT_SUCCESS);
}

```

D8. (2 puncte) Program care tipareste pe stderr toate mesajele de eroare ce se pot obtine cu functia " perror", precedate de valoarea corespunzatoare a variabilei "errno". Cu ajutorul lui sa se obtina un fisier cu aceste mesaje.

```

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

int main(int argc, char **argv)
{
    /*FILE *fp;
    f=fopen("fisier.txt", "w+");*/
    if(argc<2)
    {
        printf("Argumente insuficiente\n");
        return 0;
    }
    char *nume;
    nume=malloc(sizeof(char) * strlen(argv[1]));
    strcpy(nume, argv[1]);
    FILE *fp;
    fp=fopen(nume, "r+");
    if(fp==NULL)
    {
        printf("Eroare la deschiderea fisierului\n");
        return 0;
    }
    int i;
    for(i=0; i<=133; i++)
    {
        errno=i;

```

```
    if(i!=41 && i!=58)
        fprintf(fp, "%s %d \n", strerror(errno), errno);
}
return 0;
}
```

2015

23.01.2015

Examen sisteme de operare.

I. Modelul de comunicare al proceselor în arhitectura multiproceselor.

II. În ce condiții se este zdroi un fisier în UNIX?

III. Ce ar trebui să scrieți

rw-rw---- aranja unu fisier UNIX?

IV. Numiști deoarece tipuri de interfețe cu utilizatorul.

V. Scrieți o funcție C

void myfunct(void *f) (int rigi)

care instalață pe f ca handle pentru toate menajele posibile.

VI. Scrieți o funcție C

int myfunct (char *fis, char *s)

care îl întoarce numărul de apariții ale lui s în fisierul cu numele de fis. În caz de eroare întoarce -1.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int search(char *filename, char *str) {
    FILE * pFile;
    pFile = fopen(filename, "r");
    if (pFile == NULL) {
        return -1;
    }
    char line[256];
    int strLen = strlen(str), sol = 0;
    while (fgets(line, 256, pFile) != NULL) {
        char * ptr = line;
        while ((ptr = strstr(ptr, str)) != NULL) {
            sol += 1;
            ptr += strlen(str);
        }
    }
}
```

```

    }
    fclose(pFile);

    return sol;
}

int main() {
    int result = search("alex.txt", "bla");
    printf("%d", result);
    return 0;
}

2)

#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void handler(int signum) {
    char s[256];
    sprintf(s, "Am primit semnal %d\n", signum);
    write(1, s, sizeof(s));
}

int main() {
    int i;
    for (i=0; i<NSIG; i++) {
        signal(i, handler);
    }
    while (1) {
        sleep(1);
    }
    return 0;
}

```

2014

grupa 241 (2014)

1. Dacă o pagină are 512 octeți, câte pagini sunt în 1MB?
2. Care sunt politicile de înlocuire ale paginilor?
3. Ce este fragmentarea discului?
4. Care sunt părțile componente ale unui hard disk?
5. Scrieți o funcție în C `int myFunct(char *dir)` care returnează suma mărimilor fișierelor din acel director.

6. Scrieți o funcție în C `int myFunct(char *fis)` care citește acel fișier și scrie un număr de octeți astfel încât mărimea aceluui fișier modulo 256 să fie 0.

```
#include <unistd.h>
#include <stdio.h>
#include <dirent.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <errno.h>

int myFunct(char *dir) {
    int sz = 0;
    DIR *foo = opendir(dir);
    if (foo == NULL) {
        return -1;
    }
    struct dirent *ent;
    while ((ent = readdir(foo)) != NULL) {
        ent = readdir(foo);
        struct stat st;
        errno = 0;
        int exists = stat(ent->d_name, &st);
        if (exists < 0) {
            perror("plm");
            printf("%s\n", ent->d_name);
        } else {
            if (S_ISREG(st.st_mode)) {
                sz += st.st_size;
            }
        }
    }
    printf("%d\n", sz);
    return 0;
}

int main() {
    char *myDir = ".";
    return myFunct(myDir);
}
```

Sisteme de operare 21.01.2012 grupa 234

- I. Signatura apelului sistem **fork**
- II. Numiti cele trei tipuri de IPC SYSTEM V
- III. Numiti algoritmul folosit pentru planificarea proceselor folosit de sistemele de operare din familia UNIX
- IV. Care este teoria matematica ce sta la baza proiectarii algoritmilor de detectare a deadlock-urilor?
- V. Scrieți o funcție C

int *myFunct(char **name, int n, int mod)

care deschide cate un fisier cu numele dat de fiecare element din **nume** (care are **n** elemente), cu caracteristicile date de **mod** și drepturi de acces implicate, și întoarce un tabel de **n** elemente continand descriptorii fisierelor deschise. În caz de eroare se va întoarce NULL.

- VI. Scrieți o funcție C

int myFunct(int semid, (void *)f())

care apelează funcția data de **f** în mod semaforizat de semaforul de tip mutex cu indicatorul intern dat de **semid**.

În caz de eroare se întoarce -1 și 0 în caz de succes.

1. Signatura apelului sistem fork

```
#include <unistd.h>
pid_t fork(void);
//pag 81
```

2. Numiti cele trei tipuri de IPC SYSTEM V

1. Cozi de mesaje 1. Message Queues

2. Semafoare 2. Semaphore Sets
3. Memorie Partajata 3. Shared Memory Management
//pag 102

3. Numiti algoritmul folosit pentru planificarea proceselor folosit de sistemele de operare din familia UNIX

SCHEDULING ALGORITHM: Planificarea Round-Robin (Round-Robin scheduling)
//curs Draguluici soc7

4. Care este teoria matematica ce sta la baza proiectarii algoritmilor de detectarea deadlock-urilor?

GRAFURILE!! Teoria transmiterii informației, de detectare și corectare a erorilor sau "teorema a seriabilității".

O.S. runs a deadlock detection algorithm to detect the circular wait condition present for a set of processes. It uses a marking method to mark all processes that are not currently deadlocked.

5. Scrieti o functie C int *myFunct(char **name, int n, int mod) care deschide cate un fisier cu numele dat de fiecare element din nume (care are n elemente), cu caracteristicile date de mod si drepturi de acces implice, si intoarce un tabel de n elemente continand descriptorii fisierelor deschise. In caz de eroare se va intoarce NULL.

```
int *myFunct(char **name, int n, int mod){  
    int i, *Desc=(int*)malloc(n*sizeof(int));  
    for(i=0;i<n;i++) {  
        if((Desc[i]=open(name[i], mod))==-1) {  
            perror("open");  
            return NULL;  
        }  
    }  
    return Desc;  
}
```

6. Scrieti o functie C int myFunct(int semid, (void *)f()) care apeleaza functia data de f in mod semaforizat de semaforul de tip mutex cu indicatorul intern dat de semid. In caz de eroare se intoarce -1 si 0 in caz de succes.

```
// via Stefan Postăvaru :)  
#include <stdio.h>  
#include <sys/types.h>  
#include <sys/ipc.h>
```

```

#include <sys/sem.h>
#include <errno.h>
#include <stdio.h>
#define KEY 3536

struct sembuf inc[1],dec[1];

void so()
{
    puts("#so");
}

int myFunct(int semid, void (*f)())
{
    if(semop(semid, dec, 1)==-1) return -1;
    f();
    if(semop(semid, inc, 1)==-1) return -1;
    return 0;
}

void set_operations()
{
    inc[0].sem_num = 0;
    inc[0].sem_op = 1;
    inc[0].sem_flg = 0;

    dec[0].sem_num = 0;
    dec[0].sem_op = -1;
    dec[0].sem_flg = 0;
}

int main()
{
    set_operations();
    int semid = semget(KEY, 1, IPC_CREAT|0666); //create 1 semaphore
    semctl(semid, 0, SETVAL, 1);// set its initial value to 1(mutex
semaphore)
    int ret=myFunct(semid,&so);
    return ret;
}

```



aliasbind / Sesiune

Watch 6

Star 16

Fork 15

Code

Issues 0

Pull requests 0

Projects 0

Wiki

Insights

Sisteme de Operare

thereau edited this page on Jan 17, 2015 · 43 revisions

Sisteme de Operare

Cuprins

- [1. Fișiere](#)
- [2. Fișiere](#)
- [3. Procese](#)
- [4. Semnale](#)
- [5. Shared Memory](#)
- [6. Semafoare](#)
- [7. Procese](#)
- [8. Fișiere](#)
- [9. Directoare](#)
- [10. Fișiere](#)

Pages 9

[Home](#)[Anul II](#)[Metode de dezvoltare software](#)[Pointeri](#)[Programare Orientată pe Obiecte](#)[SGBD](#)[Sisteme de Operare](#)[Sisteme de Operare \(pe scurt\)](#)[Subiecte](#)



UPDATE 04.04

Adăugat un "Cuprins" pentru navigarea mai ușoară a paginii și două exerciții rezolvate date la grupa noastră în sesiunea trecută.

UPDATE 27.01 16:00 - 22:00

Adăugate încă două probleme cu procese și fișiere cu tot cu comentarii în sursă.

UPDATE 27.01 13:55

Adăugată o problemă cu Semafoare (Problema 6).

UPDATE 26.01 22:00

Adăugată o problemă cu Shared Memory (Memorie Partajată) (Problema 5).

UPDATE 26.01 18:15

M-am uitat puțin peste semafoare și am ajuns la concluzia că o să dureze ceva până o să le înțeleg. Sper să reușesc să le bag la cap și voi posta (sper) ceva probleme cu ele mâine.

UPDATE 26.01 17:35

Am mai adăugat o problemă cu semnale (Problema 4).

UPDATE 26.01 17:00

Am scos tot ce ținea de funcția `perror()` că se pare că nu-i convine d-lui Baranga să folosim funcția asta la examen. (Mulțumesc, Eugen!).

Recomand să vedeti și pagina despre [Pointeri](#).

Probleme rezolvate

1. Grupa 242 (2009 - 2010), Exercițiul 5. (Fișiere)

```
int myFunct(char *buf, int fd, int *poz)
```

`fd` este ID-ul unui fișier din descriptorul de fișiere al programului. Se citește din fișier un caracter și se pune în `buf` pe poziția `*poz`, după care se incrementează circular `*poz`. Returnează `0`, respectiv `-1` dacă a terminat cu success, respectiv a dat de o eroare.

Rezolvare:

```
#include <unistd.h>
// Includ biblioteca ce conține majoritatea apelurilor de sistem.

int myFunct(char *buf, int fd, int *poz)
{
    char temp;

    // Citesc '1' caracter din fișierul menționat de
    // descriptorul 'fd' și îl pun în 'temp'.
    if( read(fd, &temp, 1) >= 0 )
    {
        buf[*poz] = temp;
        (*poz)++;
        return 0;
    }

    return -1;
}
```

Linkuri ajutătoare:

- [read\(\)](#)
-

2. Grupa 242 (2009 - 2010), Exercițiul 6. (Fișiere)

```
int myFunct(char *f1, char *f2, int m, int n)
```

Se iau caracterele din fișierul `f1` dintre pozițiile `m` și `n` și se pun la sfârșitul lui `f2` (dacă nu există se creaază). Se pun numai litere și cifre, restul caracterelor se sar. Returnează `-1` în caz de eroare sau numărul de caractere puse în `f2` în caz de succes.

Rezolvare:

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
//fcntl.h conține constantele O_RDWR, O_RDONLY, O_CREAT, etc. și funcția open()

int myFunct(char *f1, char *f2, int m, int n)
{
    int fd_1, fd_2;

    //Cum în cerință scrie că trebuie să iau elementele dintre 'm' și 'n',
    //atunci trebuie să fiu sigur că 'm' este strict mai mic ca 'n'.
    //Altfel, funcția nu ar avea sens.
    if( m > n )
        return -1;

    //Deschid fișierul 'f1' doar în citire
```

```

//și memorez descriptorul său în fd_1
if( (fd_1 = open(f1, O_RDONLY)) < 0)
{
    return -1;
}

//Deschid fișierul 'f2' în citire și scriere, având niște opțiuni
//suplimentare:
//
//O_RDWR = deschid fișierul în citire și scriere
//O_CREAT = crează fișierul dacă nu a fost creat
//O_APPEND = mută cursorul de citire/scriere la sfârșitul fișierului
//0644 = masca de drepturi pentru fișierul nou creat (dacă nu a fost
//creat deja)
if( (fd_2 = open(f2, O_RDWR | O_CREAT | O_APPEND, 0644)) < 0)
{
    //nu ai putut deschide f2, dar f1 e deschis
    close(fd_1);
    return -1;
}

//Mut cursorul de citire la poziția 'm' a fișierului
if( lseek(fd_1, m, SEEK_SET) < 0)
{
    //nu ai putut face lseek, dar ele sunt tot deschise, și asta e ideea peste
    close(fd_1);
    close(fd_2);
    return -1;
}

int nr_de_char_scrisi = 0;

while(m <= n)
{
    //Declar un char în care citesc din fișier.

```

```
char temp;

//Citesc un char.
if( read(fd_1, &temp, 1) < 0 )
{
    close(fd_1);
    close(fd_2);
    return -1;
}

//Verific dacă char-ul citit este alfanumeric.
//Dacă este, atunci îl scriu în 'f2'.
if(isalnum(temp))
{
    if( write(fd_2, &temp, 1) < 0 )
    {
        return -1;
    }

    nr_de_char_scrisi++;
}

m++;
}
close(fd_1);
close(fd_2);
return nr_de_char_scrisi;
}
```

Linkuri ajutătoare:

- [open\(\)](#) A se observa că există două funcții având numele de `open`.
- [close\(\)](#)

- [lseek\(\)](#)
 - [write\(\)](#)
 - [Articol Wikipedia despre masca de drepturi](#)
 - [isalnum\(\) și altele](#)
-

3. Grupa 231 (2009 - 2010), Exercițiul 5. (Procese)

```
int myFunct(int *tab, int n)
```

tab este o listă de identificatori a n procese fiu. Funcția așteaptă terminarea tuturor proceselor din tabelă. Dacă cel puțin un proces nu s-a terminat, atunci se returnează -1 . Dacă toate procesele s-au terminat, se returnează media aritmetică a codurilor de returnare.

Rezolvare:

```
#include <stdlib.h>
#include <unistd.h>

int myFunct(int *tab, int n)
{
    int i;
    int suma_exit_statusurilor = 0;

    //Luăm pe rând toate pid-urile din vector
    for(i=0; i<n; i++)
    {
        //wait_ret = reține valoarea returnată de un apel al funcției waitpid()
        int wait_ret;

        //status = variabila în care se rețin informații legate de apelul waitpid()
```

```

int status;

//Facem un waitpid() neblocant (WNOHANG) asupra PID-ului curent (tab[i])
if( (wait_ret = waitpid(tab[i], &status, WNOHANG)) < 0)
{
    //Deși în cerință nu scria ce să facem în cazul în care waitpid()
    //eșuează am presupus că funcția ar returna, și în acest caz, -1.

    return -1;
}

//În cazul în care, waitpid() returnează 0, atunci
//înseamnă că procesul curent (tab[i]) nu și-a terminat execuția.
//Deci, putem returna -1 conform cerinței.
if(wait_ret == 0)
{
    return -1;
}

//Dacă waitpid() a returnat o valoare strict pozitivă,
//atunci luăm valoare de return a procesului fiu și o adunăm
//la sumă.
suma_exit_statusurilor += WIFEXITED(status);
}

//Returnăm media aritmetică a valorilor de return a proceselor fiu,
//având PID-urile în vectorul 'tab'.
return suma_exit_statusurilor / n;
}

```

Linkuri ajutătoare:

- [waitpid\(\)](#). Aveți aici informații și despre `WNOHANG` și `WIFEXITED`.

4. Grupa 232 (2009-2010), Exercițiul 6 (Semnale)

Scrieti o functie C `int myFunct(int *tab, int n, (void *)funct(int x), int val) (sic)` care lanseaza un proces fiu care mai intai mascheaza (**cum?**) semnalele din vectorul tab, ce are lungime `n` si apoi va executa functia desemnata de `funct` cu argumentul `val`. Se va returna `-1` in caz de eroare sau daca procesul fiu nu s-a terminat normal si `0` altfel.

NOTĂ: Nu este mentionat clar cum se face mascarea (sau nu intelegh eu), asa ca am presupus ca toate acele semnale din vectorul `tab` sunt mascate ca fiind ignore. Totodata, header-ul dat in cerinta este gresit.

Rezolvare:

```
int myFunct(int *tab, int n, void (funct) (int x), int val)
{
    //Se creeaza un proces fiu. Din acest moment exista doua procese, ambele execu
    //aceleiasi instructiuni pana la al doilea 'if'.
    pid_t child = fork();

    //Se verifică dacă fork-ul s-a realizat cu succes. În cazul în care a eșuat, î
    //că un proces nou nu a mai fost creat, iar funcția returnează -1
    if(child < 0)
    {
        return -1;
    }

    //Aici se verifică dacă procesul este fiul.
    //Dacă da, se execută instrucțiunile explicate în cerință.
    //(dacă child este 0, atunci înseamnă că ne aflăm în fiu)
    if(child == 0)
    {
```

```
//Declarăm o "multime de semnale".  
//Aceasta este un fel de vector în care sunt reținute semnalele.  
//Peste un 'sigset_t' se execută operații folosind anumite funcții  
//(vezi mai jos la Linkuri ajutătoare).  
sigset_t sset;  
  
//Golim multimea de semnale.  
if( sigemptyset(&sset) < 0)  
    exit(1);  
  
int i;  
  
//Adăugăm pe rând toate elementele din `tab` în această multime  
for(i=0; i<n; i++)  
    if( sigaddset(&sset, tab[i]) < 0)  
        exit(1);  
  
//Acum, peste toată multimea, punem o mască de ignorare.  
//Altfel spus, toate semnalele din 'sset' vor fi ignoreate de acum în colo.  
if( sigprocmask(SIG_SETMASK, &sset, NULL) < 0)  
    exit(1);  
  
//Apelăm funcția din cerință.  
funct(val);  
exit(0);  
}  
  
//Aici suntem înapoi în procesul tată.  
//Ultimul exit(0) ne asigură că procesul fiu nu v-a mai ajunge  
//vreodată să execute instrucțiunile care urmează.  
  
//Facem un 'wait' până când procesul fiu se termină de executat, după care îi  
//codul de return și, într-un final vom returna cum spune cerința.  
int status;  
wait(&status);
```

```
if(WIFEXITED(status) == 0)
    return 0;

    return -1;
}
```

Linkuri ajutătoare:

- [Functii în care se foloseste 'sigset_t'](#)
- [sigprocmask\(\)](#)
- [fork\(\)](#)
- [exit\(\)](#)

5. Grupa 243 (2009-2010), Exercițiul 5 (Shared Memory)

Problemă rezolvată de Eugen.

Să se scrie o funcție `int myFunc(key_t cheie)` care să transforme literele mici în litere mari și invers, din stringul de la începutul segmentului de memorie desemnat de `cheie`.

Rezolvare:

```
#include <ctype.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
```

```
int myFunct(key_t cheie)
{
    //Presupunem dimensiunea memoriei partajate ca fiind de 100,
    //întrucât în cerință nu se zice cât este.
    int dim = 100;

    //Accesăm prin funcția corespunzătoare cu ajutorul cheii.
    int get;
    get = shmget(cheie, dim, 0666);
    if(get < 0)
    {
        return -1;
    }

    //Atașăm segmentul de memorie partajat la variabila 'str'.
    char *str;
    str = (char *) shmat(get, NULL, 0);
    if(str == (void *) -1)
    {
        return -1;
    }

    //Odată atașat, peste acest segment se pot desfășura operații
    //ca în orice alt vector alocat cu 'malloc'
    char *ss;
    for(ss = str; (*ss) != '\\0'; ss++)
    {
        if(islower(*ss)) //caz literă mică
            (*ss) = toupper(*ss);
        else
            if(isupper(*ss)) //caz literă mare
                (*ss) = tolower(*ss);
    }

    //Detașăm segmentul de memorie.
```

```
if(shmdt(str) < 0)
{
    return -1;
}

return 0;
}
```

Linkuri ajutătoare:

- [shmget\(\)](#)
 - [shmat\(\), shmdt\(\)](#)
-

6. Grupa 231 (2009-2010), Exercițiul 6 (Semafoare)

Scriți o funcție C sub forma `int myFunct(int id, (int *)f(), int *ret)` (sic) unde `id` e identificatorul unui semafor, care apelează semaforizat funcția desemnată de `f` (înainte de apel semaforul se decrementează și după apel semaforul se incrementează). La adresa desemnată de `ret` se va depune codul de return al funcției `f`. Se lucrează doar cu primul semafor din vector. `myFunct` returnează `0` în caz de succes sau `-1` în caz de eroare.

Rezolvare:

```
#include <sys/sem.h>

//Din nou, header-ul funcției dat în cerință este greșit.
//Dacă aş fi pus '(int *) f()' în loc de 'int (f())' mi-ar fi dat eroare la compil
//Presupun că header-ul e așa întrucât am dedus în același mod header-ul de la pro
int myFunct(int id, int (f)(), int *ret)
{
```

```
//Pentru a executa operațiunile de incrementare / decrementare
//trebuie să definim o structură de tip 'sembuf' în care
//să detaliem ceea ce vrem să facem.
struct sembuf sb;

//Alegem primul (și, în cazul nostru, singurul) semafor
//din array-ul de semafoare.
sb.sem_num = 0;

//Selectăm tipul operațiunii
//(Număr negativ = scade valoarea semaforului cu acea valoare)
//(Număr pozitiv = crește valoarea semaforului cu acea valoare)
//(Zero = funcția va intra într-o stare de așteptare până când valoarea
//semaforului va fi zero).
sb.sem_op = -1;

//Setăm lista de flaguri, în cazul nostru rămânând vidă.
sb.sem_flg = 0;

//Executăm operația de decrementare.
if(semop(id, &sb, 1) < 0)
    return -1;

//Apelăm funcția dată ca parametru și memorăm valoarea în
//locul indicat de 'ret'.
*ret = f();

//Acum va trebui să facem o incrementare. Cum am definit adineauri care semafo
//flaguri avem nevoie, nu va mai trebui decât să modificăm 'sem_op' să increme
sb.sem_op = 1;

//Executarea operației de incrementare.
if(semop(id, &sb, 1) < 0)
    return -1;
```

```
    return 0;  
}
```

Linkuri ajutătoare:

- [semop\(\)](#)
- [Tutorial despre semafoare \(scurt\)](#)
- [Tutorial despre semafoare \(lung\)](#)

7. Grupa 242 (2008-2009), Exercițiul 5 (Procese)

Scrieti o functie `int *myFunct(int *n)` care asteaptă terminarea tuturor proceselor fiu, pune într-un vector pidurile acestora și returnează adresa acestuia. De asemenea `*n` va reține numărul de procese fiu.

Rezolvare:

```
#include <errno.h>  
#include <stdlib.h>  
#include <unistd.h>  
  
int *myFunct(int *n)  
{  
    //Nu știu câți fii are procesul, aşa că o să presupun că are maxim 256.  
    int *copii = (int *) malloc(sizeof(int) * 256);  
  
    //În 'wait_stat' memorez valoarea returnată de 'wait()'  
    int wait_stat;
```

```

//Aici număr câți copii avea procesul în total.
int nr_de_copii = 0;

while(1)
{
    //Aștept un fiu să termine.
    wait_stat = wait(NULL);

    //Verific dacă wait-ul s-a desfășurat cu succes.
    if(wait_stat < 0)
    {
        //În cazul în care 'wait()' eșuează și setează errno cu
        //valoarea ECHILD, înseamnă că toate procesele fiu s-au terminat.
        if(errno = ECHILD)
        {
            //Setez *n-ul și returnez vectorul de piduri conform cerinței.
            *n = nr_de_copii;
            return copii;
        }

        //Returnez NULL dacă wait eșuează cu oricare errno în
        //afară de ECHILD.
        return NULL;
    }

    //Adaug pid-ul copilului "wait-at" în vector.
    copii[nr_de_copii] = wait_stat;
    nr_de_copii++;
}
}

```

Linkuri ajutătoare:

- [errno](#)

8. Grupa 231 (sau 233) (Anul curent), Exercițiu 5 (Fișiere)

Scrieti o functie C `int myFunct (char *f1, char *f2)` care intoarce `0` dacă fișierele cu numele dat de `f1` și `f2` coincid, și `1` altfel. Se intoarce `-1` în caz de eroare.

Rezolvare:

```
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int myFunct(char *f1, char *f2)
{
    //Descriptorii fișierelor 'f1', respectiv 'f2'.
    int fd_1, fd_2;

    //Deschid 'f1' în mod read-only.
    fd_1 = open(f1, O_RDONLY);
    if(fd_1 < 0)
        return -1;

    //Analog 'f2'.
    fd_2 = open(f2, O_RDONLY);
    if(fd_2 < 0)
        return -1;

    //Bufferii în care citesc caracterele din fișiere.
    char buf1, buf2;

    //Rețin aici numărul de bytes citiți la fiecare 'read'.
    int bytes_cititi;
```

```

//Variabile care le setez cu '1' atunci când am ajuns
//la sfârșitul fișierului respectiv.
int end1 = 0, end2 = 0;

while(1)
{
    //Citesc un byte din 'f1'.
    if( (bytes_cititi = read(fd_1, &buf1, 1)) == 0 )
    {
        //În cazul în care am ajuns la sfârșitul fișierului, intru în acest
        //'if' și îi spun programului că am ajuns la sfârșitul fișierului.
        end1 = 1;
    }
    else
    {
        //Intru aici când apelul read 'eșuează'.
        if(bytes_cititi < 0)
            return -1;
    }

    //Analog ca în 'if'-ul precedent, dar pentru 'f2'.
    if( (bytes_cititi = read(fd_2, &buf2, 1)) == 0 )
    {
        end2 = 1;
    }
    else
    {
        if(bytes_cititi < 0)
            return -1;
    }

    //Verific dacă ultimul char citit din 'f1' este identic cu
    //ultimul char citit din 'f2'. Dacă da, continu. Dacă nu mă opresc.
    if(buf1 != buf2)
        return 1;
}

```

```

//Verific dacă fișierul 'f1' și 'f2' au ajuns la End-Of-File
//în același timp. Dacă da, înseamnă că fișierele sunt identice.
if(end1 == 1 && end2 == 1)
    return 0;

//Dacă unul din fișiere a terminat înaintea celuilalt,
//e clar că fișierele nu sunt identice.
if(end1 != end2)
    return 1;
}
}

```

9. Grupa 242 (Anul curent), Exercițiul 5 (Directoare)

Se consideră definiția

```

typedef struct
{
    char name[NAME_MAX];
    struct stat st;
} MYSTR;

```

Scrieți o funcție C

```
MYSTR *myFunct(char *dir, int *n)
```

care construiește un vector de structuri `MYSTR` ce conțin fiecare numele fișierelor și caracteristicile i-nodurilor din directorul cu numele `dir`. `n` (va) comtinre (*sic*) numărul de elemente din `vec`. Se

întoarce `NULL` în caz de eroare.

Rezolvare:

```
#include <dirent.h>
// Pentru operații pe directoare cum ar fi:
// * opendir()
// * readdir()
// * closedir()
#include <limits.h>
// Pentru constantele NAME_MAX și PATH_MAX
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
// Pentru funcția și structura 'stat'.

typedef struct
{
    char name[NAME_MAX];
    struct stat st;
} MYSTR;

MYSTR *myFunct(const char *dir, int *n)
{
    // Deschidem directorul aflat la calea specificată
    // de 'dir'.
    DIR *dirp = opendir(dir);
    if(dirp == NULL)
    {
        (*n) = 0;
        return NULL;
    }

    // Citim primul fișier din director pentru a verifica
```

```
// dacă directorul a fost deschis cum trebuie și se pot
// executa operațiile de citire.
struct dirent *dirs = readdir(dirp);
if(dirs == NULL)
{
    (*n) = 0;
    closedir(dirp);
    return NULL;
}

// Setăm numărul de fișiere cu 1 pentru că citisem
// adineaoară primul fișier, apoi citim restul de fișiere
// și incrementăm n-ul de fiecare dată.
// Acest pas este important pentru a ști lungimea
// finală a vectorului de structuri 'MYSTR'.
(*n) = 1;
while( (dirs = readdir(dirp)) != NULL)
{
    (*n)++;
}
closedir(dirp);

// Închidem și redeschidem directorul dat pentru a
// o lua de la capăt cu citirea datelor. De data
// asta vom citi și numele fișierelor + informațiile
// oferite de 'stat'.
dirp = opendir(dir);
if(dirp == NULL)
{
    return NULL;
}

// Alocăm vectorul de structuri mai sus menționat.
MYSTR *files = (MYSTR *) malloc(sizeof(MYSTR) * (*n));
if(files == NULL)
```

```

{
    closedir(dirp);
    return NULL;
}

// Pentru a obține informații la nivel de sistem
// despre un anumit fișier, avem nevoie de calea lui exactă.
// Astfel, vom aloca un vector de caractere (aka string)
// de lungime PATH_MAX (lungimea maximă a unei căi oarecare
// acceptată pe un sistem de operare POSIX).
char *current_file_path = malloc(sizeof(char) * PATH_MAX);
int i = 0;

// Începem citirea riguroasă a fișierelor.
// Reamintesc că la fiecare readdir(dirp) se obține o
// referință către un alt fișier din directorul referit
// de 'dirp'. Când nu mai sunt fișiere, 'readdir' returnează 'NULL'.
while( (dirs = readdir(dirp)) != NULL)
{
    // Luăm de la fiecare fișier numele și îl reținem
    // în vectorul nostru de structuri.
    strcpy(files[i].name, dirs->d_name);

    // Aici creăm calea la fișierul curent, relativă
    // la directorul 'dir'.
    // Spre exemplu, dacă 'dir' = "NewFolder" și denumirea
    // fișierului curent ar fi "NewFile", calea relativă
    // ar fi "NewFolder/NewFile".
    strcpy(current_file_path, dir);
    strcat(current_file_path, "/");
    strcat(current_file_path, files[i].name);

    // Executăm 'stat()' și punem informațiile
    // în vectorul nostru.
    if( stat(current_file_path, &files[i].st) < 0)

```

```
    {
        free(current_file_path);
        free(files);
        closedir(dirp);
        return NULL;
    }

    i++;
}

// Eliberăm memoria alocată de noi care nu mai
// folosește programului și închidem directoarele.
free(current_file_path);
closedir(dirp);

return files;
}
```

Linkuri ajutătoare:

- [opendir\(\)](#)
- [readdir\(\)](#)
- [closedir\(\)](#)
- [stat\(\)](#)

10. Grupa 242 (Anul curent), Exercițiul 6 (Fișiere)

Scrieti o functie C

```
double myFunct(char *f, double *vec, int n, int *err)
```

unde `f` este numele unui fișier ce conține în format binar elemente de tip double, iar `n` numărul de elemente din `vec`. Funcția returnează produsul scalar dintre `vec` și vectorul format din primele `n` elemente din `f`. Dacă sunt mai puțin de `n` elemente în `f`, atunci elementele lipsă vor fi considerate ca având valoarea `0.0`. `err` este parametru de ieșire și va avea valoarea `0` în caz de success și `-1` în caz de eroare.

Rezolvare:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

double myFunct(char *f, double *vec, int n, int *err)
{
    int fd = open(f, O_RDONLY);
    if(fd < 0)
    {
        *err = -1;
        return 0.0;
    }

    int bytes_read;
    int i = 0;
    double prod_scalar = 0;
    double buf;
    while( (bytes_read = read(fd, &buf, sizeof(double))) >= sizeof(double) && i <
    {
        if(bytes_read == -1)
        {
            close(fd);
            *err = -1;
        }
        else
        {
            prod_scalar += buf * vec[i];
            i++;
        }
    }
    return prod_scalar;
}
```

```
    return 0.0;
}

prod_scalar += buf * vec[i];
i++;
}

*err = 0;
close(fd);
return prod_scalar;
}
```

◀ ▶

TODO: De explicitat exercițiul ăsta.

