

Sisteme de operare

Sistem de operare = software fundamental care gestionează resursele hardware și furnizează servicii de bază pentru alte programe care rulează pe un dispozitiv

↳ Windows, macOS, Linux, iOS, Android

Linia de comandă

- ↳ trimitete comenzi directe către un S.O.
 - mai eficientă, control mai bun, etc
 - „un program ce ne permite să ruleză alte programe”
 - în ceea ce urmă:
 - Linux: Sh, Bash, Ksh...
 - Windows: Cmd, PowerShell
 - Mac OS: Terminal (Bash)
- ! case sensitive

Consola - permite userilor să interacționeze direct cu un S.O.

UNIX - S.O. ~~az dezvoltă~~ → Linux

Comenzi și combinații de taste în UNIX

- formatul unei comenzi UNIX:

comandă [opțiuni] [argumente]

- comandă: este primul cuvânt introdus în linia de comandă și este un program
- opțiuni: sunt specificate folosind caracterul - (short options) sau -- (long options)
- argumente: pot fi obligatorii, optionale sau pot să nu existe
- delimitarea comenzi de opțiuni/argumente se face prin SPAȚII
- interpretorul de comenzi face deosebire între litere mici și litere mari (case-sensitive)

- comenzi pentru navigarea în sistemul de fișiere:

Comandă	Abreviere	Efect
pwd	print working directory	Afișează directorul curent
ls	list	Afișează conținutul directorului curent
cd dir	change directories	Schimbă directorul curent cu cel specificat

- o comandă este un program

- comenzi pentru manipularea directoarelor:

Comandă	Efect
<code>mkdir nume_dir</code>	Crează un director nou
<code>cp dir_src dir_dest</code>	Copiază directorul <code>dir_src</code> în directorul <code>dir_dest</code>
<code>mv dir_src dir_dest</code>	Mută directorul <code>dir_src</code> în directorul <code>dir_dest</code>
<code>rm (sau rmdir) nume_dir</code>	Șterge directorul al căruia nume e furnizat ca argument

- comenzi pentru manipularea fișierelor:

Comandă	Efect
<code>touch nume_fisier</code>	Crează un fișier nou care nu are conținut (este gol)
<code>cp fis_src fis_dest</code>	Copiază fișierul <code>fis_src</code> în fișierul <code>fis_dest</code>
<code>mv fis_src fis_dest</code>	Mută fișierul <code>fis_src</code> în fișierul <code>fis_dest</code>
<code>rm nume_fisier</code>	Șterge fișierul al căruia nume e furnizat ca argument
<code>cat nume_fisier</code>	Afișează conținutul fișierului furnizat ca argument

- alte comenzi:

`help`, `history`, `clear`, `cut`, `file`, `grep`, `head`, `less`, `more`, `sort`, `tail`, `wc`, `who`, `whoami`, `users`, `uname`

- taste speciale:

- TAB - completare automată a liniei de comandă (completion)
- săgeată în sus (\uparrow) sau săgeată în jos (\downarrow) - navigare în istoricul de comenzi

- combinări de taste:

Combinatie	Efect
<code>ctrl-C</code>	Oprește execuția programului care rulează la momentul actual
<code>Ctrl-Z</code>	Suspendă execuția programului care rulează la momentul actual
<code>Ctrl-D</code>	Închide sesiunea de lucru (în anumite situații este echivalent cu EOF)
<code>Ctrl-S</code>	Blochează consola
<code>Ctrl-Q</code>	Deblochează consola
<code>Ctrl-K</code>	Decupează textul de la poziția curentă până la sfârșitul liniei de comandă
<code>Ctrl-Y</code>	Lipește textul decupat anterior cu <code>Ctrl-K</code>
<code>Ctrl-R</code>	Căută în istoricul de comenzi
<code>Ctrl-A</code>	Mută cursorul la începutul liniei de comandă
<code>Ctrl-B</code>	Mută cursorul înapoi cu un caracter
<code>Ctrl-F</code>	Mută cursorul înainte cu un caracter
<code>Ctrl-E</code>	Mută cursorul la sfârșitul liniei de comandă

3. CONSULTAREA MANUALULUI UNIX

- consultarea paginilor din manual pentru o anumită comandă:

man nume_comandă

- localizarea paginii de manual: apropos, whatis

- manualul este împărțit în mai multe secțiuni:

- **secțiunea 1: Comenzi utilizator**
- **secțiunea 2: Apeluri sistem**
- **secțiunea 3: Funcții din librării**
- **secțiunea 4: Fișiere speciale și a.m.d.**

- consultarea manualului pentru o anumită secțiune:

man [secțiune] nume_comandă/nume_apel/nume_funcție

- navigarea în manual:

- pagina anterioară: b, PgUp (în cazuri foarte rare se suspendă execuția)
- pagina următoare: SPACE, PgDn (în cazuri foarte rare se suspendă execuția)
- căutare: / (slash)
- ieșire: q

- varianta online: <https://linux.die.net/man/>

4. REDIRECTAREA FLUXURILOR STANDARD ȘI CONECTAREA COMENZILOR ÎN PIPE

- fluxuri standard:

- 0 = flux standard de intrare STDIN (standard input)
- 1 = flux standard de ieșire STDOUT (standard output)
- 2 = flux standard de eroare STDERR (standard error)

- simboluri utilizate pentru redirectare:

- redirectarea intrării standard: <
- redirectarea ieșirii standard:
 - > (dacă fișierul există, suprascrie conținutul acestuia)
 - >> (dacă fișierul există, adaugă ieșirea comenzi la conținutul acestuia)

- redirectarea fluxurilor standard:

- redirectarea fluxul standard de intrare:

who >users.txt sort <users.txt

- redirectarea fluxul standard de ieșire într-un fișier:

ls -l >list.txt sau ls -l 1>list.txt

ls -l >>list.txt sau ls -l 1>>list.txt

- redirectarea fluxul standard de eroare într-un fișier:

ls -l /bonus >error.log sau ls -l /bonus 2>error.log

ls -l /bonus >>error.log sau ls -l /bonus 2>>error.log

- redirectarea fluxurilor de ieșire și de eroare în același fișier:

```
ls -l /bonus >output.log 1>&2
ls -l /bonus >>output.log 1>>&2
```

- conectarea comenziilor prin pipe:

```
who | sort
sort users.txt | head -n 5
sort users.txt | tail -n 5
```

. PRIMUL PROGRAM C ÎN UNIX

- editoare de texte în UNIX: vi, nano, joe
- exemplu: hello.c

```
#include <stdio.h>

int main(int argc, char** argv)
{
    printf("Hello world!\n");
    return 0;
}
```

- compilarea: gcc -Wall -g -o hello hello.c
- lansarea în execuție: ./hello

Comenzi:

mano mamne fișier

man comandă

what is comandă

Command line editor: vim, mano, joe, etc...

Conectarea comenziilor în PIPE

→ pipe | ia output-ul comenzii de dinainte și îl pune ca input comenzii de după

C programming im Liniendekommando

2. C language

- a. String vs byte array: strings end with 0, while buffers must be accompanied by their length somehow
- b. There are no references in C, only pointers (memory addresses)
 - i. &n is the address of variable n
 - ii. *p is the content of pointer p.
- c. Command line arguments: int main(int argc, char** argv)
 - i. argv[0] - name of the command
 - ii. argv[1] is the first argument, argv[2] is the second argument, and so on
 - iii. argc - length of array argv
- d. Memory
 - i. Allocation - malloc
 - ii. Deallocation - free

→ probleme de memoie: Valgrind

Labonator 2

PROGRAMARE ÎN LIMBAJUL C

1. TIPURI DE DATE DE BAZĂ

Tip	Dimensiune de stocare	Domeniu de valori
char	1 octet	[-128, 127] sau [0, 255]
int	2 sau 4 octeți	[-32.768, 32.767] sau [-2.147.483.648, 2.147.483.647]
float	4 octeți	[1,2E-38, 3,4E+38]
double	8 octeți	[2,3E-308, 1,7E+308]

- tipurilor de date `char` și `int` li se poate aplica calificatorul: `unsigned`
- tipului de date `int` i se mai pot aplica și calificatori: `short` și `long`

Tip	Dimensiune de stocare	Domeniu de valori
<code>unsigned char</code>	1 octet	[0, 255]
<code>unsigned int</code>	2 sau 4 octeți	[0, 65.535] sau [0, 4.294.967.295]
<code>short (int)</code>	2 octeți	[-32.768, 32.767]
<code>unsigned short</code>	2 octeți	[0, 65.535]
<code>long (int)</code>	4 octeți	[-2.147.483.648, 2.147.483.647]
<code>unsigned long</code>	4 octeți	[0, 4.294.967.295]

2. CUVINTE REZERVATE (CUVINTE-CHEIE)

auto	break	case	char	const	continue
default	do	double	else	enum	extern
float	for	goto	if	int	long
register	return	short	signed	sizeof	static
struct	switch	typedef	unsigned	union	void
volatile	while				

3. CONSTANTE

- pot fi definite în două moduri:
 - folosind directiva de preprocesare `#define`:

```
#define TEN 10
#define NEWLINE '\n'
```

 - utilizând prefixul `const`:

```
const int TEN = 10;
const char NEWLINE = '\n';
```

4. VARIABILE

- definirea unei variabile:

```
tip_variabila nume_variabila;
```

- **tip_variabila** poate fi: char, int, short, long etc. (vezi pct. 1)
- **nume_variabila** poate fi alcătuit din litere, cifre și caracterul „_” (underscore)
- primul caracter trebuie să fie o literă
- cuvintele rezervate (vezi pct. 2) nu pot fi utilizate ca nume de variabile
- limbajul C face deosebire între litere mici și litere mari (case-sensitive)

- exemple:

```
int n;           int n = 10;
char c;         char c = 'a';
```

5. OPERATORI

- specifică operațiile care se efectuează cu variabile și constante

- categorii de operatori:

- **operatori aritmetici:** + - * / % ++ --

- **operatori relaționali:** == != > < >= <=

- **operatori logici:** && || !

- **operatori logici pe biți:** & | ^ ~ << >>

- **operatori de atribuire:** = += -= *= /= %= <<= >>= &= ^= |=

- **alți operatori:** sizeof() & * ?:

- prioritatea operatorilor:

https://en.cppreference.com/w/c/language/operator_precedence

6. TIPURI DERIVATE DE DATE

6.a. Tablouri unidimensionale (vectori)

```
tip_tablou nume_tablou[dimensiune_tablou];
```

- exemple:

```
int list[5];
int list[5] = {10, 20, 30, 40, 50};
double values[] = {100.0, 2.0, 300.0, 40.0, 50.0};
```

6.b. Siruri de caractere

```
char msg[] = "Hello";
char msg[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

6.c. Pointeri

- **pointer** = o variabilă care conține adresa unei alte variabile

- exemple:

```
int *p; // pointer către o variabilă de tip întreg
char *c; // pointer către o variabilă de tip float
float *f; // pointer către o variabilă de tip float
```

double *d; // pointer către o variabilă de tip double

- obținerea adresei la care este stocată variabila x: &x
- obținerea valorii variabilei x: *p (dacă p este un pointer către variabila x)

6.d. Structuri de date

- definirea unei structuri de date:

```
struct Books
{
    int id;
    char author[50];
    char title[100];
}
```

- declararea și utilizarea unei structuri de date:

```
int main(int argc, char** argv)
{
    struct Books book1;
    ...
    book1.id = 1000;
    strcpy(book1.author, "B.W. Kernighan, D.M. Ritchie");
    strcpy(book1.title, "The C Programming Language");
    ...
    return 0;
}
```

7. FUNCTII

- declararea unei funcții:

tip_returnat nume_functie(tip_param param1, tip_param param2, ...);

unde:

- *tip_returnat* poate fi orice tip de date de bază/tip de date derivat sau void
- *nume_functie* poate fi alcătuit din litere, cifre și caracterul „_” (*underscore*)
- primul caracter trebuie să fie o literă
- cuvintele rezervate (*vezi pct. 2*) nu pot fi utilizate ca nume de funcții
- *tip_param param1, tip_param param2, ...* este lista parametrilor formali sau void (dacă funcția nu are parametruii)

- exemplu:

```
void afiseaza_matrice(int** matrice)
float media_aritmetica(int a, int b)
int** citeste_matrice(FILE* file)
```

- funcția **main()**:

- este punctul de intrare principal în program (*the program main entry point*)
- prototipul recomandat este:

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

deoarece permite accesul la argumentele furnizate de utilizator în linia de comandă.

8. FUNCȚII DE INTRARE/IEȘIRE

```
int getchar(void)
int putchar(void)
char *gets(char *s)
int puts(const char *s)
int scanf(const char *format, ...)
int printf(const char *format, ...)
```

9. FUNCȚII PENTRU LUCRUL CU FIȘIERE

9.a. Pentru fișiere de tip text:

```
FILE *fopen(const char *filename, const char *mode)
int fgetc(FILE *fp)
int fgets(char *buf, int n, FILE *fp)
int fputc(int c, FILE *fp)
int fputs(const char *s, FILE *fp)
int fclose(FILE *fp)
```

9.b. Pentru fișiere binare:

```
size_t fread(void *buf, size_t bsize, size_t nbyte, FILE *fp)
size_t fwrite(const void *buf, size_t bsize, size_t nbyte, FILE *fp)
```

sau:

```
int open(const char *path, int oflag, ... )
ssize_t read(int fd, void *buf, size_t nbyte)
ssize_t write(int fd, const void *buf, size_t nbyte)
int close(int fd)
```

10. PRIMUL PROGRAM C ÎN UNIX

- editoare de texte în UNIX: vi, nano, joe
- exemplu: hello.c

```
#include <stdio.h>

// the program main entry point
int main(int argc, char** argv)
{
    printf("Hello world !\n");
    return 0;
}
```

- compilarea: gcc -Wall -g -o hello hello.c
- lansarea în execuție: ./hello

11. EXEMPLE

- obținerea numărului și a listei de argumente furnizate în linia de comandă:


```
wget http://www.cs.ubbcluj.ro/~dbota/SO/lab2/lab2_2.c
```
- obținerea și afișarea variabilelor de mediu:


```
wget http://www.cs.ubbcluj.ro/~dbota/SO/lab2/lab2_3.c
```
- utilizarea tablourilor unidimensionale:


```
wget http://www.cs.ubbcluj.ro/~dbota/SO/lab2/lab2_4.c
```
- citirea unui întreg de la tastatură:

```
wget http://www.cs.ubbcluj.ro/~dbota/SO/lab2/lab2_5.c
```

- citirea conținutului unui fișier de tip text:

```
wget http://www.cs.ubbcluj.ro/~dbota/SO/lab2/lab2_6.c
```

- citirea conținutului unei matrice dintr-un fișier de tip text:

```
wget http://www.cs.ubbcluj.ro/~dbota/SO/lab2/lab2_7.c
```

12. ERORI/ATENȚIONĂRI LA COMPILEARE

- erori de sintaxă
- omiterea unui fișier de tip antet
- folosirea unei variabile care nu a fost definită
- folosirea a două variabile definite cu același nume
- folosirea unei funcții care nu a fost declarată
- apelul unei funcții fără respectarea prototipului acesteia (număr incorect de argumente, inversarea ordinii argumentelor etc.)

13. DETECȚIA ERORILOR DE GESTIUNE A MEMORIEI

- detectia erorilor de gestiune a memoriei folosind utilitarul valgrind:

```
valgrind ./myprog
```

REFERINTE:

- Curs: <http://www.cs.ubbcluj.ro/~rares/course/os/>
- Programare C: <https://www.tutorialspoint.com/cprogramming/index.htm>
- Manual Valgrind: <http://valgrind.org/docs/manual/quick-start.html>

Expresii regulare

expresie regulară = o secvență finită de caractere care descrie un pattern de căutare
 match = s-a găsit o porțiune de text care este conform expr. reg

- Orice caracter care operează într-o expr. regulară poate avea 2 intenții
 ⇒ ca să aibă ÎNTELESUL OBISNUIT ⇒ punem \ (backslash) în fața caracterului special

Caractere speciale:

	Semnificație într-o expresie regulară
.	un singur caracter (any character)
\.	DOAR caracterul ". "
\	Escape = dacă e înaintea unui caracter special îl face normal
[abc]	un singur caracter dintr-o listă de caractere date în paranșete (a,b sau c)
[^abc]	un singur caracter care NU se află într-o listă de caractere (d,e,...,z)
[a-z]	o singură literă mică de la a la z
[A-Z]	o singură literă mare de la A la Z
[a-zA-Z]	o singură literă mică sau mare de la A la Z
[0-9]	o singură cifră de la 0 la 9
[^0-9]	un singur caracter care nu e cifră
\d	o singură cifră de la 0 la 9 (echivalent cu [0-9])
\s	un singur caracter special (inclusiv SPACE, TAB, CR, LF)
\w	un singur caracter alfumeric sau caracterul underscore „_”
\()	gruparea mai multor caractere într-o expresie
^	indicație că începeți o nouă linie → înainte de expresie
\$	indicație că se termină o nouă linie → la sfârșitul expresiei
\<	indicație că începeți unui cuvânt
\>	indicație că se termină unui cuvânt
\b	indicație că se termină frontieră unui cuvânt \b Ana\b ⇔ \<Ana\>
?	cel mult o dată (adică 0 sau 1 dată)
*	dă 0 sau mai multe ori
+	dă 1 sau mai multe ori
{m}	dă exact m ori
{m, }	dă m sau mai multe ori
{, m}	dă cel mult m ori
{m,m}	dă cel puțin m ori, dar de cel mult m ori
	Orijină părții ale expresiei regulate

→ ex. grep '^Tudor' angajati.txt
 → ex. grep '1942.\$' angajati.txt

se pun după expresie

GREP

→ afisează liniiile din fișierul de intrare care conțin pattern-ul dat

grup optiuni

\downarrow expresie

fișier

- g = suprîmă regîna, se folosește dacă vrei doar să verifici (cod încîrcat: 0-există match, 1-nu)
- c, --count = afisează nr de linii care conțin pattern-ul acela
- i, --ignore-case = casă insensitivă
- r, --invert-match = afisează liniiile care **NU** conțin pattern-ul
- A nn = afisează nn (ex 2,3,...) linii după linii selectate
 (ex): grup -A 2 'Aaa' f.txt → afisează 2 linii după
- B nn = afisează nn (ex. 2,3,...) linii înainte de linii selectate
- C nn = afisează nn (ex 2,3,...) linii înainte și după
- m nn = afisează fix nn (ex 1,2,...) linii cu match pe pattern

exercițiu: Afisează

1. toate liniiile care conțin sirul Tudor

grup 'Tudor' angajatiG.txt
 \downarrow căută fix evenimentul acesta

2. toți angajații care au numele Tudor

grup '^Tudor' angajatiG.txt
 \uparrow început de nume

3. toate liniiile care conțin 1942

grup '1942' angajatiG.txt

4. toți angajații care au salariul 1942 → salariul e ultimul într-o linie a fișierului

grup '1942 \$' angajatiG.txt
 \uparrow sfârșit de nume

5. toți angajații al căror nume de familie începe cu A

grup '\< [A][a-z]\+ \>' angajatiG.txt
 \uparrow început de nume \uparrow mai multe litere \rightarrow sfârșit de nume.

grup '\s \< [A][a-z]\+ \> \s'
 \uparrow caracter special (aceea space) \uparrow mai multe litere (≥ 1)

6. toți angajații al căror nr de tel. începe cu 041

grup '041' angajatiG.txt → cauteră fix secvența 041 nu o expresie regulară

7. toți angajații măscuți în luna martie

grup '/3/' angajatiG.txt

8. toți angajații care locuiesc în Florești

grup 'Florești' angajatiG.txt

10. toți angajații care NU locuiesc în Florești sau Cluj-Napoca

grup '-v 'Florești' angajatiG.txt | grup '-v 'Cluj-Napoca'

\uparrow inverted match

\downarrow redirectionă

\uparrow inverted match

!!!

acă nu pui fișier pentru că primește primul pipe

! Observație

! grup -E 'pattern' f6.txt

↳ activează Extended Regular Expressions Mode

dici nu mai prefixez metacaracterile

grup -e 'pattern1' -e 'pattern2' f6.txt

↳ cauteră fie pattern1 fie pattern 2

```
antonia@DESKTOP-B9HP8D2:~/SO$ grep '[0-9]\{5\}$' angajatiG.txt
Horatiu Vasilescu 23/4/1965:Piata Marasti,13,Cluj-Napoca:0741-485769 37005
Octavian Cotescu 17/12/1954:Stejarului,68,Floresti:0745-789456 32150
antonia@DESKTOP-B9HP8D2:~/SO$ grep -m 1 '[0-9]\{5\}$' angajatiG.txt
Horatiu Vasilescu 23/4/1965:Piata Marasti,13,Cluj-Napoca:0741-485769 37005
antonia@DESKTOP-B9HP8D2:~/SO$ grep -A 1 '[0-9]\{5\}$' angajatiG.txt
Horatiu Vasilescu 23/4/1965:Piata Marasti,13,Cluj-Napoca:0741-485769 37005
Adrian Pintea 11/8/1957:Lacrimioarelor,22,Cluj-Napoca:0742-258369 1942
-- 
Octavian Cotescu 17/12/1954:Stejarului,68,Floresti:0745-789456 32150
Silviu Achim 19/10/1936:Tudor Vladimirescu,18,Cluj-Napoca:0726-369147 1932
antonia@DESKTOP-B9HP8D2:~/SO$ grep -B 2 '[0-9]\{5\}$' angajatiG.txt
Tudor Alexandrescu 2/5/1963:Aleea Baisoara,53,Cluj-Napoca:0742-235641 2355
Victor Baciu 25/9/1968:Eroilor,105,Floresti:0723-162453 4560
Horatiu Vasilescu 23/4/1965:Piata Marasti,13,Cluj-Napoca:0741-485769 37005
-- 
Stela Enache 28/2/1952:Sindicatelor,75,Cluj-Napoca:0745-563214 1946
Radu Beligan 8/4/1949:Zambilei,98,Semeseni:0744-852369 1957
Octavian Cotescu 17/12/1954:Stejarului,68,Floresti:0745-789456 32150
antonia@DESKTOP-B9HP8D2:~/SO$ grep -C 3 '[0-9]\{5\}$' angajatiG.txt
Alexandru Ionescu 3/7/1971:Aleea Bibliotecii,10,Cluj-Napoca:0721-124536 3875
Tudor Alexandrescu 2/5/1963:Aleea Baisoara,53,Cluj-Napoca:0742-235641 2355
Victor Baciu 25/9/1968:Eroilor,105,Floresti:0723-162453 4560
Horatiu Vasilescu 23/4/1965:Piata Marasti,13,Cluj-Napoca:0741-485769 37005
Adrian Pintea 11/8/1957:Lacrimioarelor,22,Cluj-Napoca:0742-258369 1942
Mircea Diaconu 6/11/1946:Prieteniei,7,Semeseni:0744-147258 2565
Ovidiu Moldovan 17/1/1942:Almasului,65,Cluj-Napoca:0722-123789 1968
-- 
Olga Tudorache 24/1/1932:Florilor,41,Floresti:0744-458712 1942
Stela Enache 28/2/1952:Sindicatelor,75,Cluj-Napoca:0745-563214 1946
Radu Beligan 8/4/1949:Zambilei,98,Semeseni:0744-852369 1957
Octavian Cotescu 17/12/1954:Stejarului,68,Floresti:0745-789456 32150
Silviu Achim 19/10/1936:Tudor Vladimirescu,18,Cluj-Napoca:0726-369147 1932
Toma Voicu 27/5/1948:Sportului,43,Floresti:0740-987125 1949
Ilarion Ciobanu 4/7/1931:Xenopol,32,Cluj-Napoca:0728-456987 1946
antonia@DESKTOP-B9HP8D2:~/SO$ |
```

11. Toate linile care contin o majusculă urmată de 4 litere mici, un spatiu și o majusculă
 grep '[A-Z][a-z]{4} [] [A-Z]' angajatiG.txt

↑
 caracter special ↑ pt
 special spatiu

12. Toate localitățile de domiciliu care încep cu litera F sau S
 grep '[FS][a-z]+:' angajatiG.txt

↑ caracterul în sine și căută

Labonator 3

Expresii regulate GREP

1. EXPRESII REGULARE

- **expresie regulară** = o secvență finită de caractere care descrie un pattern de căutare
- **potrivire (match)** = o literă, o secvență de octeți sau de caractere, o porțiune de text
- caractere speciale (meta-caractere):

. period or dot	\ backslash	^ caret	\$ dollar sign
vertical bar	? question mark	* asterix or star	+ plus sign
(opening parenthesis) closing parenthesis	[square bracket	{ curly brace

- într-o expresie regulară aceste caractere au un alt înțeles decât cel obișnuit
- pentru a le reda înțelesul lor obișnuit este necesar să le prefixăm cu simbolul \ backslash

- semnificația caracterelor speciale în expresii regulate:

Expresie	Semnificație potrivire (matching)
.	un singur caracter (oricare caracter)
\.	DOAR caracterul „.”
[abc]	un singur caracter dintre cele aflate între paranteze (a, b sau c)
[^abc]	un singur caracter care NU se află între paranteze (d, e, ..., z)
[a-z]	o singură literă mică de la a la z (<i>toate literele mici</i>)
[A-Z]	o singură literă mare de la A la Z (<i>toate literele mari</i>)
[a-zA-Z]	o singură literă mică sau mare
[0-9]	o singură cifră de la 0 la 9
[^0-9]	un singur caracter care nu e cifră
\d	o singură cifră de la 0 la 9 (echivalent cu [0-9])
\s	un singur caracter special (inclusiv SPACE, TAB, CR, LF)
\w	un singur caracter alfanumeric sau caracterul underscore „_”
\(\)	gruparea mai multor caractere într-o expresie

- exemplu:

1. Dacă se dă următoarele linii:

```
abc
bdf
ceg
```

putem scrie următoarele expresii regulate pentru a selecta:

- NUMAI prima linie: 'abc'
- NUMAI a doua linie: 'bdf'
- NUMAI a treia linie: 'ceg'
- TOATE LINIILE: '...' sau, mult mai restrictiv, '[abc] [bde] [cfg]'

▪ ancore (anchors):

Simbol	Semnificație
^	indică începutul unei linii
\$	indică sfârșitul unei linii
\<	indică începutul unui cuvânt
\>	indică sfârșitul unui cuvânt
\b	indică frontieră unui cuvânt (\bAna\b este echivalent cu \<Ana\>)

▪ operatori de repetiție:

Operator	Semnificație
?	cel mult o dată
*	de 0 (zero) sau mai multe ori
+	de 1 (unu) sau mai multe ori
{n}	de exact n ori
{n, }	de n sau mai multe ori
{, m}	de cel mult m ori
{n, m}	de cel puțin n ori, dar nu mai mult de m ori

▪ exemplu:

2. Dacă un fișier de intrare conține următoarele linii:

```
aaabc
aaadf
aaace
```

- pot fi scrise următoarele expresii regulate pentru potrivire (matching) pe:
 - TOATE LINIILE: 'aaa[bdc][cfe]' sau 'a{3}[bdc][cfe]'

2. UTILITARUL grep

- afișează liniile din fișierul de intrare care conțin pattern-ul dat

▪ descrierea comenzi:

```
grep [OPTIONS] PATTERN [FILE...]
grep [OPTIONS] [-e PATTERN] [-f FILE...] [FILE...]
```

▪ OPTIONS:

-c, --count	afișează numărul de liniile care conțin pattern-ului
-i, --ignore-case	ignoră distincția dintre literele mici și mari
-v, --invert-match	afișează liniile care nu conțin pattern-ul
-A NUM, --after-context=NUM	afișează NUM liniile după liniile selectate
-B NUM, --before-context=NUM	afișează NUM liniile înainte de liniile selectate
-C NUM, -NUM --context=NUM	afișează exact NUM liniile care conțin pattern-ul

- PATTERN este furnizat, de obicei, în linia de comandă printr-o expresie regulară

- pentru a furniza mai multe pattern-uri sau un pattern care începe cu „-” (hyphen):

`-e PATTERN (--regexp=PATTERN)`

- pentru a furniza pattern-uri dintr-un fișier (câte un pattern pe linie):

`-f FILE (--file=FILE)`

SED

= Stream EDitor

= editor de text interactiv, care procesă pe rând, fiecare rând, fiecare linie din textul furnizat la intrare și afișază rezultatul procesării pe ecran (nu modifică fișierul deoarece afișază "fătă")

`sed [optioni] '[/pattern/] command' f.txt`

`sed -f script-fie.txt f.txt`

↳ permite specificarea comenzilor în fișierul `script-fie.txt`

optioni

↳ -m = suprimă afișarea automată a bufferului între linii (pattern space)

-e script = permite specificarea mai multor comenzi în linia de comandă

- dacă nu se specifică o anumită linie/pattern /mai multe linii ⇒ `command` se va executa pe TOATE linile din textul furnizat la intrare

• Specificarea linilelor de procesat

N - doar linia N

\$ - doar ultima linie

M,N - de la linia M la linia N

M~step - de la linia M, linile din step în step

/regexpr/ - doar linile care conțin patternul specificat prin regexpr

0,/regexpr/ - doar prima linie care conține patternul specificat

M,+N - de la linia M, N linii după

M,~N - de la linia M, toate linile multiple de N

• COMENZI

→ p (print)

→ d (delete)

→ s (substitute)

→ a (append)

→ c (change)

→ i (insert)

→ q (quit)

→ r (read content to file)

→ w (write content to file)

→ l (display control characters)

→ m (next)

→ y (transform)

→ h (holding)

→ g (getting)

→ x (exchange)

exerciții:

1. conținutul întregului fișier

sed ' ' angajati.txt

sed -m 'p' angajati.txt

↑ „quiet” ca să nu dubleze

2. linii de la 5 la 10

sed -m '5,10 p' angajati.txt

3. linii de la 8 la ultima

sed -m '8,\$p' angajati.txt

4. linii care contin sirul „Tudor”

sed -m '/Tudor/p' angajati.txt

patternul trebuie încastrat între 2 slash-uri

5. linii care contin cuvântul Tudor

sed -m '/\bTudor\b/p' angajati.txt
 ↑ beginning ↑ end of word

sed -m '/\<Tudor\>/p' angajati.txt

6. stergeri linii de la 1 la 3

sed '1,3 d' angajati.txt

sed -e '1d' -e '2d' -e '3d' angajati.txt
 expresii multiple

sed -f script-1 angajati.txt ; script-1

{ #!/bin/sed
 1d
 2d
 3d

7. stergeri linii care contin cuvântul Popescu

sed '/Popescu/d' angajati.txt

8. stergeri linii de la Iom la Tudor

sed '/Iom/,/Tudor/d' angajati.txt

↑ de la... la... (se șterge la prima apariție a expr. dim dn chiar dacă ar mai fi linii valide după)

9. stergeri prima linie / ultima linie

sed '1d' angajati.txt

sed '\$d' angajati.txt

10. stergeri toate linile goale

sed '/^\$/d' angajati.txt

expresie regulară: ^ începutul liniei, \$ sfârșitul liniei

11. substituiri Olga cu **Olga**

sed 's/Olga/**Olga**/i' angajati.txt

↑ substitute

12. înlocuiri toate aparițiile nr 19 cu nr 18

sed 's/19/18/g' angajati.txt

↑ TOATE APARIȚIILE = GLOBAL (g)

13. creați o copie a fișierului
sed `'m !w file-copie.txt' file.txt`
14. adăugati o linie nouă după a treia linie
sed `'3a Limie adaugata' angajati.txt`
15. adăugati o linie nouă după numele Adrian
sed `'/Adrian/a Limie nouă' angajati.txt`
16. adăugati textul TERMINAT la sfârșitul fișierului
sed `'$ a TERMINAT' angajati.txt`
funciunea e să săracă spatiu
17. înlocuiți textul de pe linia 2 cu textul SALARIAT PENSIONAT
sed `'2 c /SALARIAT PENSIONAT/' angajati.txt`
18. inserati la începutul fișierului textul DATE DESPRE PERSONAL
sed `'1i \t \t DATE DESPRE PERSONAL' angajati.txt`
prima linie = 1
inserat TAB
19. înlocuiți spațiile cu tab-uri și afișați rezultatul cu sed
sed `'s/ /t/g' angajati.txt > junk.txt`
substituie spațiu tab
regexp global
redirecțare
20. opriți execuția după afișarea celui de-a 5-a linie
sed `'5 q' angajati.txt`
21. inserati după a treia linie un text dintr-un fișier
sed `'3 n text.txt' angajati.txt`
read from file
22. inserati pe a 5-a linie DATA și ORA CURENTĂ
sed `'5 e date' angajati.txt`

Laborator 4

SED
AWK

2. sed (Stream EDitor)

- este un editor de text neinteractiv, care procesează, pe rând, fiecare linie din textul furnizat la intrare și afișează rezultatul procesării pe ecran

- sintaxa comenzi:

```
sed [-n] [-e] '[/pattern/]command' [input-file]
sed [-n] -f script-file [input-file]
```

-n suprimă afișarea automată a buffer-ului intern (*pattern space*)

-e script permite specificarea mai multor comenzi în linia de comandă

-f script-file permite specificarea comenziilor în fișierul *script-file*

- dacă nu se specifică o anumită linie, un pattern sau mai multe linii, command se va executa pe toate liniile din textul furnizat la intrare

- textul de procesat poate fi furnizat: fie de la intrarea standard (tastatură), fie dintr-un fișier *input-file* sau poate fi rezultatul execuției în pipe a altelor comenzi

- **specificarea liniilor de procesat (adresarea liniilor):**

N	doar linia N
\$	doar ultima linie
M, N	de la linia M la linia N
M~step	de la linia M, liniile din step în step
/regexp/	doar liniile care conțin pattern-ul specificat prin regexp
0, /regexp/	doar prima linie care conține pattern-ul specificat prin regexp
M, +N	de la linia M, N liniile după
M, ~N	de la linia M, toate liniile multiplu de N

- **comenzi:**

- **p (print)**

```
sed angajati.txt
sed 'p' angajati.txt
sed -n 'p' angajati.txt
sed -n '2p' angajati.txt
sed -n '/Tudor/p' angajati.txt
sed -n '2,5p' angajati.txt
sed -n '/Ion/,/Victor/p' angajati.txt
sed -e '2p' -e '5p' angajati.txt
```

- **d (delete)**

```
sed 'd' angajati.txt
```

```
sed '4d' angajati.txt
sed '/Tudor/d' angajati.txt
sed '2,5d' angajati.txt
sed '/Tudor/, $d' angajati.txt
sed -e '2d' -e '5d' angajati.txt

- s (substitute)
  sed 's/Tudor/Tudorel/' angajati.txt
  sed -n 's/Tudor/Tudorel/' angajati.txt
  sed -n 's/19/18/g' angajati.txt
  sed -n 's/1931/1932/p' angajati.txt
  sed -n 's/(Ion\)\el/\lut/p' angajati.txt
  sed -n 's/[0-9\]\[0-9\]$/&.\5/' angajati.txt
  sed -n '/Olga/, /Toma/s/$/**CONCEDIU**/' angajati.txt

- a (append)
  sed '3a Linie adaugata' angajati.txt
  sed '$a TERMINAT' angajati.txt
  sed '/Adrian/a Linie adaugata' angajati.txt

- c (change)
  sed '2c SALARIAT PENSIONAT' angajati.txt

- i (insert)
  sed '1i \t\t\tDATE DESPRE PERSONAL' angajati.txt

- q (quit)
  sed '5q' angajati.txt

- r (read content from file)
  sed '3r text.txt' angajati.txt

- w (write content to file)
  sed -n 'w angajati.bak' angajati.txt

- = (print line number)

- I (display control characters)
  sed -n 'l' test.txt

- n (next)

- y (transform)

- h (holding)

- g (getting)

- x (exchange)
```

3. awk

- nu este doar un simplu utilitar de procesare a textelor, ci un limbaj de programare interpretat cu o sintaxă asemănătoare limbajului C
- numele său vine de la inițialele creatorilor: Alfred Aho, Peter Weinberger și Brian Kernighan

AWK

- limbaj de programare

```
awk optiuni '/pattern/' file.txt
awk optiuni '{action}' file.txt
awk optiuni '/pattern/ {action}' file.txt
```

optiuni

- **F** fs \Rightarrow specifică un field separator nou (ex: -F:, -F'[:\t]')

- **f** script file \Rightarrow permite specificarea comenzilor într-un fișier

\hookrightarrow programul awk poate fi scris în linia de comandă sau într-un fișier

Variabile interne

\$0

- linia curentă

\$1, \$2, ...

- coloanele din linia curentă (despartite de fs)

NR

- nr. de ordine al înregistrării curente

NF

- nr. de coloane din linia curentă

RS

- delimitatorul de înregistrării curente

ORS

- delimitatorul pentru afisarea câmpurilor

FS

- delim. de câmpuri / coloane curent

OFS

- delim pt afisarea câmpurilor

OFMT

- formatul de afisare a numerelor

ARGC

- nr de argumente din linia de comandă

ARGV

- tabelul de argumente din linia de comandă

FILENAME

- fișierul de intrare curent

FNR

- nr de ordine al înregistrării din fișierul de intrare curent

exercitii

1. afișați conținutul întregului fișier

```
awk '{print}' f.txt
```

2. afișați conținutul întregului fișier prefixând fiecare linie cu nr acestuia

```
awk '{print NR, $0}' f.txt
```

3. afișați conținutul întregului fișier prefixând fiecare linie cu nr de fields

```
awk '{print NF, $0}' f.txt
```

4. afișați numere, prenumele și nr de tel

```
awk '{print $2, $1, $4}' f.txt
```

luăm fiecare field

5. afișați toti angajații al căror nume începe cu T

```
awk '/^T/ {print}' f.txt
```

expresie \Rightarrow / /

6. afișați nume & prenume celor măscăti în mantie

```
awk '/^/ {print $1, $2}' f.txt
```

expresie \Rightarrow / /

metacaracter

7. afişați ang. al căror salariu = 1949

awk | $\$5 == 1949$ } print '\$' f.txt
al 5-lă camp

8. afişați ang. al căror salariu < 2000

awk | $\$5 < 2000$ } print '\$' f.txt

9. afişați linile care conțin mai mult de 42 de caractere

awk | length (\$0) > 42 } print NR, \$0 f.txt

10. calculați și afişați nr. total de angajați:

awk | BEGIN { total = 0 } { total++ } END { print "Nr angajați: ", total } f.txt

awk | END { print NR } f.txt

11. calculați și afişați suma necesară pt plată salariailor

awk | BEGIN { sumar = 0 } { sumar += \$5 } END { print "Sumar: ", sumar } f.txt

12. afişați nr de ang. furnizat în linia de comandă

awk | BEGIN { print "Nr de argumente", ARGV[0] } unde doi trei patru cîinei

13. afişați argumentele furnizate în linia de comandă

script-1 ⇒ } BEGIN { for (i=0; i < ARGV.length(); i++) }
printf "ARGV[%d] = %s \n", i, ARGV[i] }

awk -f script-1 unde doi trei patru

- sintaxa comenzi:

```
awk [OPTIONS] '/pattern/' [input-file]
awk [OPTIONS] '{action}' [input-file]
awk [OPTIONS] '/pattern/{action}' [input-file]
```

-F fs permite specificarea unui nou delimitator de câmpuri *fs*
-f script-file permite specificarea comenziilor în fișierul *script-file*

- awk procesează, pe rând, fiecare linie din textul furnizat la intrare (la fel ca grep sau sed)
- fiecare linie de text constituie o înregistrare (record)
- delimitator de înregistrări implicit: CR (Carriage Return)
- înregistrarea curentă este stocată în variabila internă \$0
- awk tratează fiecare înregistrare ca pe un text structurat și o separă în câmpuri (fields)
- delimitatori de câmpuri implicați: SPACE sau TAB
- variabile interne (*built-in variables*):

\$0	înregistrarea (linia) curentă
\$1, \$2, ...	câmpurile (coloanele) din înregistrarea (linia) curentă
NR	numărul de ordine al înregistrării curente
NF	numărul de câmpuri (coloane) din înregistrarea (linia) curentă
RS	delimitatorul de înregistrări (<i>input record separator</i>) curent
ORS	delimitatorul pentru afișarea înregistrărilor (<i>output record separator</i>)
FS	delimitatorul de câmpuri (<i>input field separator</i>) curent
OFS	delimitatorul pentru afișarea câmpurilor (<i>output field separator</i>)
OFMT	formatul de afișare a numerelor
ARGC	numărul de argumente furnizate în linia de comandă
ARGV	tabloul cu argumentele furnizate în linia de comandă
FILENAME	fișierul de intrare curent
FNR	numărul de ordine al înregistrării din fișierul de intrare curent
ENVIRON	tabloul cu variabile de mediu

- exemple:

- tipărirea întregului fișier:

```
awk '{print}' angajati.txt
awk '{print $0}' angajati.txt
```

- afișarea liniilor care conțin un pattern dat:

```
awk '/Tudor/' angajati.txt
awk '/Tudor/{print}' angajati.txt
awk '/Tudor/{print $0}' angajati.txt
```

- modificarea delimitatorului de câmpuri implicit:

```
awk -F: '{print $1}' /etc/passwd
awk -F: '{print NR, $1}' /etc/passwd
awk -F'[ :\t]' '{print $1, $2, $3}' angajati.txt
```

▪ **operatori relaționali:**

Operator	Semnificație	Utilizare
<	mai mic decât	x < y
<=	mai mic sau egal cu	x <= y
==	egal cu	x == y
!=	nu e egal cu	x != y
>=	mai mare sau egal decât	x >= y
>	mai mare decât	x > y
~	coresponde cu expresia regulară	x ~ /regexp/
!~	nu corespunde cu expresia regulară	x !~ /regexp/

▪ **exemple:**

- utilizare operatori:

```
awk '$5 < 2000' angajati.txt
awk '$5 < 2000 {print}' angajati.txt
awk '$5 == 1942 {print NR, $1}' angajati.txt
```

- utilizare operatori cu expresii regulate:

```
awk '$1 ~ /Tudor/ {print}' angajati.txt
awk '$1 !~ /Tudor/ {print}' angajati.txt
```

▪ **operatori logici:** `&& || !`

▪ **operatori aritmetici:** `+ - * / % ^`

▪ **operatori de atribuire:** `= += -= *= /= %= ^=`

▪ **expresii condiționale:**

conditie ? expresie1 : expresie2 este echivalentă cu:

```
if (conditie)
    expresie1
else
    expresie1
```

▪ **script-uri (fișiere de comenzi):**

- blocul **BEGIN** conține comenzi care se execută ÎNAINTE DE ÎNCEPEREA procesării textului
- blocul **END** conține comenzi care se execută DUPĂ ÎNCHEIEREA procesării textului
- comenzi cuprinse între acolade **{}** dintre blocurile BEGIN și END se execută pe timpul procesării textului

- **exemple:**

```
awk 'BEGIN{FS = ":"}' /etc/passwd
awk 'BEGIN{FS = ":"; OFS="\t"} {print $1, $2}' /etc/passwd
awk '/Ion/{cnt++}END{print "Ion apare de " cnt " ori."}' angajati.txt
awk 'END{print "Nr. angajati: " NR}' angajati.txt
awk 'BEGIN{total=0} {total++} END{print "Total: " total}' angajati.txt
```

UNIX Shell Programming - Bash Scripting

- cat - display ceea ce e in fisier
- echo ang - display ang

Bash Script \hookrightarrow se scrie in fisier **mumne.sh**

\rightarrow prima linie: **#!/bin/bash**

- execution permission: **chmod u+x mumne.sh**
- run: **./mumne.sh**

Variabile

(ex 1) $\begin{array}{l} \text{FIRST_NAME=Antonia} \\ \text{echo Hello \$FIRST_NAME} \end{array}$

(ex 3) \rightarrow Interaction cu user

```
antonia@DESKTOP-B9HP8D2:~/SO/Shell$ cat hello.sh
#!/bin/bash

echo What is your first name?
read FIRST_NAME
echo What is your last name?
read LAST_NAME
echo Hello $FIRST_NAME $LAST_NAME

antonia@DESKTOP-B9HP8D2:~/SO/Shell$ chmod u+x hello.sh
antonia@DESKTOP-B9HP8D2:~/SO/Shell$ ./hello.sh
What is your first name?
Antonia
What is your last name?
Moga
Hello Antonia Moga
antonia@DESKTOP-B9HP8D2:~/SO/Shell$
```

(ex 2) $\begin{array}{l} \text{#!/bin/bash} \\ \text{FIRST_NAME=Antonia} \end{array}$

LAST_NAME=Nume

echo Hello \\$FIRST_NAME \\$LAST_NAME

\Rightarrow **chmod u+x hello.sh**

\Rightarrow **./hello.sh** \Rightarrow Hello Antonia Nume

Positional Arguments

```
#!/bin/bash
echo Hello $1 $2
 $\Rightarrow$  chmod u+x ex.sh
 $\Rightarrow$  ./ex.sh ARGUMENT_1 ARGUMENT_2
```

$\downarrow \Rightarrow$ **./ex.sh Nume Prenume**
Hello Nume Prenume

Piping \rightarrow folosim pipe (,|), grep, sed, awk

Redirectarea output-ului

> \rightarrow scrie intr-un fisier
>> \rightarrow append la continutul unui fisier

```
$ echo Hello World! > hello.txt
```

```
$ echo Hello world >> hello.txt
```

Redirectarea input-ului

(ex) $\begin{array}{l} \text{wc -w hello.txt} \Rightarrow G hello.txt \\ \text{wc -w < hello.txt} \Rightarrow G \\ \text{wc -w << "Hello there word count!"} \\ \hookrightarrow 3 \end{array}$

```
herbert@DESKTOP-UFIAGQU:~/tutorial-bash$ cat << EOF
> I will
> write some
> text here
> EOF
I will
write some
text here
herbert@DESKTOP-UFIAGQU:~/tutorial-bash$
```

asteapta cuvantul
acesta ca sa se
opreasca

Ajax

```
LIST=(one two three four five)
— echo $LIST → one
— echo ${LIST[@]} → one two three four five (toate lista)
— echo ${LIST[0]} → one (elementul de pe pozitie)
```

```

1 #!/bin/bash
2
3 # Script ./users.sh user1 n1 user2 n2 user3 n3 .....
4 # Afisam un mesaj daca user1 s-a conectat de n1 ori in sistem .
5
6 if [ `expr $# % 2` -eq 1 ]; then
7     echo "Ne trebuie numar par de argumente"
8     exit 1
9 fi
10
11 while [ $# -gt 0 ]; do
12     echo $1 $2
13     if grep -E -q "^$1:" /etc/passwd ; then
14         if echo $2 | grep -E "^[1-9][0-9]*$"; then
15             echo $2 este numar
16         else
17             echo $2 nu este numar, se ignora perechea de argumente.
18         fi
19     else
20         echo $1 nu este user valid din sistem, se ignora.
21     fi
22     shift 2
23 done
24
```

~
~
~
~
~

-DELA CĂLIN < 24L, 595C 18,9 All

→ variabile sunt în mod implicit tratate ca siruri de caractere:

```
#!/bin/bash
A=1234
B=5678
echo "$A + $B" → 12345678
```

Sheff

variable sociale

- \$0 - numerele fisierului lansat în execuție
 - \$1, ..., \$9 - argumentele furnizate în linia de comandă
 - \$# - numărul de argumente din linia de comandă
 - \$* - sirul de argumente din linia de comandă
 - \$@ - lista individuală a argumentelor din linia
 - \$? - exit statusul ultimei comenzi executate
 - \$& - PID-ul procesului curent
 - \$! - PID-ul ultimei comenzi lansate în background

5.1 STANDARD INPUT/OUTPUT/ERROR AND I/O REDIRECTIONS

- 0 = standard input - where you read from when you use "scanf" or "gets" in C, "cin" in C++, or "input" in Python.
 - 1 = standard output - where you write when you use "printf" or "puts" in C, "cout" in C++, or "print" in Python.
 - 2 = standard error - similar to the standard output, but conventionally used to display errors, in order to avoid mixing results with errors.
 - What's the deal with 0, 1, and 2?
 - When you open a file in a program (written in any language), you get back some kind of variable that allows you to operate on the file (FILE* from fopen, int from open, etc). This variable contains an integer, representing the roughly the order number of the file opened by the program.
 - Whenever you start a program, it will have three files already open: 0, 1, and 2. Yeah, the program treats the command line like a file: it writes to it and it reads from it. Very natural, isn't it?
 - Many of the commands we will use act as filters: read the standard input, process it, and then print the results to the standard output. The errors will be printed to the standard error.
 - I/O redirections
 - What if I want the output of a command to be stored in a file?
 - ls -l --color=never /etc > output.txt
 - What if I want to add the output of another command to the same file?
 - ps -ef >> output.txt
 - What if I want the standard output of a command to be sent to the standard input of another command?
 - ls | sort → pipe
 - What if I want the standard input to be taken from a file?
 - sort < a.txt → & in dm a.txt
 - Redirect the errors of a command to a file
 - rm some-file-that-does-not-exist.c 2> output.err
 - Redirect both the standard and error output in the same file
 - rm some-file-that-does-not-exist.c > output.all 2>&1 - read as, redirect standard output to output.all, and the error output to the same place where the standard output goes
 - /dev/null
 - The file that contains nothing, and everything that you write to it, disappears. The Windows equivalent is NUL.
 - Used mainly to hide program output (either standard or error)

Redirectors: >, >>, <, |
↑
append

/dev/null = fizieră care nu conține nimic, orișe se scrie DISPARÈ

5.2 COMMAND TRUTH VALUES

1. The truth value of a command execution is determined by its exit code. The rule is the opposite of the C convention, with 0 being true, and anything else being false. Basically, there is only one way a command can be executed successfully, but many ways in which it can fail. The exit code is not the output of the command.
2. There are two standard commands `true` and `false`, that simply return 0 or 1.
3. Command `test` offers a lot of options for comparing integers, strings and verifying file and directory attributes

5.2.1 LECTURE 2

1. Commands can be chained using logical operators `&&` and `||`. Lazy logical evaluation can be used to nice effects. The negation operator `!` reverses the truth value of a command.
 - a. `true || echo This should not be displayed`
 - b. `false || echo This should definitely be displayed`
 - c. `true && echo This should also be displayed`
 - d. `false && echo Should never be displayed as well`
 - e. `grep -E -q "=" /etc/passwd || echo There are no equal signs in file /etc/passwd`
 - f. `test -f /etc/abc || echo File /etc/abc does not exist`
 - g. `test 1 -eq 2 || echo Not equal`
 - h. `test "asdf" == "qwer" || echo Not equal`
 - i. `! test -z "abc" || echo Empty string`
2. Test command conditional operators
 - a. String: `==`, `!=`, `-n`, `-z`
 - b. Integers: `-lt`, `-le`, `-eq`, `-ne`, `-ge`, `-gt`
 - c. File system: `-f`, `-d`, `-r`, `-w`, `-x`

Operatorul `||` execută comanda din partea dreaptă doar dacă ea din stânga e falsă
 Operatorul `&&` execută comanda din partea dreaptă doar dacă ea din stânga e adevarată

④ `test -f /etc/abc` → verifică dacă există `/etc/abc`

Operatorul `!` (NOT) înversează statusul comenzii

Semimar 2 - SO - Programare Shell

DREPTURI DE ACCES. PROGRAMARE SHELL

1. DREPTURI DE ACCES (FILES AND DIRECTORIES PERMISSIONS)

- pentru fiecare fișier sau director sunt specificate drepturi de acces (*permissions*)
- afisare drepturi de acces: `ls -l`

```
- rwx rwx rwx      drwxr-xr-x 2 antonia antonia 4096 Apr 25 12:09 CURS
      u     g     o          u     g     o - înseamnă fără permisiune
```

- reprezentarea simbolică a rolurilor:
 - `u` (owner) - proprietarul fișierului/directorului
 - `g` (group) - grupul din care face parte proprietarul fișierului/directorului
 - `o` (others) - utilizatorii care nu sunt în grupul din care face parte proprietarul
- reprezentarea simbolică a drepturilor de acces pentru fișiere:
 - `r` (read) - dreptul de a citi conținutul fișierului
 - `w` (write) - dreptul de a scrie în fișier
 - `x` (execute) - dreptul de a lansa în execuție fișierul

- reprezentarea simbolică a drepturilor de acces pentru directoare:
 - `r` (read) - dreptul de a citi conținutul directorului (vizualizare fișiere/sub-directoare)
 - `w` (write) - dreptul de a scrie în director (creare fișiere/sub-directoare)
 - `x` (execute) - dreptul de a intra în director

- reprezentarea numerică a drepturilor de acces:

`r (read) = 4 w (write) = 2 x (execute) = 1`

- modificarea drepturilor de acces:

`chmod 755 file` \Rightarrow $\begin{matrix} 7=4+1+1 \\ 5=4+1 \\ 5=4+1 \end{matrix}$ $\begin{matrix} u & \Leftrightarrow & rwx \\ g & \Leftrightarrow & r-x \\ o & \Leftrightarrow & r-x \end{matrix}$

`chmod +x file`

`chmod g+r file`

`chmod u+r, g+r-w, g+r file`

`chmod u=rwx, g=rw, o=r file`

2. PROGRAMARE SHELL

2.1. NOTIUNI INTRODUCTIVE

- shell = un program special care furnizează o interfață între utilizator și nucleul sistemului de operare
- shell-uri: `sh (Bourne shell)`, `csh (C shell)`, `ksh (Korn shell)`, `bash (GNU Bourne-again shell)`
- script = un fișier de tip text care conține comenzi (interne sau externe)

- un prim exemplu de script bash:

```
#!/bin/bash

pwd
ls
```

Observații:

- secvența `#!` de pe prima linie a script-ului **NU ESTE** un comentariu (este numită *shebang*)
- după această secvență este definită calea absolută către programul care trebuie rulat pentru toate celelalte linii ale script-ului

- rularea unui script bash:

```
chmod +x script_1.sh
./script_1.sh
```

- **comentarii:** încep cu caracterul `#` (hash)

- **variabile:**

- numele unei variabile poate conține litere, cifre și caracterul „`_`” (*underscore*)
- primul caracter trebuie să fie o literă
- cuvintele rezervate nu pot fi utilizate ca nume de variabile
- shell-ul face deosebire între litere mici și litere mari (*case-sensitive*)

- **exemple:**

```
n=45
name=Ana
msg="Enter a number:"
```

- **cuvinte rezervate (keywords):**

```
if then else elif fi
for while until do done
case in esac
```

- **comenzi interne (built-in commands):**

- afișarea listei cu comenzi interne: `help`
- afișarea informațiilor despre o comandă: `help command`
- **exemple:** `echo read printf test`

- **exemple de script-uri bash:**

- citirea și afișarea unui număr: `script_2.sh`
- citirea și afisarea unui sir de caractere: `script_3.sh`

- **variabile speciale:**

<code>\$0</code>	numele fișierului lansat în execuție
<code>\$1, ..., \$9</code>	argumentele furnizate în linia de comandă
<code>\$#</code>	numărul de argumente furnizate în linia de comandă
<code>\$*</code>	șirul de argumente din linia de comandă

\$@	lista individuală a argumentelor din linia de comandă
\$?	codul de ieșire (exit status) al ultimei comenzi executate
\$\$	PID-ul procesului curent
\$!	PID-ul ultimei comenzi lansate în background

- exemplu: script_4.sh

2.2. EXPRESII ARITMETICE CU NUMERE ÎNTREGI

- variabilele shell sunt tratate, în mod implicit, ca siruri de caractere
- exemplu: script_5.sh

2.2.a. Comanda expr

expr expresie

- evaluează și afișează la ieșirea standard valoarea unei expresii aritmetice cu numere întregi
- operatori:

+ - * / % Sumă, diferență, produs, cât sau modulo

= != Comparații numerice:

\> \>= – valoarea 1 dacă relația dintre s și d este adevărată

\< \<= – valoarea 0 în caz contrar

\(\)

S \| D valoarea S dacă S nu este nici NULL nici 0, valoarea D în caz contrar

S \& D valoarea S dacă S cât și D nu sunt nici NULL nici 0, 0 în caz contrar

length S Lungimea sirului S

index S CHAR\$ Poziția primei apariții în S sau 0 (numerotarea începe de la 1)

substr S P L Subșirul care începe în S pe poziția P și are lungimea L

2.2.b. Comanda let

- evaluează și afișează la ieșirea standard valoarea unei expresii aritmetice cu numere întregi
- operatori: ++ -- ! ~ ** * / % + - << >> <= >= < > == != & ^ | && ||

2.2.c. Parantezele duble

- exemplu: script_6.sh

2.3. Comanda test *test comanda* \Leftrightarrow [comanda]

- sintaxă:

test conditie sau [conditie]

- evaluează conditie și returnează 0 dacă condiția este adevărată sau o valoarea NENULĂ în caz contrar

- permite compararea numerelor întregi, compararea sirurilor de caractere sau opțiuni de verificare a fișierelor

a. Compararea numerelor întregi

- operatori: -lt -le -eq -ne -ge -gt

b. Compararea sirurilor de caractere

-z s	Verifică dacă sirul are lungimea 0
-n s1	Verifică dacă sirul are lungime nenulă
s1 = s2	Verifică dacă cele două siruri sunt egale
s1 != s2	Verifică dacă cele două siruri sunt diferite

c. Opțiuni de verificare a fișierelor

-e fisier	Verifică dacă fișierul există
-s fisier	Verifică dacă fișierul există și are lungimea nenulă
-r fisier	Verifică dacă fișierul există și dacă se poate citi
-w fisier	Verifică dacă fișierul există și dacă se poate scrie
-x fisier	Verifică dacă fișierul există și este executabil
-f fisier	Verifică dacă fișierul există și este un fișier obișnuit
-d fisier	Verifică dacă fișierul există și este un director
-L fisier	Verifică dacă fișierul există și este o legătură simbolică
-p fisier	Verifică dacă fișierul există și este un pipe
-c fisier	Verifică dacă fișierul există și este un fișier special de tip caracter
-b fisier	Verifică dacă fișierul există și este un fișier special de tip bloc

2.4. IF/THEN/ELIF/ELSE/FI

- sintaxă:

```
if condition
then
    statement(s) to be executed
elif condition
then
    statement(s) to be executed
elif condition; then
    statement(s) to be executed
else
    statement(s) to be executed
fi
```

- exemple: if_1.sh, if_2.sh

2.5. FOR/DO/DONE

- sintaxă:

```
for var in list
do
    statement(s) to be executed
done
```

- exemple: for_1.sh, for_2.sh, for_3.sh, for_4.sh, for_5.sh

- filename wildcards:



Orice secvență de caractere, chiar și vidă (nu primul punct din numele fișierului)



Un singur caracter (nu primul punct din numele fișierului)

- [abc] Orice caracter care se găsește în lista de caractere
- [!abc] Orice caracter care nu se găsește în lista de caractere

- exemplu:

- list the files whose names begin with a letter followed by a dot and exactly 2 chars:

```
ls [a-zA-Z]*.??
```

2.6. WHILE/DO/DONE, UNTIL/DO/DONE

- sintaxă:

while condition	until condition
do	do
statement(s) to be executed	statement(s) to be executed
done	done

- exemplu: while_1.sh, while_2.sh, while_3.sh, while_4.sh, while_5.sh

2.7. CASE/ESAC

- sintaxă:

```
case var in
  pattern_1
    statement(s) to be executed if pattern_1 is matched;;
  pattern_2
    statement(s) to be executed if pattern_2 is matched;;
  ...
  *)
    default condition to be executed;;
esac
```

- exemplu: case_1.sh

2.8. ALTE COMENZI UTILE

a. Comanda cut

```
cut -d: -f 1 /etc/passwd
cut -d ":" -f 1 /etc/passwd
who | cut -d " " -f 1
```

b. Comanda find directori curent toate fișierele .sh

find . -type f -name "*.sh" find nume_din_care_caută din → tot ce-i în director

```
find /tmp -type d -empty
```

c. Comanda shift

shift [n] deplasează spre stânga cu n poziții argumentele furnizate în linia de comandă

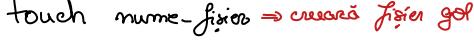
d. Comanda sleep

sleep [n] suspendă execuția procesului curent pentru n secunde

e. Comanda exit

exit [n] încheie execuția procesului curent și revine în procesul din care a fost lansat

Comenzi și explicații

1. `date`: Afisează data și ora curentă.
2. `last`: Afisează lista utilizatorilor care s-au conectat recent la sistem, împreună cu informații despre aceste conexiuni.
3. `grep`: Este utilizată pentru a căuta siruri de caractere în fișiere sau ieșirile altor comenzi. Poate fi folosită pentru a filtra ieșirile altor comenzi după un anumit şablon.
4. `sort`: Sortează liniile unui fișier sau ieșirea unei alte comenzi în ordine alfabetică sau numerică.
5. `uniq`: Elimină liniile consecutive duplicate dintr-o listă sortată. Este adesea folosită împreună cu `sort`.
6. `who`: Afisează informații despre utilizatorii care sunt conectați la sistem în acel moment.
7. `ps`: Afisează o listă a proceselor care rulează în prezent în sistem, cu detalii precum PID-ul procesului și utilizarea resurselor.
8. `finger`: Oferă informații detaliate despre utilizatorii sistemului, cum ar fi numele, directorul de pornire și altele.
9. `wc`: Afisează numărul de linii, cuvinte și caractere dintr-un fișier sau din ieșirea altor comenzi.
10. `find`: Utilizată pentru a căuta fișiere și directoare într-un sistem de fișiere pe baza unor criterii specificate.
11. `expr`: Este folosită pentru evaluarea expresiilor în shell-uri, cum ar fi adunare, scădere, înmulțire și altele. 
12. `break`: Utilizată pentru a ieși dintr-o buclă (cum ar fi `while` sau `for`) prematur, întrerupând execuția repetată a acesteia.
13. `mv`: Utilizată pentru a muta sau redenumi fișiere și directoare în sistemul de fișiere.
14. `rm`: Utilizată pentru a șterge fișiere sau directoare din sistemul de fișiere.
15. `chmod`: Este folosită pentru a schimba permisiunile fișierelor sau directoarelor în sistemul de fișiere.
16. `ls`: Afisează conținutul unui director.
17. `sed`: Este un editor de flux utilizat pentru a efectua transformări de text pe fluxuri de intrare.
18. `df`: Afisează informații despre spațiul pe disc folosit și disponibil pe diferitele sisteme de fișiere montate.
19. `awk`: Este un limbaj de procesare a textului care poate fi folosit pentru a efectua diferite operații pe fișiere text, cum ar fi căutarea, filtrarea și manipularea datelor.
20. `md5sum`: Calculează și afisează suma de control MD5 a unui fișier. Suma de control MD5 este o valoare hash de 128 de biți (16 octeți) generată pe baza conținutului fișierului. Această valoare este unică pentru fiecare conținut de fișier și este utilizată pentru a verifica integritatea datelor.
21. `head`: Afisează primele linii dintr-un fișier sau din ieșirea unei alte comenzi.
22. `tail`: Afisează ultimele linii dintr-un fișier sau din ieșirea unei alte comenzi.
 touch nume_fisier 

PROCESE

- procese care se execută simultan și numesc „concurante”

Într-un sistem de operare, există adesea mai multe procese decât unități de procesare disponibile (CPU-uri și nuclei). Aceasta ridică întrebarea cum pot toate aceste procese să fie active în același timp, dacă nu sunt suficiente unități de procesare. Iată explicația:

- a. Sistemul alocă fiecărui proces o cantitate de timp pe CPU și comută între procese după fiecare cantitate alocată de timp.
- b. Aceste intervale de timp sunt imperceptibil de mici, astfel încât toate procesele par să progreseze simultan.

Explicație

Multitasking și Time-Slicing

1. Multitasking: Sistemul de operare folosește o tehnică numită multitasking, care permite mai multor procese să pară că rulează simultan. În realitate, sistemul comută rapid între procese, oferind fiecărui un mic interval de timp pentru a utiliza CPU-ul.
2. Time-Slicing: Fiecare proces primește o "feliu" de timp (quantum de timp) pe CPU. Acest timp este foarte scurt, de obicei de ordinul milisecundelor. După ce acest timp expira, sistemul de operare comută la următorul proces în coadă.
3. Imperceptibil pentru utilizator: Aceste felieri de timp sunt atât de scurte încât utilizatorii nu observă comutările. Aceasta dă iluzia că toate procesele rulează simultan și fac progrese continuu.

Problema: Condiția de Cursă (Race Condition)

Problema: Procesele pot produce rezultate incorecte atunci când lucrează asupra acelorași resurse ca alte procese, chiar dacă în mod normal ar produce rezultate corecte. O astfel de situație se numește condiție de cursă.

Explicație:

- Resurse Partajate: Într-un sistem multitasking, mai multe procese pot accesa aceeași resurse, cum ar fi fișiere, variabile de memorie, sau alte componente partajate.
- Condiție de Cursă: Dacă două sau mai multe procese accesează și modifică aceeași resurse simultan, fără o sincronizare adecvată, pot apărea conflicte. Aceste conflicte pot duce la rezultate incorecte sau comportamente neașteptate.
- Exemplu: Gândește-te la două procese care încearcă să actualizeze aceeași variabilă în memorie. Dacă unul dintre procese citește valoarea variabilei, o modifică, iar celălalt proces face același lucru înainte ca prima modificare să fie completă, rezultatul final al variabilei poate fi incorect.

Soluții: Mecanisme de Sincronizare

3. Show and explain the race condition created by running the inc.c program below simultaneously on the same file

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

int main(int argc, char** argv) {
    int f, k, i;

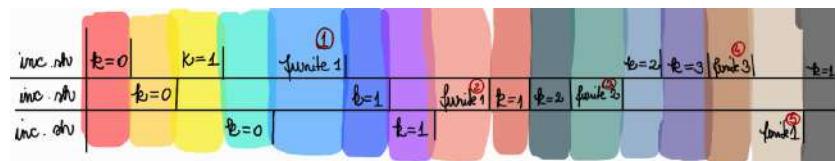
    f = open(argv[1], O_RDWR);
    if(argc > 2 && strcmp(argv[2], "reset") == 0) {
        k = 0;
        write(f, &k, sizeof(int));
        close(f);
        return 0;
    }
}

for(i=0; i<255*255; i++) {
    lseek(f, 0, SEEK_SET);
    read(f, &k, sizeof(int));
    k++;
    lseek(f, 0, SEEK_SET);
    write(f, &k, sizeof(int));
}
close(f);
return 0;
}

/* Script for simultaneous execution
#!/bin/bash

./inc b.dat reset
./inc b.dat &
./inc b.dat &
./inc b.dat &
*/
```

instrucțiunile care nu sunt atomică care crează race condition pt că pot fi întrerupte



Până la momentul curent, trebuie să fie 5 (dacă s-a executat fără întrerupere), dar poate să fie la fel de laine 1, 2, 3, 4, în funcție de cum are loc procesul.

Procese UNIX

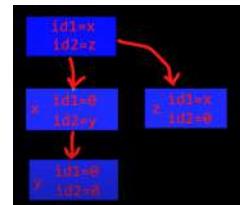
- proces = program aflat în execuție
- nucleul (kernel) SO-ului întărită permanent și tabelă cu procese: **ps -e**

Functia fork()

- prototip:

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

- crează un nou proces prin duplicarea procesului apelant (procesul părinte)
- noul proces este denumit proces copil și este o copie aproape exactă a procesului părinte
- cele două procese își continuă execuția cu instrucțiunea care urmează apelului **fork()**
- returnează:
 - valoarea 0 - în procesul copil
 - identifierul procesului copil (child PID) - în procesul părinte → ex: $PID=1612, \bar{PID}=1613, \dots$
 - valoarea -1 - dacă apelul a eșuat
- apelul funcției **fork()** eșuează în următoarele situații:
 - nu există spațiu de memorie suficient pentru duplicarea procesului părinte
 - numărul total de procese depășește limita maximă admisă



`getpid();` → returnează pid-ul procesului care a apelat
`getppid();` → returnează pid-ul părintelui procesului care a apelat;

Functiile wait(), waitpid()

- prototipuri:

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

- **wait()** suspendă execuția procesului apelant până la terminarea unui proces copil
- apelul **wait(&status)** este echivalent cu **waitpid(-1, &status, 0)**

↪ nu putem să garantăm ordinea dacă avem nevoie de wait

if($\bar{PID} \neq 0$) `wait();` → ne asigurăm că suntem în procesul părinte/principal

sleep(m); ← procesul așteaptă

int main() {
 id1 = fork();
 id2 = fork();
 cout << getpid();
}

- `waitpid()` suspendă execuția procesului apelant până la apariția unuia dintre următoarele evenimente:
 - procesul fiu specificat prin argumentul `pid` și-a terminat execuția
 - procesul fiu specificat prin argumentul `pid` a fost opriț printr-un semnal
 - procesul fiu specificat prin argumentul `pid` a fost repornit printr-un semnal
- semnificația valorilor argumentului `pid`:

<code>pid</code>	Semnificație
< -1	Se așteaptă terminarea tuturor proceselor copil al căror identificator de grup (GID) este egal cu valoarea absolută a parametrului <code>pid</code>
-1	Se așteaptă terminarea tuturor proceselor copil
0	Se așteaptă terminarea tuturor proceselor copil al căror identificator de grup (GID) este egal cu GID-ul procesului părinte
> 0	Se așteaptă terminarea procesului cu PID-ul specificat prin parametrul <code>pid</code>

```
// Using wait() system call (man 2 wait)
//
// If the parent process calls wait() system call,
// then the execution of parent is suspended until the child is terminated.
//

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

int main(int argc, char* argv[])
{
    int pid = fork();
    if (pid == -1)           // fork() failed
    {
        perror("fork() error: ");
        exit(EXIT_FAILURE);
    }

    if (pid == 0)            // in child process
    {
        printf("[In CHILD] I started ...\\n");
        printf("[In CHILD]\\tMy PID is %d Parent PID is %d\\n", getpid(), getppid());
        sleep(10);
        printf("[In CHILD]\\tI AM NOT an ORPHAN process ...\\n");
        printf("[In CHILD]\\tMy PID is %d Parent PID is %d\\n", getpid(), getppid());
        printf("[In CHILD] I finished my job.\\n");
    }
    else                     // in parent process
    {
        printf("[In PARENT] I started ...\\n");
        printf("[In PARENT]\\tMy PID is %d Child PID is %d\\n", getpid(), pid);
        int status;
        wait(&status);
        printf("[In PARENT]\\tchild has finished with exit status: %d\\n", status);
        printf("[In PARENT] I finished.\\n");
    }

    return 0;
}
```

```
/*
// A ZOMBIE process
//
// If the parent decides not to wait for the child's termination and it executes its subsequent task,
// then at the termination of the child, the exit status is not read.
// Hence, there remains an entry in the process table even after the termination of the child.
// This state of the child process is known as the ZOMBIE state.
//

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

int main(int argc, char* argv[])
{
    int pid = fork();
    if (pid == -1)           // fork() failed
    {
        perror("fork() error: ");
        exit(EXIT_FAILURE);
    }

    if (pid == 0)            // in child process
    {
        printf("\\n[In CHILD] I started ...\\n");

        int i;
        for (i = 0; i < 3; i++)
        {
            sleep(5);
            printf("[In CHILD]\\tMy PID is %d Parent PID is %d\\n", getpid(), getppid());
        }

        printf("[In CHILD] I finished my job.\\n");
        printf("\\n[In CHILD]\\tI am in the ZOMBIE state now ...\\n");
        printf("[In CHILD]\\tcould you killed me, please ?\\n");
    }
    else                     // in parent process
    {
        printf("[In PARENT] I started ...\\n");
        printf("[In PARENT]\\tMy PID is %d Child PID is %d\\n", getpid(), pid);
        while(1);
        printf("[In PARENT] I finished.\\n");
    }

    return 0;
}
```

Procese ZOMBIE

- dacă procesul copil se termină înainte ca părintele să apeleze `wait`, să time procesul copil într-un „zombie state” (nu execută nimic, dar este în lista de procese [dormit])
- un proces zombie și termină când părintele apelează `wait` sau `waitpid`
- ! ca să eviti procesele zombie, apelează `wait` pentru fiecare proces copil pe care-l creazi
- `wait` → așteaptă să se termine orice proces copil, `waitpid` pentru unul specific

Functia signal();

- prototip:

```
#include <signal.h>

sighandler_t signal(int signum, sighandler_t handler);
```

- stabilește modul de acțiune la apariția unui semnal
- semnale:
 - man 7 signal
- dacă semnalul *signum* poate fi livrat unui proces, atunci acesta poate să aleagă:
 - să ignore semnalul - SIG_IGN
 - să-l trateze în mod implicit - SIG_DFL
 - să specifică o funcție care definește acțiunile care se execută la apariția semnalului
- prevenirea apariției proceselor de tip „zombie”:
 - signal(SIGCHLD, SIG_IGN)
 - nu summașă și doar înregistrează un handler
 - (ex): CTRL C ⇒ trimite signal 2 (SIGINT) ⇒ stop process

Functia kill();

- prototip:

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

- permite livrarea unui semnal unui proces sau grup de procese
- semnificația valorilor argumentului *pid*:

pid	Semnificație
> 0	Semnalul este livrat procesului <u>cu PID-ul specificat</u> prin parametrul <i>pid</i>
0	Semnalul este livrat tuturor proceselor al căror identificator de grup (GID) <u>este egal cu GID-ul procesului apelant</u>
-1	Semnalul este livrat <u>tuturor proceselor</u> pentru care procesul apelant are dreptul de a livra semnale (cu excepția procesului <i>init</i>)
< -1	Semnalul este livrat tuturor al căror identificator de grup (GID) <u>este egal cu valoarea absolută a parametrului pid</u>
- semnale:
 - man 7 signal

6. FAMILIA DE FUNCȚII exec()

- prototipuri:

```
#include <unistd.h>

int execl(const char *path, const char *arg, ...);
int execvp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char *const envp[]);

int execv(const char *path, char *const argv[]);
int execvpe(const char *file, char *const argv[]);
int execve(const char *file, char *const argv[], char *const envp[]);
```

- lansează în execuție un nou program (fișier executabil)
- imaginea în memorie a procesului curent este înlocuită cu imaginea celui lansat în execuție
- exemple: exec_1.c, exec_2.c, exec_3.c

1. To run another program from an existing process, the UNIX system offers the exec system calls. There are six of them, but we will only present four of them, and use probably just one or two. The table below shows the four variants differing by whether the program arguments are an array or specified directly as function arguments, and whether the program is given with an absolute path or whether it should be searched for in the PATH.

		Search PATH for the program	
		Yes	No
Arguments passed as	Array	<pre>char* a[] = {"grep", "-E", "/an1/gr911/", "/etc/passwd", NULL}; execvp("grep", a);</pre>	<pre>char* a[] = {"/bin/grep", "-E", "/an1/gr911/", "/etc/passwd", NULL}; execv("/bin/grep", a);</pre>
	List	<pre>execvp("grep", {"grep", "-E", "/an1/gr911/", "/etc/passwd", NULL});</pre>	<pre>execl("/bin/grep", {"bin/grep", "-E", "/an1/gr911/", "/etc/passwd", NULL});</pre>

2. What exactly is the PATH? A UNIX Shell variable (you can however find it in Windows under same name) that contains paths where the Shell should look for the program you run, unless you specify an absolute or relative path like ./myprog
3. Unexpected behavior: the exec system calls re-use the current process to run the other program. Essentially they wipe out the current process code, and replace it with the code of the new program. If the exec call fails (usually due to the program not being found) the calling process continues to execute.

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

int main(int argc, char** argv) {
    execvp("grep", {"grep", "-E", "/an1/gr211/", "/etc/passwd", NULL});
    printf("If grep is in the PATH, then execvp succeeds, and this will never be printed.\n");
    return 0;
}
```

! dacă se execuțiază cu succes ceea ce din familia exec atunci nu se execuțiază nimic de după (soluție: nuții în fără)

4. As the first argument of a program (argument 0) is always the command name, we need to pass that argument explicitly, hence the duplication of the command name.
5. The last NULL argument is required in order to mark the end of the arguments.

3. Procese Unix (în C): fork, exec, exit, wait system, signals

Contents

3.	PROCESE UNIX (ÎN C): FORK, EXEC, EXIT, WAIT SYSTEM, SIGNALS1
3.1.	STANDARDUL POSIX DE GESTIUNE A ERORILOR ÎN APELURI SISTEM: ERRNO.....	1
3.2.	PRINCIPALELE APELURI SISTEM UNIX CARE OPEREAZĂ CU PROCESE.....	1
3.3.	EXEMPLE DE LUCRUL CU PROCESE.....	2
3.3.1.	<i>Utilizări simple fork, exit, wait</i>	2
3.3.2.	<i>Utilizări simple exec, execp, execv, system</i>	5
3.3.3.	<i>Un program care compileaza și rulează alt program</i>	6
3.3.4.	<i>Capitalizarea cuvintelor dintr-o listă de fișiere text</i>	7
3.3.5.	<i>Câte perechi de argumente au suma un număr par?</i>	8
3.4.	SEMNALE UNIX; EXEMPLE DE UTILIZARE	10
3.4.1.	<i>Evitarea proceselor zombie.....</i>	10
3.4.2.	<i>Schema client / server: adormire și deșteptare</i>	10
3.4.3.	<i>Aflarea unor informații de stare</i>	11
3.4.4.	<i>Tastarea unei linii în timp limitat.....</i>	11
3.4.5.	<i>Blocarea tastaturii</i>	12
3.5.	PROBLEME PROPUSE.....	12

3.1. Standardul POSIX de gestiune a erorilor în apele sisteme: errno

Marea majoritate a funcțiilor C și practic toate apelele sistem Unix întorc un rezultat care "spune" dacă funcția/apelul s-a derulat normal sau dacă a apărut o situație deosebită. În caz de eșec funcția / apelul sistem întoarce fie un întreg nenul (valoarea 0 este rezervată pentru succes), fie un pointer NULL etc. Metodologic, se recomandă **SA SE APELEZE**:

```
if (functie( - - - ) == SUCCES) { tratare normala }
else      { tratare situație de derulare anormală }
```

NU (așa cum din comoditate apeleză mulți programatori) :

```
functie( - - - ); tratare normala
```

Evident forma a doua este mai scurtă, dar nu sunt tratate situațiile de excepție.

Pentru o abordare unitară a acestor tratamente, standardul POSIX oferă prin **#include <errno.h>** o variabilă întreagă **errno**, (care nu trebuie declarată) a carei valoare este setată de sistem nunai în caz de derulare anormală a apelului! La o situație de derulare anormală sistemul fixează o valoare nenulă ce indică cauza erorii. Pentru detalii vezi man **errno**, precum și lista completă a cazurilor de erori, aflată de exemplu la <http://www.virtsync.com/c-error-codes-include-errno>

La apelul cu succes al unei funcții sistem errno nu se setează la 0! Pentru a se vedea și în clar eroarea depistată se pot folosi funcțiile **strerror** și **perror** dău detalii pentru fiecare valoare a lui **errno**:

```
#include <errno.h>
- -
if (functie( - - - ) == SUCCES) { tratare normala }
else      { perror("Eroarea depistată este:");
            // sau printf("Eroarea depistată este:%s", strerror(errno); }
```

3.2. Principalele apele sisteme care operează cu procese

3 Unix fork_exec_wait_exit_system

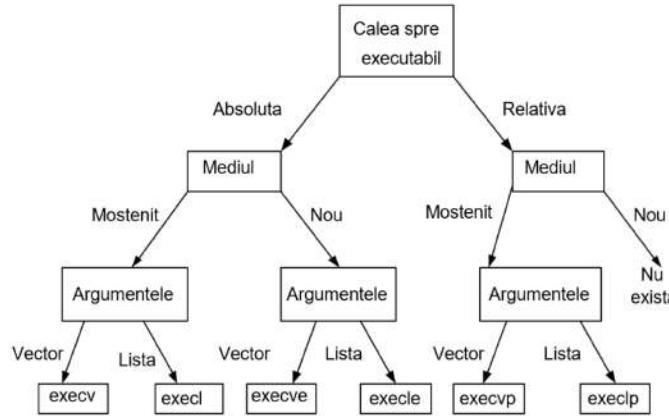
Page 2 of 13

2

Tabelul următor prezintă sintaxele principalelor apeluri sistem Unix care operează cu procese:

Funcții specifice proceselor
fork()
exit(n)
wait(p)
exec*(c, lc)
system(c)

Tipurile de exec:



Prototipurile lor sunt descrise, de regula, în <unistd.h>. Parametrii sunt:

- n este intreg – codul de return cu care se termină procesul;
- p este un pointer la un întreg unde fiul întoarce codul de return (extras cu funcția WEXITSTATUS);
- c este o comandă Unix;
- lc este linia de comandă (comanda c urmată de argumentele liniei de comandă);
- f este un tablou de doi intregi – descriptori de citire / scriere din / în pipe;
- nume este numele (de pe disc) al fișierului FIFO, iar drepturi sunt drepturile de acces la acesta;
- fo și fn descriptori de fisiere: fo deschis în program cu open, fn poziția în care e duplicat fo.

In caz de eșec, funcțiile întorc -1 (NULL la fopen) și poziționează errno se depistează ce eroare a apărut.

Funcțiile system și popen au comanda completă (un string), interpretabilă de shell: aceste funcții lansează mai întâi un shell, apoi în acesta lansează comanda c. Această lansare se face simplu folosind un apel sistem execl: `execl("/bin/sh", "sh", "-c", c, NULL);`

Comanda c din apelurile exec* NU permite specificări folosite uzuale în liniile shell. Astfel, în c NU trebuie să apară specificări generice de fișiere, redirectări de intrări / ieșiri standard, variabile shell, captări de ieșiri prin construcții ` - - comanda - - ` etc. Dacă totuși se dorește acest lucru, trebuie să se lanseze, ca mai sus, interpretorul sh cu opțiunea -c și apoi să se specifică comanda.

3.3. Exemple de lucru cu procese

3.3.1. Utilizări simple fork, exit, wait

Vom prezenta și discuta două exemple de programe care utilizează apelurile sistem fork, exit, wait. Să considerăm **programul f1.c** căruia i-am numerotat liniile sursă:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5 int main() {
6     int p, i;
7     p=fork();
8     if (p == -1) {perror("fork imposibil!"); exit(1);}
9     if (p == 0) {
10         for (i = 0; i < 10; i++)
11             printf("Fiu: i=%d pid=%d, ppid=%d\n", i, getpid(), getppid());
12         exit(0);
13     } else {
14         for (i = 0; i < 10; i++)
15             printf("Parinte: i=%d pid=%d ppid=%d\n", i, getpid(), getppid());
16         wait(0);
17     }
18     printf("Terminat; pid=%d ppid=%d\n", getpid(), getppid());
19 }

```

↳ se execută în copil pt că are PID returnat = 0

↳ se execută în părinte

Vom analiza comportamentul acestui program în diverse situații, făcând o serie de modificări în această sursă.

Rularea în forma inițială: Sunt afișate 21 linii: 10 ale fiului de la linia 11 cu pidul lui și al părintelui. 11 ale fiului, 10 de la linia 15 și ultima de la linia 18. Părintele părintelui este pidul shell. Este posibil ca ordinea primelor 20 de linii să apară amestecate, linii ale fiului și liniile ale părintelui. Dacă la linia 10 și la linia 14 se înlocuiește 10 cu 1000, se vor afișa 2001 linii iar amestecarea între liniile fiului și ale părintelui va fi mai evidentă.

Comentarea liniei 12: Procesul fiu se termină la linia 18, ca și părintele. Se vor tipări 22 linii, linia 18 se va tipări de două ori: odată de părinte și odată de fiu.

Comentarea liniei 16: Părintele nu mai așteaptă terminarea fiului și acesta din urmă rămâne în starea zombie. Se tipăresc cele 21 de linii ca în primul caz. O observație interesantă: dacă ieșirea programului se redirecțiază într-un fișier pe disc, apar cele 21 linii. În schimb, dacă ieșirea se face direct pe terminal, apar doar liniile fiului. De ce oare? Rămâne un TO DO pentru studenți.

Comentarea liniilor 12 și 16: Se tipăresc 22 linii, cu aceeași observație de mai sus, de la comentarea liniei 16. Aici recomandăm modificări ale numărului liniilor tipărite de fiu (linia 10) și a celor tipărite de părinte (linia 14). Se vor vedea efecte interesante.

Să considerăm **programul f2.c**:

```
main() { fork(); if (fork()) {fork();} printf("Salut\n");}
```

Care este efectul execuției acestui program? (Acoladele nu sunt necesare, dar le-am pus pentru a evidenția mai bine corpul lui if). Să facem o primă analiză:

- Primul fork naște un proces fiu. Ambele procese au de executat secvența: if (fork()) fork(); printf("Salut\n");
- Condiția fork din if mai naște câte un proces fiu cărora le rămâne de făcut doar printf("Salut\n"); In același timp, cele două procese care evaluatează if mai au de făcut {fork();} printf("Salut\n"); Până aici avem patru procese.
- Fiecare fork dintre accolade mai naște câte un proces fiu căruia îi mai rămâne de făcut printf("Salut\n"); Avem încă două procese în plus.
- În concluzie, avem șase (6) procese care au de executat printf("Salut\n"); In consecință, se va tipări de 6 ori Salut.

Merită să studiem mai atent acest exemplu. Principala carență a lui este aceea că nici un părinte care naște un fiu nu așteaptă terminarea lui prin wait. Consecință, vor rămâne câteva procese în starea zombie.

3 Unix_fork_exec_wait_exit_system

Page 4 of 13

4

Pentru a aprofunda analiza, să rescriem puțin programul f2.c:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
    printf("START: pid=%d ppid=%d\n", getpid(), getppid());
    int i=-2, j=-2, k=-2;
    i=fork();
    if (j=fork())
        {k=fork();}
    printf("Salut pid=%d ppid=%d i=%d j=%d k=%d\n",getpid(),getppid(),i,j,k);
}
```

In fapt, am reținut în variabilele *i*, *j*, *k* valorile PID-urilor create pe parcursul execuției. Rezultatul execuției este:

```
START: pid=3998 ppid=3810
Salut pid=3998 ppid=3810 i=3999 j=4000 k=4001
florin@ubuntu:~/c$ Salut pid=4001 ppid=1700 i=3999 j=4000 k=0
Salut pid=4000 ppid=1700 i=3999 j=0 k=-2
Salut pid=3999 ppid=1700 i=0 j=4002 k=4003
Salut pid=4003 ppid=1700 i=0 j=4002 k=0
Salut pid=4002 ppid=1700 i=0 j=0 k=-2
```

Să analizăm ordinea în care se execută aceste instrucțiuni:

- 3810 este PID-ul shell care afișează prompterul, iar 3998 este PID-ul programului inițial.
- Procesul 3998 crează fiul *i* cu PID-ul 3999, fiul *j* cu PID-ul 4000 și fiul *k* cu PID-ul 4001. Apoi își face tipărire și se termină - se vede tipărirea prompterului.
- Cele trei procese 3999, 4000 și 4001 rămân active dar sunt în starea zombie (PPID-ul lor este 1700).
- Procesul 4001 preia controlul procesorului, valorile *i* și *j* sunt moștenite de la 3998, iar *k* = 0 fiind vorba de fork în fiu, face tipărirea și se termină.
- Procesul 4000 preia controlul procesorului, valorile *i* și *k* sunt moștenite de la 3998 - *i* creat, *k* încă necreat, iar *j* = 0 fiind vorba de fork în fiu, face tipărirea și se termină.
- Procesul 3999 preia controlul procesorului, *i* = 0 fiind vorba de fork în fiu, crează fiul *j* cu PID-ul 4002 și fiul *k* cu PID-ul 4003. Apoi își face tipărirea și se termină.
- Procesul 4003 preia controlul procesorului, valorile *i* și *j* sunt moștenite de la 3999, iar *k* = 0 fiind vorba de fork în fiu. Apoi își face tipărirea și se termină.
- Procesul 4002 preia controlul procesorului, valorile *i* și *k* sunt moștenite de la 3999, iar *j* = 0 fiind vorba de fork în fiu. Apoi își face tipărirea și se termină.

Tabelul următor prezintă cele 6 procese: ce valori moștenesc de la părinte, ce cod mai au de executat și ce valori finale au (ce tipăresc).

PID 3998 PPID 3810 <i>i</i> =-2 <i>j</i> =-2 <i>k</i> =-2 <i>i</i> =fork(); if (<i>j</i> =fork()) { <i>k</i> =fork();} printf --- <i>i</i> =3999 <i>j</i> =4000 <i>k</i> =4001				
	PID 4001 PPID 3998 <i>i</i> =3999 <i>j</i> =4000 <i>k</i> =0 printf --- <i>i</i> =3999 <i>j</i> =4000 <i>k</i> =0	PID 4000 PPID 3998 <i>i</i> =3999 <i>j</i> =0 <i>k</i> =-2 printf --- <i>i</i> =3999 <i>j</i> =0 <i>k</i> =-2	PID 3999 PPID 3998 <i>i</i> =0 <i>j</i> =-2 <i>k</i> =-2 if (<i>j</i> =fork()) { <i>k</i> =fork();} printf --- <i>i</i> =0 <i>j</i> =4002 <i>k</i> =4003	

5

				PID 4003 PPID 3999 i=0 j=4002 k=0 printf - - - i=0 j=4002 k=0	PID 4002 PPID 3999 i=0 j=0 k=-2 printf - - - i=0 j=0 k=-2
--	--	--	--	---	---

In acest tabel, valoarea PPID este cea reală a părintelui creator, deși la momentul terminării fiului părintele nu mai există, așa că procesul intră în starea zombie.

Pentru a evita starea acumularea de procese în starea zombie, se poate folosi, spre exemplu, secvența:

```
---  
#include <signal.h>  
int main() {  
    signal(SIGCHLD, SIG_IGN);  
---
```

In acest fel se cere ignorarea trimiterii de către fiu a semnalului SIGCHLD, pe care părintele ar trebui să îl primească (să fie în viață), să îl trateze cu un wait. Prin această ignorare, procesul fiu fiu este sters din sistem imediat după terminarea lui.

3.3.2. Utilizări simple execl, execlp, execv, system

Urmatoarele două programe, desă diferențe, au același efect. Toate trei folosesc o comandă de tip exec, spre a lansa din ea comanda shell:

```
ls -l
```

Programul 1:

```
#include <stdio.h>  
#include <unistd.h>  
int main() {  
    char* argv[3];  
    argv[0] = "/bin/ls";  
    argv[1] = "-l";  
    argv[2] = NULL;  
    execv("/bin/ls", argv);  
}
```

Aici se pregătește linia de comandă în vectorul argv spre a o lansa cu execv.

Programul 2:

```
#include <stdio.h>  
#include <unistd.h>  
#include <stdlib.h> // trebuie pentru system  
int main() {  
    //execl("/bin/ls", "/bin/ls", "-l", NULL);  
    // execlp("ls", "ls", "-l", NULL);  
    // execl("/bin/ls", "/bin/ls", "-l", "pl.c", "execl.c", "fork1.c", "xx", NULL);  
    // execl("/bin/ls", "/bin/ls", "-l", "*.*", NULL);  
    system("ls -l *.c");  
}
```

Aici se executa, pe rand, numai una dintre cele 5 linii, comentând pe celelalte 4. Ce se va întâmpla?

- Primul execl lansează ls prin cale absolută și are același efect ca și programul 1.
- Al doilea lansează ls prin directoarele din PATH, efectul este același.
- Al treilea cere ls pentru o listă de fișiere. Pentru cele care nu există, se dă mesajul: /bin/ls: cannot access 'xx': No such file or directory (în loc de xx apar numele fișierelor inexistente);

6

- Al patrulea exec va da mesajul: /bin/ls: cannot access *.c: No such file or directory
Nu este interpretat asa cum ne-am asteptă! De ce? Din cauza faptului ca specificarea *.c reprezinta o specificare generica de fisier, dar numai shell "stie" acest lucru si el (shell) inlocuieste aceasta specificare, in cadrul uneia dintre etapele de tratare a liniei de comanda. La fel stau lucrurile cu evaluarea variabilelor de mediu, \${---}, inlocuirea dintre apostroafele inverse ` --- ', redirectarea I/O standard etc.
- Apelul system are efectul așteptat, făcând rezumatul tuturor fișierelor de tip c din directorul curent.

Să ne oprim puțin asupra funcției system. Ea creaază prin fork un proces fiu, în care lansează comanda execl("/bin/sh", "sh", "-c", c, NULL) aşa cum am arătat mai sus și întoarce codul de return cu care s-a terminat execuția comenzii. Doritorii pot să vadă sursa system.c, care este o funcție simplă, de maximum 100 linii în care se includ comentariile, tratările cu errno ale posibilelor erori și manevrarea unor semnale specifice. Sursa poate fi găsită la: <http://man7.org/tlpi/code/online/dist/procexec/system.c>

3.3.3. Un program care compileaza și rulează alt program

Exemplul care urmeaza are același efect ca și scriptul sh:

```
#!/bin/sh
if gcc -o ceva $1
then ./ceva $*
else echo "Erori de compilare"
fi
```

Noi nu îl vom implementa în sh, ci vom folosi programul compilerun.c.

```
// Similar cu scriptul shell:
// #!/bin/sh
// if gcc -o ceva $1; then ./ceva $*
// else echo "Erori de compilare"
// fi
#include<stdio.h>
#include<unistd.h>
#include <stdlib.h>
#include<string.h>
#include <sys/wait.h>
int main(int argc, char* argv[]) {
    char comp[200];
    char* run[100];
    int i;
    strcpy(comp, "gcc -o ceva ");
    strcat(comp, argv[1]); // Fabricat comanda de compilare
    if (WEXITSTATUS(system(comp)) == 0) {
        run[0] = "./ceva";
        for (i = 1; argv[i]; i++) run[i] = argv[i];
        run[i] = NULL; // Fabricat comanda pentru execv
        execv("./ceva", run);
    }
    printf("Erori de compilare\n");
}
```

Compilarea lui se face

```
gcc -o compilerun compilerun.c
```

Executia se face, de exemplu, prin

```
./compilerun argvenvp.c a b c
```

Cefect, daca compilarea sursei argument (`argvenvp.c`) este corecta, atunci compilatorul gcc creeaza fisierul `ceva` si intoarce cod de return 0, dupa ceva este lansat prin `execv`. Daca esueaza compilarea, se va tipari doar mesajul "Erori de compilare".

Am ales ca si exemplu de program `argvenvp.c`:

```
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[], char *envp[]) {
    int i;
    printf("Argumentele:\n");
    for (i = 1; argv[i]; i++) printf("%s\n", argv[i]);
    printf("Cateva variabile de mediu:\n");
    for (i = 0; envp[i]; i++)
        if (strncmp("HOME", envp[i], 4)==0 || strncmp("LOGNAME", envp[i], 7)==0)
            printf("%s\n", envp[i]);
}
```

Secventa de executie este:

```
florin@ubuntu:~/c$ gcc -o compilerun compilerun.c
florin@ubuntu:~/c$ ./compilerun argvenvp.c a b c
Argumentele:
argvenvp.c
a
b
c
Cateva variabile de mediu:
HOME=/home/florin
LOGNAME=florin
florin@ubuntu:~/c$
```

3.3.4. Capitalizarea cuvintelor dintr-o listă de fișiere text

Se cere un program care primește la linia de comandă o listă de fișiere text. Se cere ca toate aceste fișiere să fie transformate în altele, cu același conținut, dar în care fiecare cuvânt să înceapă cu literă mare. Se vor lansa procese paralele pentru prelucrarea simultană a tuturor fișierelor.

Pentru aceasta, vom crea mai întâi un program cu numele `cap` din sursa `cap.c`. Acesta primește la linia de comandă numele a două fișiere text, primul de intrare, al doilea de ieșire cu cuvintele capitalizate:

```
#include <stdio.h>
#include <string.h>
#define MAXLINIE 100
main(int argc, char* argv[]) {
    printf("Fiu: %d ...> %s %s\n", getpid(), argv[1], argv[2]);
    FILE *fi, *fo;
    char linie[MAXLINIE], *p;
    fi = fopen(argv[1], "r");
    fo = fopen(argv[2], "w");
    for ( ; ; ) {
        p = fgets(linie, MAXLINIE, fi);
        linie[MAXLINIE-1] = '\0';
        if (p == NULL) break;
        if (strlen(linie) == 0) continue;
        linie[0] = toupper(linie[0]); // Pentru cuvantul care incepe in coloana 0
        for (p = linie; ; ) {
            p = strstr(p, " ");
            if (p == NULL) break;
            p++;
        }
        fprintf(fo, "%s", linie);
    }
}
```

```
8
```

```

        if (*p == '\n') break;
        *p = toupper(*p); // Caracterul de dupa spatiu este facut litera mare
    }
    fprintf(fo, "%s", linie);
}
fclose(fo);
fclose(fi);
}

```

Al doilea program, numit master.c va crea câte un proces pentru fiecare nume de fișier primit la linia de comandă și în acel proces va lansa cap fi fi.CAPIT

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
main(int argc, char* argv[]) {
    int i, pid;
    char argvFiu[200];
    for (i=1; argv[i]; i++) {
        pid = fork();
        if (pid == 0) {
            strcpy(argvFiu, argv[i]);
            strcat(argvFiu, ".CAPIT");
            execl("./cap", "./cap", argv[i], argvFiu, NULL);
        } else
            printf("Parinte, lansat fiul: %d ...> %s %s \n", pid, argv[i], argvFiu);
    }
    for (i=1; argv[i]; i++) wait(NULL);
    printf("Lansat simultan %d procese de capitalizare\n", argc - 1);
}
```

Compilari:

```
>gcc -o cap cap.c
>gcc -o master master.c
```

Lansare master f1 f2 ... fi ... fn

3.3.5. Câte perechi de argumente au suma un număr par?

La linia de comandă se dau n perechi de argumente despre care se presupune ca sunt numere întregi și positive. Se cere numărul de perechi care au suma un număr par, numărul de perechi ce au suma număr impar și numărul de perechi în care cel puțin unul dintre argumente nu este număr strict pozitiv.

Rezolvarea: În procesul părinte se va crea câte un proces fiu pentru fiecare pereche. Oricare dintre fiu întoarce codul de return:

- 0 dacă perechea are suma pară,
- 1 dacă suma este impară,
- 2 dacă unul dintre argumente este nul sau nenumeric.

Părintele așteaptă terminarea fiilor și din codurile de return întoarse de aceștia va afisa rezultatul cerut.

Vom da două soluții:

1. Solutia 1 cu textul complet într-un singur fisier sursă
2. Solutia 2 cu două texte sursă și unul să îl apeleze pe celalalt prin exec.

Solutia 1:

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

```
9
```

```

main(int argc, char* argv[]) {
    int pare = 0, impare = 0, nenum = 0, i, n1, n2;
    for (i = 1; i < argc-1; i += 2) {
        if (fork() == 0) {
            n1 = atoi(argv[i]); // atoi intoarce 0
            n2 = atoi(argv[i+1]); // si la nenumeric
            if (n1 == 0 || n2 == 0) exit(2);
            exit ((n1 + n2) % 2);
        }
    }
    for (i = 1; i < argc-1; i += 2) {
        wait(&n1);
        switch (WEXITSTATUS(n1)) {
            case 0: pare++; break;
            case 1: impare++; break;
            default: nenum++;
        }
    }
    printf("Pare %d, impare %d, nenumeric %d\n", pare, impare, nenum);
}

```

Solutia 2:

Se creaza programul `par.c` care primește la linia de comandă o pereche de argumente. Din această sursă se va constui prin `gcc -o par par.c` executabilul `par`:

```

main(int argc, char *argv[]) {
    int n1, n2;
    n1 = atoi(argv[1]); // atoi intoarce 0
    n2 = atoi(argv[2]); // si la nenumeric
    if (n1 == 0 || n2 == 0) exit(2);
    exit ((n1 + n2) % 2);
}

```

Se creaza programul `master.c` care primește la linia de comandă n perechi de argumente. El va crea n procese fii și în fiecare va lansa prin `exec` programul `par`. Din aceasta sursă se va constui prin `gcc -o master master.c` executabilul `master`:

```

main(int argc, char* argv[]) {
    int pare = 0, impare = 0, nenum = 0, i, n1, n2;
    for (i = 1; i < argc-1; i += 2) {
        if (fork() == 0) {
            execl("./par", "./par", argv[i], argv[i+1], NULL);
        }
    }
    for (i = 1; i < argc-1; i += 2) {
        wait(&n1);
        switch (WEXITSTATUS(n1)) {
            case 0: pare++; break;
            case 1: impare++; break;
            default: nenum++;
        }
    }
    printf("Pare %d, impare %d, nenumeric %d\n", pare, impare, nenum);
}

```

Intrebare la ambele solutii: Ce se întamplă daca `wait` și `switch` nu sunt plasate în cicluri `for` succesive ci în același `for` care crează procesele fii?

10

3.4. Semnale Unix; exemple de utilizare

3.4.1. Evitarea proceselor zombie

Versiunea Unix System V Release 4 și Linux: este suficient ca în partea de initializare, înainte de crearea proceselor ce pot deveni zombie:

```
#include <signal.h>
-----
signal(SIGCHLD, SIG_IGN);
```

Versiunea Unix BSD, efectul unui apel sistem signal este valabil o singură dată:

```
#include <signal.h>
-----
void waiter() { // Functie de manipulare a apelurilor signal
    wait(0); // Sterge fiul recent terminat
    signal(SIGCHLD, waiter); // Reinstalare handler signal
} // waiter
-----
signal(SIGCHLD, waiter); // Plasat în partea de initializare
```

3.4.2. Schema client / server: adormire și deșteptare

Programul *server* este pus în adormire și va fi trezit de fiecare *client* ca să-i satisfacă o cerere, după care intră din nou în adormire. Prezentăm două variante de server: *iterativ* și *concurrent*.

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
main() {
    for ( ; ; ) {
        printf("%d doarme . . .\n", getpid());
        kill(getpid(), SIGSTOP); // Doarme, asteptand cereri
}
```

Server iterativ	Server concurrent
<pre>printf("Servesc cererea . . ."); serveșteCerereClient(. . .);</pre>	<pre>if (fork() == 0) { printf("Servesc cererea. . ."); serveșteCerereClient(. . .); }</pre>

Trezirea se poate face fie printr-o comandă:

```
$ kill -SIGCONT pidserver
```

fie printr-un program client de forma:

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
main(int argc, char ** argv) {
    int pidserver = atoi(argv[1]);
    // -- Clientul află pid-ul serverului și prepară o cerere
    kill(pidserver, SIGCONT); // Desteptarea, server!
    // -- Clientul tratează răspunsul
}
```

II

3.4.3. Aflarea unor informații de stare

Pentru un program care dureaza mult, se vrea din cand in cand sa se afle stadiul calculelor:

```

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

// informatii globale de stare
int numar;

void tipareste_stare(int semnal) {
// Tipareste informatiile de stare solicitate
    printf("Numar= %d\n", numar);
}//handlerul de semnal

main() {
    signal(SIGUSR1,tipareste_stare);
    // - -
    for(numar=0; ; numar++) {
        // - -
        //for
    }//main
}

```

Pentru tipărirea stadiului curent, cunoscând (ps) pidul programului, se dă comanda:

```
$ kill -SIGUSR1 pid
```

3.4.4. Tastarea unei linii în timp limitat

SE cere ca tastarea unei linii de la terminal să se facă în timp limitat (în cazul nostru 5 secunde), altfel se anulează citirea și programul se aduce în starea dinaintea lansării citirii:

```

#include <stdio.h>
#include <setjmp.h>
#include <sys	signal.h>
#include <unistd.h>
#include <string.h>

jmp_buf tampon;

void handler_timeout (int semnal) {
    longjmp (tampon, 1);
}//handler_timeout

int t_gets (char *s, int t) {
    char *ret;
    signal (SIGALRM, handler_timeout);
    if (setjmp (tampon) != 0)
        return -2;
    alarm (t);
    ret = fgets (s, 100, stdin);
    alarm (0);
    if (ret == NULL)
        return -1;
    return strlen (s);
}//t_gets

main () {
    char s[100];
    int v;
    while (1) {

```

```
12
```

```

printf ("Introduceti un string: ");
v = t_gets (s, 5);
switch (v) {
    case -1:
        printf ("\nSfarsit de fisier\n");
        return(1);
    case -2:
        printf ("timeout!\n");
        break;
    default:
        printf("Sirul dat: %s are %d caractere\n", s, v-1);
    } //switch
} //while
} //t_gets.c

```

3.4.5. Blocarea tastaturii

Se blochează tastatura până când se tastează parola cu care s-a făcut login:

```

#define _XOPEN_SOURCE
#include <stdio.h>
//#include <sys/types.h>
#include <unistd.h>
#include<string.h>
#include <pwd.h>
#include <shadow.h>
#include <signal.h>
main() {
    char *cpass, pass[15];
    struct passwd *pwd;
    struct spwd *shd;
    signal(SIGHUP, SIG_IGN);
    signal(SIGINT, SIG_IGN);
    signal(SIGQUIT, SIG_IGN);
    setpwent();
    pwd = getpwuid(getuid());
    endpwent();
    setspent();
    shd = getspnam(pwd->pw_name);
    endsspent();
    setuid(getuid()); // Redevin userul real
    for ( ; ; ) {
        strcpy(pass, getpass("...tty LOCKED!!"));
        cpass = crypt(pass, shd->sp_pwdp);
        if (!strcmp(cpass, shd->sp_pwdp))
            break;
    } //for
} //main
//lockTTY.c
// Compilare: gcc -lcrypt -o lockTTY lockTTY.c
// User root: chown root.root lockTTY
// User root: chmod u+s lockTTY
// Executie: ./lockTTY

```

3.5. Probleme propuse

1. Programul apelat compara doua sau mai multe numere primite ca argumente si returneaza cod 0 daca toate sunt egale, 1 altfel. Programul apelant citeste niste numere si spune daca sunt egale.

Seminar 4

Comunicarea între procese în UNIX

1. COMUNICAREA ÎNTRE PROCESE ÎN UNIX

- se poate realiza:
 - a. între procese care rulează pe același computer (local):
 - prin pipe
 - prin FIFO
 - prin cozi de mesaje
 - prin semafoare
 - prin zone de memorie partajată
 - b. între procese care rulează pe computere diferite (aflate la distanță):
 - prin socket-uri
 - reguli de scriere într-un canal unidirecțional de comunicare prin flux de octeți:
 - orice octet scris nu mai poate fi recuperat și după el se va scrie octetul următor, fie în scrierea curentă, fie în cea următoare
 - dacă un proces dorește să scrie n octeți, însă canalul mai are doar p ($p < n$) octeți liberi, procesul va putea scrie doar primii p octeți din cei n octeți și scrierea se consideră terminată
 - procesul care scrie decide dacă dorește sau nu să scrie și ceilalți $n-p$ octeți printr-o scriere ulterioară
 - reguli de citire dintr-un canal unidirecțional de comunicare prin flux de octeți:
 - un octet odată citit este eliminat din flux și octetul următor este cel care va fi citit, fie în citirea curentă, fie în cea următoare
 - dacă un proces dorește să citească n octeți, însă canalul de comunicare are doar p ($p < n$) octeți disponibili, procesul va putea citi doar cei p octeți existenți și citirea se consideră terminată
 - procesul care citește decide dacă dorește sau nu să obțină și ceilalți $n-p$ octeți printr-o citire ulterioară

2. COMUNICAREA PRIN PIPE

- pipe = flux de date unidirecțional gestionat de către nucleul sistemului de operare

- crearea unui pipe în limbajul C (vezi man 3 pipe):

```
#include <unistd.h>

int pipe(int fd[2]);
```

- returnează 0 dacă apelul a fost executat cu succes sau -1, în caz contrar
- tabloul fd[] conține descriptorul de citire fd[0] și descriptorul de scriere fd[1]

- poate fi utilizată numai de către procese înrudite (procesele care comunică trebuie să fie descendenți ai procesului care a creat pipe-ul)

- descriptorii de citire/scriere din/în pipe sunt unici și sunt moșteniți datorită apelului fork()
- un pipe este un canal unidirectional de comunicare prin flux de octeți, prin urmare programatorul are următoarele obligații:

- să stabilească sensul de comunicare (de la părinte la copil sau invers)
- să închidă descriptorii de citire/scriere din/în pipe pe care nu îi utilizează
- exemplu: pipe_1.c, pipe_2.c

3. COMUNICAREA PRIN FIFO (NAMED PIPE)

- FIFO = flux de date unidirectional rezident în sistemul de fișiere
- Înlătură dezavantajele comunicării prin pipe datorită faptului că poate fi accesat de orice proces (FIFO e un fișier cu nume care se află în sistemul de fișiere)
- crearea unui FIFO din Bash (vezi man 1 fifo, man 1 mknod):

```
mkfifo ./mypipe
mkfifo -m 0640 /tmp/mypipe

mknod ./mypipe p
mknod -m 0640 /tmp/mypipe p
```

- ștergerea unui FIFO din Bash:

```
rm /tmp/mypipe
```

- crearea unui FIFO în limbajul C (vezi man 3 fifo):

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *path, mode_t mode);
```

sau (vezi man 2 mknod):

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

int mknod(const char *path, mode_t mode, dev_t dev);
```

- ștergerea unui FIFO în limbajul C (vezi man 2 unlink):

```
#include <unistd.h>

int unlink(const char *path);
```

Comunicarea prin Pipe

pipe = buffer în memorie care e deschis ca un fișier, de două ori, și poate să citească și să scrie

Descriptori:

- 0 pt citire
- 1 pt scriere

↳ read / write au componente pentru sincronizare: așteaptă pârțea primă date / așteaptă pârțea a două

și geste

```
/*
// pipe_1.c
//
// IPC - Using half-duplex pipe (pipe() system call)
//
// A half-duplex pipe resides within the kernel itself,
// and not within the bounds of any physical file system.
//


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

int main(int argc, char* argv[])
{
    int fd[2];           // file descriptors
    int a[] = {1, 2, 3, 4};

    int res = pipe(fd);
    if (res == -1)      // fail to create pipe
    {
        perror("pipe()");
        exit(EXIT_FAILURE);
    }

    int pid = fork();
    if (pid == -1)      // fail to create child
    {
        perror("fork()");
        exit(EXIT_FAILURE);
    }

    if (pid == 0)        // in child process
    {
        close(fd[0]);    // close the read descriptor

        a[0] += a[1];
        printf("[CHILD] Sum: %d\n", a[0]);

        write(fd[1], &a[0], sizeof(int)); // write partial sum to pipe

        close(fd[1]);      // close the write descriptor
        exit(0);
    }

    close(fd[1]);        // close the write descriptor

    a[2] += a[3];

    read(fd[0], &a[0], sizeof(int)); // read the partial sum from pipe

    int status;
    wait(&status);
    printf("[PARENT] Child has finished with exit status: %d\n", status);

    a[0] += a[2];

    printf("[PARENT] Sum: %d\n", a[0]);

    close(fd[0]);        // close the read descriptor
    return 0;
}
```

! PIPE SE MOSTENEȘTE din părinte în copil

! programul trebuie să închidă descriptorii de pipe când nu se folosește direcția respectivă

```

// pipe_2.c
//
// Se va implementa un proces care creaza un proces copil cu care comunic
// prin pipe. Procesul parinte trimite prin pipe procesului copil doua numere
// intregi, iar procesul copil returneaza prin pipe suma lor.

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

int main(int argc, char* argv[])
{
    int c2p[2];           // child to parent file descriptors
    int res = pipe(c2p);
    if (res == -1)        // fail to create pipe
    {
        perror("pipe(c2p)");
        exit(EXIT_FAILURE);
    }

    int p2c[2];           // parent to child file descriptors
    res = pipe(p2c);
    if (res == -1)        // fail to create pipe
    {
        perror("pipe(p2c)");
        exit(EXIT_FAILURE);
    }

    int pid = fork();
    if (pid == -1)        // fail to create child
    {
        perror("fork()");
        exit(EXIT_FAILURE);
    }

    // in the child process
    if (pid == 0)
    {
        // close the unused file descriptors
        close(c2p[0]);
        close(p2c[1]);

        while(1)
        {
            // read first integer
            int a;
            read(p2c[0], &a, sizeof(int));
            printf("\t[CHILD] a: %d\n", a);

            // stop
            if (a == 0)
                break;

            // read the second integer
            int b;
            read(p2c[0], &b, sizeof(int));
            printf("\t[CHILD] b: %d\n", b);

            // send the sum to parent
            int sum = a + b;
            write(c2p[1], &sum, sizeof(int));

            printf("\t[CHILD] Sum: %d\n", sum);
        }

        // close the file descriptors
        close(c2p[1]);
        close(p2c[0]);
        exit(EXIT_SUCCESS);
    }

    // close the unused file descriptors
    close(c2p[1]);
    close(p2c[0]);

    while(1)
    {
        // read first integer
        int a;
        printf("[PARENT] a: ");
        scanf("%d", &a);

        // send to the child
        write(c2p[1], &a, sizeof(int));

        if (a == 0)
            break;

        sleep(2);

        // read the second integer
        int b;
        printf("[PARENT] b: ");
        scanf("%d", &b);

        // send to the child
        write(p2c[1], &b, sizeof(int));

        // read the sum from child
        int sum = 0;
        read(c2p[0], &sum, sizeof(int));

        printf("[PARENT] Sum: %d\n", sum);
    }

    // wait for child
    int status;
    wait(&status);
    printf("\n[PARENT] Child has finished with exit status: %d\n", status);

    // close the file descriptors
    close(c2p[0]);
    close(p2c[1]);
}

return 0;
}

```

Labonator 9

Comunicarea între procese în UNIX Message Queue

3. COMUNICAREA PRINTR-O COADĂ DE MESAJE (MESSAGE QUEUE)

- coada de mesaje = flux de date bidirectional gestionat de către nucleul SO
- pași necesari:
 - crearea unei cozi de mesaje (din Bash sau din limbajul C)
 - obținerea identificatorului (*msqid*) cozii de mesaje
 - efectuarea schimbului bidirectional de mesaje (*send/receive*) între procese
 - ștergerea cozii de mesaje (optional)
- crearea unei cozi de mesaje din Bash (vezi man 1 ipcmk):
`ipcmk -Q`
- afișarea informațiilor despre resursele IPC din Bash (vezi man 1 ipcs):
`ipcs`
- ștergerea unei cozi de mesaje din Bash (vezi man 1 ipcrm):
`ipcrm -q msgid`
- crearea unei cozi de mesaje sau obținerea identificatorului acesteia (vezi man 2 msgget):

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgfls);
```
- efectuarea schimbului de mesaje (*send/receive*) (vezi man 2 msgsnd):

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msfls);
ssize_t msgrcv(int msqid, const void *msgp, size_t msgsz, long msqtype,
int msfls);
```
- operații de control (ștergere) asupra unei cozi de mesaje (vezi man 2 msgctl):

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```
- exemplu: `msg_client.c`, `msg_server.c`

4. COMUNICAREA PRINTR-UN SEGMENT DE MEMORIE PARTAJATĂ (SHARED MEMORY SEGMENT)

- pași necesari:
 - crearea unui segment de memorie partajată
 - obținerea identificatorului (*shmid*) segmentului de memorie partajată
 - atașarea segmentului de memorie partajată (*attach*) la spațiul de adrese al procesului
 - prelucrarea datelor din segmentul de memorie partajată în modul în care se dorește
 - detașarea segmentului de memorie partajată (*detach*) de spațiul de adrese al procesului
 - ștergerea segmentului de memorie partajată (optional)
- crearea unui segment de memorie partajată din Bash (vezi `man 1 ipcmk`):
`ipcmk -M size`
- afișarea informațiilor despre resursele IPC din Bash (vezi `man 1 ipcs`):
`ipcs`
- ștergerea unui segment de memorie partajată din Bash (vezi `man 1 ipcrm`):
`ipcrm -m shmid`
- crearea unui segment de memorie partajată sau obținerea identificatorului acestuia (vezi `man 2 shmget`):

```
#include <sys/IPC.h>
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmfllags);
```

- atașarea/detașarea segmentului de memorie partajată (vezi `man 2 shmat`):

```
#include <sys/types.h>
#include <sys/shm.h>

void* shmat(int shmid, const void *shmaddr, int shmfllags);
int shmdt(const void *shmaddr);
```

- operații de control (ștergerea) asupra unui segment de memorie partajată (vezi `man 2 shmctl`):

```
#include <sys/IPC.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- exemplu: `shm_client.c`, `shm_server.c`

REFERINTE:

- Curs: <http://www.cs.ubbcluj.ro/~rares/course/os/>
- The Linux Programmer's Guide: <https://www.tldp.org/LDP/lpg/lpg.html>

SHARED MEMORY

= mecanism de comunicare între procese care permite mai multor procese să aducăze și să modifice același zonă de memorie

viteză & partajare

- pt probleme de sincronizare se folosesc mutex / semafoare

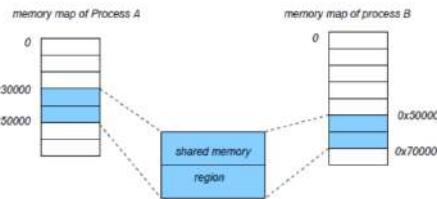
Funcții:

- **shmget** = crearea unui segment de memorie partajat

int shmget (key_t key, size_t size, int shmflg)

↳ returnă identificatorul memoriei partajate

- dacă **shmflg** specifică **IPC_CREAT** și **IPC_EXCL** și un segm. de mem există doar pt clasa resp
 ↳ **shmget()** este creată



- **shmat** = atașarea segm. de mem. partajată la spațiul de adresare al procesului

void * shmat (int shmid, const void * shmidaddr, int shmflg)

↳ dacă **shmidaddr** e NULL, se va alege o adresă potrivită (înghesuită) căreia îi este atașat segmentul (aligare frecventă); altfel **shmidaddr** trebuie să fie o adresă aliniată cu pagina la care atașarea are loc

- **shmdt** = detasarea segmentului de memorie partajată

int shmdt (const void * shmidaddr)

↳ detasarea adresau segm. de mem partajată specificată de **shmidaddr** de spațiul de adresare a procesului curent

- **shmctl** = controalează segmentul de memorie partajată (ex pt a sterge segmentul)

int shmctl (int shmid, int cmd, struct shmid_ds *buf)

```
struct shmid_ds {
    struct ipc_perm shm_perm; /* Ownership and permissions */
    size_t shm_segsz; /* Size of segment ( bytes ) */
    time_t shm_atime; /* Last attach time */
    time_t shm_dtime; /* Last detach time */
    time_t shm_ctime; /* Last change time */
    pid_t shm_cpid; /* PID of creator */
    pid_t shm_lpid; /* PID of last shmat (2) / shmdt (2) */
    shmat_t shm_nattach; /* No. of current attaches */
    ...
};
```

↳ pointer spre structura **shmid_ds**

- some possible values for cmd:

- **IPC_STAT**: copy information from the kernel data structure associated with **shmid** into the **shmid_ds** structure pointed to by **buf**.
- **IPC_SET**: write the value of some member of the **shmid_ds** structure pointed to by **buf** to the kernel data structure associated with this shared memory segment, updating also its **shm_ctime** member.
- **IPC_RMID**: mark the segment to be destroyed. The segment will be destroyed after the last process detaches it (i.e., **shm_nattach** is zero).

```
int main() {  
    // Generăm o cheie unică pentru segmentul de memorie partajată  
    key_t key = ftok("shmfile", 65);  
  
    // Cream un segment de memorie partajată de 1024 bytes  
    int shmid = shmget(key, 1024, 0666|IPC_CREAT);  
  
    // Atașăm segmentul de memorie partajată la spațiul nostru de adresare  
    char *str = (char*) shmat(shmid, (void*)0, 0);  
  
    // Scriem date în segmentul de memorie partajată  
    printf("Scriem in memoria partajata: ");  
    fgets(str, 1024, stdin);  
  
    // Detașăm segmentul de memorie partajată din spațiul nostru de adresare  
    shmdt(str);  
  
    // Citim date din segmentul de memorie partajată  
    str = (char*) shmat(shmid, (void*)0, 0);  
    printf("Datele din memoria partajata: %s\n", str);  
  
    // Detașăm din nou segmentul de memorie partajată  
    shmdt(str);  
  
    // Stergem segmentul de memorie partajată  
    shmctl(shmid, IPC_RMID, NULL);  
  
    return 0;  
}
```

Threads - PROGRAMARE CONCURENTĂ

thread = cea mai mică unitate de procesare care poate fi programată de un SO.

↳ folosite pt a permite unui program să execute operațiuni în paralel \Rightarrow performanță + eficiență

Procese vs. Thread-uri

- **Procese:** Sunt instanțe independente ale unui program în execuție. Fiecare proces are propriul spațiu de memorie, iar comunicarea între procese (IPC - Inter-Process Communication) poate fi relativ costisitoare.
- **Thread-uri:** Partajează același spațiu de memorie și resursele unui proces. Acest lucru permite o comunicare mai rapidă și eficientă între thread-uri, dar necesită mecanisme de sincronizare pentru a preveni conflictele de acces la resurse partajate.

- pot exista mai multe thread-uri într-un singur proces
- cum thread-urile partajează spațiul de memorie, există aceleși variabile

Implementare:

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

void *f(void * a)
{
    printf("f \m");
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t t;           handler thread
    pthread_create(&t, NULL, f, NULL); → crează și permite un thread
    printf("main \m");      argumentele funcției
    pthread_join(t, NULL);
    return 0;               pointer care primește rezultatul din thread
}
```

gcc -Wall -g -o a a.c -pthread

Race condition = apare atunci când se sau mai multe thread-uri accesează resurse partajate (variabile, struct. de date) în același timp și cel puțin unul modifică resursa.
 \rightarrow rezultatul final depinde de ordinea în care fiindcă de execuție accesează resursa ea ce poate duce la comportamente imprevizibile și erori grele de detectat.

→ apare pt că instrucțiunile nu sunt atomicice
 \rightarrow apare unde avem multicore processors

Mijloace de sincronizare

MUTEX: - rezolvă nacee condițion

- un fil de lucăt în jurul unei secțiuni din cod

```
exemplu: void* f() {
    for(int i=0; i<100000; i++) {
        if(lock == 1) {
            // wait until the lock is 0
            lock = 1;
            mails++;
            lock = 0;
        }
    }
}
```

instructione care nu sunt atomică

↳ protejează variabilele ca să nu fie modificată de alte thread-uri în același timp

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
void* f() {
    for(int i=0; i<100000; i++) {
        pthread_mutex_lock(&mtx);
        mails++;
        pthread_mutex_unlock(&mtx);
    }
}
```

un singur thread execută pt că e protejat de mutex

Creația mai multor thread-uri

```
pthread_t t[10]; ← tablou unde stoacăm
for(int i=0; i<10; i++)
    pthread_create(&t[i], NULL, &f, NULL);
for(int i=0; i<10; i++)
    pthread_join(t[i], NULL);
```

Get a return value from a thread

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <time.h>

void* roll_dice() {
    int value = (rand() % 6) + 1; ← variabilă locală
    int* result = malloc(sizeof(int)); ← pointer
    *result = value; ← valoarea din result = value;
    // printf("%d\n", value);
    printf("Thread result: %p\n", result);
    return (void*) result;
}

int main(int argc, char* argv[]) {
    int* res;
    srand(time(NULL));
    pthread_t th;
    if(pthread_create(&th, NULL, &roll_dice, NULL) != 0) {
        return 1;
    }
    if(pthread_join(th, (void**) &res) != 0) {
        return 2;
    }
    printf("Main res: %p\n", res);
    printf("Result: %d\n", *res);
    free(res);
    return 0;
}
```

! Obs
! pthread_mutex_trylock(&mtx)

- 0 dacă poate să-și dia lock
- 1 dacă nu poate

! SECTIUNE CRITICĂ = zonă de cod unde o variabilă este modificată

How to pass arguments to threads in C

- creaă 10 thread-uri, fiecare alege un unic din tablou.

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

int primes[10] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };

void* routine(void* arg) {
    sleep(1);
    int index = *(int*)arg; → pointer pt index
    printf("%d ", primes[index]);
    free(arg); → abia după ce thread-ul și termină execuție
}

int main(int argc, char* argv[]) {
    pthread_t th[10];
    int i;
    for (i = 0; i < 10; i++) {
        int* a = malloc(sizeof(int)); ← pointer
        *a = i; ← valoarea lui a devine i
        if (pthread_create(&th[i], NULL, &routine, a) != 0) {
            perror("Failed to create thread");
        }
    }
    for (i = 0; i < 10; i++) {
        if (pthread_join(th[i], NULL) != 0) {
            perror("Failed to join thread");
        }
    }
    return 0;
}
```

Suma elementelor într-un tablou

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

int primes[10] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };

void* routine(void* arg) {
    int index = *(int*)arg;
    int sum = 0;
    for (int j = 0; j < 5; j++) {
        sum += primes[index + j];
    }
    printf("Local sum: %d\n", sum);
    *(int*)arg = sum;
    return arg;
}

int main(int argc, char* argv[]) {
    pthread_t th[2];
    int i;
    for (i = 0; i < 2; i++) {
        int* a = malloc(sizeof(int)); ← 2 malloc
        *a = i * 5;
        if (pthread_create(&th[i], NULL, &routine, a) != 0) {
            perror("Failed to create thread");
        }
    }
    int globalSum = 0;
    for (i = 0; i < 2; i++) {
        int* r;
        if (pthread_join(th[i], (void**) &r) != 0) {
            perror("Failed to join thread");
        }
        globalSum += *r;
        free(r); ← 2 free
    }
    printf("Global sum: %d\n", globalSum);
    return 0;
}
```

→ Împărțim tabloul în 2 părți;
→ Thread-ul 1 face 0-4, thread 2 5-9

VARIABILE DE CONDIȚIE

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

pthread_mutex_t mutexFuel;
pthread_cond_t condFuel;
int fuel = 0;

void* fuel_filling(void* arg) {
    for (int i = 0; i < 5; i++) {
        pthread_mutex_lock(&mutexFuel);
        fuel += 15;
        printf("Filled fuel... %d\n", fuel);
        pthread_mutex_unlock(&mutexFuel); → unlock
        pthread_cond_signal(&condFuel);
        sleep(1);
    }
}
```

void* car(void* arg) {
 pthread_mutex_lock(&mutexFuel); → unmeză operării care nu sunt atomice ; și dă lock ea să nu se modifice, DAR dacă fuel < 40
 while (fuel < 40) { → pt că fuel = 40 verificăm că există fuel > 40 trebuie să se modifice și să se aplice fuel-filling
 printf("No fuel. Waiting...\n");
 pthread_cond_wait(&condFuel, &mutexFuel); → nu semnalizarea că e adev. cond ci că poate să schimbă ceva

```
    // Equivalent to:
    // pthread_mutex_unlock(&mutexFuel);
    // wait for signal on condFuel
    // pthread_mutex_lock(&mutexFuel);
}
fuel -= 40;
printf("Got fuel. Now left: %d\n", fuel);
pthread_mutex_unlock(&mutexFuel); → unlocks mutex
}
```

```
int main(int argc, char* argv[]) {
    pthread_t th[2];
    pthread_mutex_init(&mutexFuel, NULL);
    pthread_cond_init(&condFuel, NULL);
    for (int i = 0; i < 2; i++) {
        if (i == 1) {
            if (pthread_create(&th[i], NULL, &fuel_filling, NULL) != 0) {
                perror("Failed to create thread");
            }
        } else {
            if (pthread_create(&th[i], NULL, &car, NULL) != 0) {
                perror("Failed to create thread");
            }
        }
    }

    for (int i = 0; i < 2; i++) {
        if (pthread_join(th[i], NULL) != 0) {
            perror("Failed to join thread");
        }
    }
    pthread_mutex_destroy(&mutexFuel);
    pthread_cond_destroy(&condFuel);
    return 0;
}
```

- condiția nu este în variabila de condiție ci în while
- initializare: `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
- `pthread_cond_wait(&cond, &mtx);`
 ↳ mutex-ul trebuie să fie locked ⇒ va fi unlocked temporar mutex-ul și va aștepta după cond
- { `pthread_cond_signal(&cond)` → tragește un thread care este așteptă cond.
`pthread_cond_broadcast(&cond)` → tragește toate thread-urile
 → mecanism pt că nu e operăre atomică

variabile de condiție

BARIERA

- `pthread_barrier_t barrier;`
- `pthread_barrier_init(&barrier, NULL, 3);`
- `pthread_barrier_destroy(&barrier);`

```
#define THREAD_NUM 8
int dice_values[THREAD_NUM];
int status[THREAD_NUM] = {0}; ← won / lost pt thread
```

← fixare thread care arunca un zar,
salveaza val intr-un tablou
thread-ul principal calculeaza castigatoare

```
pthread_barrier_t barrierRolledDice;
pthread_barrier_t barrierCalculated;
```

```
void* roll(void* args) {
    int index = *(int*)args;
    while (1) {
        dice_values[index] = rand() % 32 + 1;
        pthread_barrier_wait(&barrierRolledDice);
        pthread_barrier_wait(&barrierCalculated);
        if (status[index] == 1)
            printf("%d rolled %d I won\n", index, dice_values[index]);
        else
            printf("%d rolled %d I lost\n", index, dice_values[index]);
    }
    free(args);
}
```

```
int main(int argc, char *argv[]) {
    srand(time(NULL));
    pthread_t th[THREAD_NUM];
    int i;
    pthread_barrier_init(&barrierRolledDice, NULL, THREAD_NUM + 1);
    pthread_barrier_init(&barrierCalculated, NULL, THREAD_NUM + 1);
    for (i = 0; i < THREAD_NUM; i++) {
        int* a = malloc(sizeof(int));
        *a = i;
        if (pthread_create(&th[i], NULL, &roll, (void*) a) != 0)
            perror("Failed to create thread");
    }
}

while (1) {
    pthread_barrier_wait(&barrierRolledDice);
    // Calculate winner
    int max = 0;
    for (i = 0; i < THREAD_NUM; i++) {
        if (dice_values[i] > max)
            max = dice_values[i];
    }
    for (i = 0; i < THREAD_NUM; i++) {
        if (dice_values[i] == max)
            status[i] = 1;
        else
            status[i] = 0;
    }
    sleep(1);
    printf("==== New round starting ====\n");
    pthread_barrier_wait(&barrierCalculated);
}
for (i = 0; i < THREAD_NUM; i++) {
    if (pthread_join(th[i], NULL) != 0)
        perror("Failed to join thread");
}
pthread_barrier_destroy(&barrierRolledDice);
pthread_barrier_destroy(&barrierCalculated);
return 0;
}
```

```
pthread_barrier_t barrier;

void* routine(void* args) {
    while (1) {
        printf("Waiting at the barrier..\n");
        sleep(1);
        pthread_barrier_wait(&barrier);
        printf("We passed the barrier\n");
        sleep(1);
    }
}

int main(int argc, char *argv[]) {
    pthread_t th[10];
    int i;
    pthread_barrier_init(&barrier, NULL, 10); ← create thread-urile trebuie sa ajungă la
                                                barrier ca să se rezolve
    for (i = 0; i < 10; i++) {
        if (pthread_create(&th[i], NULL, &routine, NULL) != 0)
            perror("Failed to create thread");
    }
    for (i = 0; i < 10; i++) {
        if (pthread_join(th[i], NULL) != 0)
            perror("Failed to join thread");
    }
    pthread_barrier_destroy(&barrier);
    return 0;
}
```

Deadlocks

→ me doi unlock \Rightarrow deadlock
 → multiple lock-unlock in ordini differenti

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

#define THREAD_NUM 8

pthread_mutex_t mutexFuel;
int fuel = 50;
pthread_mutex_t mutexWater;
int water = 10;

void* routine(void* args) {
    if (rand() % 2 == 0) {
        pthread_mutex_lock(&mutexFuel);
        sleep(1);
        pthread_mutex_lock(&mutexWater);
    } else {
        pthread_mutex_lock(&mutexWater);
        sleep(1);
        pthread_mutex_lock(&mutexFuel);
    }

    fuel += 50;
    water = fuel;
    printf("Incremented fuel to: %d set water to %d\n", fuel, water);
    pthread_mutex_unlock(&mutexFuel);
    pthread_mutex_unlock(&mutexWater);
}

int main(int argc, char *argv[]) {
    pthread_t th[THREAD_NUM];
    pthread_mutex_init(&mutexFuel, NULL);
    pthread_mutex_init(&mutexWater, NULL);
    int i;
    for (i = 0; i < THREAD_NUM; i++) {
        if (pthread_create(&th[i], NULL, &routine, NULL) != 0) {
            perror("Failed to create thread");
        }
    }

    for (i = 0; i < THREAD_NUM; i++) {
        if (pthread_join(th[i], NULL) != 0) {
            perror("Failed to join thread");
        }
    }

    printf("Fuel: %d\n", fuel);
    printf("Water: %d\n", water);
    pthread_mutex_destroy(&mutexFuel);
    pthread_mutex_destroy(&mutexWater);
    return 0;
}
```

Semaphores VS Mutex

- There is a similarity between binary semaphore and mutex. But they are not the same. The purpose of mutex and semaphore are different. Due to similarity in their implementation a mutex would be referred as binary semaphore.
- A **mutex** is **locking mechanism** used to synchronize access to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. It means there is **ownership** associated with mutex, and only the owner can release the lock (mutex).
- A **semaphore** is a **signaling mechanism**.

SEMAFOARE Posix

- `#include <semaphore.h>`
- `sem_t semp;` ↓ note foloseste
 0 → cu un singur proces
- `sem_init(&sem, 0, 1)`
- `sem_destroy(&sem);`
- `sem_wait(&sem);` → verifică val sem $\stackrel{<=0}{\Rightarrow}$ wait
 $\stackrel{>0}{\Rightarrow}$ sem--
- `sem_post(&sem);` → sem++

- There are two operations which can be used to access and change the value of the semaphore variable:
 - P - is also called **wait**, **sleep** or **down** operation
 - V - is also called **signal**, **wake-up** or **up** operation
 - Both operations are **atomic** and semaphore is always initialized to value 1
 - A critical section is surrounded by both operations to implement process synchronization.

```

P(semaphore s){
    while (s==0);
    s=s-1;
}

V(semaphore s){
    s=s+1;
}

```

SO - Semimar 5

PROGRAMARE CONCURENTĂ ÎN UNIX

1. NOTIUNI GENERALE

- proces = un calcul care poate fi executat concurent sau în paralel cu alte calcule
= un program aflat în execuție
- thread (fir de execuție) = entitate de execuție din cadrul unui proces, compusă dintr-un context și o secvență de instrucțiuni de executat → funcție
 - un thread execută o procedură sau o funcție, în cadrul aceluiași proces, concurent sau în paralel cu alte thread-uri
 - toate firele de execuție din cadrul unui proces partajează același spațiu de adrese
 - toate firele de execuție ale unui proces utilizează în comun instrucțiunile, majoritatea datelor și contextul de execuție
 - fiecare fir de execuție are un identificator unic (TID), un set de registri și o stivă proprie
 - singurul spațiu ocupat exclusiv de către un fir de execuție este spațiul de stivă
 - avantajele utilizării firelor de execuție:
 - crearea unui fir de execuție durează mai puțin decât crearea unui proces
 - distrugerea unui fir de execuție durează mai puțin decât distrugerea unui proces
 - trecerea de la un fir de execuție la altul (context switch) este foarte rapidă
 - comunicarea între firele de execuție produce o încărcare (overhead) mai mică a sistemului

2. THREAD-URI POSIX

- implementate de către biblioteca pthreads (POSIX threads)
- crearea unui thread (vezi `man 3 pthread_create`):

```
#include <pthread.h>

int pthread_create(pthread_t *tid, pthread_attr_t *attr, → pthread_create(&t, NULL, f, NULL)
                  void *(*start_routine)(void *), void *arg);
```

- așteptarea terminării unui thread (vezi `man 3 pthread_join`):

```
#include <pthread.h>

int pthread_join(pthread_t tid, void **retval);
```

- terminarea unui thread (vezi `man 3 pthread_exit`):

```
#include <pthread.h>

void pthread_exit(void *retval);
```

- transmiterea unei cereri de abandon unui thread (vezi `man 3 pthread_cancel`):

```
#include <pthreads.h>

int pthread_cancel(pthread_t tid);
```

- exemplu: /exemplu/thread_1.c, thread_2.c, thread_3.c, thread_4.c

- compilare:

```
gcc -Wall -o thread_1 thread_1.c -lpthread
gcc -pthread -Wall -o thread_1 thread_1.c
```

3. SINCRONIZAREA FIRELOR DE EXECUȚIE ÎN LINUX

OBIECTE UTILIZATE PENTRU SINCRONIZARE

Obiect	Declarare	Creare/distrugere	Operații
Semafor	sem_t	sem_init() sem_destroy()	sem_wait() sem_post()
Mutex (MUtual EXclusion)	pthread_mutex_t	pthread_mutex_init() pthread_mutex_destroy()	pthread_mutex_lock() pthread_mutex_unlock()
RW lock (Read-Write lock)	pthread_rwlock_t	pthread_rwlock_init() pthread_rwlock_destroy()	pthread_rwlock_wrlock() pthread_rwlock_rdlock() pthread_rwlock_unlock()
Barieră (Barrier)	pthread_barrier_t	pthread_barrier_init() pthread_barrier_destroy()	pthread_barrier_wait()
Variabilă de condiție (Condition variable)	pthread_cond_t	pthread_cond_init() pthread_cond_destroy()	pthread_cond_wait() pthread_cond_signal() pthread_cond_broadcast()

3.1. SEMAFOARE POSIX

- implementate de către sistemul de operare
- crearea unui semafor (vezi man 3 sem_init):

```
#include <semaphore.h>

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

- blocarea/deblocarea unei resurse folosind un semafor (vezi man 3 sem_wait):

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

- distrugerea unui semafor (vezi man 3 sem_destroy):

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

- exemplu: /exemplu/locks/lock_3.c

3.2. MUTEX

- implementat de către biblioteca pthreads

- crearea unui mutex (vezi `man 3 pthread_mutex_init`):

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex,
                      const pthread_mutexattr_t *attr);
```

sau prin alocare statică:

```
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- blocarea/deblocarea unei resurse folosind un mutex (vezi `man 3 pthread_mutex_lock`):

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- distrugerea unui mutex (vezi `man 3 pthread_mutex_destroy`):

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- exemplu: `/exemplu/locks/lock_2.c`

3.3. RW (READ-WRITE) LOCK

- implementat de către biblioteca pthreads

- crearea unui RW lock (vezi `man 3 pthread_rwlock_init`):

```
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *rwlock,
                      const pthread_rwlockattr_t *attr);
```

- blocarea pentru citire/scriere sau deblocarea unei resurse folosind RW lock (vezi `man 3 pthread_rwlock_rdlock`):

```
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

- distrugerea unui RW lock (vezi `man 3 pthread_rwlock_destroy`):

```
#include <pthread.h>

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

- exemplu: `/exemplu/locks/lock_4.c`

3.4. BARIERA

- implementată de către biblioteca pthreads

- crearea unei bariere (vezi `man 3 pthread_barrier_init`):

```
#include <pthreads.h>

int phtread_barrier_init(pthread_cond_t *restrict barr,
                         const pthread_barrierattr_t *attr, unsigned count);
```

- utilizarea unei bariere (vezi man 3 pthread_barrier_wait):

```
#include <pthreads.h>

int phtread_barrier_wait(pthread_barrier_t *barr);
```

- distrugerea unei bariere (vezi man 3 pthread_barrier_destroy):

```
#include <pthreads.h>

int phtread_barrier_destroy(pthread_barrier_t *barr);
```

- exemplu: /exemplu/barrier.c

3.5. VARIABILA DE CONDITIE

- implementată de către biblioteca pthreads
- crearea unei variabile de condiție (vezi man 3 pthread_cond_init):

```
#include <pthreads.h>

int phtread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
```

sau prin alocare statică:

```
#include <pthreads.h>

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- blocarea/deblocarea folosind o variabilă de condiție (vezi man 3 pthread_cond_wait):

```
#include <pthreads.h>

int phtread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int phtread_cond_signal(pthread_cond_t *cond);
int phtread_cond_broadcast(pthread_cond_t *cond);
```

- distrugerea unei variabile de condiție (vezi man 3 pthread_cond_destroy):

```
#include <pthreads.h>

int phtread_cond_destroy(pthread_cond_t *cond);
```

- exemplu: /exemplu/cond_var.c

1. Threads are another mechanism for implementing concurrent programs that allow faster creation, reduced memory usage, and much simpler communication. POSIX threads (Pthreads) are the native threads implementation in Linux, but every modern operating system provides thread creation libraries. Below is a very basic thread creation example, using the Pthreads library.

<pre>#include <stdio.h> #include <pthread.h> void* f(void* a) { printf("f\n"); return NULL; }</pre>	<pre>int main(int argc, char** argv) { pthread_t t; pthread_create(&t, NULL, f, NULL); printf("main\n"); pthread_join(t, NULL); return 0; }</pre>	<p><i>↳ din acest punct programul are 2 threaduri: · main · f</i></p> <p><i>atâtut care ar modifica funcționarea feță de</i></p>
--	--	--

- a. A thread always executes a given function, in our case `f()`.
 - b. The Pthread library requires the thread function to have a specific signature. It should have a **single void*** argument and it should **return void***.
 - c. The call to `pthread_create` creates and starts a thread that executes the given function pointer `f`, and populates the thread handler `t`.
 - d. The second argument to `pthread_create` is a pointer to `pthread_attr_t` which controls various aspects of the thread creation and execution. In our case, by setting it to `NULL`, we use the default settings.
 - e. The fourth argument to `pthread_create` is the argument to be passed to the function `f()`. In our case, `f()` doesn't use its argument, so we pass `NULL`.
 - f. The call to `pthread_join` waits for the thread identified by the handle `t` to finish.
 - g. The second argument to `pthread_join` will contain the value returned by function `f()`. Passing it as `NULL` means we do not need this value, so we ignore it.
 - h. To compile a program that uses Pthreads, you need to either pass the `-pthread` argument to `gcc`, or to tell it to use the Pthreads library, which you can do passing the `-lpthread` argument.
- `gcc -Wall -g -o a a.c -pthread or gcc -Wall -g -o a a.c -lpthread`
- i. The output of this program depends on how the operating system schedules the execution of these threads, consequently we may see the two `printf`-ed values displayed in any order. This is essential to understanding how concurrency works.
2. It is difficult to see the non-deterministic aspect of thread scheduling using the program above, because the thread execution is extremely short, and it takes a lot of trials to see the lines printed in a different order. Running the program on a busier computer would make this a little easier to see, but we can make this easily obvious using thread functions that do a lot more.

<pre>#include <stdio.h> #include <pthread.h> int n = 1; void* fa(void* a) { int i; for(i=0; i<n; i++) { →printf("fa\n"); } return NULL; } void* fb(void* a) { int i; for(i=0; i<n; i++) { →printf("fb\n"); } return NULL; }</pre>	<pre>int main(int argc, char** argv) { int i; pthread_t ta, tb; if(argc > 1) { sscanf(argv[1], "%d", &n); } pthread_create(&ta, NULL, fa, NULL); pthread_create(&tb, NULL, fb, NULL); for(i=0; i<n; i++) { →printf("main\n"); } pthread_join(ta, NULL); pthread_join(tb, NULL); return 0; }</pre>	<p><i>↳ am mai multe de la mainis folosesc un struct</i></p> <p><i>↳ am mai multe de la mainis folosesc un struct</i></p>
--	---	---

- a. We now have two thread functions that is each run in a separate thread
- b. The command line argument specifies how many iterations each thread will do. Notice that each thread has access to the global variables.
- c. To show the thread scheduling without having long printouts, we pipe the output of the program through `uniq -c` without sorting the out first, thus displaying how many iterations each thread did before the scheduler switched to another. Below are two executions of the same program, taken a few moments apart.

| ./tb 10000 uniq -c |
|--------------------|--------------------|--------------------|--------------------|--------------------|
| 201 main | 189 fa | 1 fa | 1 fb | 1 fb |
| 1 fa | 1 main | 27 main | 15 fa | 28 fa |
| 51 main | 50 fa | 1 fa | 1 fb | 2 fa |
| 1 fa | 2 main | 1 main | 1 fa | 1 fb |
| 1 main | 1 fa | 35 fa | 34 fb | 31 fa |
| 1 fa | 1 main | 1 main | 1 fa | 1 fb |
| 27 main | 1 fa | 34 fa | 36 fb | 32 fa |
| 1 fa | 1 main | 1 main | 1 fa | 1 fb |
| 30 main | 31 fa | 1 fa | 36 fb | 35 fa |
| 1 fa | 1 main | 34 main | 1 fa | 1 fb |
| 9690 main | 31 fa | 1 fa | 31 fa | 1 fa |
| 10000 fb | 1 main | 1 main | 1 fb | 1 fb |
| 9995 fa | 34 fa | 1 fa | 35 fb | 1 fb |
| | 1 main | 33 main | 1 fa | 37 fa |
| | 35 fa | 1 fa | 34 fb | 1 fb |
| | 1 main | 36 main | 1 fa | 1 fa |
| | 168 fa | 1 fa | 37 fb | 9107 fa |
| | 1 main | 9855 main | 1 fa | 4466 fb |
| | 11 fa | 5153 fb | 115 fb | 34 fb |
| | 1 main | 1 fa | 1 fa | |

7.2 PTHREAD ARGUMENT PASSING

1. The last program can be written using a single thread function that gets value to display in the console as argument.

```
#include <stdio.h>
#include <pthread.h>

int n = 1;

void* f(void* a) {
    int i;
    for(i=0; i<n; i++) {
        printf("%s\n", (char*)a);
    }
    return NULL;
}

int main(int argc, char** argv) {
    int i;
    pthread_t ta, tb;
    if(argc > 1) {
        sscanf(argv[1], "%d", &n);
    }
    pthread_create(&ta, NULL, f, "fa");
    pthread_create(&tb, NULL, f, "fb");

    for(i=0; i<n; i++) {
        printf("main\n");
    }

    pthread_join(ta, NULL);
    pthread_join(tb, NULL);

    return 0;
}
```

2. If we want to pass n as an argument to the thread too (instead of having it a global variable), we need to declare a struct since the thread function can only get one argument.

```
#include <stdio.h>
#include <pthread.h>

struct arg_t {
    char* name;
    int count;
};

void* f(void* a) {
    int i;
    struct arg_t* x = (struct arg_t*)a;
    for(i=0; i<x->count; i++) {
        printf("%s\n", x->name);
    }
    return NULL;
}

int main(int argc, char** argv) {
    int i, n = 1;
    pthread_t ta, tb;
    struct arg_t aa, ab;
    if(argc > 1) {
        sscanf(argv[1], "%d", &n);
    }
    aa.name = "fa"; ab.name = "fb";
    aa.count = n; ab.count = n;

    pthread_create(&ta, NULL, f, &aa);
    pthread_create(&tb, NULL, f, &ab);

    for(i=0; i<n; i++) {
        printf("main\n");
    }

    pthread_join(ta, NULL);
    pthread_join(tb, NULL);

    return 0;
}
```

3. Let's create 10 threads, pass to each of them the order number in which it was created, and have it displayed. The program will include a serious bug that causes to a race condition and yields puzzling outputs.

```
#include <stdio.h>
#include <pthread.h>

void* f(void* a) {
    printf("%d\n", *(int*)a);
    return NULL;
}

int main(int argc, char** argv) {
    int i;
    pthread_t t[10];
    for(i=0; i<10; i++) {
        pthread_create(&t[i], NULL, f, &i);
    }
    for(i=0; i<10; i++) {
        pthread_join(t[i], NULL);
    }
    return 0;
}
```

- a. Multiple executions of this program yielded the outputs below

A	B	C	D	E
1	1	1	1	0
2	5	5	2	1
6	2	3	6	1
6	6	4	6	1
3	6	2	3	1
4	6	7	4	1
7	7	6	7	1
8	8	8	8	1
9	9	4	8	1
4	4	5	8	1

- b. Analyzing the outputs above, we notice that:
- i. executions A, B, C, and D do not display 0
 - ii. execution D does not display 9
 - iii. execution E displays one 0 and then nine 1s
 - iv. all executions display duplicate numbers
- c. The root cause of these behaviors is the passing of `&i` to all threads. While we intend to pass to each thread the order number at which it was created, in reality, we pass to all threads the same value: *the address of i*. The main function changes the value of `i` in two `for` cycles (one for thread creation and one for joining). Depending how the threads get the CPU, they will print whatever they find at the memory location `&i`.
- d. To avoid this situation, we can either have an array of `ints` and send each thread the address of another element (first solution below), or dynamically allocate an `int` before creating each thread (second solution below).

<pre>#include <stdio.h> #include <pthread.h> void* f(void* a) { printf("%d\n", *(int*)a); return NULL; } int main(int argc, char** argv) { int i, a[10]; pthread_t t[10]; for(i=0; i<10; i++) { a[i] = i; pthread_create(&t[i], NULL, f, &a[i]); } for(i=0; i<10; i++) { pthread_join(t[i], NULL); } return 0; }</pre>	<pre>#include <stdio.h> #include <stdlib.h> #include <pthread.h> void* f(void* a) { printf("%d\n", *(int*)a); free(a); return NULL; } int main(int argc, char** argv) { int i; int* p; pthread_t t[10]; for(i=0; i<10; i++) { p = (int*)malloc(sizeof(int)); *p = i; pthread_create(&t[i], NULL, f, p); } for(i=0; i<10; i++) { pthread_join(t[i], NULL); } return 0; }</pre>
---	--

- e. Notice where we free the allocated memory in the second solution. What would happen if we freed it right after the call to `pthread_create`?
- f. The outputs of these implementations will still be non-deterministic due to the order of the `printf`s, but they will contain all the expected values.

A B C D E

. /a 1 => LO
 . /a 10 => 100
 :
 . /a 1000 => 9876

*→ număr random
 (în cătă numărul n este mai mare,
 cu atât
 probabil să
 existe astfel
 de "suprapunere"
 crește)*

0	0	1	1	0
5	5	4	4	6
1	2	5	0	1
6	1	2	2	4
3	8	3	5	2
4	6	6	3	7
2	3	0	6	3
7	7	8	7	8
9	9	7	8	9
8	4	9	9	5

LOAD : mov Ax,[count]
 INCREMENT: inc Ax
 STORE: mov [count],Ax

↓
 decide că multe sunt atomic
 (pt mai multe procese paralele)

threadul pierde procesorul

8 POSIX SYNCHRONIZATION MECHANISMS

1. Let's write a program that creates 10 threads, each of which increments a global variable as many times given as a command line argument.

A | B
 L7
 L7
 18
 58
 18
 58

```
#include <stdio.h>
#include <pthread.h>

int count = 0;

void* f(void* a) {
    int i;
    for(i=0; i<*(int*)a; i++) {
        count++;
    }
    return NULL;
}

int main(int argc, char** argv) {
    int i, n = 10;
    pthread_t t[10]; ← 10 threads
    if(argc > 1) {
        sscanf(argv[1], "%d", &n); ← citește din linia de comandă
    }

    for(i=0; i<10; i++) {
        pthread_create(&t[i], NULL, f, &n); ← și creează 10 threads
    }

    for(i=0; i<10; i++) {
        pthread_join(t[i], NULL);
    }

    printf("%d\n", count);
    return 0;
}
```

2. We mentioned before that race conditions do not always manifest, and that the best way to make them more likely to appear is to increase the concurrency. We will run this program with various arguments and repeating those executions to see whether the output gets corrupted or not.

Correct result:	10	100	1000	10000	100000	1000000
for n in 10 100 1000 10000 100000 1000000	100	1000	10000	86337	610108	5068606
do	100	1000	10000	74379	641742	5342264
k=0	100	1000	10000	9446	590373	5777329
while [\$k -lt 10]; do	100	1000	10000	85734	674556	5122867
./th \$n	100	1000	10000	82615	81122	601525
k=`expr \$k + 1`	100	1000	10000	9526	601525	5477098
done	100	1000	10000	9754	79517	5288377
done	100	1000	10000	79977	671355	5405598
	100	1000	10000	78419	599169	5340372
	100	1000	10000	9115	81924	559812
	100	1000	10000	10000	77140	5411218
	100	1000	10000	557539	557539	2303521

3. You can see that as the concurrency increases, the values are more and more corrupted. Even though with lower concurrency things appear to be correct, it is just a matter of time until the data will be corrupted there as well.
 4. This situation is called race condition, with count being the critical resource and the line ; being the .
 5. Race conditions are solved using synchronization mechanisms.

8.1 MUTEXES – MUTUAL EXCLUSION

1. A mutex is a synchronization mechanism that has two main operations: lock and unlock.
2. Only one thread can complete the lock operation at one time, any other threads attempting to lock the mutex, will have to wait until the "winning" thread calls unlock.
3. The race condition in the code above can be avoided using a mutex as shown below

```

#include <stdio.h>
#include <pthread.h>

int count = 0;
pthread_mutex_t m;

void* f(void* a) {
    int i;
    for(i=0; i<*(int*)a; i++) {
        pthread_mutex_lock(&m);
        count++;
        pthread_mutex_unlock(&m);
    }
    return NULL;
}

int main(int argc, char** argv) {
    int i, n = 10;
    pthread_t t[10];
    if(argc > 1) {
        sscanf(argv[1], "%d", &n);
    }
    pthread_mutex_init(&m, NULL);
    for(i=0; i<10; i++) {
        pthread_create(&t[i], NULL, f, &n);
    }
    for(i=0; i<10; i++) {
        pthread_join(t[i], NULL);
    }
    pthread_mutex_destroy(&m);
    printf("%d\n", count);
    return 0;
}

```

- We wrap the critical section between mutex lock/unlock calls.
- The lock/unlock operations could be moved outside the for loop. However, that would drastically reduce the concurrency, as the entire code of the thread would be executed non concurrently.
- The second argument to pthread_mutex_init is a pointer to pthread_mutexattr_t which controls various aspects of the mutex creation and execution. In our case, by setting it to NULL, we use the default settings.

8.2 READ-WRITE LOCKS *pthread_rwlock*

tip: pthread_rwlock_t

- 
- Consider a situation where the threads acting on a critical resource can be grouped into readers (only consult the resource value without modifying it) and writers (modify the resource value). Also assume that the number of reader threads is significantly larger than that of writer threads (a frequent situation in everyday life).
 - Synchronizing the access to the critical resource with a mutex would prevent writers from corrupting the value, and readers from reading dirty values (partially new and partially old due to a simultaneous write). However, the mutex will also prevent more than one reader from consulting the resource at any one time, although any number of readers could do so safely since they do not change it.
 - Read-write locks are designed to address this situation by providing two types of locking:
 - Write lock:** exclusive lock just like a mutex lock, only one thread can get a write lock at any one time, but only if there are no read locks already present. An existing write lock will cause any read lock attempts to wait.
 - Read lock:** shared lock that can be obtained any number of times as long as there is no write lock present. The presence of any read locks will cause a write lock attempt to wait.
 - Here is an example simulating the purchasing product X of which there is an initial stock of 100 pieces. Most shoppers just check the availability of the product, and only some of them buy it.

<pre> #include <stdio.h> #include <pthread.h> int x = 100; pthread_rwlock_t rwl; void* shopper(void* a) { long id = (long)a; if(id % 10 == 0) { // buyer pthread_rwlock_wrlock(&rwl); if(x > 0) { x--; printf("%ld: bought %d\n", id, x+1); } else { printf("%ld: none left\n", id); } pthread_rwlock_unlock(&rwl); } else { //onlooker pthread_rwlock_rdlock(&rwl); printf("%ld: available %d\n", id, x); pthread_rwlock_unlock(&rwl); } return NULL; } </pre>	<pre> int main(int argc, char** argv) { int i; pthread_t t[200]; pthread_rwlock_init(&rwl, NULL); for(i=0; i<200; i++) { pthread_create(&t[i], NULL, shopper, (void*)(long)i); } for(i=0; i<200; i++) { pthread_join(t[i], NULL); } pthread_rwlock_destroy(&rwl); return 0; } </pre>
--	--

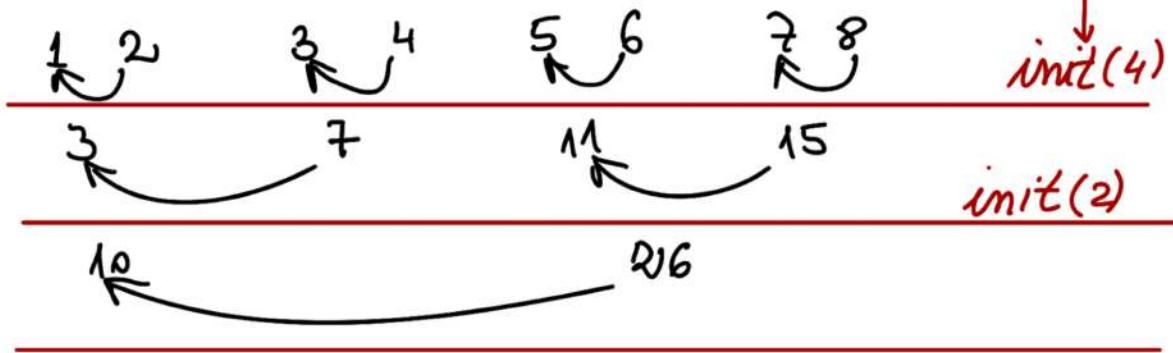
- The purchase is protected by write locks because it modifies the value, while availability checks are protected by read locks because they only consult the value without modifying it.

- b. Notice the "dirty" trick we are using to pass the value of *i* to the thread, by converting to a `void*` via a `long` and then back to a `long`.

8.3 CONDITIONAL VARIABLES

	asynchron	the septe
<code>pthread_cond-</code>	<code>→ init</code> <code>→ wait</code> <code>→ signal</code> <code>→ broadcast</code> <code>→ destroy</code>	
	<code>lock(&m)</code>	<code>lock(&m)</code>
	<code>while (!conditie){</code> <code> Wait(&c,&m)</code> <code>}</code> <code> unlock(&m)</code> <code> asynchron semnal</code> <code> lock(&m)</code>	<code>if (conditie){</code> <code> signal(&c)</code>
	<code>unlock(&m)</code>	<code>unlock(&m)</code>

8.5 BARRIERS - "nu plecăm pînă nu plecăm toți" nu trece mai departe pînă nu sunt 4



`pthread_barrier-` { `init`
`wait`
`destroy`

tip: `pthread_barrier_t`

$$x = y + z$$

`wait`

8.4 SEMAPHORES

sem - {
 → unit
 → wait
 → post
 → destroy

tip: sem_t

initializat cu
① - semafor binar

`**popen**` este o funcție în limbaje de programare precum C și Python care deschide un proces nou și permite interacțiunea cu acest proces prin intermediul unui flux (pipe). Aceasta funcție este utilă pentru a rula comenzi în sistemul de operare și pentru a citi sau scrie date către sau din acel proces.

Funcția `popen` în C

În limbajul C, `popen` este definită în biblioteca standard `<stdio.h>`. Iată o scurtă descriere a utilizării sale:

Sintaxă

C

[Copy code](#)

```
FILE *popen(const char *command, const char *type);
```

- `command`: Un sir de caractere care reprezinta comanda ce va fi executata de shell.
- `type`: Un sir de caractere care specifica modul de deschidere. Poate fi:
 - `"r"` pentru a deschide fluxul in modul citire (citirea ieșirii procesului).
 - `"w"` pentru a deschide fluxul in modul scriere (scrierea intrării pentru proces).

```
    return 0;
}
```

6.9 POPEN

6.9.1 LECTURE 6

1. If you need to run a program or any general Shell command from C code, and get back its standard output, or send data to its standard input, you need `popen`.
2. Write a program that generates the "99 bottles of beer" song, and displays it in a pager like `less`.

```
int main(int argc, char** argv) {
    int i;
    FILE* fp;

    fp = popen("less", "w");
    for(i=99; i>0; i--) {
        fprintf(fp, "%d bottles of beer on the wall,\n", i);
        fprintf(fp, "     %d bottles of beer.\n", i);
        fprintf(fp, "If one of those bottles should happen to fall,\n");
        fprintf(fp, "     %d bottles of beer on the wall.\n\n", i-1);
    }
    pclose(fp);
    return 0;
}
```

3. Write a C program that displays the user with the most processes in the system, and how many processes he or she has. Use a Shell command to find the data.

```
int main(int argc, char** argv) {
    int n;
    char u[64];
    FILE* fp;

    fp = popen("ps -ef | awk '{print $1}' | sort | uniq -c | sort -nr | head -n 1", "r");
    fscanf(fp, "%d %s", &n, u);
    pclose(fp);
    printf("%s: %d\n", u, n);
    return 0;
}
```

6.10 UNIX PROCESS FILE DESCRIPTOR MANIPULATION WITH DUP() AND DUP2()

6.10.1 LECTURE 6

- The goal of this section is to see behind the scenes of input/output redirections, and eventually be able to write in C the equivalent of the command `ps -ef | grep -E '^root' | awk '{print $2}'`
- The process file descriptor table is an array holding handles for accessing the files open by the process.
- Consider the program below.

```
int main(int argc, char**argv) {
    int fd, myfifo, pa[2], pb[2];

    fd = open("a.txt", O_RDWR);
    myfifo = open("myfifo", O_RDONLY);
    pipe(pa);
    pipe(pb);

    return 0;
}
```

The file descriptor table will look as follows

Index	Value
0	Handle for reading from the console (<u>standard input</u>)
1	Handle for writing to the console (<u>standard output</u>)
2	Handle for writing to the console (<u>standard error</u>)
3	Handle for reading/writing to file a.txt. Variable <code>fd</code> has the value of the index, i.e. 3
4	Handle for reading from FIFO myfifo. Variable <code>myfifo</code> has the value of the index, i.e. 4
5	Handle for reading from the first PIPE created. Variable <code>pa[0]</code> has the value of the index, i.e. 5
6	Handle for writing to the first PIPE created. Variable <code>pa[1]</code> has the value of the index, i.e. 6
7	Handle for reading from the second PIPE created. Variable <code>pb[0]</code> has the value of the index, i.e. 7
8	Handle for writing to the second PIPE created. Variable <code>pb[1]</code> has the value of the index, i.e. 8

- The file descriptor table can be manipulated using the system calls `dup()` and `dup2()`.
 - `int dup(int oldfd)` makes a copy of the handle at index `oldfd` to a new entry, and returns the index of the entry just created. For example, adding line `int x = dup(myfifo)` to the program above, just before the `return`, will result in the following line being added to the file descriptor table, and variable `x` will have the value 9.
 - `int dup2(int oldfd, int newfd)` makes a copy of the handle at index `oldfd` to index `newfd`, overwriting the existing value (it also closes it silently before overwriting it). For example, adding line `dup2(pa[0], 0)` to the program above, just before the `return`, will result in line 0 containing the handle for reading from the first PIPE created. Consequently, any reading from the standard input will use the first PIPE created.
- Let's implement in C a program that does the equivalent of running the command `ps -ef | grep -E '^root' | awk '{print $2}'`, by running the three programs and transferring the data between them using pipes.

```

int main(int argc, char** argv) {
    int p2g[2], g2a[2];
    pipe(p2g); pipe(g2a);
    if(fork() == 0) {
        close(p2g[0]); close(g2a[0]); close(g2a[1]);
        dup2(p2g[1], 1);
        execvp("ps", "ps", "-ef", NULL);
        exit(1);
    }
    if(fork() == 0) {
        close(p2g[1]); close(g2a[0]);
        dup2(p2g[0], 0); dup2(g2a[1], 1);
        execvp("grep", "grep", "-E", "^root", NULL);
        exit(1);
    }
}

```

↑
Solve
↓
CIDE

```

if(fork() == 0) {
    close(p2g[0]); close(p2g[1]); close(g2a[1]);
    dup2(g2a[0], 0);
    execvp("awk", "awk", "{print $2}", NULL);
    exit(1);
}

close(p2g[0]); close(p2g[1]);
close(g2a[0]); close(g2a[1]);

wait(0); wait(0); wait(0);

return 0;
}

```

6. How do you undo a dup2 call: use dup to make a copy of the entry that will be overwritten, and when you want to reset it, use again dup2 with the copy.

```

int x = dup(1);
dup2(p[1], 1);
...
dup2(x, 1);

```

6.10.2 SEMINAR 4

1. Implement a simple popen/pclose API using fork, exec, and dup2.

```

FILE* mypopen(char* cmd, char* type) {
    int p[2], caller_idx, child_idx;
    pipe(p);

    caller_idx = 0;
    if(type[0] == 'w') {
        caller_idx = 1;
    }
    child_idx = (caller_idx+1) % 2;

    if(fork() == 0) {
        close(p[caller_idx]);
        dup2(p[child_idx], child_idx);
        if(execvp("bash", "bash", "-c", cmd, NULL) < 0) {
            close(p[child_idx]);
            exit(1);
        }
    }

    close(p[child_idx]);
    return fdopen(p[caller_idx], type);
}

```

```

void mypclose(FILE* fd) {
    fclose(fd);
    wait(0);
}

// Usage example
FILE* f=mypopen("who", "r");
// read from f
mpclose(f);

FILE* f=mypopen("less", "w");
// write to f
mpclose(f);

```

- a. Variable `caller_idx` is the pipe end to be returned to the caller. As the child process needs to do the opposite operation on the pipe (i.e. if the caller wants to read, the child process must write, and vice versa), the child process will close it. Variable `child_idx` is the other pipe end, which will be closed by the caller but used by the child process.

Read

- Extrage date
- așteaptă pe gol
 - până să apar niste date
 - până nu mai sunt rezultări

Write

- adaugă date
- așteaptă pe plin
 - până acine niste rezultări
 - până nu mai sunt cititori

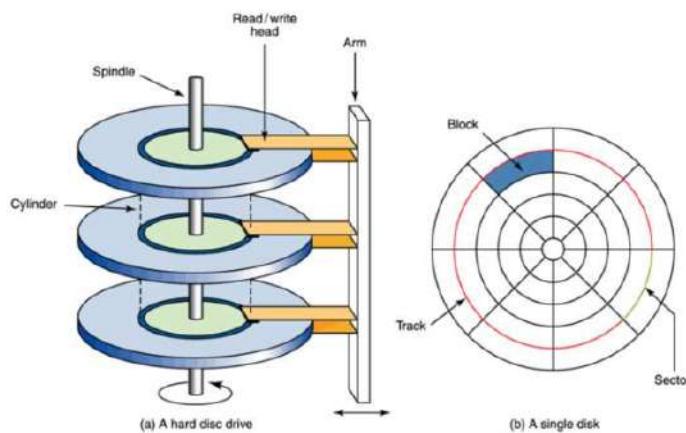
Open, → așteaptă deschiderea pentru opțiunea complementară
→ fereață: _ONDELAY (ar nu se folosi pentru a rezolva buguri).

rand, rand() ← random in C

- .a = .lib (static library)
- .so = .dll (dynamic library)
- .o = .obj

SISTEM DE FISIERE

- **fisier** = o colecție de date/informații stocate pe un dispozitiv de stocare
= named, ordered collection of information
- **director** = container utilizat pentru a organiza fisierele și alte directoare într-un sistem de fisiere
↳ poate conține fisiere și subdirector
- **sistem de fisiere** = set de fisiere și directoare de pe un singur drive. Informația (raw data) de pe drive este tradusă în această vizualizare abstractă a fisierelor și directoarelor de către managerul sistemului de fisiere conform specificațiilor standardului sist. de fisiere
- **managerul sistemului de fisiere** = set de servicii ale SO-ului care furnizează fisiere și directoare pentru aplicații utilizatorului



- **sector:** a physical space on a disk that holds information. When a disk is formatted, tracks are defined. Each track is divided into slices, which are sectors. On hard drives each sector can hold 512 bytes of data.

- **block:** a group of sectors that the operating system can address (point to). It may be one sector, or several sectors (2,4,8, or even 16).

- **file:** stored on a storage medium. Each storage is a linear space for reading or both reading and writing digital information. Each byte of information on it has its offset from the storage start address and is referenced by this address. A storage can be presented as a grid with a set of numbered cells (each cell is a single byte). Any file saved to the storage gets its own cells.
- **file system:** a process that manages how and where data is stored, accessed and managed on a storage disk. It is a logical disk component that manages a disk's internal operations as it relates to a computer and is abstract to a human user.
- There are several common approaches to storing information on disk. However, in comparison to memory management, there are some key differences in managing disk media:
 - Data can only be written in fixed size chunks.
 - Access times are different for different locations on the disk. Seeking is usually a costly operation.
 - Data throughput is very small compared to RAM
 - Data commonly has to be maintained

Seminar 7

File Systems im Linux

1. BASIC CONCEPTS

- a file system controls how data is stored on and retrieved from a storage medium
- file system concepts: *partition, file, directory, path*
- storage mediums: magnetic tape, floppy disk, hard disk, optical disk, SSD (Solid State Drive)
- storing data on disks: *track, sector, disk block*
- file systems:
 - for disks: *FAT (FAT12/FAT16/FAT32), NTFS, exFAT, HFS, HPFS, APFS, ext2fs, ext3fs, ext4fs, ReiserFS, XFS, ZFS etc.*
 - for optical disk: *ISO 9660, UDF (Universal Disk Format)*

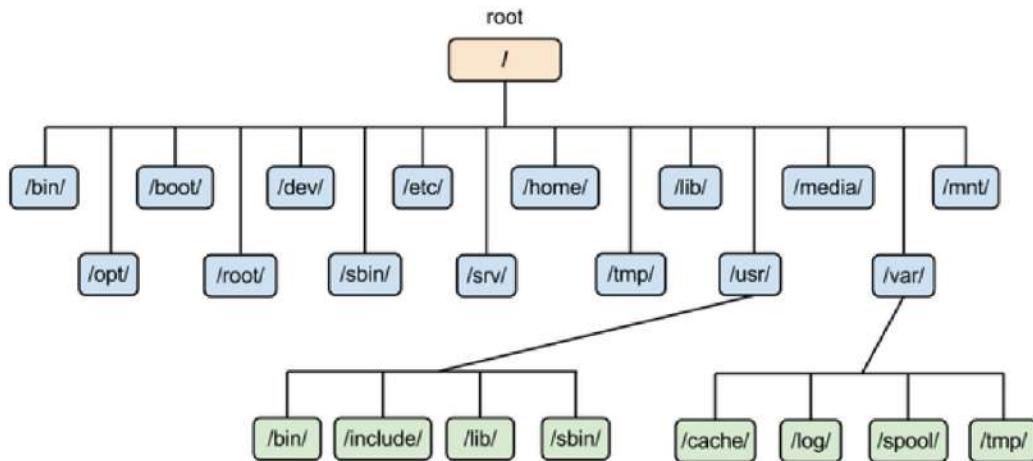
2. JOURNALING FILE SYSTEMS

- issues in classic file systems:
 - the consistency of file system is affected by failures (i.e. system crashes, software crashes, improper shutdowns, power outages)
 - following a crash the *fsck (file system consistency check)* utility has to be run
 - *fsck* will scan the entire file system validating all entries and making sure that blocks are allocated and referenced correctly; if it finds a corrupt entry it will attempt to fix the problem
- solution: fault-resilient file systems
- the first fault-resilient file system was the IBM Journaled File System (JFS)
- it was first released in 1990 by IBM Corp. for its AIX operating system
- a journaling file system = *a file system that maintains a special file called a "journal" that is used to repair any inconsistencies that occur as the result of a failure*
- a journaling file system records changes to the file system; in this way it is able to recover after a failure with minimal loss of data
- a journaling file system write metadata (i.e. data about files and directories) into the journal that is flushed to the disk before each command returns
- examples:
 - Windows: *NTFS (New Technology File System)*
 - Unix/Linux: *ext3fs* (third extended file system), ext4fs, ReiserFS, JFS/JFS2, XFS etc.
- the most common journaling strategies:

- writeback mode - only the metadata is journaled, and the data blocks are written directly to their location on the disk
- ordered mode - metadata journaling only, but writes the data before journaling the metadata
- data mode - both metadata and data are journaled
- advantage: the structure of the file system is always internally consistent
- drawbacks: journaling does not give a performance boost; in fact, the journaling operation reduces the speed slightly, in exchange for reliability

3. FILE SYSTEMS IN LINUX

- Linux supports almost 100 types of file systems
- in Linux and many other operating systems, directories can be structured in a tree-like hierarchy; the Linux directory structure is well defined and documented in the *Linux Filesystem Hierarchy Standard (FHS)*



- mount a file system:
 - refers back to the early days of computing when a tape or removable disk pack would need to be physically mounted on an appropriate drive device;
 - after being physically placed on the drive, the filesystem on the disk pack would be logically mounted by the operating system to make the contents available for access by the OS, application programs and users
- mounting point, mount/umount commands

4. AN OVERVIEW OF DIRECTORY TREE

4.1. Background

<https://tldp.org/LDP/sag/html/fs-background.html>

4.2. The `root` filesystem

<https://tldp.org/LDP/sag/html/root-fs.html>

4.3. The /etc directory

<https://tldp.org/LDP/sag/html/etc-fs.html>

4.4. The /dev directory

<https://tldp.org/LDP/sag/html/dev-fs.html>

4.5. The /usr filesystem

<https://tldp.org/LDP/sag/html/usr-fs.html>

4.6. The /var filesystem

<https://tldp.org/LDP/sag/html/var-fs.html>

4.6. The /proc filesystem

<https://tldp.org/LDP/sag/html/proc-fs.html>

REFERENCES:

- Storing and organizing data files on disks

<http://ntfs.com/hard-disk-basics.htm>

<http://www.mathcs.emory.edu/~cheung/Courses/377/Syllabus/1-files/intro-disk.html>

- Anatomy of Linux journaling file systems

<https://www.ibm.com/developerworks/library/l-journaling-filesystems/index.html>

- An introduction to Linux filesystems

<https://opensource.com/life/16/10/introduction-linux-filesystems>

- Linux Filesystem Hierarchy Standard (FHS)

<http://www.pathname.com/fhs/>

- How to Mount and Unmount File Systems in Linux

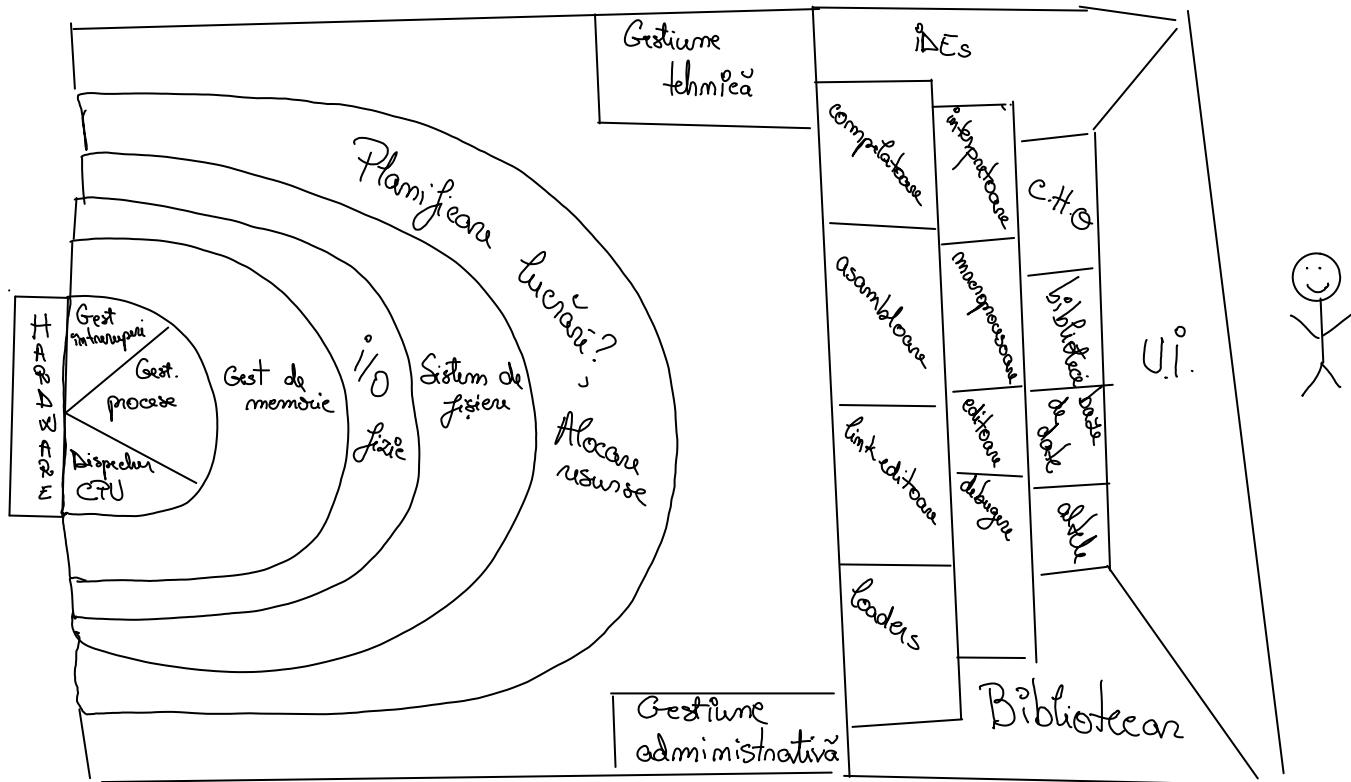
<https://linuxize.com/post/how-to-mount-and-unmount-file-systems-in-linux/>

- ZFS (Zettabyte File System)

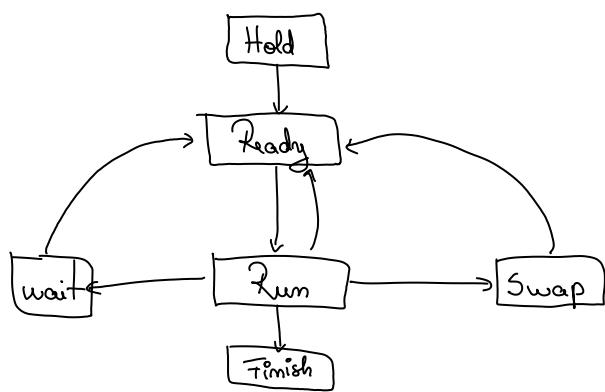
<https://arstechnica.com/information-technology/2020/05/zfs-101-understanding-zfs-storage-and-performance/>

Curs 11 - SO

Kernel - nucleu



Procese



Scheduling

- FCFS - „primul venit, primul servit”
- SJF - „shortest job first” → dacă un proces mare este apărat de procesor și nu-l permite să lanseze altul

- Priorități

- Dead line scheduling / Planificare la termen

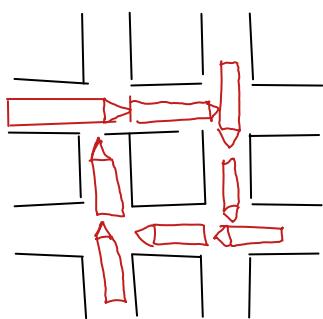
Durată	Termen
A	4
B	1
C	4

→ BAC

„sisteme în timp real”

- Round robin (morm. actual)

Deadlocks

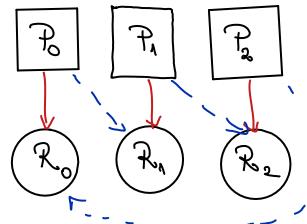


↳ New York, camioane

→ iesirea din deadlock

- Cum detectăm un deadlock?

→ Graf al alocării resurselor



→ durime
--- viață (asteptă să obțină)

↳ ciclu în graf = deadlock

- Cum prevenim deadlock?

→ Bloacăm în același ordin

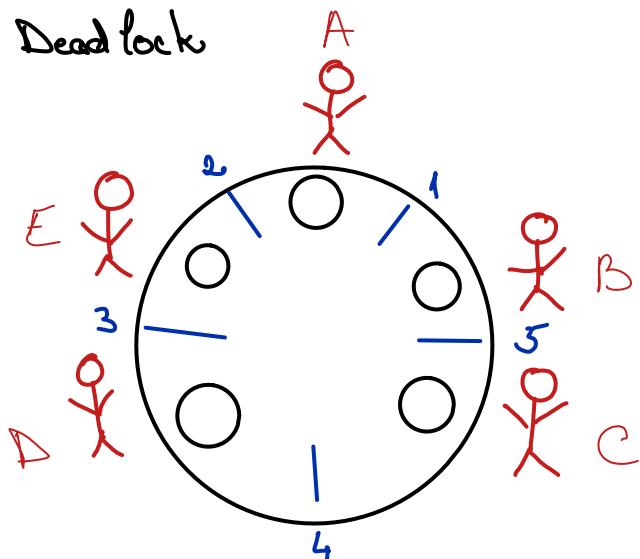
↳ iesirea din deadlock e traumatică
↳ trebuie omorât ceva

Curs 12

-filosofii în AIAU -

deadlock vs. livelock

Deadlock

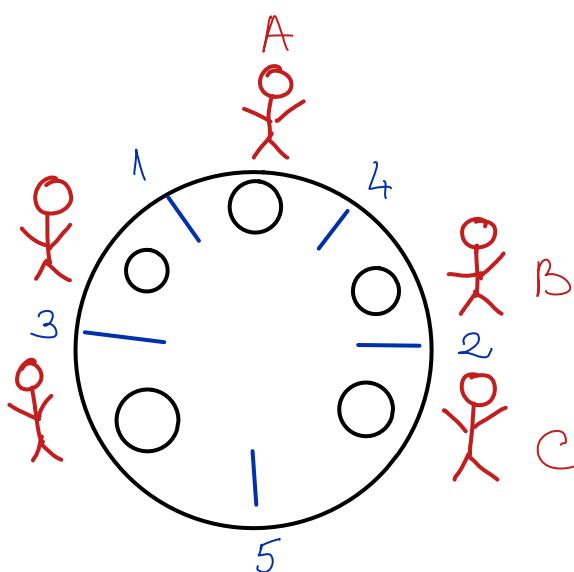


↳ blocarea resurselor în același ordine

"s-a blocat programul" = deadlock

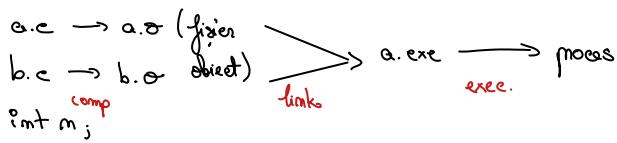
Prevenirea deadlock-ului

1. Excludere mutuală → e pt prevenirea concurrii datelor
 2. Occupă și astăptă
 3. Neprecumpă?
 4. Astăptare circulară
- ↳ facem asta imposibilă



Gestirea memoriei

- Calculul adresei



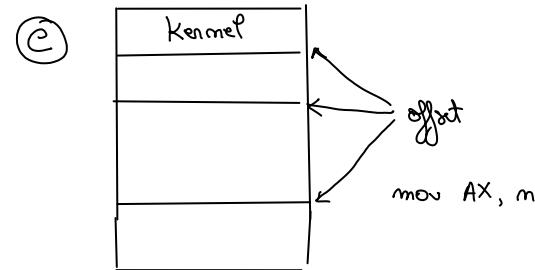
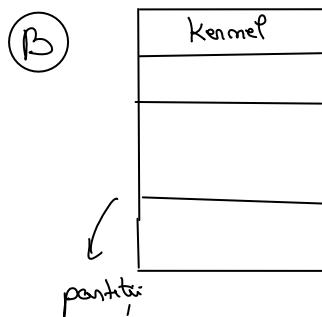
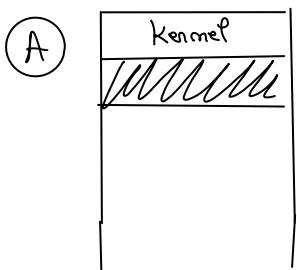
- Tehnici de alocare

- alocare reală

- (A) ↗ pt SO single-tasking
- ↗ pt SO multi-tasking
- cu partitii fixe
- (B) - absolute
- (C) • relocaabile
- (D) → cu partitii variabile

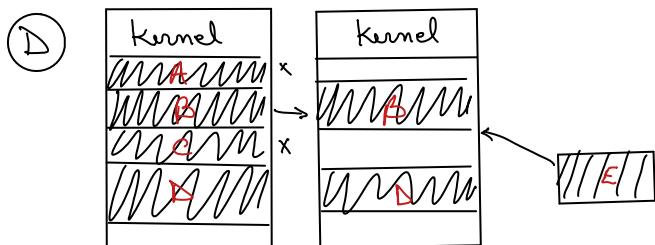
- alocare virtuală

- (E) → paginată
- (F) → segmentată
- (G) → paginat-segmentată

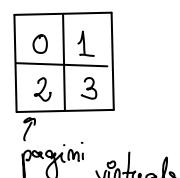
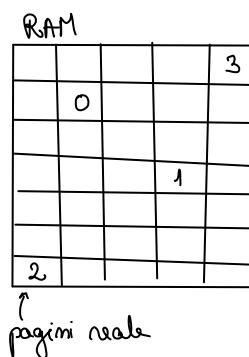


- compilatorul / linkeditorul
 handcodează în executabil
 adrese fixie

↪ se face adunarea ale offset-urilor partitiei p+ fiecare instr.



(E)



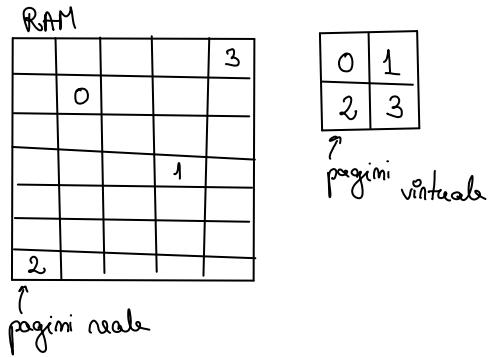
↑ pagini reale

Politica de înlocuire

- toate la permanență programului → Dezavantaj: încasarea memoriei cu pagini nefolosite
→ Avantaj: nulare mai rapidă
- la memorie: → Dezavantaj: permanență chiar mai lentă, execuție mai lentă
- principiu al vecinătății ↪ folosirea în momentul actual
 - ↪ prima înlocuire a unei ce statistică ar fi necesar

Curs 13

Paginare



Politici de înlocuire (pdf)

- FIFO nu oferă nici o producție solidă
- NRU: accesele la fiecare pagină sunt —? — pe 2 la împărțit

?

o

- LRU: - dacă memoria are N pagini folosim o matrice $N \times N$ de laiți
 - când o pagină e accesată, linia ei se populează cu 1 și coloana cu 0
 - victimă se alege dintre paginile cu sumă minimă pe linii

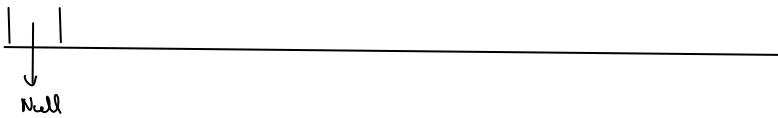
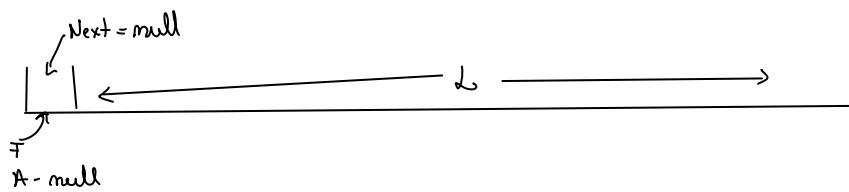
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
		↓ 1	
0	0	0	0
1	0	1	0
0	0	0	0
0	0	0	0

$\xrightarrow{3}$

0	0	0	0
1	0	1	0
0	0	0	0
0	0	0	0

1 1 1 0

Politici de alocare



? dinse

First Fit

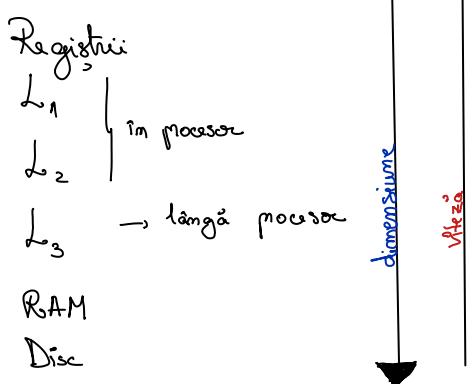
- încărcă, iterez, în primul loc unde începe să punem
- dezavantaj
- avantaj

Best Fit

Worst Fit

$$2^n = 2^0 + 2^1 + 2^2 + \dots + 2^{n-1}$$

Cache



Direct mapping

- locația în cache a unei pagini este nr-paginii % dim Cache
- ④ Dacă cache are 16 pagini

0	1	2	1*	33	49	1	4
0	1	2	1	1	1	1	4
1*	33	1*	33	1*	33		

Cache asociativ

de la Ceață la Mădă

Curs 14

→ presupunem că avem 3 procese

A

:
A
B
C
D

B

:
e
f
g

C

:

Tabula sistem a
fiierelor deschise

:
k
l
m
:
n
o
:
p
q
r

Tabela de
î-moduri

→ link simbolie

→ fiieri spre care permitem să un link simbolie

↳ Cum se face stocarea datelor? Cum se recuperă datele

8 Sisteme de operare – prezentare generală

8.1 Tipuri de sisteme de operare (SO); clasificări

In prezent există un mare număr de SO în funcțiune și numărul acestora este în continuă creștere. Nu există un criteriu unitar de comparare a acestora. Noi vom da mai multe clasificări, în funcție de următoarele criterii:

- după gradul de partajabilitate a resurselor;
- după tipurile de interacțiuni permise;
- după organizarea internă a programelor ce compun SO.

8.1.1 Clasificare după gradul de partajabilitate a resurselor

După acest criteriu, distingem trei categorii de sisteme de operare [10]:

- Sisteme monouser și monotasking;
- Sisteme monouser și multitasking;
- Sisteme multiuser.

Sisteme de operare monouser sunt aceleia care permit, la un moment dat, unui singur utilizator să folosească sistemul. Un sistem este monotasking dacă admite la un moment dat execuția unui singur program.

In forma cea mai simplă, un sistem monouser și monotasking execută un singur program la un moment dat și acesta rămâne activ din momentul lansării lui și până la terminarea lui completă. Un astfel de sistem este de exemplu cel care deservește un telefon mobil sau un terminal PDA (Personal Digital Assistant).

Un sistem monouser și multitasking este acela în care un singur utilizator este conectat la sistem, dar el poate să își lanseze simultan mai multe procese în lucru. Tipice în acest sens sunt SO din familia Windows 9X, ca și cele din familia NT care nu sunt servere. Tot în această categorie putem îngloba sistemele Linux instalate pe calculatoare neconectate în rețea. Pentru sistemele de acest fel care au memoria internă relativ mică, se aplică așa-numita tehnică swapping, adică evacuarea temporară a unui program, din memoria internă pe discul magnetic. În timpul evacuării, în memoria internă este încărat un alt program, care la rândul lui este executat parțial și este apoi evacuat și el. Cât timp un program este în memorie, el are acces la toate resursele sistemului. Programele sunt astfel execute pe porțiuni, momentele de execuție alternând cu cele de evacuare. După unii autori ([4]), SO de acest tip se află la granița dintre sistemele monoutilizator și cele multiutilizator.

Sisteme de operare multiutilizator permit accesul simultan al mai multor utilizatori la sistem. Ele trebuie să aibă în vedere partajarea procesorului, a memoriei, a timpului, a perifericelor și a altor tipuri de resurse, între utilizatorii activi la un moment dat. Vom reveni cu detalii asupra metodelor de partajare a procesorului, timpului și a memoriei. La astfel de sisteme sunt necesare tehnici mai sofisticate, de gestiune și protecție a utilizatorilor. La un moment dat aici există mai multe procese active care se execută concurrent sub controlul SO.

8.1.2 Clasificare după tipurile de interacțiuni permise

In funcție de resursele pe care le are un sistem și în funcție de destinația acestuia, este necesară stabilirea unor strategii de interacțiune cu utilizatorul. Conform acestui criteriu de diferențiere, distingem [34]:

- Sisteme seriale (care prelucrează loturi de lucrări);
- Sisteme cu timp partajat (time – sharing);
- Calculatoare personale și stații de lucru (workstations);
- Sisteme autonome (embedded systems);
- Sisteme portabile, destinate comunicațiilor (telefoane mobile, PDA-uri etc.);
- Sisteme conectate în rețea.

Sisteme seriale. In cadrul acestor sisteme lucrările se execută pe loturi pregătite în prealabil. Practic, din momentul predării unei lucrări la ghișeul centrului de calcul și până la eliberarea ei, utilizatorul nu poate interveni spre a influența execuția programului său. SO afectate acestor sisteme pot lucra atât monoutilizator, cât și multiutilizator. Din punct de vedere istoric sunt printre primele sisteme. In prezent, în această categorie intră supercalculatoarele, precum și o serie de sisteme medii – mari cum ar fi AS-400. Sistemul de operare trebuie să gestioneze loturi de lucrări pe care să le pregătească spre a fi introduse în sistem și eventual să pregătească livrarea rezultatelor (listare etc.). In general gradul de concurență la acest tip de sisteme este relativ redus. In [10] sunt date mai multe detalii despre acest tip de sisteme.

Sistemele cu timp partajat trebuie să suporte, într-o manieră interactivă, mai mulți utilizatori. Fiecare utilizator este în contact nemijlocit cu programul său. In funcție de unele rezultate intermediare, el poate decide modul de continuare a activității programului său. SO trebuie să gestioneze printre altele și terminalele de teletransmisie la capătul căror se află utilizatori care-i pot transmite diverse comenzi. Sistemul trebuie să disponă de mecanisme mai sofisticate decât cele seriale pentru partajarea procesorului, a memoriei și a timpului. De regulă, acest tip de sisteme practică partajarea timpului, astfel. Se fixează o cantă de timp. Pe durata cuantei, un singur proces are acces la procesor (și la celealte resurse aferente). După epuizarea cuantei, procesul este suspendat, pus la sfârșitul cozii de procese și un alt proces ocupă procesorul pentru următoarea cantă ș.a.m.d. Dimensiunea cuantei și strategiile de comutare sunt astfel alese încât să se reducă timpii de așteptare și să se realizeze o servire rezonabilă a proceselor. Tipice acestui tip de sisteme sunt sistemele Unix, dar și Windows din familia NT / server.

Calculatoare personale și stații de lucru. La acest tip de sisteme, toate resursele mașinii sunt dedicate utilizatorului care este conectat. Așa că nu se pune problema partajării resurselor între utilizatori, ci doar a datelor între procesele pe care userul le lansează simultan. Practic, sarcina principală a SO este aceea de a oferi userului o interfață prietenoasă de exploatare a sistemului.

Sistemele autonome sunt sisteme dedicate unui anumit proces industrial, cum ar fi: funcționarea unui robot, coordonarea activităților dintr-un satelit geostaționar, supravegherea unui baraj de apă, a unei stații de radiolocație etc. Fiind conectate la diverse procese tehnologice, aceste sisteme trebuie să fie capabile să deservească în timp prestatabilitatea fiecare serviciu care î se cere. Dacă sistemul nu este capabil să dea un răspuns, este posibilă oprirea procesului supravegheat. Astfel de sisteme sunt în prezent în plină dezvoltare, mai ales din cauza dezvoltării rapide a tehnologiilor multimedia.

Sisteme portabile, destinate comunicațiilor. Acestea formează cea mai nouă categorie de sisteme. Exemplele tipice sunt telefoane mobile și PDA-urile. Sunt mașini portabile, de dimensiuni mici și au facilități puternice de comunicare. Facilitatea lor principală este conexiunea wireless printr-o dintre tehnologiile existente: radio, infraroșu, Bluetooth etc. Evident, sistemul de operare aferent trebuie să fie capabil să gestioneze eficient aceste comunicări. Din cauza formatului mic, memoria internă, dar mai ales memoria pe suport extern este foarte limitată. Sistemul de operare trebuie să facă față acestei penurii de resurse. Securizarea accesului la un astfel de sistem este esențială. Fiind echipamente mici pot fi ușor pierdute / furate, iar această securizare trebuie să le facă de nefolosit de către persoanele neautorizate.

Sisteme conectate în rețea. Practic, astăzi mareea majoritate a sistemelor nu mai sunt independente, ci sunt conectate la rețele de calculatoare, inclusiv la rețeaua publică Internet. Din această cauză, una dintre sarcinile fundamentale ale sistemelor de operare din această categorie trebuie să fie gestionarea accesului la rețea. De asemenea, accesul din exterior la resursele locale trebuie să fie bine protejat, spre a interzice acceselor neautorizate.

În legătură cu acest criteriu de clasificare, trebuie arătat că el este relativ. Astfel, există sisteme seriale care permit interacțiunea cu programele, sistemele interactive au facilități puternice de lucru serial etc. Mai mult, același SO permite în același timp să lucreze, spre exemplu, în timp real, dar în același timp să servească, evident cu o prioritate mai mică, și alte programe de tip serial sau interactiv, în regim de multiprogramare clasică.

8.1.3 Clasificare după organizarea internă a programelor ce compun SO.

Să vedem cum arată un SO văzut din "interior". Vom vedea patru tipuri de structuri, în evoluția lor istorică.

- Sisteme monolitice;
- Sisteme cu nucleu de interfață hardware;
- Sisteme cu structură stratificată;
- Sisteme organizate ca mașini virtuale

Sisteme monolitice. Un astfel de SO este o colecție de proceduri, fiecare dintre acestea putând fi apelată după necesitate. Execuția unei astfel de proceduri nu poate fi întreruptă, ea trebuie să-și execute complet sarcina pentru care a fost apelată. Un program utilizator rulat sub un sistem monolitic se comportă ca o procedură apelată de SO. La rândul lui, programul poate apela în maniera apelurilor sistem (vezi 4.5 pentru Unix) diverse rutine din SO. Singura structură posibilă aici este legată de cele două moduri de lucru: nucleu și user (vezi pentru Unix 4.5 și pentru Windows 7.1.3). Esențial pentru aceste tipuri de sisteme este legătura de tip apel-revenire, fără posibilitatea de întrerupere a fluxului normal al execuției. În prezent, aceste tipuri de SO sunt pe cale de dispariție.

Sisteme cu nucleu de interfață hardware. Un astfel de sistem concentrează sarcinile vitale de nivelul cel mai apropiat de hardware într-o colecție de rutine care formează nucleul SO. În acesta sunt înglobate inclusiv rutinile de întrerupere. Componentele nucleului se pot executa concurrent. Toate acțiunile utilizatorului asupra echipamentului hard trec prin acest nucleu. Unele SO mai plasează între nucleu și utilizator încă o interfață. Exemple de nuclee ale SO pot fi considerate componentele BIOS ale mașinilor PC deși fac parte din hardware.

Sisteme cu structură stratificată este o generalizare a organizării cu nucleu. Un astfel de sistem este construit nivel după nivel, componentele fiecărui nivel folosind toate serviciile oferite de nivelul inferior. În prezent aceste sisteme sunt cele mai răspândite. "Părinții" lor sunt SO THE (Dijkstra, 1968) și SO MULTICS. Ultimul dintre ele oferă o organizare mai interesantă, și unică în felul ei, aceea a inelelor de protecție. Pentru detalii se pot consulta [48], [49]. În secțiunea următoare vom ilustra structura stratificată a unui SO ipotetic.

Sisteme organizate ca mașini virtuale. Echipamentul hard al acestor tipuri de sisteme servește, în regim de multiprogramare, eventual în time-sharing, un număr de procese. Fiecare proces dispune, în mod exclusiv, de o serie de resurse, cea mai importantă fiind memoria. Fiecare dintre procesele deservite este un sistem de operare, care are la dispoziție toate resursele alocate procesului respectiv de către echipamentul hard. În acest mod, pe același echipament hard se poate lucra simultan sub mai multe SO. Primele sisteme de acest tip au fost VM/370 (Virtual Machines for IBM-370), care coordonează mai multe SO conversaționale monoutilizator de tip CMS (Conversational Monitor System). Fiecare utilizator lucrează sub propriul CMS, pentru el fiind transparent faptul că lucrează sub CMS singur, pe un calculator mic, sau că este legat, împreună cu alții la același echipament hard. Sistemele actuale Windows oferă mașini virtuale (ferestre) pentru lucru sub DOS. Implementările Linux, FreeBSD și Solaris actuale oferă mașina virtuală Wine care emulează platforme Windows. Pentru detalii, se pot consulta [4], [49].

8.2 Structura și funcțiile unui sistem de operare

8.2.1 Stările unui proces și fazele unui program

8.2.1.1 Stările unui proces

În secțiunile precedente am folosit destul de des termenul de proces, privit intuitiv ca un program în execuție. Am văzut în capitolele precedente noțiunea de proces sub Unix și sub Windows. În fig. 5.18 din 5.2.3 sunt prezentate stările unui proces Unix. Există [62] și stări în care se poate afla un proces Windows.

Sintetizând stările proceselor sub diverse sisteme de operare, putem defini stările unui proces într-un sistem de operare. Aceste stări sunt:

- HOLD – proces pregătit pentru intrarea în sistem;
- READY – procesul este în memorie și este gata de a fi servit de (un) procesor;
- RUN – un procesor execută efectiv instrucțiuni (mașină) ale procesului;
- WAIT – procesul așteaptă terminarea unei operații de intrare / ieșire;
- SWAP – imaginea procesului este evacuată temporar pe disc;
- FINISH – procesul s-a terminat, trebuie livrate doar rezultatele.

HOLD este starea unui proces la un sistem serial în care procesului îi sunt citite datele de lansare din cadrul lotului din care face parte, se face o analiză preliminară a corectitudinii lor și apoi procesul este depus într-o coadă pe disc (numită HOLD) spre a fi preluat spre prelucrare. De obicei, aceste sisteme au o componentă specializată pentru astfel de prelucrări – SPOOL-ing de intrare (Simultaneous Peripheral Operations OnLine – vezi [10]).

FINISH este, de asemenea, stare a unui proces la un sistem serial. Procesul se execută și rezultatele lui sunt plasate pe disc într-o coadă FINISH. După terminare, acestea vor fi listate,

fie pe o imprimantă locală, fie pe una aflată la distanță. Componenta din SO care face preia din coada FINISH și listează se numește SPOOL-ing de ieșire (vezi [10]).

READY este starea în care un proces se află în memoria internă dar nu este servit de procesor. Este posibil ca el să fi fost servit parțial, dar pe moment să fie suspendat în defavoarea altui proces, urmând să i se continue execuția mai târziu.

SWAP este starea în care un proces este evacuat temporar într-un spațiu rezervat pe disc (SWAP) pentru a face temporar loc altui proces în memoria internă. Ulterior procesul va fi reîncărcat în memoria internă și i se va continua execuția.

WAIT este starea în care un proces a cerut executarea unei operații de intrare / ieșire. Cât timp procesul așteaptă să se termine operația, cedă altor procese procesorul.

RUN este starea principală, aceea în care procesorul execută efectiv instrucțiuni mașină ale programului procesului. Într-un sistem monoprocesor doar un singur proces se află în starea RUN. Dacă sistemul are n procesoare, atunci maximum n proceze se vor afla în starea RUN.

Acste stări nu se vor întâlni la toate tipurile de sisteme de operare. De exemplu, la un sistem monoutilizator (vezi 8.1) nu va fi starea SWAP, numai sistemele seriale (vezi [10]) au stările HOLD și FINISH etc.

8.2.1.2 Fazele unui program

Fazele unui program reprezintă etapele prin care trece acesta din momentul în care s-a început proiectarea lui și până când devine cod mașină executabil spre a fi integrat într-un proces. Aceste faze, bine cunoscute de către programatori, sunt:

1. Editarea textului sursă într-un limbaj de programare, executată de către programator cu ajutorul unui editor de texte.
2. Compilarea, executată de un compilator specializat în limbajul respectiv. Ca rezultat se obține un fișier obiect.
3. Editarea de legături, fază în care se grupează mai multe module obiect rezultate din compilări și se obține un fișier executabil. Editarea se face cu un program specializat, numit editor de legături
4. Execuția programului înseamnă mai întâi încărcarea fișierului executabil în memorie cu ajutorul unui program încărcător (loader) și lansarea lui în execuție.
5. Testarea și depanarea programului, fază care alternează cu cele de mai sus până când proiectantul are certitudinea că programul lui este corect. Uneori această depanare poate fi asistată de către un program specializat numit depanator.

In secțiunile următoare vom reveni cu detalii, atât asupra stărilor unui proces cât și asupra fazelor unui program.

Menirea unui SO, indiferent de tipul lui, este pe de o parte de a facilita accesul la sistem a unuia sau mai multor utilizatori, iar pe de altă parte de a asigura o exploatare eficientă a echipamentului de calcul. Vom încerca să arătăm mai în detaliu care sunt funcțiile posibile ale unui sistem, independent de tipul acestuia. Desigur, dependență de tip va fi ponderea în cadrul sistemului a uneia sau alteia dintre funcții. Din motive care se vor clarifica mai târziu, vom folosi în loc de termenul program termenul de proces sau task. Printr-un proces vom înțelege un calcul care poate fi efectuat concurrent cu alte calcule (asupra acestei noțiuni vom reveni).

S-au proiectat și se proiectează încă multe tipuri de SO. Fiecare dintre acestea trebuie să aibă ca obiective fundamentale:

- optimizarea utilizării resurselor;
- minimizarea efectului uman de programare și exploatare;
- automatizarea operațiilor manuale în toate etapele de pregătire și exploatare a SC;
- creșterea eficienței utilizării SC prin scăderea prețului de cost al prelucrării datelor.

Principalele funcții ale SO sunt legate de acțiunile acestuia pentru tranziția între diverse stări ale procesului. În fig. 8.1 [10], [47] sunt schematizate stările unui proces. În dreptul fiecărei tranziții sunt trecute principalele acțiuni efectuate de către SO. Caracteristicile fiecăreia dintre cele șase stări posibile (HOLD, READY, RUN, SWAP, WAIT, FINISH) le-am prezentat în secțiunea precedentă.

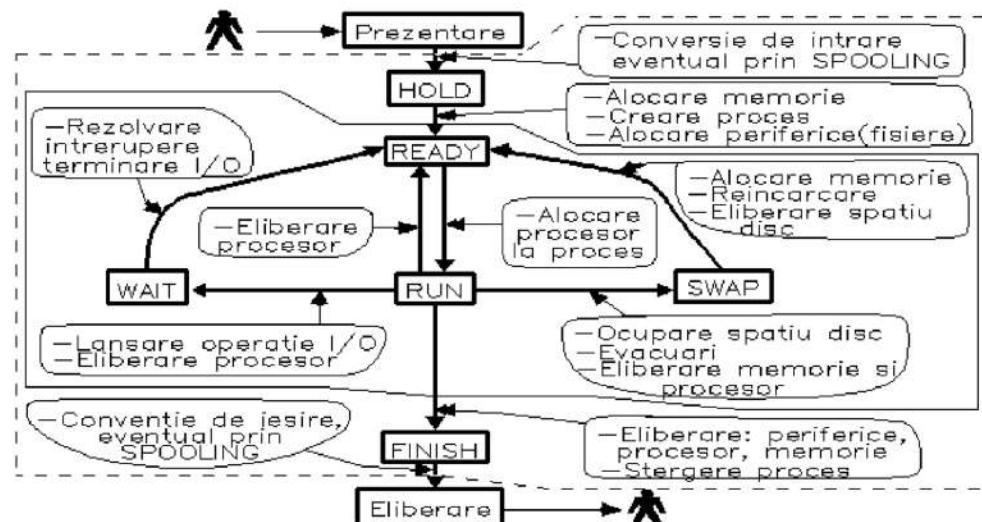


Figura 8.1 Stările unui proces și acțiunile SO aferente

Acțiunile SO ilustrate în fig. 8.1 sunt executate de către componenta nucleu a sistemului de operare. La nivelul componentei user, o mare parte dintre acțiunile SO sunt focalizate mai ales în sprijinul utilizatorului pentru dezvoltarea de programe, adică de asistare a userului la trecerea programului prin diverse faze (vezi secțiunea precedentă 8.2.1.2).

Am exclus de aici, de fapt am amânat numai până la o secțiune viitoare, o funcție primordială: punerea în lucru a SO. După cum vom vedea, în mod (aparent) paradoxal, această sarcină revine tot SO! La prima vedere pare a fi vorba de o antinomie, cum este posibil ca ceva (cineva) să-și dea viață singur? Vom vedea.

8.2.2 Structura generală a unui sistem de operare

În această secțiune încercăm să oferim o imagine asupra structurii unui SO ipotetic. Un astfel de sistem nu va fi regăsit nicăieri în practică, însă structura fiecărui SO real va fi apropiată de acestuia. Evident, în funcție de tipul de SO real, unele dintre prezentele componente vor fi atrofiate sau vor lipsi cu desăvârșire, iar altele vor putea fi eventual mult mai dezvoltate decât

aici. Fără pretenția de "gen-proxim" și "diferență-specifică", încercăm să definim principalele noțiuni, structuri și componente cu care operează orice SO.

Să trecem acum la structura propriu-zisă.

8.2.2.1 Structura unui SO

Schema generală a unui SO este ilustrată în fig. 8.2.

Curs :

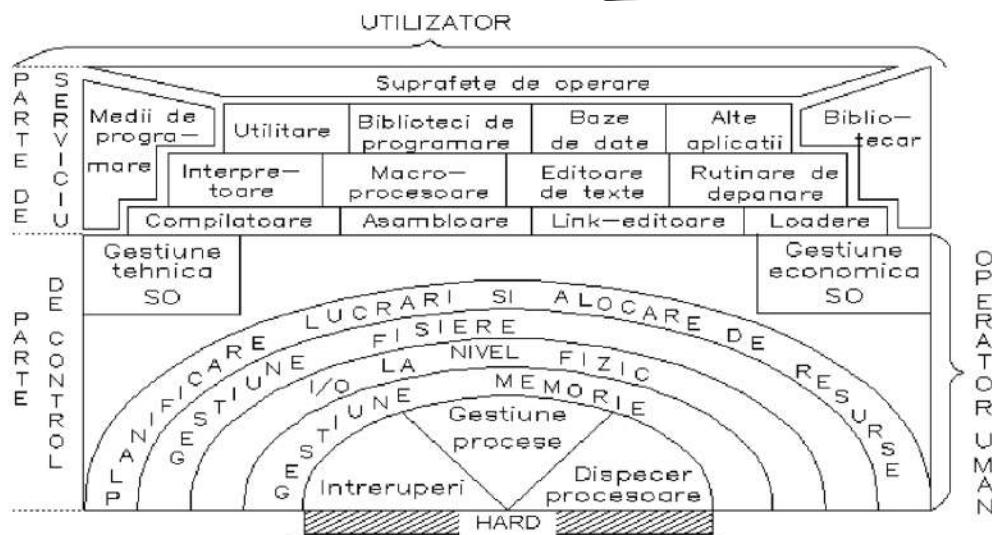


Figura 8.2 Structura generală a unui SO

Orice SO conține două părți mari: partea de control și partea de servicii.

PARTEA DE CONTROL este în legătură directă cu operatorul uman și indirectă cu utilizatorul. SO execută în mod master componentele acestei părți. Principalul ei rol este de a realiza legătura cu SC, deci cu echipamentul hard.

PARTEA DE SERVICIU lucrează în mod slave, folosind facilitățile părții de control. Este în legătură directă cu utilizatorul și indirectă cu operatorul uman. Sarcina de bază a ei este de a ține legătura cu utilizatorul, facilitându-i acestuia accesul la resursele SC și SO.

Ideea schemei din fig. 8.2 este preluată din [10]. În continuare vom descrie succint rolul fiecărei componente.

8.2.2.2 Partea de control

INTRERUPERI. Un ansamblu de rutine, fiecare dintre ele fiind activată la apariția unui anumit semnal fizic de intrerupere. Corespunzător, SO efectuează acțiunile aferente intreruperii respective. Semnificația conceptului de intrerupere este descrisă în fiecare manual de arhitectura calculatoarelor. Pentru intreruperile 8086, se poate consulta [52].

Funcționarea concretă a componentei (sistemului) de intreruperi depinde foarte mult de particularitățile SC, deci nu se poateză prea bine la abordări cu caracter general. Pe de altă parte, detalii despre aceste componente interesează în primul rând pe programatorii și inginerii de sistem, mai puțin pe utilizatorii obișnuiți, cărora ne adresăm noi. Din aceste motive vom schița numai mecanismul de funcționare a intreruperilor.

Fiecare tip de intrerupere are asociată o locație fixă de memorie. În această locație se află o adresă. Această adresă indică locul din memorie la care se află o secvență de instrucțuni, numită handler, secvență care deservește intreruperea respectivă.

La apariția semnalului de intrerupere, după ce instrucțiunea mașină în curs s-a executat, se execută, în această ordine, următoarele activități:

- se salvează într-o zonă de memorie starea programului în curs de desfășurare;
- se activează handlerul asociat intreruperii respective;
- handlerul execută acțiunile necesare servirii intreruperii;
- se restaurează starea programului care a fost întrerupt.

GESTIUNE PROCESE. Creează procese și rezolvă problemele privind cooperarea și concurența acestora. La terminarea unui proces, acesta este șters din sistem. Asupra conceptului de proces vom mai reveni.

DISPECER PROCESOARE. La sistemele multiprocesor, repartizează sarcinile de calcul solicitate de procese, procesoarelor care sunt libere și care pot executa sarcinile cerute.

GESTIUNEA MEMORIEI. Alocă necesarul de memorie internă solicitat de procese și asigură protecția memoriei interprocese. Componenta este parțial realizată prin hard și parțial este realizată soft, de către o componentă a SO. Vom reveni și asupra problemelor legate de gestiunea memoriei.

I/O LA NIVEL FIZIC. Asigură efectuarea operațiilor elementare de I/O cu toate tipurile de periferice din sistem realizând, acolo unde este posibil, desfășurarea simultană a uneia sau mai multor operații I/O, cu activitatea procesorului central. Această componentă este, de asemenea, puternic dependentă hard, în special de multitudinea de echipamente periferice conectate la sistem. Așa după cum am văzut, există niște rutine de interfață numite drivere, care realizează compatibilizarea convențiilor de I/O între SC și echipamentele periferice. Driverele printre altele, fac parte din această componentă. Vom reveni într-o secțiune viitoare cu detalii.

GESTIUNEA FISIERELOR. O colecție de module prin intermediul cărora se asigură: deschiderea, închiderea și accesul utilizatorului la fișierele rezidente pe diferite suporturi de informații. Componentă de bază a SO, este cel mai des invocată de către utilizatori și de către operator. Vom reveni și asupra ei într-un capitol special.

PLANIFICAREA LUCRARILOR SI ALOCAREA RESURSELOR. Resursele unui SC se împart în: resurse fizice (procesoare, memorie internă, periferice etc.) și resurse logice (proceduri comune, tabele ale sistemului etc.). Fiecare proces solicită la un moment dat anumite resurse. Prin această componentă a SO se asigură planificarea proceselor astfel, încât ele să poată obține de fiecare dată resursele necesare, în mod individual sau partajate cu alte procese. Elemente ale planificării se răsfrâng, după cum vom vedea mai târziu, asupra gestiunii proceselor, gestiunii memoriei, intrărilor și ieșirilor etc. Atunci când vom discuta despre procese, memorie și I/O la nivel fizic vom aborda și problematica planificărilor și a alocărilor de resurse.

GESTIUNEA TEHNICA a SO. Tine evidența erorilor hard apărute la echipamentele SC. La cerere, furnizează informații statistice asupra gradului de utilizare a componentelor SC.

GESTIUNEA ECONOMICA a SO. Tine evidența utilizatorilor care folosesc sistemul, lucrările executate de către acestia și resursele consumate de aceste lucrări (timpi de rulare, memorie internă ocupată, echipamente și suporti folosiți etc). Periodic (zilnic, lunar, anual) editează liste cuprinzând lucrările rulate și facturi de plată către beneficiari.

8.2.2.3 Partea de servicii

COMPILETOARELE sunt programe care traduc texte sursă scrise în limbaje de nivel înalt (C, C++, Java, FORTRAN, COBOL, Pascal, Ada etc.) în limbaj mașină. La această oră există o mare varietate de tehnici de compilare, având ca nucleu teoria limbajelor formale. Există de asemenea sisteme automate de elaborare a diverselor componente ale compilatorului. Rezultatul unei compilări este un modul obiect memorat într-un fișier obiect. Detalii privind compilatoarele se pot obține consultând [33].

ASAMBLORUL este un program care traduce texte scrise în limbajul de asamblare propriu SC într-un modul obiect. Din punct de vedere al SO problemele legate de asamblor sunt incluse în problemele compilatoarelor. De exemplu, programarea în limbach de asamblare 8086 este descrisă în [52].

LINK-EDITORUL sau EDITORUL DE LEGATURI. Grupează unul sau mai multe module obiect, împreună cu subprograme de serviciu din diverse biblioteci, într-un program executabil. De cele mai multe ori această componentă oferă posibilitatea segmentării programelor mari, adică posibilitatea ca două componente diferite a programului să poată ocupa în execuție, la momente diferite de timp, aceeași zonă de memorie. Rezultatul editării de legături este un fișier executabil. În 5.1 este descris formatul executabil de sub Unix, iar în [52] fișierele executabile COM și EXE de pe platformele Microsoft.

LOADER (PROGRAM DE INCARCARE) este un program al SO care are trei sarcini:

- citirea unui program executabil de pe un anumit suport;
- încărcarea acestuia în memorie;
- lansarea lui în execuție.

Dacă programul executabil este segmentat, atunci loaderul încarcă la început numai segmentul rădăcină al programului, după care, la cerere, în timpul execuției, încarcă de fiecare dată segmentul solicitat în memorie. În [52] este descris mecanismul de încărcare de sub MS-DOS.

INTERPRETOR este un program care execută pas cu pas instrucțiunile descrise într-un anumit limbaj. Tehnica interpretării este aplicată la diferite limbaje de programare, cum ar fi Java, BASIC, LISP, PROLOG etc. Pentru SO, cea mai importantă categorie de interpretoare sunt "interpretoarele de comenzi". În 2.1.1 am descris interpretoarele de comenzi Shell de sub Unix, iar în [10] [52] interpretorul de comenzi COMMAND.COM de pe platformele Microsoft.

MACROPROCESOR (PREPROCESOR) este un program cu ajutorul căruia se transformă o machetă de program, pe baza unor parametri, într-un program sursă compilabil. Spre exemplu, fie macheta:

```

MACRO    ADUNA    &1, &2, &3
LOAD     &1
ADD      &2
STORE    &3
ENDMACRO

```

Dacă într-un program (dintr-un limbaj de asamblare) se scrie:

```
ADUNA A, B, C
```

atunci prin macroprocesoare se generează:

```

LOAD     A
ADD      B
STORE   C

```

Practic, macroprocesarea este tehnica prin care se realizează o scriere mai compactă a programelor și o posibilă parametrizare a acestora pentru realizarea comodă a unor implementări înrudite.

Astfel de macroprocesoare există la toate SO moderne. Remarcăm în mod deosebit interpretorul de comenzi "shell" de sub Unix, descris în 2.2. Acesta dispune, printre altele, de cele mai puternice facilități de macroprocesare dintre toate interpretoarele de comenzi cunoscute.

Unele limbi au prevăzute în standardele lor specificații de macroprocesare. În acest sens amintim preprocesorul "C" care tratează în această manieră liniile care încep cu "#" și construcțiile "typedef", precum și facilitățile generice ale limbajului ADA .

EDITORUL DE TEXTE este un program interactiv destinat introducerii și corectării textelor sursă sau a unor texte destinate tipăririi. Orice SO interactiv dispune cel puțin de câte un astfel de editor de texte. Până vom ajunge la capitolul dedicat editoarelor de texte, să enumerez câteva editoare "celebre" unele prin vechimea lor, iar altele prin performanțele lor:

- ed și vi sub Unix;
- Notepad și EDIT sub Windows;
- xedit și emacs de sub mediul X WINDOWS.

RUTINELE DE DEPANARE (DEPANATOARELE) asistă execuția unui program utilizator și-l ajută să depisteze în program cauzele erorilor apărute în timpul execuției lui. Întreaga execuție a programului utilizator se face sub controlul depanatorului. La prima apariție a unui eveniment deosebit în execuție (depășire de indici, adresă inexistentă, cod de operație necunoscut etc.) depanatorul primește controlul. De asemenea, el poate primi controlul în locuri marcate în prealabil de către utilizator. Un astfel de loc poartă numele de breakpoint. La primirea controlului, depanatorul poate afișa valorile unor variabile sau locații de memorie, poate reda, total sau parțial istoria de calcul, poate modifica valorile unor variabile sau locații de memorie etc. Unele depanatoare mai performante permit execuția reversibilă, concept tratat teoretic dar mai puțin aplicat practic. La această oră sunt cunoscute două tipuri de depanatoare:

- depanatoare mașină;
- depanatoare simbolice.

Depanatoarele mașină operează cu elemente cum ar fi: adrese fizice, configurații de biți, locații de memorie etc. Fiecare SO dispune de câte un astfel de instrument de depanare. Pentru

exemplificări se pot consulta documentațiile MS_DOS pentru DEBUG, precum și documentația Unix despre depanatoarele adb și gdb [47].

Depanatoarele simbolice operează cu elemente ale textelor sursă ale programelor. Întâlnim aici termeni ca linie sursă, nume de variabilă, nume de procedură, stivă de apel a procedurilor etc. Evident, depanatoarele simbolice sunt mult mai tentante pentru utilizator, și este bine că este așa. De obicei, un astfel de depanator însوtește implementarea unui limbaj de programare de nivel înalt. Primul și cel mai răspândit limbaj la care au fost atașate depanatoare simbolice este limbajul Pascal. Amintim aici depanatorul Pascal de pe lângă implementarea TURBO Pascal, un depanator simbolic complet interactiv, cu o grafică destul de bună pentru momentul implementării lui și cu facilități puternice. Firma BORLAND, un adevărat vârf de lance în acest domeniu, a realizat printre altele produsul TURBO DEBUGER, care recunoaște în vederea depanării toate limbajele pentru care s-au realizat medii TURBO: Pascal, "C", limbajul de asamblare TASM, Basic, PROLOG etc. Produsul lucrează atât simbolic, cât și la nivel mașină.

BIBLIOTECARUL este un program cu ajutorul căruia utilizatorul poate comanda păstrarea programelor proprii în fișiere speciale de tip bibliotecă, poate copia programe dintr-o bibliotecă în alta, șterge, inserează și modifică programe în aceste biblioteci. Exemple de programe biblioteci sunt LBR sub MS-DOS, ar (archive) sub Unix etc.

MEDIILE DE PROGRAMARE sunt componente complexe destinate în principal activității de dezvoltare de programe. O astfel de componentă înglobează, relativ la un limbaj, cel puțin următoarele sub componente:

- editor de texte;
- compilator interactiv;
- compilator serial (clasic);
- editor de legături sau un mecanism echivalent acestuia destinat creerii de programe executabile;
- interpretor pentru execuția rezultatului compilării;
- depanator, de regulă simbolic;
- o componentă de legătură cu mediul exterior, de regulă cu SO sub care se lucrează.

De regulă, filozofia de manevrare a mediului are un caracter unitar, este simplă și puternică. Facilitățile grafice ale acestor medii sunt destul de mult utilizate, făcând deosebit de agreabil lucrul cu ele.

SUPRAFETELE DE OPERARE au apărut tot la calculatoarele IBM-PC și a celor compatibile cu ele. Ele au rolul principal de a "îmbrăca" sistemul de operare (Windows sau Unix), pentru a-i face mai ușoară operarea. Inițial au fost destinate utilizatorilor neinformaticieni. La această oră, nici profesioniștii nu se mai pot "dezlipi" de o suprafață de operare cu care s-a obișnuit să lucreze. De regulă, suprafețele de operare înlocuiesc limbajul de control (de comandă) al SO respectiv. Utilizatorul, în loc să tasteze o comandă a SO respectiv, poartă un dialog cu suprafața de operare, iar la finele dialogului suprafața generează sau execută comanda respectivă. Ponderea dialogului o deține suprafața însăși, utilizatorul răspunde de regulă prin apăsarea uneia sau maximum a două taste, sau unul sau două "clicuri" cu un mouse. Suprafața deține și un mecanism deosebit de puternic de HELP, conducând utilizatorul din aproape în aproape spre scopul dorit. Deși se adresează în primul rând neinformaticienilor, ele sunt din ce în ce mai mult folosite și de către specialiști.

O remarcă specială trebuie făcută pentru WINDOWS, care deține un mecanism propriu de dezvoltare a unor aplicații.

Din rațiuni impuse de standarde, sub Unix se folosește aproape peste tot suprafața X-WINDOWS [43].

Din scurta prezentare a componentelor principale ale unui SO rezultă caracterul modular al SO și organizarea lui ierarhică. Modularitatea este termenul prin care se descrie partaționarea unui program mare în module mai mici, cu sarcini bine precizate. Caracterul ierarhic este reflectat în faptul că orice componentă acționează sub controlul componentelor interioare acesteia, apelând primitivele oferite de acestea. Aceste facilități asigură o implementare mai ușoară a SO, o înțelegere mai facilă a SO de către diverse categorii de utilizatori, o întreținere și depanare mai ușoară a sistemului și creează premise simulației lui pe alt sistem.

8.3 Incărcarea (lansarea în execuție) a unui sistem de operare

Una dintre sarcinile de bază ale unui SO este aceea de a se putea autoîncărca de pe disc în memoria internă și de a se autolansa în execuție. Această acțiune se desfășoară la fiecare punere sub tensiune a SC, precum și ori de câte ori utilizatorul (operatorul uman) crede de cuviință să reîncarce SO. Acțiunea este cunoscută sub numele de încărcare a SO sau lansare în execuție a SO.

In cele ce urmează dorim să punem în evidență mecanismul de lansare. O vom face însă pentru un SO ipotetic, fără să ne legăm de un anume sistem de operare concret. Simplificând la maximum expunerea.

Pentru lansare se utilizează un mecanism combinat hard și soft, numit bootstrap. Pentru a putea înțelege mai bine modul lui de funcționare, să considerăm un exemplu.

Să notăm cu $M[0]$, $M[1]$, $M[2]$, ... locațiile de memorie ale unui calculator ipotetic. Presupunem că în fiecare locație $M[i]$ începe un număr întreg. Presupunem, de asemenea că sistemul nostru dispune de o memorie ROM și un disc (dischetă, CD, DVD, hard disc, partație disc etc.) numit disc de boot, de la care se citesc, secvențial, începând din primul sector, numere întregi.

Prin $\text{read}(x)$ vom înțelege că se citește următorul număr de la intrarea standard și valoarea lui se depune în locația x .

Prin $\text{transferla}(y)$ înțelegem că următoarea instrucțiune mașină de executat este cea din (al cărei cod se află în) locația y .

Mecanismul bootstrap intră în lucru la apăsarea butonului de pornire <START>. Ca prim efect, sistemul extrage din ROM un sir de octeți reprezentând echivalentul în limbaj mașină al programului din fig. 8.3.

```
APASASTART:
for (i = 0; i < 100 ; i++) {
    read(M[i]);
}
transferla(M[0]);
```

Figura 8.3 Programul APASASTART

Sirul de octeți citit este depus în memorie la o adresă pe care o vom nota APASASTART. După depunerea la adresa respectivă, sistemul execută instrucțiunea:

```
transferla(APASASTART);
```

Cu aceasta, sistemul transferă controlul programului din fig. 8.3. Execuția acestui program înseamnă că sistemul citește de pe discul de boot primele 100 de numere, le depune în primele 100 de locații de memorie, după care trece la executarea instrucțiunii al cărei cod se află în prima locație.

Este împedite că în 100 de locații nu se poate scrie decât un program simplu. Ori pentru lansarea în execuție a unui SO, probabil că este necesară punerea în lucru a unui program mai sofisticat. Un astfel de program poate fi scris pe mai multe sectoare consecutive, folosind, de exemplu reprezentarea din fig. 8.4. Să presupunem că într-un sector se pot înregistra S numere întregi.

AM1	n1	n1 întregi de program sau date	S – n1 – 2 întregi zero
AM2	n2	n2 întregi de program sau date	S – n2 – 2 întregi zero
---	---	---	---
AMk	Nk	nk întregi de program sau date	S – nk – 2 întregi zero
ADL	-1		S – 2 întregi zero

Figura 8.4 Structura unui fișier ce conține un program executabil

Deci fișierul executabil din fig. 8.4 poate avea un număr oarecare k de înregistrări a către un sector, conținând numere întregi. Semnificațiile câmpurilor sunt:

- AMi adresa din memorie de unde începe introducerea porțiunii de cod de program sau date din înregistrarea i;
- ni este numărul efectiv de locații ocupat de porțiunea de program sau date a înregistrării i. Valoarea -1 indică ultimul sector al a programului.
- ADL se află în ultimul sector al programului și este adresa de lansare în execuție a programului reprezentat: un număr între AM1 și AMk+nk-1.

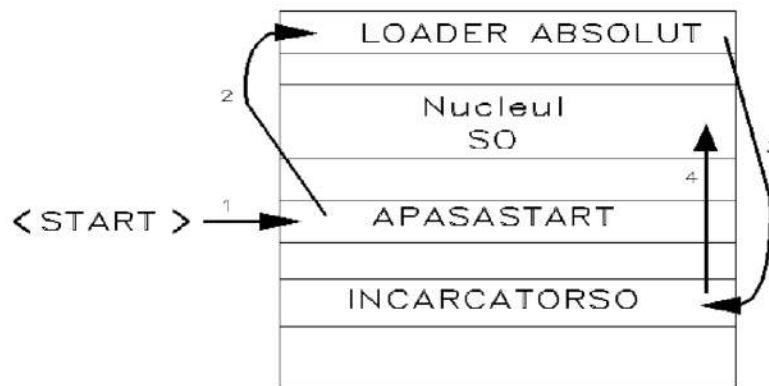
Să considerăm acum programul din fig. 8.5, în care presupunem că începând de la locația M[r] există S locații libere.

```
LOADERABSOLUT:
do {
    for ( i=0; i < S; i++ ) {
        read(M[r+i]);
    }
    A = M[r];
    n = M[r+1];
    if (n== -1)
        break;
    for ( i=0; i< n; i++ ) {
        M[A+i] = M[r+2+i]
    }
    while (FALSE);
    LANSARE:
    transferla(A);
```

Figura 8.5 Incărător (loader) de programe cu structura din fig. 8.4

Avem acum posibilitatea să citim de pe disc și să lansăm în execuție, cu programul din fig. 8.5, programe oricât de mari cu formatul din fig. 8.4. De fapt, în fig. 8.5 avem de-a face cu un exemplu simplu de program de încărcare (loader).

Cheia (saltul calitativ al) încărcării: În cele 100 de numere care vor fi citite în momentul startului cu programul APASASTART (din fig. 8.3), se trece codul mașină al programului LOADERABSOLUT (din fig. 8.5). Să vedem acum, pe etape, cum are loc încărcarea și lansarea în execuție a SO. În fig. 8.6 sunt ilustrate zonele de memorie folosite în faza de încărcare și lansare în execuție a SO.

**Figura 8.6 Incărarea unui SO**

Etapele încărcării sunt:

- 1) Se apasă <START>. Ca efect, programul APASASTART citește, în primele 100 de locații de memorie, codul mașină al programului LOADERABSOLUT.
- 2) După transfera(M[0]) intră în lucru programul LOADERABSOLUT.
- 3) Acesta citește la rândul lui un program, care de această dată poate fi oricât de sofisticat, cu formatul din fig. 8.4. Acest program îl vom numi INCARCATORSO.
- 4) După ce se execută și ultima instrucțiune transferla(A) a programului LOADERABSOLUT, intră în lucru programul INCARCATORSO. Dat fiind faptul că formatul de reprezentare a codului lui mașină este cel din fig. 8.4, acesta poate fi practic oricât de mare. Așa cum îi spune și numele, acest program va încărca de pe disc programele și rutinile (nucleului) sistemului de operare.

La încheierea încărcării SO, INCARCATORSO execută următoarele acțiuni:

- șterge din memorie codul programului APASASTART;
- șterge din memorie codul programului LOADERABSOLUT;
- eliberează spațiul de memorie ocupat de el, cu excepția unei securi porțiuni, despre care vorbim mai jos;
- execută o instrucțiune transferla(ASSO), unde ASSO este adresa de start a sistemului de operare. Codul necesar acestei instrucțiuni este cel pe care nu-l poate, în mod firesc, elibera. Însă dacă acest cod este plasat într-o zonă folosită ca buffer, practic nu mai rămâne memorie ocupată.
- după aceasta, încărătorul își încheie activitatea.

In sfârșit, toate SO moderne lansează la terminarea încărcării un fișier căruia o să-i spunem fișier de comenzi initiale. Acesta este un fișier de comenzi, care la rândul lui poate lansa alte fișiere de comenzi și.a.m.d. text în care sunt trecute toate comenzile pe care administratorul le dorește a fi executate la pornirea sistemului. Printre ultimele acțiuni desfășurate la lansarea So amintim:

- montarea partițiilor locale;
- efectuarea unor operații de întreținere asupra sistemului de fișiere (filesystem cleanup);
- pornirea diferitelor servicii (de exemplu serviciile de printare);
- setare nume sistem și inițializare ceasuri în funcție de zona geografică;
- configurația rețea;
- montează sisteme de fișiere aflate la distanță (așa-numitele sisteme de fișiere remote)
- permite conectarea utilizatorilor;

Proiectantul SO va avea grijă ca programul încărcațor să fie depus într-o zonă de memorie care să nu fie ocupată de către programele SO pe care el le încarcă.

Față de modelul de încărcare exemplificat de noi mai sus, realitatea, conceptual, este foarte aproape. Primele calculatoare executați APASASTART citind o cartelă, pe care era înregistrat programul LOADERABSOLUT. Aceasta realiza citirea programului INCARCATORSO de pe bandă sau de pe disc. Multe calculatoare foloseau în acest scop banda perforată (care uneori trebuia "ajutată" cu mâna să defileze prin cititor ☺). Imaginea "arhaică" a unei astfel de porniri este similară -cei în vîrstă probabil că își aduc aminte- cu pornirea în anii '50 a celebrului tractor KD, sau a unei drujbe, pornire care se face trăgând de o sfoară. ☺)

In prezent suportul magnetic de încărcare este primul sector (sectorul zero) al unui disc magnetic. Programele care realizează încărcarea se vor deosebi de cele de mai sus, numai prin înlocuirea lui 100 cu 128, 256, 512 sau 1024, deci cu lungimea unui sector.

Unele SO pretind o modalitate prin care i se indică sistemului de pe ce disc să încarce programele LOADERABSOLUT și INCARCATORSO: specificarea se face în memoria CMOS, la cheile pupitrului calculatorului etc. Alte sisteme stabilesc singure, în mod automat, de pe ce disc să facă încărcarea.

10 Gestiunea memoriei

10.1 Structură; calculul de adresă

10.1.1 Problematica gestiunii memoriei

Pentru a fi executat, un program are nevoie de o anumită cantitate de memorie. Dacă se lucrează în multiprogramare este necesar ca în memorie să fie prezente simultan mai multe programe. Fiecare program folosește zona (zonele) de memorie alocată (alocate) lui, independent de eventuale alte programe active.

Se știe că, în general, pe durata execuției unui program, necesarul de memorie variază. Această necesar variabil de memorie apare din două surse: folosirea *variabilelor dinamice* și *segmentarea*. Variabilele dinamice sunt alocate de către proces, de regulă dintr-o zonă de memorie numită *heap* și spațiul este eliberat tot de către proces. Alocarea și eliberarea se face prin perechi de funcții new – dispose, malloc – mfree, constructor – destructor etc.

Segmentarea este o tehnică prin care un program executabil este ocupat în entități distincte numite *segmente*. Segmentele pot fi: de cod, date sau de stivă. Pe durata vieții programului unele dintre segmente pot fi prezente în memorie, altele nu. Programul însuși poate cere încărcarea sau reîncărcarea unui segment, fie într-o zonă de memorie liberă, fie în locul altui (altor) segment(e). La construirea programului executabil se poate defini *structura de acoperire a segmentelor*. Spre exemplu, să presupunem că un program este segmentat și are forma din fig. 10.1a. Atunci segmentele active la un moment dat pot fi: A, AC, AB, AF, ACE sau ACD, aşa cum reiese din fig. 10.1b.

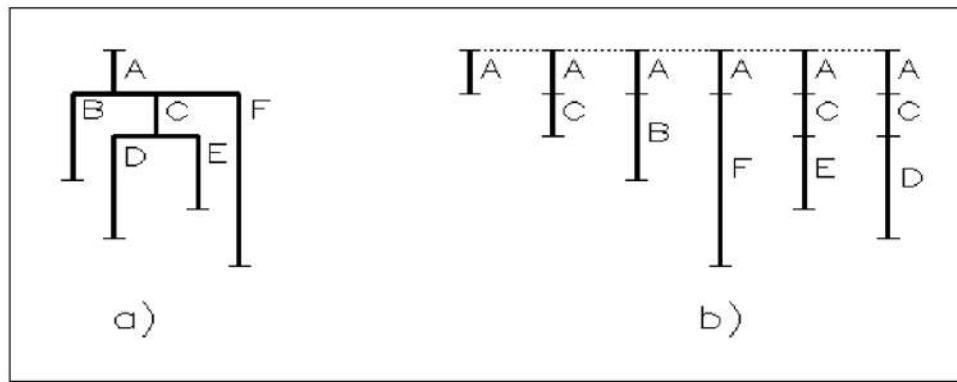


Figura 10.1 Instanțe posibile ale unui program segmentat

Evident, la fiecare din situațiile precedente se solicită o altă cantitate de memorie.

Spațiul de memorie principală al unui **SC**, adică ceea ce poate fi accesat în mod direct de către **CPU**, este în prezent încă *limitat*. Chiar dacă spațiul de memorie oferit de actualele **SC** este mult mai mare decât cel oferit acum 20 de ani sau chiar 10 ani (iar prețul de cost al memoriei a scăzut drastic), limitarea totuși rămâne. Din această cauză, **SO** cu sprijinul **SC** trebuie să gestioneze eficient folosirea acestui spațiu.

Problemele privind gestiunea memoriei sunt rezolvate la nivelul inferior de către **SC**, extins eventual cu o componentă de *management al memoriei*. La nivelul superior, rezolvarea se face de către **SO** în colaborare cu **SC**. **Principalele obiective ale gestiunii memoriei** sunt:

- Calculul de translatăre a adresei (relocare) ;
- Protecția memoriei;
- Organizarea și alocarea memoriei operative;
- Gestiona memoria secundare;
- Politici de schimb între proces, memoria operativă și memoria secundară.

Vom aborda fiecare dintre aceste obiective, cu excepția protecției memoriei. În general protecția memoriei este puternic dependentă de hardware, motiv pentru care nu o tratăm aici, într-un cadru general. Fiecare pereche sistem de calcul – sistem de operare are propria politică de protecție a memoriei, deci aceasta trebuie studiată în context concret.

10.1.2 Structura ierarhică de organizare a memoriei

În structura sa actuală, memoria unui sistem de calcul apare ca în fig. 10.2.

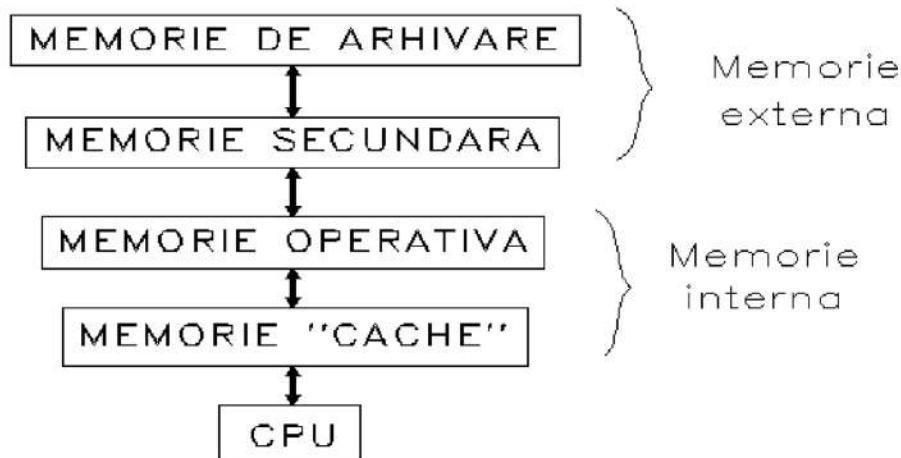


Figura 10.2 Structura memoriei unui sistem de calcul

Să prezintăm componentele de jos în sus.

Memoria "cache" conține informațiile cele mai recent utilizate de către **CPU**. Ea are o capacitate relativ mică, dar cu timp de acces foarte rapid. La fiecare acces, **CPU** verifică dacă data invocată se află în memoria "cache" și abia apoi solicită memoria operativă. Dacă este, atunci are loc schimbul între **CPU** și ea. Dacă nu, atunci data este căutată în nivelele superioare. Aplicând principiul vecinătății [40], data respectivă este adusă din nivelul la care se află, dar odată cu ea se aduce un număr de locații vecine ei, astfel încât, împreună să umple memoria "cache". Principiul de lucru este cel al memorilor tampon temporare, prezentat în 11.1. Intr-o secțiune viitoare vom prezenta în detaliu cum funcționează memoria cache.

In ce constă principiul vecinătății? P. Denning afirmă, pe baza unor studii de simulare temeinice, că dacă la un moment dat se solicită o dată dintr-un anumit loc atunci solicitarea din momentul următor se va face, cu mare probabilitate, la o dată din apropierea precedentei.

Memoria "cache" este practic prezentă la toate sistemele de calcul moderne. De exemplu, un notebook actual, Intel Pentium 2GHz poate avea o memorie cache de până la 1Mo.

Memoria operativă conține programele și datele pentru toate procesele existente în sistem. În momentul în care un proces este terminat și distrus, spațiul de memorie operativă pe care l-a ocupat este eliberat și va fi ocupat de alte procese. Capacitatea memoriei operative variază azi de la 128Mo până la 8 Go sau chiar mai mult. Viteza de acces este foarte mare, dar mai mică decât a memoriei "cache".

Memoria secundară apare la **SO** care dețin mecanisme de memorie virtuală. De asemenea, tot în cadrul memoriei secundare poate fi inclus spațiul disc de swap (vezi 8.2.1.1). Această memorie este privită ca o extensie a memoriei operative. Suportul ei principal este discul magnetic. Accesul la această memorie este mult mai lent decât la cea operativă.

Memoria de arhivare este gestionată de utilizator și constă din fișiere, baze de date și rezidente pe diferite suporturi magnetice (discuri, benzi, etc.).

Memoria "cache" și memoria operativă formează ceea ce cunoaștem sub numele de *memoria internă*. Accesul **CPU** la acestea se face în mod direct. Pentru ca **CPU** să aibă acces la datele din memoria secundară și de arhivare, acestea trebuie mai întâi mutate în memoria internă.

Din punct de vedere a performanțelor, privind fig. 10.2 de jos în sus se disting următoarele caracteristici ale componentelor memoriei:

- viteza de acces la memorie scade;
- prețul de cost pe unitatea de alocare scade;
- capacitatea de memorare crește.

10.1.3 Mecanisme de translatare a adresei

Adresarea memoriei constă în realizarea legăturii între un obiect al programului și adresa corespunzătoare din memoria operativă a **SC**. Pentru a explica mecanismele de translatare, să adoptăm câteva *notării*:

- **OP** notăm spațiul de nume al obiectelor din programul sursă: nume de constante, de variabile, de etichete, de proceduri etc.
- **AM** este adresa relativă din cadrul unui modul compilat.
- **AR** este adresa relocabilă – adresa relativă din cadrul unui segment dintr-un fișier executabil.
- **AF** notăm mulțimea adreselor fizice din memoria operativă la care face referire programul în timpul execuției.

Calculul de adresă este modalitatea prin care se ajunge de la un obiect sursă din **OP** la adresa lui fizică din **AF**. După cum se vede, acest calcul necesită trei faze, corespunzătoare fazelor în care se poate afla un program (vezi 8.2.1.2). Matematic vorbind, calculul de adresă se realizează prin compunerea a trei funcții: **c**, **l**, **t**, astfel:

$$OP \xrightarrow{c} AM \xrightarrow{l} AR \xrightarrow{t} AF$$

In fig. 10.3 sunt ilustrate fazele prin care trece un program componentele invocate și etapele calculului de adresă, de la textul sursă până la execuție.

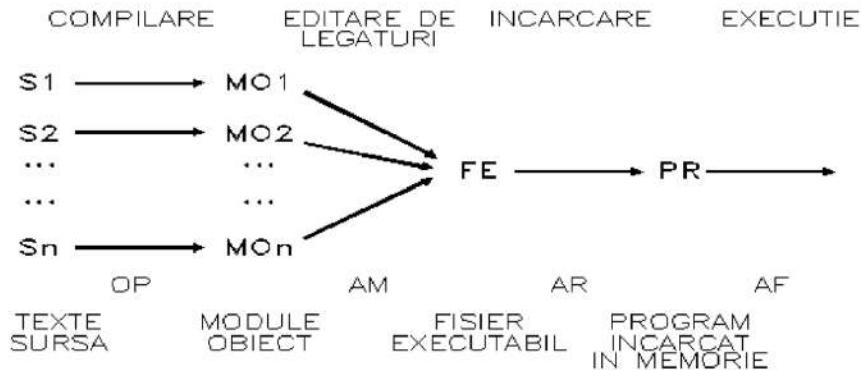


Figura 10.3 Fazele translatării unui program

10.1.3.1 Faza de compilare

Faza de *compilare* transformă un text sursă S_i într-un *modul obiect* MO_i . Corespunzător, numele obiectelor program sunt transformate în numere reprezentând **AM**, adică adrese în cadrul modulului obiect. În cadrul fiecărui modul obiect aceste adrese încep de la 0. Deci prima funcție din calculul de adresă:

$$c: OP \rightarrow AM$$

este executată de către compilator sau asamblor. Modul ei de evaluare depinde de limbajul, de compilatorul și de **SO** concret. Teoria compilării [33] are în vedere această evaluare, ea nu intră în scopurile noastre actuale. În fig. 10.4, prima parte, este prezentat un exemplu de aplicare a acestei funcții.

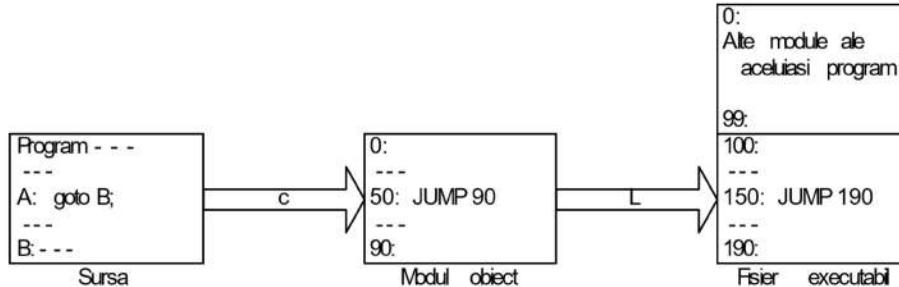


Figura 10.4 Translatare de la sursă la fișier executabil

10.1.3.2 Faza editării de legături

Faza de *editare de legături* grupează mai multe module formând un *fișier executabil*. Editorului de legături îi revine sarcina evaluării celei de-a doua funcții de calcul a adresei. Această funcție transformă adresele din cadrul modulelor în așa zisele *adrese relocabile* (*relocatable*). Funcția de legare este:

$$l: AM \rightarrow AR$$

Particularitățile de evaluare a acestei funcții sunt proprii editorului de legături. În fig. 10.4 sugerăm modul în care acționează compunerea celor două funcții.

10.1.3.3 Faza de încărcare și execuție

Funcția

$$t: AR \rightarrow AF$$

este *funcția de translatăre (relocare) a adresei*. În mod obișnuit ea este executată de către **CPU**. Translatarea depinde de tipul sistemului de calcul, în particular de existența dispozitivului de management al memoriei. Pentru a vedea principiul ei de lucru, presupunem că fișierul executabil conține înregistrări succesive cu instrucțiuni de forma celei din fig. 10.5.

ARI CO A1 A2 ... An

Figura 10.5 Formatul unei instrucțiuni mașină

Semnificațiile câmpurilor instrucțiunii sunt:

- *ARI* este adresa relocabilă a instrucțiunii;
- *CO* este codul operației,
- *A1, ..., An* sunt argumentele instrucțiunii mașină: nume de registri, constante (argumente *immediate*), adrese relocabile din memorie

Presupunem că fiecare instrucțiune începe într-o locație de memorie și că instrucțiunile sunt plasate una după cealaltă în locații succesive. Atunci încărcarea unui astfel de fișier se poate face folosind un încărcător analog lui LOADERABSOLUT, descris în 8.3 fig. 8.5.

Să presupunem că mașina dispune de un singur registru general pe care-l vom numi *A* (de la *registrator acumulator*). De asemenea, presupunem că între *A1, ..., An* există o singură adresă relocabilă. Evident că în realitate structura unui fișier executabil este mai complexă, dar pentru moment aceasta ne este suficientă.

Să adoptăm următoarele notații:

- *M[0..m]* conține locațiile memoriei operative;
- *pc* (Program Counter) indică adresa fizică a instrucțiunii care urmează a fi executată;
- *w* este conținutul instrucțiunii curente;
- *Opcode(w)* este o funcție care furnizează codul operației din instrucțiunea curentă.

Pentru fixarea ideilor, să presupunem că:

- 1 este codul adunării;
- 2 este codul operației de memorare (depunere din *A*) într-o anumită locație;
- 3 este codul instrucțiunii de salt necondiționat;
- §.a.m.d.
- *Address(w)* este o funcție care furnizează valoarea adresei relocabile aflată între argumentele instrucțiunii curente.

Cu aceste notații, în fig. 10.6 este descris modul de funcționare a **CPU** și de acțiune a funcției de translatăre *t*. De fapt, în fig. 10.6 este schițat un *interpreter* al limbajului care are instrucțiuni de forma celei din fig. 10.5.

```

pc = t(adresa-de-start-a-programului);
do {
    w = M[pc];           // operația fetch
    co = Opcode(w);
    adr = Adress(w);
    pc = pc+1;
    switch (co) {
        1: A:=A+M[t(adr)]; // adunare
        2: M[t(adr)]:=A;   // memorare
        3: pc:=t(adr);    // salt necondiționat
        - - -
    }
} while (false);

```

Figura 10.6 Funcționarea CPU și calculul funcției de translatare

10.2 Scheme simple de alocare a memoriei

10.2.1 Clasificarea tehniciilor de alocare

Problema alocării memoriei se pune în special la sistemele multiutilizator, motiv pentru care în continuare ne vom ocupa aproape exclusiv numai de aceste tipuri de sisteme. Tehnicile de alocare utilizate la diferite **SO** se împart în două mari categorii, fiecare categorie împărțindu-se la rândul ei în alte subcategorii, ca mai jos [19] [10].

- alocare reală:
 - la **SO** monoutilizator;
 - la **SO** multiutilizator:
 - cu partiții fixe (statică):
 - absolută;
 - relocabilă;
 - cu partiții variabile (dinamică);
- alocare virtuală:
 - paginată;
 - segmentată;
 - segmentată și paginată.

10.2.2 Alocarea la sistemele monoutilizator

La sistemele monoutilizator este disponibil aproape întreg spațiul de memorie. Gestiunea acestui spațiu cade exclusiv în sarcina utilizatorului. El are la dispoziție tehnici de *suprapunere (overlay)* pentru a-și putea rula programele mari. In fig. 10.6 este ilustrat acest mod de lucru.

Porțiunea dintre adresele 0 și $a-1$ este rezervată nucleului **SO**, care rămâne acolo de la încărcare și până la oprirea sistemului. Intre adresele c și $m-1$ (dacă memoria are capacitatea de m locații) este spațiu nefolosit de către programul utilizator activ. Evident, adresa c variază de la un program utilizator la altul.

10.2.3 Alocarea cu partiții fixe

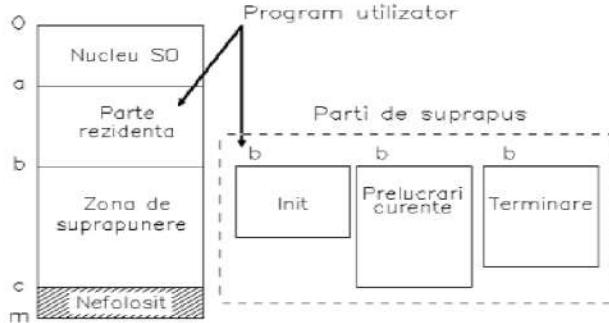


Figura 10.7 Alocarea memoriei la sistemele monoutilizator

Acest mod de alocare mai poartă numele de alocare statică sau alocare MFT - Memory Fix Tasks. El presupune decuparea memoriei în zone de lungime fixă numite *partiții*. O partiție este alocată unui proces pe toată durata execuției lui, indiferent dacă o ocupă complet sau nu. Un exemplu al acestui mod de alocare este descris în fig. 10.8. Zonele hașurate fac parte din partiții, dar procesele active din ele nu le utilizează.

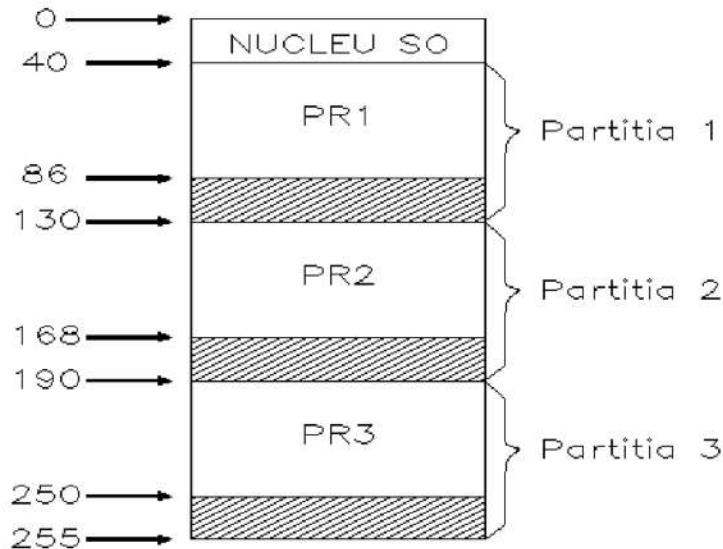


Figura 10.8 Exemplu de alocare cu partiții fixe

Alocarea absolută se face pentru programe pregătite de editorul de legături pentru a fi rulate într-o zonă de memorie prestabilită și numai acolo.

Mult mai folosită este **alocarea relocabilă**, la care adresarea în partiție se face cu bază și deplasament. La încărcarea unui program în memorie, în registrul lui de bază se pune adresa de început a partiției. În cazul sistemelor seriale cu multiprograme, dacă un proces este plasat spre execuție într-o partiție insuficientă, el este eliminat din sistem fără a fi executat.

De obicei, partițiiile au lungimi diferite. Una dintre problemele cele mai dificile este fixarea acestor dimensiuni. Dificultatea constă în faptul că nu se pot prevedea în viitor cantitățile de memorie pe care le vor solicita procesele încărcate în aceste partiții. Alegerea unor dimensiuni

mai mari scade probabilitatea ca unele procese să nu poată fi executate, dar scade și numărul proceselor active din sistem.

Acest mod de alocare este utilizat preponderent de către sistemele seriale. La fiecare partitie există un șir de procese care așteaptă să fie executate. Modul în care se organizează acest sistem de așteptare poate influența performanțele de ansamblu ale sistemului și poate eventual atenua efectul unei dimensionări defectuoase a partitiilor. În general există două moduri de legare a proceselor la partiti:

- *Fiecare partitie are coada proprie;* operatorul stabilește de la început care sunt procesele care vor fi executate în fiecare partitie.
- *O singură coadă pentru toate partitiile;* SO alege, pentru procesul care urmează să intre în lucru, în ce partitie se va executa.

Legarea prin cozi proprii partitiilor este mai simplă din punctul de vedere al SO. Primele sisteme multiutilizator au adoptat acest mod de legare. În schimb, legarea cu o singură coadă este mai avantajoasă, pentru faptul că se poate alege partitia cea mai potrivită pentru plasarea unui proces.

10.2.4 Alocarea cu partiti variabile

Acest mod de legare mai este cunoscut și sub numele de alocare dinamică sau alocare MVT - Memory Variable Task. El reprezintă o extensie a alocării cu partiti fixe, care permite o exploatare mai suplă și mai economică a memoriei SC. *In funcție de solicitările la sistem și de capacitatea de memorie încă disponibilă la un moment dat, numărul și dimensiunea partitiilor se modifică automat.*

În fig. 10.9 sunt prezentate mai multe stări succesive ale memoriei, în acest mod de alocare.

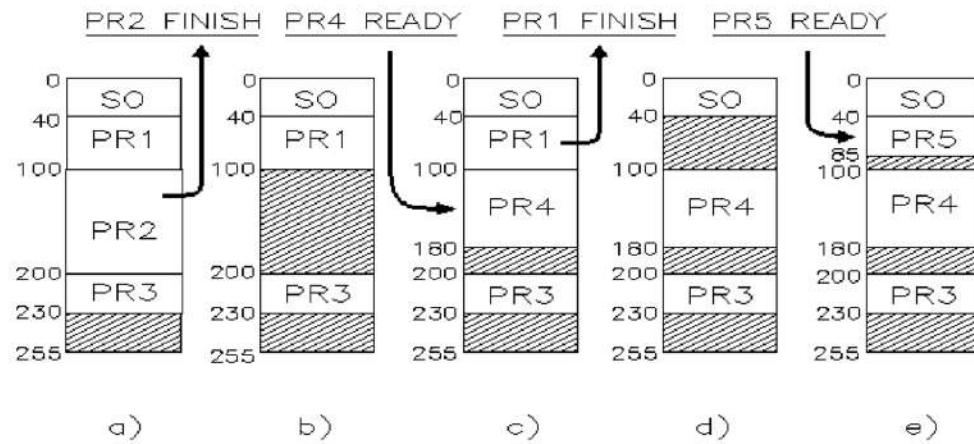


Figura 10.9 Evoluția proceselor la alocarea cu partiti variabile

In momentul în care procesul intră în sistem, el este plasat în memorie într-un spațiu în care începe cea mai lungă ramură a sa. Spațiul liber în care a intrat procesul este acum descompus în două partiti: una în care se află procesul, iar cealaltă într-un spațiu liber mai mic. Este ușor de observat că dacă sistemul funcționează timp îndelungat, atunci numărul spațiilor libere va crește, iar dimensiunile lor vor scădea. Fenomenul este cunoscut sub numele de *fragmentarea*.

internă a memoriei. După cum se va vedea, acest fenomen poate avea efecte neplăcute. În momentul în care un proces nu are spațiu în care să se încarce, **SO** poate lua una din următoarele trei decizii:

- Procesul așteaptă* până când i se eliberează o cantitate suficientă de memorie.
- SO** încearcă *alipirea unor spații libere vecine - colationare*, în speranța că se va obține un spațiu de memorie suficient de mare. Spre exemplu, dacă momentul care urmează după cel din fig. 10.9d este terminarea procesului *PR4*, atunci în gestiunea sistemului apar trei zone libere adiacente: prima de 15Ko, a doua de 80Ko, iar a treia de 20Ko (vezi fig. 10.10). **SO** poate (nu întotdeauna o și face în mod automat) să formeze din aceste trei spații unul singur de 115Ko, așa cum se vede în fig. 10.10.
- SO** decide efectuarea unei operații de *compactare a memoriei (relocate)* adică de deplasare a partițiilor active către partitia monitor pentru a se absorbi toate "fragmentele" de memorie neutilizate. Este posibil ca spațiul astfel obținut să fie suficient pentru încărcarea procesului. În fig. 10.11b este dat un exemplu de compactare.

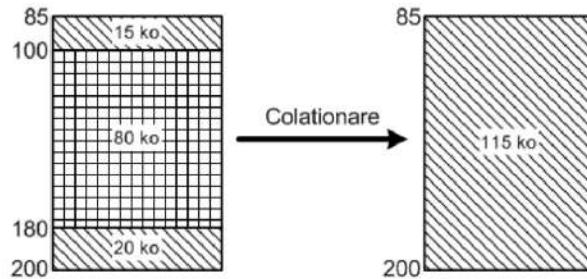


Figura 10.10 Colaționarea de spații libere vecine

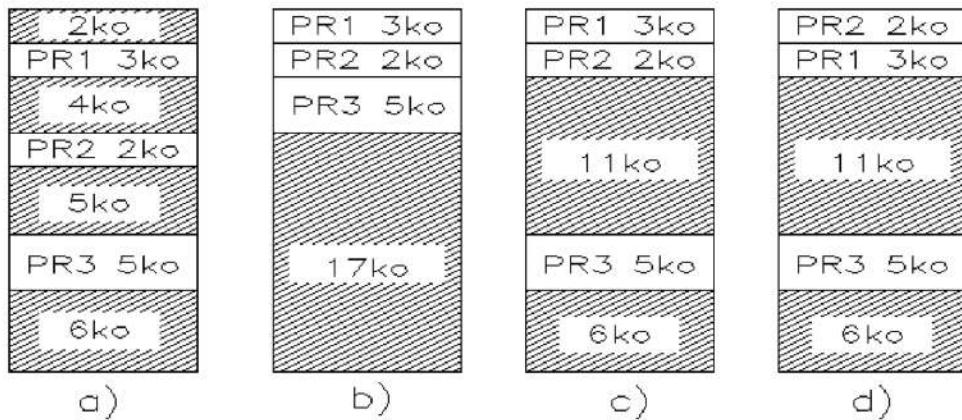


Figura 10.11 Posibilități de compactare prin relocate totală sau parțială

De regulă, compactarea este o operație costisitoare și în practică se aleg soluții de compromis, cum ar fi:

- Se lansează periodic compactarea (de exemplu la 10 secunde), indiferent de starea sistemului. În intervalul dintre compactări memoria apare ca un mozaic de spații ocupate care alternează cu spații libere. Procesele care nu au loc în memorie așteaptă compactarea sau terminarea altui proces.

- Se realizează o compactare parțială pentru a asigura loc numai procesului care așteaptă. Spre exemplu, dacă harta memoriei este cea din fig. 10.11a și un proces cere 10Ko, se poate realiza numai compactarea parțială din fig. 10.11c.
- Se încercă numai mutarea unora dintre procese cu coloanarea spațiilor rămase libere. Dacă reluăm exemplul de mai sus, este posibilă mutarea procesului PR2 în primul spațiu liber și problema este rezolvată, harta fiind cea din fig. 10.11d.

Intre alocările de tip MFT și MVT nu există practic diferențe hard. Alocarea MVT este realizată de cele mai multe ori prin intermediul unor rutine specializate, eventual microprogramate, deci de către **SO**.

Alocarea MVT a fost utilizată mai întâi la **SC IBM-360** sub **SO OS-360 MVT**, apoi la **PDP 11/45**.

10.3 Mecanisme de memorie virtuală

Termenul de *memorie virtuală* este de regulă asociat cu capacitatea de a adresa un spațiu de memorie mai mare decât este cel disponibil la memoria operativă a **SC** concret. Conceptul este destul de vechi, el apărând odată cu **SO ATLAS** al Universității Manchester, Anglia, 1960 [19]. Se cunosc două metode de virtualizare, mult înrudite după cum vom vedea în continuare. Este vorba de alocarea paginată și alocarea segmentată. Practic, toate sistemele de calcul actuale folosesc, într-o formă sau alta, mecanisme de memorie virtuală.

10.3.1 Alocarea paginată

Alocarea paginată a apărut la diverse **SC** pentru a evita fragmentarea excesivă, care apare la alocarea MVT, și drept consecință, la evitarea aplicării relocării. Această alocare presupune cinci lucruri și anume:

- Instrucțiunile și datele fiecărui program sunt împărțite în zone de lungime fixă, numite *pagini virtuale*. Fiecare **AR** (notațiile din 10.1.3) aparține unei pagini virtuale. Paginile virtuale se păstrează în memoria secundară.
- Memoria operativă este împărțită în zone de lungime fixă, numite *pagini fizice*. Lungimea unei pagini fizice este fixată prin hard. Paginile virtuale și cele reale au aceeași lungime, lungime care este o putere a lui 2, și care este o constantă a sistemului (de exemplu 1Ko, 2Ko etc).
- Fiecare **AR** este o pereche de forma: *(p,d)*, unde *p* este numărul paginii virtuale, iar *d* adresa în cadrul paginii.
- Fiecare **AF** este de forma *(f,d)*, unde *f* este numărul paginii fizice, iar *d* adresa în cadrul paginii.
- Calculul funcției de translatăre $t : AR \rightarrow AF$ se face prin hard, conform schemei din fig. 10.12.

Dacă prin **M[0..m]** notăm memoria operativă, prin *k* puterea lui 2 (numărul de biți) care dă lungimea unei pagini, prin **TP** adresa de start a tabelei de pagini, atunci algoritmul calculul funcției *t* este:

$$t(p, d) = M[TP + p] * 2^k + d$$

Acest calcul este valabil atunci când tabela de pagini ocupă un spațiu în memoria operativă. Există însă **SC** ce dispun, hard, de o memorie specială de capacitate mică, numită *memorie asociativă*. Calitatea ei fundamentală este *adresarea prin conținut*, ceea ce înseamnă că

găsește locația care are un conținut specificat, căutând simultan în toate locațiile ei. Memoria asociativă conține atâtea locații câte pagini fizice are. În fiecare locație a memoriei associative este trecut numărul paginii virtuale care se află în pagina fizică având numărul de ordine identic cu numărul de ordine al locației de memorie asociativă. Atunci când se dă un număr de pagină virtuală, se obține automat numărul paginii fizice care o găzduiește.

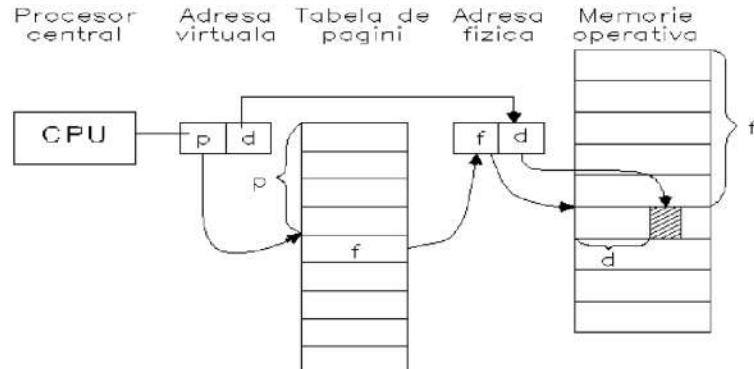


Figura 10.12 Translatarea unei pagini virtuale într-o fizică

Fiecare proces are propria lui tabelă de pagini, în care este trecută adresa fizică a paginii virtuale, dacă ea este prezentă în memoria operativă. La încărcarea unei noi pagini virtuale, aceasta se depune într-o pagină fizică liberă. Deci, în memoria operativă, paginile fizice sunt distribuite în general necontiguu, între mai multe procese. Spunem că are loc o *proiectare a spațiului virtual peste cel real*. Este posibil, la un moment dat, să existe situația din fig. 10.13.

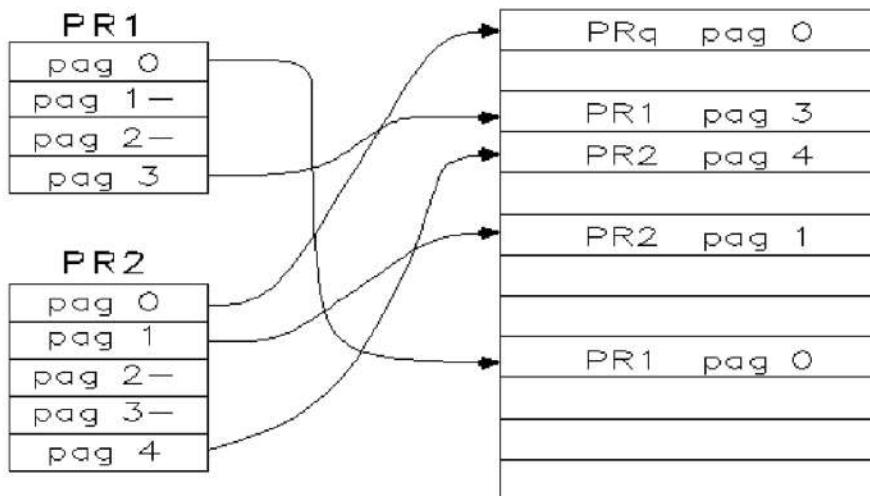


Figura 10.13 Două procese, într-o alocare paginată

Acest mecanism are avantajul că folosește mai eficient memoria operativă, fiecare program ocupând numai memoria strict necesară la un moment dat. Un avantaj colateral, dar nu de neglijat, este folosirea în comun a unei porțiuni de cod.

Să presupunem că avem un editor de texte al cărui cod (instrucțiuni pure, fără date) ocupă două pagini. Mai presupunem că fiecare utilizator consumă câte o pagină pentru datele proprii

de editat. Dacă sunt trei utilizatori, ei trebuie în mod normal să "consume" nouă pagini din memoria operativă. În fig. 10.14 se arată cum pot fi consumate numai cinci pagini.

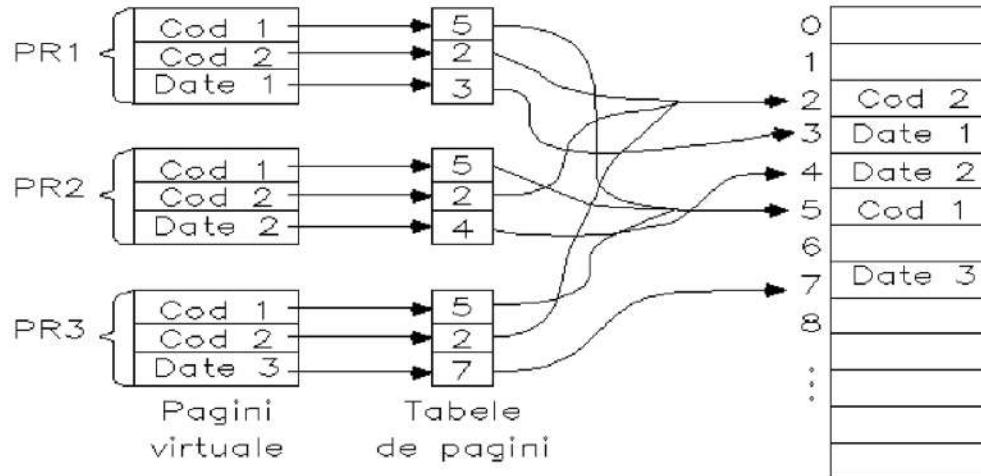


Figura 10.14 Folosirea în comun a unui cod

10.3.2 Alocare segmentată

Atunci când am vorbit despre alocarea reală, am văzut că fiecare proces trebuia să ocupe un spațiu contigu de memorie, numit partii. Ceea ce introduce nou mecanismul de alocare segmentată este faptul că textul unui program poate fi plasat în zone de memorie distincte, fiecare dintre ele conținând o bucată de program numită *segment*. Singura deosebire principală dintre alocarea paginată și cea segmentată este aceea că segmentele sunt de lungimi diferite. În fig. 10.15 am ilustrat această situație cu locurile ce vor fi ocupate de un program format din trei segmente.

În mod analog cu alocarea paginată, o adresă virtuală este o pereche (s, d) , unde s este numărul segmentului, iar d este adresa în cadrul segmentului. Adresa reală (fizică) este o adresă obișnuită. Fiecare proces activ are o *tabelă de segmente*. Fiecare intrare în această tabelă conține *adresa de început a segmentului*. Calculul de adresă se face analog celui de la alocarea paginată. Presupunem că la adresa **TS** se află începutul tabelei de segmente. Cu notațiile obișnuite, funcția t de translatare a adresei se calculează astfel:

$$t(s, d) = M[TS+s]+d$$

Pe lângă avantajul net față de alocările pe partiții, alocarea segmentată mai prezintă încă două avantaje:

- Se pot crea *segmente reentrantă*, - cod pur - care pot fi folosite în comun de către mai multe procese. Pentru aceasta este suficient ca toate procesele să aibă în tabelele lor aceeași adresă pentru segmentul pur. A se vedea aceeași problemă discutată la alocarea paginată.
- Se poate realiza o foarte bună protecție a memoriei. Fiecare segment în parte poate primi alte drepturi de acces, drepturi trecute în tabela de segmente. La orice calcul de adresă se pot face și astfel de verificări. Pentru detalii se pot consulta lucrările [10], [19].

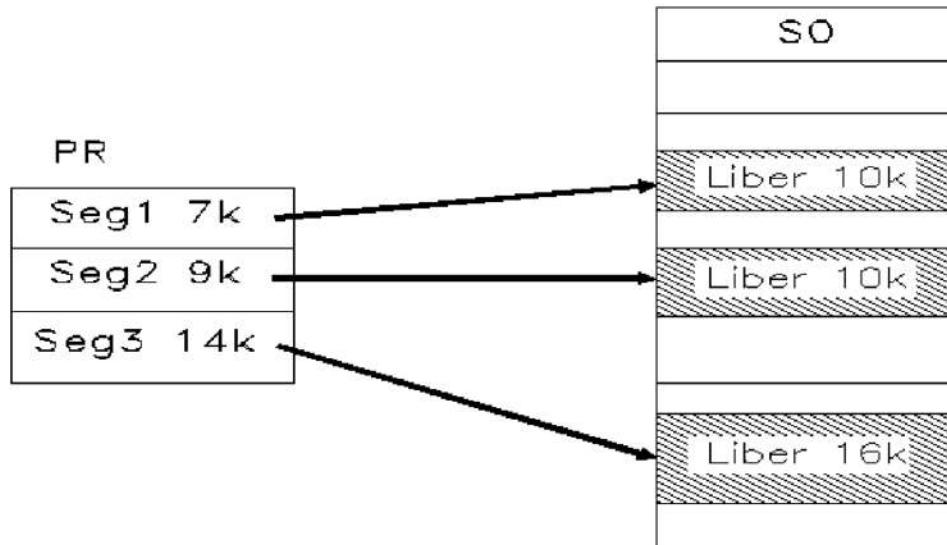


Figura 10.15 Alocare necontiguă prin segmentare

10.3.3 Alocare segmentată și paginată

La alocarea segmentată am arătat că adresa fizică este una oarecare. Este deci posibil să apară fenomenul de fragmentare, despre care am vorbit la alocarea cu partiții variabile (10.2.4). Ideea alocării segmentate și paginate este aceea că alocarea spațiului pentru fiecare segment să se facă paginat.

Pentru aceasta, mai întâi *fiecare proces* activ are *propria lui tabelă de segmente*. Apoi, *fiecare segment* dintre cele încărcate în memorie are *propria lui tabelă de pagini*. Fiecare intrare în tabela de segmente are un câmp rezervat adresei de început a tablei de pagini proprii segmentului respectiv, aşa cum se vede fig. 10.16.

O adresă virtuală este de forma: (s, p, d) , în care s este numărul segmentului, p este numărul paginii virtuale în cadrul segmentului, iar d este deplasamentul în cadrul paginii. O adresă fizică este de forma: (f, d) , unde f este numărul paginii fizice, iar d este deplasamentul în cadrul paginii.

Fie k constanta ce dă dimensiunea unei pagini (2^k), TS adresa de început a tablei de segmente a unui proces și presupunem că primul câmp al fiecărei intrări din tabela de segmente este pointerul spre tabela lui de pagini, atunci funcția t de translatare se calculează astfel:

$$t(s, p, d) = M[M[TS+s]+p]*2^k+d$$

Printre **SO** remarcabile care utilizează acest mod de alocare trebuie amintit în primul rând MULTICS, cel care a introdus de fapt acest mod de alocare. De asemenea, **SO VAX/VMS** adoptă acest mod de alocare.

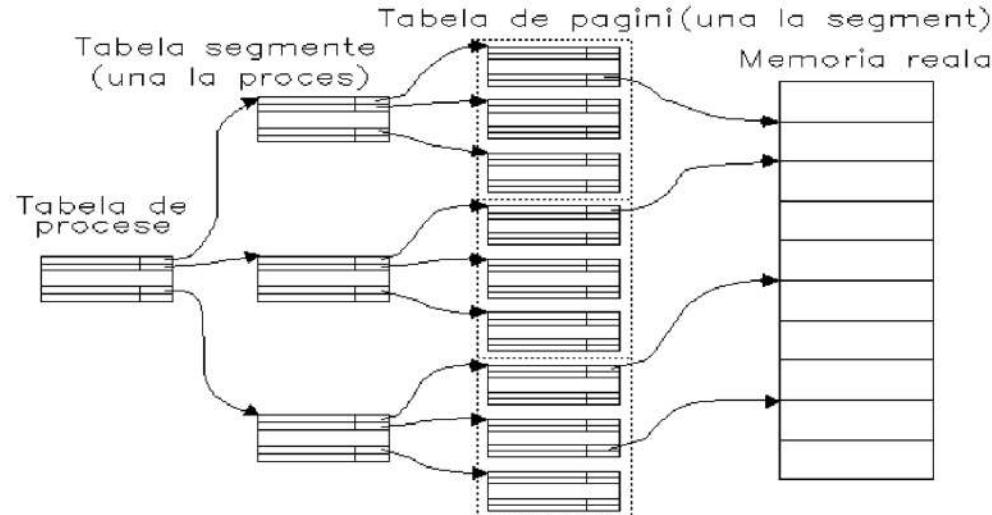


Figura 10.16 Alocare segmentată și paginată

Calculatoarele IBM-PC cu microprocesor cel puțin 80386, dispun de un mecanism hard de gestiune paginată și segmentată a memoriei extinse. Conceptual el funcționează așa cum am arătat mai sus, completat evident cu o serie de particularități legate de adresarea și protecția memoriei la acest tip de mașină.

10.4 Planificarea schimburilor cu memoria

10.4.1 Intrebările gestiunii memoriei și politici de schimb.

Pe lângă mecanismele de alocare descrise în 10.2 și 10.3, sistemul de operare trebuie să rezolve o serie de probleme care se pot ivi la modul de alocare respectiv. Rezolvarea acestor probleme înseamnă răspunsul la întrebările: CAT? UNDE? CAND? CARE?

Intrebarea "CAT?", apare atunci când se pune problema cantității de memorie alocată. La alocarea pe partii, se alocă la început toată cantitatea cerută; avem deci o *alocare statică*. La alocarea paginată avem o *alocare dinamică*, fiecare program consumă numai memoria necesară la un moment dat. În 10.3 am prezentat aceste tehnici.

Intrebarea "UNDE?", apare la alocarea cu partii variabile. Atunci când un program cere intrarea în sistem, trebuie luată decizia: dintre locurile goale pe care le poate ocupa, unde va fi plasat programul? Rezolvările posibile sunt cunoscute în literatură sub numele de *politici de plasare*. Acestea au o utilizare răspândită, depășind cadrul sistemului de operare. Din acest motiv (cât și datorită farmecului lor) le vom dedica o secțiune specială.

Intrebarea "CAND?", apare cel puțin în două situații: la sistemele cu paginare și la alocarea cu partii variabile. În sistemele cu paginare, trebuie stabilit momentul în care o pagină virtuală este depusă într-o pagină fizică. În literatură, tehniciile de răspuns sunt cunoscute sub numele de *politici de încărcare (fetch)*. Le vom dedica și acestora o secțiune specială.

Aceeași întrebare "*CAND?*", apare pentru a decide momentele de *compactare (relocare)* a memoriei la alocarea cu partiții variabile. În 10.2.4 am arătat în ce constă compactarea și am dat trei posibilități de acțiune în acest sens.

Intrebarea "CARE?", apare la sistemele cu paginare. Să presupunem că la un moment dat toate paginile fizice sunt ocupate. Dacă un program mai cere încărcarea unei pagini, atunci este necesar ca una din paginile fizice să fie evacuată, în memoria secundară, pentru a-i se face loc noiui pagini. Această manevră poartă numele (după cum am mai spus) de *swapping*. Alegerea paginii care va fi înlocuită face obiectul unor *politici de înlocuire (replacement)*, cărora le vom dedica o secțiune separată.

Deși, face parte dintre metodele implementate hard, deci nu implică deloc sistemul de operare, credem că este cazul să vedem puțin *cum funcționează o memorie cache?* Vom dedica și acesteia o secțiune specială.

10.4.2 Politici de plasare.

10.4.2.1 Metode de plasare și structuri de date folosite

Nu numai sistemul de operare, ci foarte multe programe de aplicații solicită în timpul execuției lor diverse cantități de memorie. Vom presupune că întreaga cantitate de memorie solicitată la un moment dat este formată dintr-un sir de octeți consecutivi. De asemenea, presupunem că există un depozit de memorie (numit *heap* în limbajele de programare) de unde se poate obține memorie liberă.

Rezolvarea cererilor presupune existența a două rutine. O primă rutină are sarcina de a *ocupa (aloca)* o zonă de memorie și de a întoarce adresa ei de început. De exemplu, funcția **malloc** din limbajul C și operatorul **new** din limbajul C++ au acest rol. O a doua rutină are rolul de a *elibera* spațiul alocat anterior, în vederea refolosirii lui.

Analogia dintre cele spuse mai sus și alocarea cu partiții variabile este mai mult decât evidentă. Problema politicilor de plasare nu lipsește practic din nici un curs de sisteme de operare. Dintre lucrările mai consistente în acest domeniu amintim [4], [10], [22], [49].

Dată fiind importanța acestor politici, vom da câteva metode de organizare a spațiului de memorie și algoritmii de ocupare și alocare corespunzători. Pe lângă cerințele de funcționalitate, este bine ca plasarea succesivă a programelor (zonelor alocate) să împiedice, pe cât posibil, fragmentarea excesivă a memoriei operative. Altfel, este posibil ca cererile de memorie mai mari să nu poată fi servite deși sistemul dispune per total de memoria necesară.

Dintre metodele de plasare, cele mai răspândite sunt următoarele patru:

- Metoda primei potriviri (First-fit).
- Metoda celei mai bune potriviri (Best-fit).
- Metoda celei mai rele potriviri (Worst-fit).
- Metoda alocării prin camarați (Buddy-system).

Vom analiza pe rând fiecare dintre ele. Primele trei sunt foarte asemănătoare, iar ultima este oarecum deosebită. Pentru început ne vom ocupa de primele trei.

Dată fiind fragmentarea inherentă a memoriei cea mai convenabilă structură de date pentru regăsirea zonelor libere este *lista înlanțuită*. Fiecare nod al listei va descrie o zonă de memorie liberă specificându-i adresa de început, lungimea și adresa nodului următor. Cum însă această listă înlanțuită trebuie și ea să fie stocată în undeva memorie, cea mai convenabilă soluție este ca fiecare nod să fie stocat la începutul zonei de memorie pe care o descrie. În această abordare, un nod va conține doar lungimea zonei de memorie și adresa următoarei zone libere (adresa următorului nod). Un nod nu va mai conține adresa de început a zonei la care se referă pentru că este implicită prin adresa lui de memorie. Un nod al acestei liste înlanțuite se numește în literatura de specialitate *cuvânt de control*.

Vom prezenta în continuare câțiva algoritmi de plasare a cererilor de memorie. Pentru a trata unitar alocarea și eliberarea octetelor, vom adopta următoarea convenție. Fiecare zonă liberă sau ocupată de memori începe cu un cuvânt de control. Câmpul **lung** (aflat în a doua jumătate a cuvântului de control) al lui indică numărul de octeți liberi de *după* cuvântul de control. Pentru zonele libere de memorie pointerul **next** (aflat în prima jumătate a cuvântului de control) indică următoarea zonă liberă. Pentru zonele de memorie ocupate acest pointer nu este folosit. În fig. 10.17, 10.18 și 10.19, cuvintele de control sunt subliniate cu linii îngroșate, iar pointerii la zone sunt indicați prin săgeți. De asemenea, lungimea cuvântului de control este notată cu *c* și este o constantă specifică sistemului de operare. După cum vom vedea apar situații în care numărul de octeți alocati depășește (cu maximum *c*) numărul de octeți solicitați. O zonă ocupată este reperată *după* cuvântul ei de control.

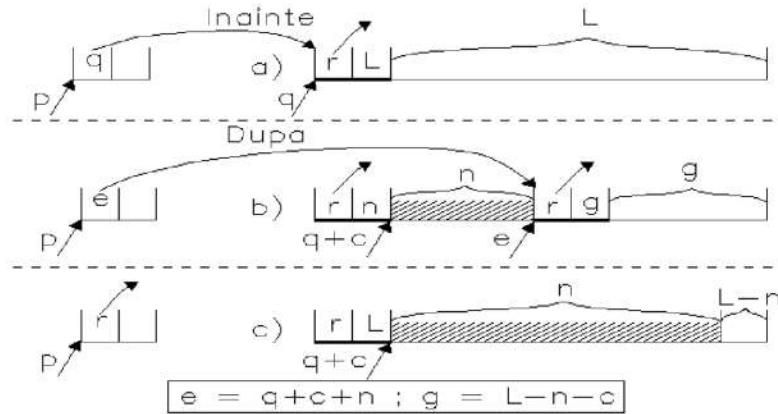
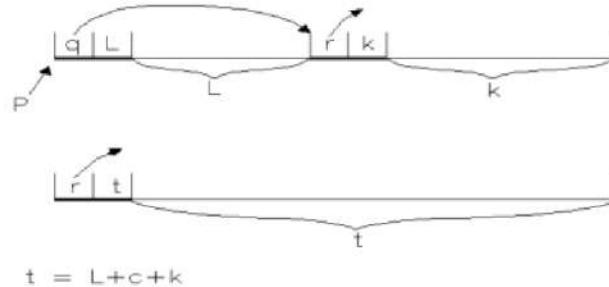


Figura 10.17 Alocarea de octeți într-o zonă liberă

În fig. 10.17a este prezentată o zonă liberă al cărei cuvânt de control începe la adresa *q* și are *L* octeți liberi. Presupunem că se cere alocarea de *n* octeți și că *L>n*. Ca rezultat al alocării, va apărea una dintre situațiile din fig. 10.17b sau 10.17c. Zona hașurată reprezintă octeții ceruți pentru alocare. Situația din fig. 10.17b apare dacă *L-n > c+1*, adică spațiul rămas în zonă permite crearea unei zone libere de cel puțin un octet. În cazul în care această condiție nu este satisfăcută, se alocă întreaga zonă și ultimii *L-n* octeți rămân nefolosiți (fig. 10.17c).

Inainte de a descrie procedura inversă (de eliberare), să ne ocupăm de problema *comasării a două zone libere adiacente*. Pentru simplificare, vom presupune că lista zonelor libere este păstrată în ordinea crescătoare a adreselor. Situația în care este posibilă concatenarea a două zone libere este ilustrată în fig. 10.18.

**Figura 10.18 Comasarea a două zone libere adiacente**

In fig. 10.19 este ilustrat un exemplu de eliberare a unei zone. La eliberare, partitia devenită liberă se repune în lista de spații libere. De asemenea, se verifică dacă nu cumva partitia proaspăt eliberată poate fi comasată cu una vecină din dreapta sau din stânga ei. Verificând sistematic, la fiecare eliberare, dacă este posibilă concatenarea, nu vor exista două zone libere adiacente.

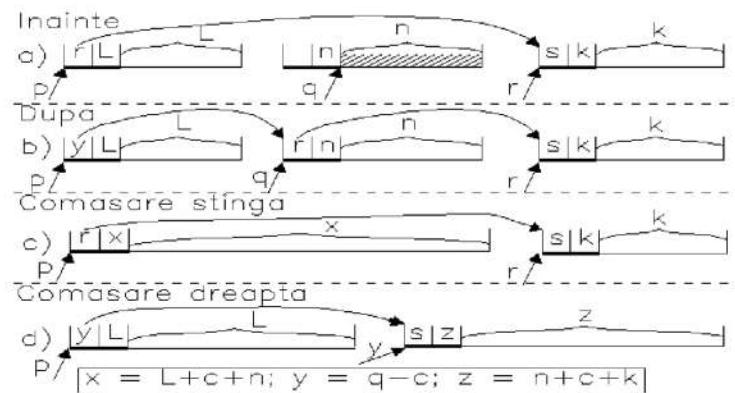
In fig. 10.19a este prezentată situația înainte de eliberarea zonei. In fig. 10.19b se prezintă situația în care nu este posibilă nici un fel de comasare. Posibilitatea de *comasare la stânga* are loc dacă:

$$p+2*c = q$$

Ca rezultat se obține zona din fig. 10.19c. O *comasare la dreapta* are loc dacă:

$$q+n = r$$

Ca efect se obține zona din fig. 10.19d.

**Figura 10.19 Eliberarea unei zone ocupate**

Acum este momentul să prezintăm cele patru politici de plasare. Primele trei le vom detalia utilizând descrierile procedurilor de mai sus. Punctul de pornire al fiecărui astfel de algoritm este faptul ca în momentul pornirii sistemului, se alocă întreaga zonă de memorie sub forma unei singure zone libere.

10.4.2.2 Metoda primei potriviri (First-fit).

Esența metodei constă în aceea că partitia solicitată este alocată în prima zonă liberă, în care începe. Principalul avantaj al metodei este simplitatea căutării de spațiu liber. Pentru această metodă, structura listei de spații libere prezentate mai sus este cea mai adecvată.

10.4.2.3 Metoda celei mai bune potriviri (Best-fit).

Esența metodei constă în căutarea acelei zone libere, care lasă după alocare cel mai puțin spațiu liber. Metoda Best-Fit a fost larg utilizată mulți ani. Ea pare a fi destul de bună, deoarece economisește zonele de memorie mai mari astfel încât dacă ulterior va fi nevoie de ele vor fi disponibile. Există însă și obiecții, dintre care amintim două: timpul suplimentar de căutare și proliferarea blocurilor libere de lungime mică, adică *fragmentarea internă excesivă* (vezi 10.2.4).

Primul neajuns este eliminat parțial dacă lista de spații libere se păstrează, nu în ordinea crescătoare a adreselor, ci *în ordinea crescătoare a lungimilor spațiilor libere*. Din păcate, în acest caz problema comasării zonelor libere adiacente se complică foarte mult.

10.4.2.4 Metoda celei mai rele potriviri (Worst-fit).

Metoda Worst-Fit este oarecum duală metodei Best-Fit. Esența ei constă în căutarea acelei zone libere care lasă după alocare cel mai mult spațiu liber.

Deși, numele ei sugerează că este vorba de o metodă slabă, în realitate nu este chiar aşa. Faptul, că după alocare rămâne un spațiu liber mare, este benefic, deoarece în spațiul rămas poate fi plasată, în viitor, o altă partitură. Fragmentarea internă probabil că nu evoluează prea rapid, însă timpul de căutare este mai mare decât cel de la metoda primei potriviri.

Să în acest caz este posibil ca lista de spații libere să se păstreze nu în ordinea crescătoare a adreselor, ci *în ordinea descrescătoare a lungimilor spațiilor libere*. Din păcate, în acest caz problema comasării zonelor libere adiacente se complică, de asemenea, foarte mult.

10.4.2.5 Metoda alocării prin camarazi (Buddy-system).

Metoda alocării prin camarazi (Buddy) este deosebit de interesantă. Ea exploatează reprezentarea binară a adreselor și faptul că din rațiuni tehnologice, dimensiunea memoriei interne este un multiplu al unei puteri a lui doi. Fie $c * 2^n$ dimensiunea memoriei interne. De exemplu, memoria unui IBM PC XT (unul dintre primele calculatoare personale) era 640 Ko, adică $10 * 2^{16}$ octeți.

Să notăm cu n cea mai mare putere a lui 2 prin care se poate exprima dimensiunea memoriei interne. În exemplul de mai sus $n = 16$. Din rațiuni practice, se stabilește ca unitate de alocare a memoriei tot o putere a lui 2. Fie m această putere a lui 2. Pentru exemplul de mai sus să considerăm că unitatea de alocare este 256 octeți, adică 2^8 .

La sistemele Buddy, dimensiunile spațiilor ocupate și a celor libere sunt de forma 2^k , unde $m \leq k \leq n$. Ideea fundamentală este de a păstra liste separate de spații disponibile pentru

fiecare dimensiune 2^k dintre cele de mai sus. Vor exista astfel $n-m+1$ liste de spații disponibile. În exemplul de mai sus, vom avea 9 liste: lista de ordin 8 având dimensiunea unui spațiu de 256 octeți, lista de ordin 9 cu spații de dimensiune 512 etc. Ultima listă va fi de ordinul 16 și poate avea maximum 10 spații a către 65536 (2^{16}) octeți fiecare.

Prin definiție, fiecare spațiu liber sau ocupat de dimensiune 2^k are adresa de început un multiplu de 2^k .

Tot prin definiție, două spații libere de ordinul k se numesc *camarazi (Buddy)* de ordin k , dacă adresele lor A1 și A2 verifică:

$$A1 < A2, \quad A2 = A1 + 2^k \text{ și } A1 \bmod 2^{(k+1)} = 0$$

sau

$$A2 < A1, \quad A1 = A2 + 2^k \text{ și } A2 \bmod 2^{(k+1)} = 0$$

Atunci când într-o listă de ordin k apar doi camarazi, sistemul îi concatenează într-un spațiu de dimensiune $2^{(k+1)}$.

Alocarea într-un sistem Buddy se desfășoară astfel:

- Se determină cel mai mic număr p , cu $m \leq p \leq n$, pentru care numărul o de octeți solicită verifică: $o \leq 2^p$
- Se caută, în această ordine, în listele de ordin $p, p+1, p+2, \dots, n$ o zonă liberă de dimensiune cel puțin o .
- Dacă se găsește o zonă de ordin p , atunci aceasta este alocată și se șterge din lista de ordinul p .
- Dacă se găsește o zonă de ordin $k > p$, atunci se alocă primii 2^p octeți, se șterge zona din lista de ordin k și se crează în schimb alte $k-p$ zone libere, având dimensiunile:

$$2^p, 2^{(p+1)}, \dots, 2^{(k-1)}$$

In fig. 10.20 este dat un astfel de exemplu. Se dorește alocarea a 1000 octeți, deci $p = 10$. Nu s-au găsit zone libere nici de dimensiune 2^{10} , nici 2^{11} și nici 2^{12} . Prima zonă liberă de dimensiune 2^{13} are adresa de început $5*2^{13}$ și am notat-o cu **I** în fig. 10.20. Ca rezultat al alocării a fost ocupată zona **A** de dimensiune 2^{10} și au fost create încă trei zone libere: **B** de dimensiune 2^{10} , **C** de dimensiune 2^{11} și **D** de dimensiune 2^{12} . Zonele **B**, **C**, **D** se trec respectiv în listele de ordine 10, 11 și 12, iar zona **I** se șterge din lista de ordin 13.

Eliberarea într-un sistem Buddy a unei zone de dimensiune 2^p este un proces invers alocării. Astfel:

1. Se introduce zona respectivă în lista de ordin p .
2. Se verifică dacă zona eliberată are un camarad de ordin p . Dacă da, atunci zona este comasată cu acest camarad și formează împreună o zonă liberă de dimensiune $2^{(p+1)}$. Atât zona eliberată cât și camaradul ei se șterg din lista de ordinul p , iar zona nou apărută se trece în lista de ordin $p+1$.
3. Se execută pasul 2 în mod repetat, mărind de fiecare dată p cu o unitate, până când nu se mai pot face comasări.

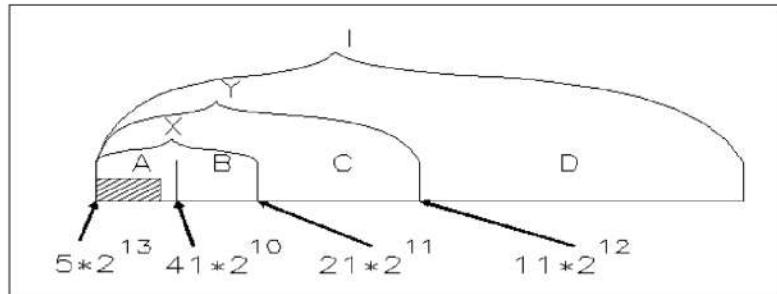


Figura 10.20 Alocare în sistem Buddy

De exemplu, să presupunem că în fig. 10.20 sunt libere la un moment dat zonele **A**, **C**, **D**, iar zona **B** este ocupată. La momentul următor, se eliberează și zona **B**. În conformitate cu pașii descriși mai sus, se execută următoarele acțiuni:

- Se trece zona **B** în lista de ordin **10**.
- Se depistează că zonele **A** și **B** sunt camarazi. Drept urmare, cele două zone sunt comasate și formează o nouă zonă **X**. Zona **X** se trece în lista de ordin **11**, iar zonele **A** și **B** se sterg din lista de ordin **10**.
- Se depistează că zonele **X** și **C** sunt camarazi. Drept urmare, ele sunt comasate și formează o zonă **Y** care se trece în lista de ordin **12**, înlocuind zonele **X** și **C** din lista de ordin **11**.
- În sfârșit, se depistează că **Y** și **D** sunt camarazi. Ele sunt sterse din lista de ordin **12**, iar în lista de ordin **13** se introduce rezultatul comasării lor.

Alocarea Buddy are multe avantaje. Unul dintre ele, deloc de neglijat, se referă la manipularea comodă a adreselor de zone. Se știe că un număr binar este multiplu de **2** dacă el se termină cu **k** zerouri binare. Rezultă deci că adresele a doi camarazi de ordin **k** diferă doar prin bitul de pe poziția **k** (numerotarea începând cu **0**) și ambele se termină prin **k** zerouri. Spre exemplu, adresele zonelor **A** și **B** din fig. 10.20 sunt:

$$\begin{array}{ll} A: & 101000000000000000 \\ B: & 101001000000000000 \end{array}$$

^

Evident, astfel de teste se fac deosebit de ușor folosind instrucțiunile mașină care operează asupra biților.

Până în prezent nu s-a găsit un criteriu solid de comparare a acestor patru metode de plasare. Compararea lor se face empiric, eventual prin simulare. De multe ori, se adoptă o metodă mai simplă, poate și numai din rațiuni tehnologice. În orice caz, trebuie necondiționat să se aibă în vedere și concluziile, mai mult sau mai puțin empirice, rezultate din experiență. Una dintre acestea spune că nu întotdeauna o metodă mai sofisticată este și mai bună!

10.4.3 Politici de încărcare.

La sistemele cu alocare paginată, în momentul lansării în execuție a programului acesta nu are nici o pagină în memorie. La prima solicitare a programului, sistemul de operare îi va aduce în memorie numai pagina solicitată. Dacă este vorba de un program mare, acesta va funcționa normal un timp, după care va cere din nou o pagină care nu este în memorie etc. Intrebarea

care se pune este: *când să se aducă o anumită pagină în memorie, pentru ca cererile de pagini să se reducă?*

O soluție simplă, dar evident ineficientă, este *încărcarea la început a tuturor paginilor*. Prin aceasta va dispărea însuși efectul mecanismului de paginare! O altă modalitate constă în aducerea unei pagini *la cerere*, adică atunci când este ea solicitată. Această modalitate pare a fi cea mai naturală, și ea este într-adevăr și cea utilizată în sistemele de operare moderne.

Există însă metode de încărcare prin care se aduc pagini *în avans*. Astfel, odată cu o pagină se aduc și câteva pagini vecine, în ipoteza că ele vor fi invocate în viitorul apropiat. O evidență statistică a utilizării paginilor, poate furniza, cu o oarecare probabilitate, care ar fi paginile invocabile în viitor. Dacă se poate, acestea sunt aduse în avans în memoria operativă.

Legat de încărcarea în avans, încă din 1968, P.J. Denning [10] a emis *principiul vecinătății*: adresele de memorie solicitate de un program nu se distribuie uniform pe întreaga memorie folosită, ci se grupează în jurul unor centre. Apelurile în apropierea acestor centre sunt mult mai frecvente decât apelurile de la un centru la altul.

Acest principiu sugerează o politică simplă de încărcare în avans a unor pagini. Se stabilește o așa zisă *memorie de lucru* [48] compusă din câteva pagini. Atunci când se cere aducerea unei pagini de pe disc, în memoria de lucru sunt încărcate câteva pagini vecine acesteia. În conformitate cu principiul vecinătății, este foarte probabil ca următoarele referiri să fie făcute în cadrul memoriei de lucru.

10.4.4 Politici de înlocuire.

In mod natural, numărul total al paginilor programelor active poate deveni mai mare decât numărul paginilor fizice din memoria operativă. Din această cauză, uneori apare situația că un program cere încărcarea unei pagini virtuale, dar nu există o pagină fizică disponibilă pentru a o găzdui. Intrebarea este *care* dintre paginile fizice va fi evacuată pentru a crea spațiul necesar?

Răspunsul optim este simplu, dar imposibil de realizat: *se evacuează acea pagină care va fi solicitată în viitor cel mai târziu !?!* Desigur, acest lucru nu poate fi prevăzut în prealabil, dat fiind faptul că evoluția unui program la un moment dat este dependentă de datele concrete asupra cărora operează. Totuși, Belady [4] a descris un model de evidență statistică prin care se poate prevedea cu o oarecare probabilitate care este pagina care va fi solicitată cel mai târziu.

Dintre metodele mai "ortodoxe", de înlocuire, descrise printre altele în [22], [10], noi vom detalia trei:

- înlocuirea unei pagini care nu a fost recent utilizată (NRU - Not Recently Used);
- înlocuirea în ordinea încărcării paginilor (FIFO - First In First Out);
- înlocuirea paginii nesolicitata cel mai mult timp (LRU - Least Recently Used).

Evident, metodele de înlocuire prezentate mai sus sunt foarte simplu de descris. Pentru implementare trebuie avut în vedere faptul că întreținerea unei structuri de date care să permită decizia trebuie făcută *la fiecare acces la memorie*. În această situație, nu este permisă nici măcar întreținerea unei liste simplu înlănțuite! De cele mai multe ori aceste metode se implementează prin hard, făcându-se uneori compromisuri.

10.4.4.1 Metoda NRU.

Fiecare pagină fizică are asociați doi biți, prin intermediul cărora se va decide pagina de evacuat. Bitul **R**, numit bit de *referire*, primește valoarea 0 la încărcarea paginii. La fiecare referire a paginii, acest bit este pus pe 1. Periodic (de obicei la 20 milisecunde), bitul este pus iarăși pe 0. Bitul **M**, numit bit de *modificare*, primește valoarea 0 la încărcarea paginii. El este modificat numai la scrierea în pagină, când i se dă valoarea 1. Acești doi biți împart în fiecare moment paginile fizice în patru clase:

0. clasa 0: pagini nereferite și nemodificate;
1. clasa 1: pagini nereferite (în intervalul fixat), dar modificate de la încărcarea lor;
2. clasa 2: pagini referite dar nemodificate;
3. clasa 3: pagini referite și modificate.

Atunci când o pagină trebuie înlocuită, pagina "victimă" se caută mai întâi în clasa 0, apoi în clasa 1, apoi în clasa 2 și în sfărșit în clasa 3. Dacă pagina de înlocuit este în clasa 1 sau clasa 3, conținutul ei va fi salvat pe disc înaintea înlocuirii. Acest algoritm simplu, deși nu este optimal, s-a dovedit în practică a fi foarte eficient.

10.4.4.2 Metoda FIFO.

Implementarea acestei metode este foarte simplă. Se crează și se întreține o listă a paginilor în ordinea încărcării lor. Această listă se actualizează *la fiecare nouă încărcare de pagină* (nu la fiecare acces la memorie!). Atunci când se cere înlocuirea este substituția prima (cea mai veche) pagină din listă. Bitul **M** de modificare indică dacă pagina trebuie sau nu salvată înaintea înlocuirii.

O primă îmbunătățire ar fi combinarea algoritmilor NRU și FIFO, în sensul că se aplică întâi NRU, iar în cadrul aceleiași clase se aplică FIFO.

O altă îmbunătățire este cunoscută sub numele de *metoda celei de-a doua șanse*. La această metodă, se testează bitul **R** de referință la pagina cea mai veche. Dacă acesta este 0, atunci pagina este înlocuită imediat. Dacă este 1, atunci este pus pe 0 și pagina este pusă ultima în listă, ca și cum ar fi intrat recent în memorie. Apoi căutarea se reia cu o nouă listă. Evident, șansa de scăpare a unei pagini victimă este să existe o pagină mai "tânără" ca ea și care să nu fi fost referită.

Si acum o curiozitate! Bunul simț ne spune că șansa ca o pagină să fie înlocuită scade pe măsură ce numărul de pagini fizice crește. Si totuși nu este așa! În [4] [22] este dat un contraexemplu, cunoscut sub numele de *anomalia lui Belady*, pe care îl ilustrăm în fig. 10.21a și 10.21b. Este vorba de un program care are 5 pagini virtuale, pe care le solicită în ordinea:

0 1 2 3 0 1 4 0 1 2 3 4

In fig. 10.21a este ilustrată evoluția când există 3 pagini fizice, iar în fig. 10.21b când există 4 pagini fizice.

a) Trei pagini fizice:

Solicitare pagina: 0 1 2 3 0 1 4 0 1 2 3 4

Pagina recenta : 0 1 2 3 0 1 4 4 4 2 3 3
 . 0 1 2 3 0 1 1 1 4 2 2

Pagina mai veche : . . 0 1 2 3 0 0 0 1 4 4

Inlocuire pagina : I I I I I I I I I = 9

b) Patru pagini fizice:

Solicitare pagina: 0 1 2 3 0 1 4 0 1 2 3 4

Pagina recenta : 0 1 2 3 3 3 4 0 1 2 3 4
 . 0 1 2 2 2 3 4 0 1 2 3

Pagina mai veche : . . . 0 0 0 1 2 3 4 0 1

Inlocuire pagina : I I I I I I I I I I = 10

Figura 10.21 Anomalia lui Belady

10.4.4.3 Metoda LRU.

LRU (pagina mai puțin folosită în ultimul timp) este un algoritm bun de înlocuire. El are la bază următoarea observație, reieșită (tot) din principiul vecinătății. O pagină care a fost solicitată mult de către ultimele instrucțiuni, va fi probabil solicitată mult și în continuare. Invers, o pagină solicitată puțin (sau deloc), va rămâne probabil tot așa pentru câteva instrucțiuni.

Problema este cum să se țină evidența utilizărilor? Se exclude din start întreținerea unei liste înlănțuite care să fie modificată la fiecare acces la memorie. Prețul plătit este mult prea mare. Iată două posibile rezolvări.

Numărătorul de accese se implementează hard. Există un registru numit *contor* reprezentat (de regulă) pe 64 de biți. La fiecare acces, valoarea lui este mărită cu o unitate. În tabela de pagini, există câte un spațiu rezervat pentru a memora valoarea contorului. În momentul accesului la o pagină, valoarea contorului este memorată în acest spațiu rezervat din tabela de pagini. Atunci când se impune o înlocuire, este înlocuită pagina care a reținut cea mai mică valoare a contorului.

Matricea de referințe. Pentru un sistem de calcul care are n pagini fizice, se utilizează o matrice binară de $n \times n$. La pornire, toate elementele au valoarea 0. Atunci când se face referire la o pagină k , linia k a matricei este înlocuită peste tot cu 1, după care coloana k este înlocuită peste tot cu 0. În fiecare moment, numărul de cifre 1 de pe o linie oarecare l arată de câte ori a fost referită pagina l după încărcare.

Iată, spre exemplu, în fig. 10.22, cum arată evoluția matricei de referințe într-un sistem de calcul care are patru pagini fizice, solicitate în ordinea:

0 1 2 3 2 1 0 3 2 3:

0	1	2	3	2	1	0	3	2	3
0111	0011	0001	0000	0000	0111	0110	0100	0100	
0000	1011	1001	1000	1000	1011	0011	0010	0000	0000
0000	0000	1101	1100	1101	1001	0001	0000	1101	1100
0000	0000	0000	1110	1100	1000	0000	1110	1100	1110

Figura 10.22 Evoluția unei matrice de referințe

Implementarea mecanismului matricei de referințe se face destul de ușor prin hard, în orice caz mult mai ușor decât implementarea mecanismului cu numărător de referințe.

Desigur, politicile de înlocuire descrise mai sus pot fi simulate foarte bine prin soft. Din păcate, eficiența mecanismelor scade drastic. Acesta este motivul pentru care unele implementări ale memoriei virtuale (nu dăm aici nume) au eşuat lamentabil. Nu-i nimic, s-a câştigat experiență și asta nu este lucru puțin.

10.4.5 Cum funcționează o memorie cache?

Apariția memoriei *cache* a fost dictată de necesitatea creșterii performanțelor sistemului de calcul. Memoria cache conține copii ale unor blocuri din memoria operativă. Când procesorul încearcă citirea unui cuvânt de memorie, se verifică dacă acesta există în memoria cache. Dacă există, atunci el este livrat procesorului. Dacă nu, atunci el este căutat în memoria operativă, este adus în memoria cache împreună cu blocul din care face parte, după care este livrat procesorului. Datorită vitezei mult mai mari de acces la memoria cache, randamentul general al sistemului crește.

Aici apar o serie de probleme: cât de mare este o memorie cache? care este dimensiunea optimă a blocului de memorie destinat schimbului cu această memorie? În lucrarea [42] se dau răspunsuri la aceste întrebări. Tot acolo se fac cunoștințele evaluări ale raportului cost/performanță pentru dimensionarea unei memorii cache, precum și care este probabilitatea (hit ratio) ca o adresă cerută de procesor să fie găsită în memoria cache.

Noi ne rezumăm la ilustrarea modului în care se face corespondența dintre memoria operativă și memoria cache, precum și politicile de schimb dintre cele două tipuri de memorie.

Memoria cache este împărțită în mai multe părți egale, numite *sloturi* (poziții). Un slot are dimensiunea unui bloc de memorie, care este în mod obligatoriu o putere a lui 2. Fiecare slot conține în plus câțiva biți (numiți generic *tag* = etichetă) care indică blocul de memorie operativă depus în slotul respectiv. Dimensiunea unui tag depinde de mecanismul de schimb dintre cele două memorii, dar noi nu ne vom ocupa aici de el. De exemplu, în fig. 10.23 și 10.24 am ales o memorie operativă de 64 Ko și o memorie cache de 1 Ko. Dimensiunea unui slot, identică cu dimensiunea unui bloc de memorie, am ales-o de 8 octeți (64 de biți).

Se cunosc [42] mai multe metode de proiectare a spațiului memoriei operative pe memoria cache. Cea mai simplă metodă este *proiectarea directă*. Dacă *C* indică numărul total de sloturi din memoria cache, *A* este o adresă oarecare din memoria operativă, atunci numărul *S* al slotului în care se proiectează adresa *A* este:

$$S = A \bmod C$$

In fig. 10.23 este ilustrată această corespondență.

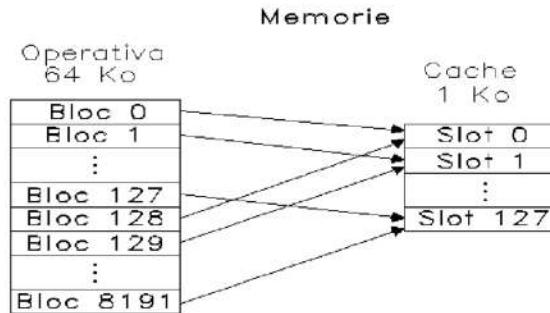


Figura 10.23 Proiectare directă pe memoria cache

Principalul ei avantaj este simplitatea. În schimb, există un mare dezavantaj, generat de faptul că fiecare bloc are o poziție fixă în memoria cache. Dacă, spre exemplu, se cer accese alternative la două blocuri care ocupă același slot, atunci are loc un trafic intens între cele două memorii, fapt care face să scadă mult randamentul general al sistemului.

O a doua metodă poartă numele de *proiectare asociativă*, apărută pentru eliminarea dezavantajului de mai sus. Această metodă plasează un bloc de memorie într-un slot oarecare liber.

Gestiunea sloturilor memoriei cache în acest caz ridică aceleasi probleme ca și alocarea paginată a memoriei. Principala problemă care se ridică aici este cea legată de *politica de înlocuire*. În paragraful 2.4.3 am tratat această problemă, descriind algoritmii NRU, FIFO și LRU. Tot ceea ce s-a spus acolo este valabil, fără modificări și la înlocuirea unui slot atunci când un bloc solicitat din memoria operativă nu mai are loc în memoria cache.

O a treia metodă poartă numele de *proiectarea set-asociativă*, și ea este o combinație a precedentelor două metode. Ideea ei este următoarea. Memoria cache este împărțită în *I seturi*, un set fiind compus din *J* sloturi. Avem deci relația $C = I \times J$. La cererea de memorie de la adresa A , se calculează numărul K al setului în care va intra blocul, astfel:

$$K = A \bmod I$$

Având fixat numărul setului, blocul va ocupa unul dintre sloturile acestui set. Alegerea slotului este de această dată o problemă de planificare. În fig. 10.24 este prezentat cazul $I=64$ și $J=2$.

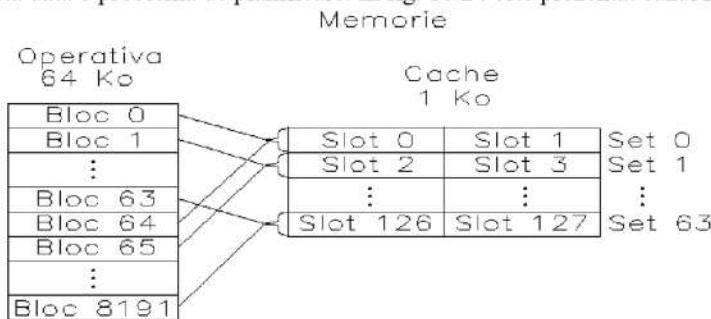


Figura 10.24 Proiectarea set - asociativă

Metoda set-asociativă este mult folosită în practică. Compromisul proiectării directe și a celei asociative face să fie aplicată o politică de înlocuire numai atunci când într-un slot există deja **J** blocuri, ceea ce se întâmplă foarte rar. În rest se aplică proiectarea directă, care este mult mai simplă.

4 Sistemul de fișiere Unix

4.1 Structura arborescentă și legături suplimentare

4.1.1 Tipuri de fișiere și sisteme de fișiere

In cadrul unui sistem de fișiere, apelurile sistem Unix gestionează opt tipuri de fișiere și anume:

1. Normale (obișnuite)
2. Directori
3. Legături hard (hard links)
4. Legături simbolice (symbolic links)
5. Socketuri (sockets)
6. FIFO - pipe cu nume (named pipes)
7. Periferice caracter
8. Periferice bloc

Pe lângă aceste opt tipuri, mai există încă patru entități, pe care apelurile sistem le văd, din punct de vedere sintactic, tot ca și fișiere. Aceste entități sunt gestionate de nucleul Unix, au suportul fizic tot în nucleu și folosite la comunicări între procese. Aceste entități sunt:

9. Pipe (anonymous pipes)
10. Segmente de memorie partajată
11. Cozi de mesaje
12. Semafoare

In acest capitol, sau în cele care urmează, vom trata aceste entități, exceptând 5, 10, 11, și 12 care vor face obiectul unei lucrări viitoare, destinate special sistemelor de operare distribuite.

Fișierele obișnuite sunt privite ca siruri de octeți, accesul la un octet putându-se face fie secvențial, fie direct prin numărul de ordine al octetului.

Fișierele directori. Un fișier director se deosebește de un fișier obișnuit numai prin informația conținută în el. Un director conține lista de nume și adrese pentru fișierele subordonate lui. Uzual, fiecare utilizator are un director propriu care punctează la fișierele lui obișnuite, sau la alți subdirectorii definiți de el.

Fișierele speciale. In această categorie putem include, pentru moment, ultimele 6 tipuri de fișiere. In particular, Unix privește fiecare dispozitiv de I/O ca și un fișier de tip special. Din punct de vedere al utilizatorului, nu există nici o deosebire între lucrul cu un fișier disc normal și lucrul cu un fișier special.

Fiecare director are două intrări cu nume speciale și anume:

- " . " (punct) denumește generic (punctează spre) însuși directorul respectiv;
- " .. " (două puncte succesive), denumește generic (punctează spre) directorul părinte.

Fiecare sistem de fișiere conține un director principal numit **root** sau **/**.

In mod obișnuit, fiecare utilizator folosește un *director curent*, atașat utilizatorului la intrarea în sistem. Utilizatorul poate să-și schimbe acest director (`cd`), poate crea un nou director subordonat celui curent, (`mkdir`), să șteargă un director (`rmdir`), să afișeze *calea de acces* de la `root` la un director sau fișier (`pwd`) etc.

Apariția unui mare număr de distribuitori de Unix a condus, inevitabil, la proliferarea unui număr oarecare de "sisteme de fișiere extinse" proprii acestor distribuitori. De exemplu:

- Solaris utilizează sistemul de fișiere `ufs`;
- Linux utilizează cu precădere sistemul de fișiere `ext2` și mai nou, `ext3`;
- IRIX utilizează `xfs`
- etc.

Actualele distribuții de Unix permit utilizarea unor sisteme de fișiere proprii altor sisteme de operare. Printre cele mai importante amintim:

- Sistemele FAT și FAT32 de sub MS-DOS și Windows 9x;
- Sistemul NTFS propriu Windows NT și 2000.

Din fericire, aceste extinderi sunt transparente pentru utilizatorii obișnuiți. Totuși, se recomandă prudență atunci când se efectuează altfel de operații decât citirea din fișierele create sub alte sisteme de operare decât sistemul curent. De exemplu, modificarea sub Unix a unui octet într-un fișier de tip `doc` creat cu Word sub Windows poate ușor să compromită fișierul așa încât el să nu mai poată fi exploatat sub Windows!

Administratorii sistemelor Unix trebuie să țină cont de sistemele de fișiere pe care le instalează și de drepturile pe care le conferă acestora vis-a-vis de userii obișnuiți.

Principiul structurii arborescente de fișiere este acela că orice fișier sau director are un singur părinte. Automat, pentru fiecare director sau fișier există o singură cale (path) de la rădăcină la directorul curent. Legătura între un director sau fișier și părinte o vom numi *legătură naturală*. Evident ea se creează odată cu crearea directorului sau fișierului respectiv.

legături suplimentare

4.1.2 Legături hard și legături simbolice

In anumite situații este utilă partajarea unei porțiuni a structurii de fișiere între mai mulți utilizatori. De exemplu, o bază de date dintr-o parte a structurii de fișiere trebuie să fie accesibilă mai multor utilizatori. Unix permite o astfel de partajare prin intermediul *legăturilor suplimentare*. O legătură suplimentară permite referirea la un fișier pe alte căi decât pe cea naturală. Legăturile suplimentare sunt de două feluri: *legături hard* și *legături simbolice (soft)*.

Legăturile hard sunt identice cu legăturile naturale și ele pot fi create numai de către administratorul sistemului. O astfel de legătură este o intrare într-un director care punctează spre o substructură din sistemul de fișiere spre care punctează deja legătura lui naturală. Prin aceasta, substructura este văzută ca fiind descendenta din două directoare diferite! Deci, printr-o astfel de legătură un fișier primește efectiv două nume. Din această cauză, la parcurgerea unei structuri arborescente, fișierele punctate prin legături hard apar duplicate. Fiecare duplicat apare cu numărul de legături către el.

De exemplu, dacă există un fișier cu numele `vechi`, iar administratorul dă comanda:

```
#ln vechi linknou
```

atunci în sistemul de fișiere se vor vedea două fișiere identice: vechi și linknou, fiecare dintre ele având marcat faptul că sunt două legături spre el.

Legăturile hard pot fi făcute numai în interiorul aceluiași sistem de fișiere (detalii puțin mai târziu).

Legăturile simbolice sunt intrări speciale într-un director, care punctează (referă) un fișier (sau director) oarecare în structura de directori. Această intrare se comportă ca și un subdirector al directorului în care s-a creat intrarea.

In forma cea mai simplă, o legătură simbolică se creează prin comanda:

```
ln -s caleInStructuraDeDirectori numeSimbolic
```

După această comandă, caleInStructuraDeDirectori va avea marcată o legătură în plus, iar numeSimbolic va indica (numai) către această cale. Legăturile simbolice pot fi utilizate și de către userii obișnuiți. De asemenea, ele pot puncta și în afara sistemului de fișiere (detalii puțin mai târziu).

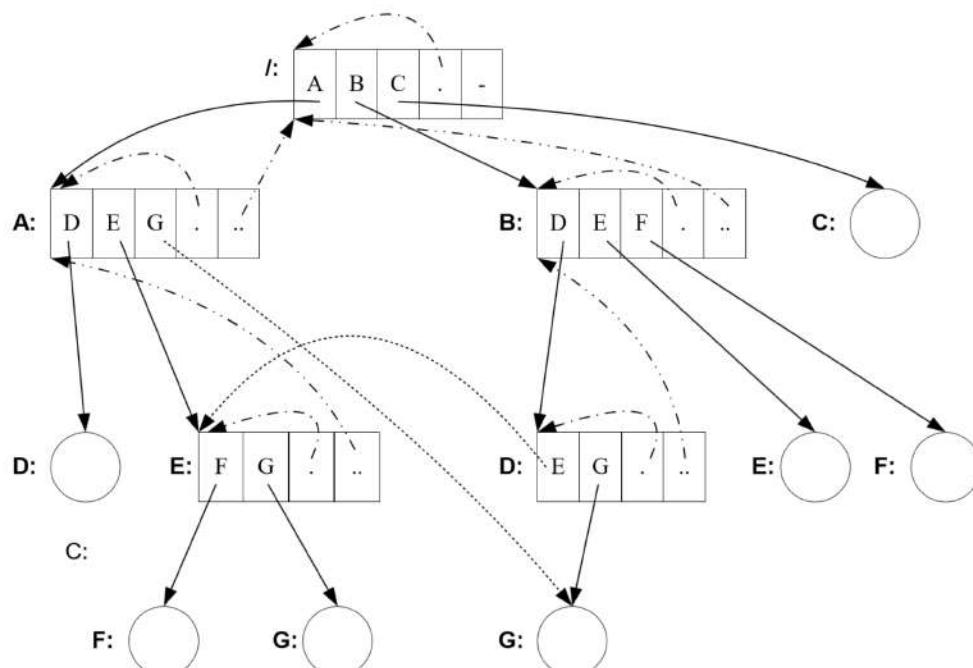


Figura 4.1 O structură arborescentă cu legături

Structura arborescentă împreună cu legăturile simbolice sau hard conferă sistemului de fișiere Unix o structură de graf aciclic. In fig. 4.1 este prezentat un exemplu simplu de structură de fișiere. Prin literele mari A, B, C, D, E, F, G am indicat nume de fișiere obișnuite, nume de directori și nume de legături. Este evident posibil ca același nume să apară de mai multe ori în structura de directori, grație structurii de directori care elimină ambiguitățile. Fișierele obișnuite sunt reprezentate prin cercuri, iar fișierele directori prin dreptunghiuri.

Legăturile sunt reprezentate prin săgeți de trei tipuri:

- linie continuă – legăturile naturale;
- linie întreruptă – spre propriul director și spre părinte;
- linie punctată – legături simbolice sau hard.

In fig. 4.1 există 12 noduri - fișiere obișnuite sau directori. Privit ca un arbore, deci considerând numai legăturile naturale, el are 7 ramuri și 4 nivele.

Să presupunem că cele două legături (desenate cu linie punctată) sunt simbolice. Pentru comoditate, vom nota legătura simbolică cu ultima literă din specificarea căii. Crearea celor două legături se poate face, de exemplu, prin succesiunea de comenzi:

cd /A	
ln -s /A/B/D/G G	Prima legătură
cd /A/B/D	
ln -s /A/E E	A doua legătură

Să presupunem acum că directorul curent este B. Vom parcurge arborele în ordinea director urmat de subordonații lui de la stânga spre dreapta. Următoarele 12 linii indică toate cele 12 noduri din structură. Pe aceeași linie apar, atunci când este posibil, mai multe specificări ale aceluiași nod. Specificările care fac uz de legături simbolice sunt subliniate. Cele mai lungi 7 ramuri vor fi marcate cu un simbol # în partea dreaptă.

/	..					
/A	./A					
/A/D	./A/D					#
/A/E	./A/E	<u>D/E</u>	<u>./D/E</u>			
/A/E/F	./A/E/F	<u>D/E/F</u>	<u>./D/E/F</u>			#
/A/E/G	./A/E/G	<u>D/E/G</u>	<u>./D/E/G</u>			#
/B	.					
/B/D	D	./D				
/B/D/G	D/G	./D/G	<u>/A/G</u>	<u>./A/G</u>		#
/B/E	E	./E				#
/B/F	F	./F				#
/C	./C					#

Ce se întâmplă cu ștergerea în cazul legăturilor multiple? De exemplu, ce se întâmplă când se execută una dintre următoarele două comenzi?

rm D/G					
rm /A/G					

Este clar că fișierul trebuie să rămână activ dacă este șters numai de către una dintre specificări.

Pentru aceasta, în descriptorul fișierului respectiv există un câmp numit *contor de legare*. Acesta are valoarea 1 la crearea fișierului și crește cu 1 la fiecare nouă legătură. La ștergere, se radiază legătura din directorul părinte care a cerut ștergerea, iar contorul de legare scade cu 1. Abia dacă acest contor a ajuns la zero, fișierul va fi efectiv șters de pe disc și blocurile ocupate de el vor fi eliberate.

4.1.3 Conceptul de montare

Spre deosebire de alte sisteme de operare ca DOS, Windows, etc. în specificarea fișierelor Unix *nu apare zona de periferic*. Acest fapt nu este întâmplător, ci este cauzat de filozofia generală de acces la fișierele Unix. Conceptul esențial prin care se rezolvă această problemă este cunoscut în Unix prin termenii de *montare și demontare* a unui sistem de fișiere.

Operația de *montare* constă în conectarea unui sistem de fișiere, de pe un anumit disc, la un director existent pe sistemul de fișiere implicit. Administratorul lansează comanda de montare sub forma:

```
# mount [ opțiuni ] sistemDeFișiere directorDeMontare
```

Efectul este conectarea indicată prin *sistemDeFișiere* la *directorDeMontare* existent pe sistemul implicit de fișiere. Opțiunile pot să indice caracteristicile montării. De exemplu opțiunea *rw* permite atât citirea, cât și scrierea în subsistemul montat, în timp ce opțiunea *ro* permite numai citirea din subsistemul montat. Opțiunea *-t* indică tipul sistemului de fișiere care se montează, și în funcție de tip, argumentul *sistemDeFișiere* poate fi */dev/periferic* sau *dev/dsk/periferic* sau */root/periferic* și.a.m.d. Pentru detalii se va putea consulta manualul *mount* al sistemului de operare curent.

Operația de *demontare* are efectul invers și ea se face cu comanda:

```
#/etc/umount directorDeMontare
```

Să urmărim cele ilustrate în fig. 4.2. În fig. 4.2a este dată structura sistemului de fișiere activ. Directorul B este vid (nu are descendenți). În fig 4.2b este dată structura de fișiere de pe un disc aflat (să zicem) pe unitatea de disc nr. 3, cu care intenționăm să lucrăm. Pentru aceasta, se va da comanda *privilegiată* Unix:

```
#/etc/mount /dev/fd3 /B
```

Prin ea se indică legarea discului al cărui fișier special (driver) poartă numele */dev/fd3* (și care se referă la discul nr. 3), la directorul vid */B*. Urmare a legării, se obține structura de fișiere activă din fig. 4.2c.

Cu scuzele de rigoare față de cititorul pe care-l plătim, vom face câteva precizări. Deși ele sunt firești, practica arată că sunt de multe ori încălcate, ceea ce provoacă neplăceri (și nu numai atât).

- Nu se va cere accesul la un fișier de pe un disc decât dacă acesta este montat în structura de fișiere implicită.
- Nu se va cere demontarea unei substructuri decât dacă a fost montată în prealabil.
- *Scoaterea unui suport montat de pe o unitate și înlocuirea lui cu un alt suport poate provoca pagube mari.* Să presupunem, spre exemplu, că s-a montat o structură existentă pe o anumită dischetă. Dacă înaintea demontării se scoate discheta din unitate și se introduce alta în loc, atunci este posibil să se piardă informații de pe noua dischetă, și, în unele cazuri, este posibilă blocarea întregului sistem! De altfel, sistemele Unix, mai nou, nici nu permit scoaterea din unitate a unui suport până când nu se efectuează operația *umount*.

- Nu este posibilă efectuarea de legături simbolice decât dacă directorul de unde se leagă și fișierul / directorul care se leagă se află pe același suport fizic. (Deși pe unele sisteme s-ar putea ca o astfel de legare să fie posibilă, noi nu o recomandăm :(

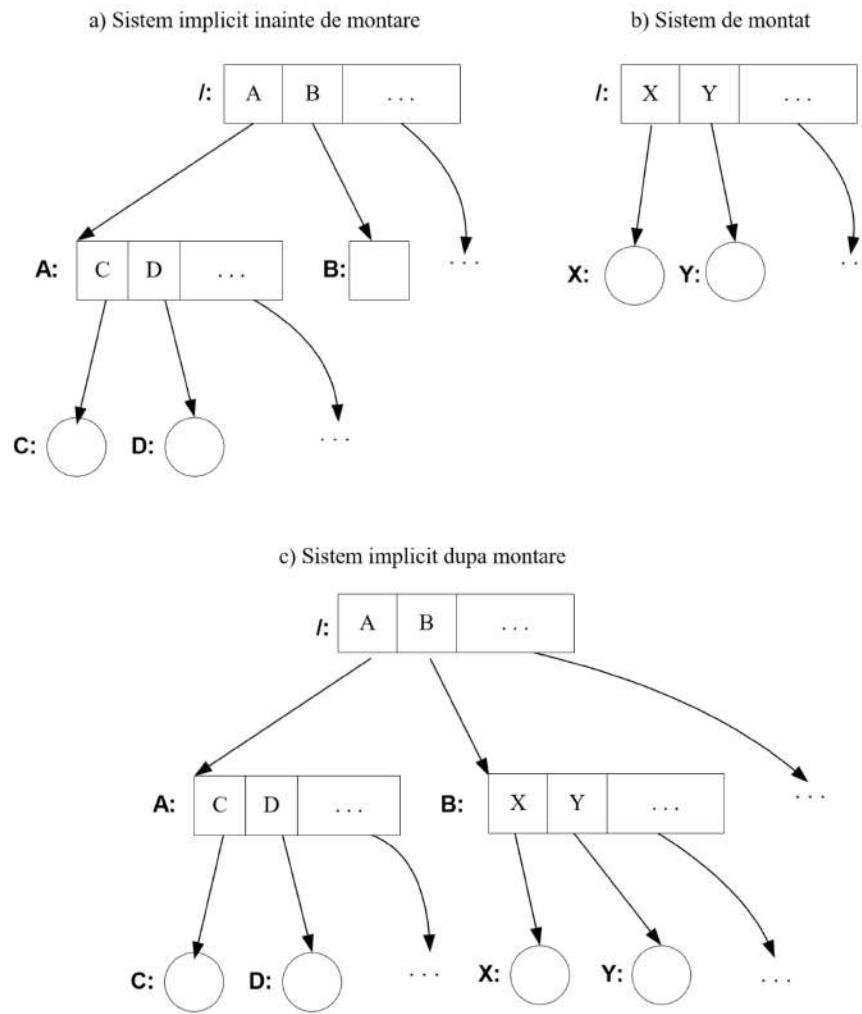


Figura 4.2 Operația de montare

In practica implementărilor Unix, la încărcarea **SO** se fac automat o serie de operații de montare, în conformitate cu configurația sistemului. Indicațiile de montare automată sunt trecute în fișierul */etc/fstab*. Acesta este un fișier text, având pe fiecare linie câte o montare, în care se specifică:

- partitia Unix (*periferic, sistemDeFisiere*) care se montează;
- directorDeMontare sub care este montată partitia;
- tipul sistemului de fișiere conținut;
- diverse opțiuni.

Iată, spre exemplu, o porțiune din acest fișier:

/dev/hda2	/	ext3	defaults	1 1
/dev/hda7	/home	ext3	defaults,nosuid,nodev,noexec	0 0
/dev/cdrom	/mnt/cdrom	iso9660	noauto,owner,ro	0 0
/dev/fd0	/mnt/floppy	auto	noauto,owner	0 0
/dev/hda3	/usr	ext3	defaults,nodev	1 2
/dev/hda5	/usr/local	ext3	defaults,nodev	1 2
/dev/hda6	/var	ext3	defaults,nosuid,nodev,noexec	0 0
/dev/hda1	swap	swap	defaults	0 0

De asemenea, operația de montare este practicată pentru *înglobarea SO Unix într-o rețea de calculatoare*. Cele mai cunoscute astfel de sisteme care dirijează montările de fișiere în rețele Unix (și nu numai) sunt *NFS* (*Network File System* al firmei Sun Microsystems), *RFS* (*Remote File System* al firmei AT&T), *SAMBA* (produs open source). Sistemul care montează o structură de directoare de pe o altă mașină poartă numele de *client* (NFS, RFS, SAMBA, etc.). Mașina care oferă structura spre montare (exportă) se numește *server* (NFS, RFS, SAMBA, etc.).

Montările pentru NFS sunt specificate tot în fișierul */etc/fstab*, prin linii de forma:

server1:/export/home	/home	nfs	rw,bg,intr	0 0
server1:/export/usr/local	/usr/local	nfs	rw,bg,intr	0 0
server2:/export/var/spool/mail	/var/spool/mail	nfs	rw,bg,intr	0 0

In acest context se poate da o nouă caracterizare a diferențelor dintre legăturile hard și cele simbolice: legăturile hard funcționează numai în interiorul același sistem de fișiere, în timp ce legăturile simbolice pot puncta și spre noduri ale altui sistem de fișiere montat împreună cu sistemul de fișiere ce conține legătura limbolică.

4.1.4 Protecția fișierelor Unix

4.1.4.1 Drepturi de acces

Reamintim iarăși principalele entități participante la Unix: useri, fișiere, procese și mai ales reamintim faptul că între ele există multe interdependențe. În această secțiune vom trata una dintre aceste interdependențe.

Vis-a-vis de un fișier sau de un director, utilizatorii se împart în trei categorii:

- proprietarul fișierului (*u - user*).
- grupul de utilizatori (*g - group*), de exemplu o grupă de studenți participă la un același proiect, motiv pentru care administratorul poate constitui un astfel de grup, cu drepturi specifice – de regulă mai slabe decât ale proprietarului, dar mai puternice decât ale restului utilizatorilor.
- restul utilizatorilor (*o - others*) cei care nu sunt în primele două categorii.

Nucleul SO Unix identifică utilizatorii prin numere naturale asociate unic, numite **UID-uri** (User IDentifications). De asemenea, identifică grupurile de utilizatori prin numere numite **GID-uri** (Group IDentifications).

Un utilizator aparte, cu drepturi depline asupra tuturor fișierelor este *root* sau superuserul.

Pentru fiecare categorie de utilizatori, fișierul permite maximum trei drepturi:

- dreptul de citire (*r* – *read*)
- dreptul de scriere (*w* – *write*) care include crearea de subdirectorii, stergerea de subdirectorii, adăugarea sau ștergerea de intrări în director, modificarea fișierului, etc.
- dreptul de execuție (*x* – *execution*) care permite lansarea în execuție a unui fișier. Acest drept, conferit unui director, permite accesul în directorul respectiv (*cd*).

In consecință, pentru specificarea drepturilor de mai sus asupra unui fișier sau director sunt necesari 9 (nouă) biți. Reprezentarea externă a acestei configurații se face printr-un grup de 9 (nouă) caractere: `rwxrwxrwx` în care absența uneia dintre drepturi la o categorie de useri este indicată prin – (minus).

Modul de atribuire a acestor drepturi se poate face cu ajutorul comenzi Shell `chmod`. Cea mai simplă formă a ei este:

```
chmod 010203 fisier . . .
```

unde `010203` sunt trei cifre octale. Biții 1 ai primeia indică drepturile userului, biții 1 ai celei de-a doua indică drepturile grupului, iar biții 1 ai celei de-a treia cifră indică drepturile restului userilor. De exemplu, comanda:

```
chmod 754 A B
```

fixează la fișierele `A` și `B` toate drepturile pentru user, drepturile de citire și de execuție pentru grup și doar dreptul de citire pentru ceilalți useri. În urma acestei comenzi, drepturile fișierelor `A` și `B` vor deveni: `rwxr-xr--`

Invităm cititorul să consulte manualul comenzi `chmod` pentru prezentarea completă a acestei comenzi.

4.1.4.2 Drepturi implicite: umask

În momentul intrării unui user în sistem, acestuia îi se vor acorda niște drepturi implicite pentru fișiere nou create. Toate fișierele și directoarele create de user pe durata sesiunii de lucru îl vor avea ca proprietar, iar drepturile vor fi cele primite implicit. Fixarea drepturilor implicite, sau aflarea valorii acestora se poate face folosind comanda `umask`. Drepturile implicite sunt stabilite scăzând (octal) masca definită prin `umask` din 777.

Pentru a se afla valoarea măștii se lansează:

```
umask
```

Rezultatul este afișarea măștii, de regulă 022. Deci drepturile implicite vor fi: $777-022=755$, adică userul are toate drepturile, iar grupul și restul vor avea doar drepturi de citire și de execuție.

Dacă se dorește schimbarea acestei măști pentru a da userului toate drepturile, grupului de citire și execuție iar restului nici un drept, adică drepturile implicate 750, fiindcă $777 - 027 = 750$. Deci se va da comanda:

```
umask 027
```

Efectul ei va rămâne valabil până la un nou umask sau până la încheierea sesiunii.

4.1.4.3 Drepturi de lansare, drepturi program executabil, biți setuid și setgid

In această secțiune vom analiza, din punct de vedere al drepturilor de acces, relația dintre un utilizator, programul executabil pe care îl lansează și fișierele asupra cărora acționează programul în cursul execuției lui.

Pentru fixarea ideilor, să considerăm un exemplu. Presupunem că:

1. un utilizator U, care face parte dintr-un grup G, lansează în execuție un program P. Pe durata execuției, programul P acționează asupra unui fișier F.
2. programul P are ca proprietar utilizatorul UP care face parte din grupul GP, iar drepturile fișierului executabil P sunt `rwxr-xr-x`.
3. fișierul F asupra căruia se acționează are proprietarul UF care face parte din grupul GF, iar drepturile fișierului F sunt `rwxr-xr--`.

In aceste ipoteze, userul U poate să lanseze în execuție programul P, deoarece acesta conferă drepturi de execuție tuturor utilizatorilor. (Dacă drepturile lui P ar fi fost `rwxr--xr--`, atunci lansarea în execuție ar fi fost posibilă numai dacă $U = UP$. Dacă drepturile lui P ar fi fost `rwxr-xr--`, atunci lansarea ar fi fost posibilă numai dacă $U = UP$ sau $G = GP$.)

După lansarea în execuție de către U a lui P, în mod implicit acțiunile pe care programul P le poate efectua asupra fișierului F sunt cele permise de drepturile pe care U le are asupra lui F. In exemplul nostru, dacă $U = UF$ atunci P poate citi, scrie sau executa F. Dacă $G = GF$ atunci P poate citi sau executa F, iar dacă $G \neq GF$ sunt diferite atunci P poate numai să citească din F. (Intr-o secțiune următoare vom arăta cum un program poate să lanseze în execuție un alt program).

Această regulă, prin care drepturile lansatorului de program permit sau nu unele operații pe care programul lansat le aplică unui fișier, este regula evasigenerală de acțiune asupra fișierelor.

Există însă situații, nu foarte frecvente, în care această regulă se poate schimba. Schimbarea este cunoscută sub numele *setuid / setgid* și se referă, cu notațiile de mai sus, la faptul că: După lansarea în execuție de către U a lui P, în mod setuid / setgid acțiunile pe care programul P le poate efectua asupra fișierului F sunt cele permise de drepturile proprietarului programului P și ale grupului din care face parte acesta.

Este vorba de doi biți importanți relativ la drepturile de acces, așa numiții *bit setuid (set-user-id)* care pus pe 1 schimbă drepturile lui U cu drepturile lui UP și bitul *setgid*, care pus pe 1 schimbă drepturile lui G cu drepturile lui GP. (Aceste schimbări au loc numai la execuția programului P.)

Cu alte cuvinte, dacă pentru un fișier executabil bitul **setuid** este 1, atunci **un utilizator care lansează în execuție acest fișier** (evident, dacă are dreptul să-l lanzeze) **primește, pe timpul execuției, aceleași drepturi de acces la resurse** (fișiere, semafoare, zone de memorie etc.) ca și proprietarul fișierului executabil.

Să vedem o situație concretă în care este utilă folosirea bitului **setuid**. Un utilizator cu numele **profesor** întreține un fișier **note** al cărui proprietar este. Din rațiuni lesne de înțeles, drepturile fișierului **note** sunt fixate la **r-----**. Utilizatorul **profesor** dorește să permită utilizatorilor din grupul **studenti** să vadă unele informații din fișierul **note**.

Pentru aceasta, **profesor** creează un fișier executabil **examen** (proprietarul lui **examen** este **profesor**) care permite citirea (eventual selectivă) de informații din fișierul **note**. Proprietarul atribuie pentru **examen** drepturile **rwx---x--x** și pune bitul **setuid** al lui **examen** pe 1. Această atribuire se face cu comanda **chmod +s examen**. Noile drepturi afișate ale lui **examen** sunt: **rws---x--x**

Utilizatorii **studenti**, în momentul lansării programului **examen**, primesc aceleași drepturi de acces la fișiere ca și **profesor**. În particular programul **examen** poate accesa fișierul **note** (vezi drepturile acestui fișier) chiar dacă el nu a fost lansat în execuție de către **profesor**. În absența lui **setuid** pentru **examen**, acesta poate fi, totuși lansat în execuție, însă nu poate să acceseze fișierul **note**.

După cum spuneam mai sus, nucleul SO Unix identifică utilizatorii prin numere naturale asociate unic, numite **UID-uri** (User IDentifications). De asemenea, identifică grupurile de utilizatori prin numere numite **GID-uri** (Group IDentifications).

Pe parcursul execuției programului **examen** acestuia î se mai asociază în plus identificatorul **EUID** (effective **UID**), care coincide cu **UID-ul** lui **profesor**, prin care asigură accesul la resurse.

Mecanismul **setuid** permite o foarte elastică manevrare a fișierelor. În schimb, dacă superuserul gestionează prost acest mecanism, atunci potențialii infractori au un câmp larg de acțiune. Să considerăm, din rațiuni evidente, doar un scenariu simplu: Presupunem ca **root** este proprietar al unui fișier executabil cu bitul **setuid** setat și cu drept de scriere pentru alții. În această situație, un răuvoitor poate să modifice acest fișier executabil aşa încât să aibă o acțiune malefică ce presupune acces la resurse ale superuserului!. Acțiunea se va putea executa deoarece EUID-ul este **UID-ul** lui **root**!

Modificarea drepturilor de acces la fișiere se poate face numai de către proprietarul fișierului (sau de către superuser), folosind comanda **chmod**. Modificarea proprietarului sau a grupului se poate face, în aceleași condiții folosind comanda **chown**.

Un exemplu tipic în care se folosește **setuid** este comanda **/usr/bin/passwd**. Aceasta este lansată de către fiecare utilizator atunci când dorește să-și schimbe parola. Efectul ei se răspândește asupra fișierului **/etc/shadow**. În acest scop, se stabilesc drepturile:

```
-r-s---x--x 1 root root 22312 Sep 25 18:52 /usr/bin/passwd
-r----- 1 root root 10256 Mar 2 14:40 /etc/shadow
```

Deci programul `passwd` are `setuid`, ceea ce permite accesul la `/etc/shadow` numai prin programul `/usr/bin/passwd`.

4.1.5 Principalele directoare ale unui sistem de fișiere Unix

De-a lungul evoluției sistemelor din familia Unix, partea superioară a structurii sistemului de fișiere a avut mai mult sau mai puțin o formă standard. De fapt, fiecare versiune Unix și-a fixat o structură specifică a părții superioare din sistemul de fișiere. Diferențele între aceste structuri nu sunt prea mari. Mai mult, din rațiuni de compatibilitate, versiunile mai noi definesc legături suplimentare hard (nu legături simbolice), pentru a asigura compatibilitatea cu sistemele de fișiere mai vechi. Din această cauză este cel mai nimerit să se studieze o reuniune a celor mai răspândite structuri. O astfel de structură este prezentată de noi în fig. 4.3.

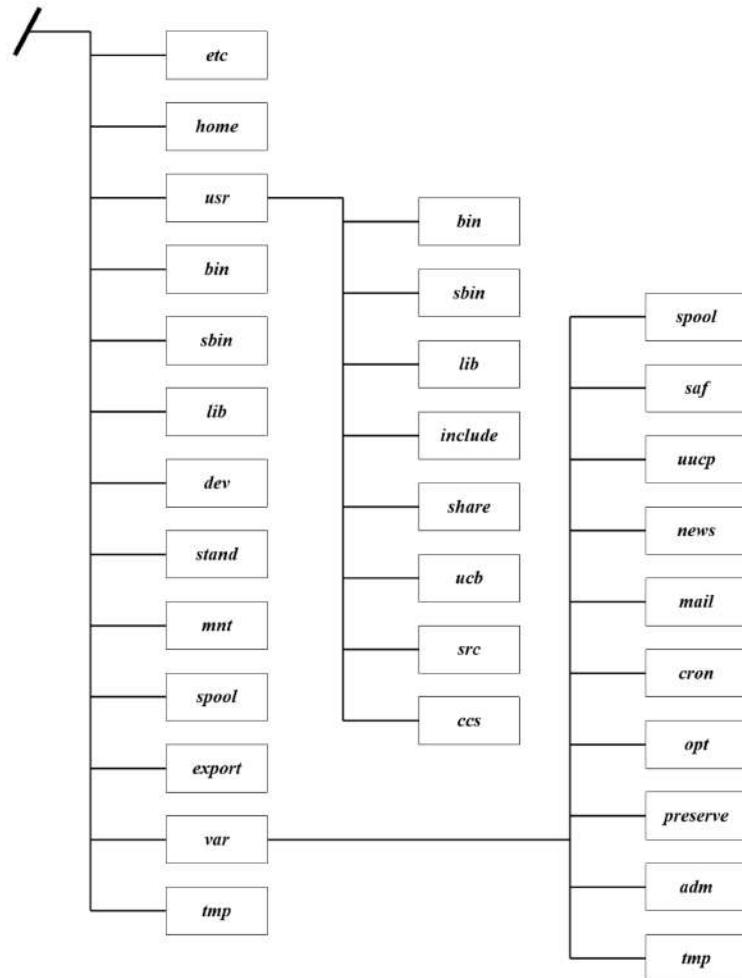


Figura 4.3 Structura superioară a unui sistem de fișiere Unix

Directorul `/etc` conține informații specifice mașinii necesare întreținerii sistemului. Acestea sunt de fapt datele restrictive și periculoase din sistem. Un exemplu de fișier ce conține informații specifice mașinii este, spre exemplu, `/etc/rc2.d` care este un director ce conține programe shell executate de procesul `init` la schimbarea stării. Tot aici sunt plasate fișierele `/etc/passwd`, `/etc/group`, `/etc/shadow` care sunt folosite pentru administrarea utilizatorilor.

Directorul `/home` este folosit pentru directorii gazdă ai utilizatorilor. În momentul intrării în sistem, fiecare utilizator indică directorul lui, care este fixat ca director curent (`home directory`) în `/etc/passwd`. Deși se poate trece ușor de la un director la altul, în mod normal fiecare utilizator rămâne în propriul lui director, unde își dezvoltă propria lui structură arborescentă de directori.

Directorul `/usr` este folosit în mod tradițional pentru a stoca fișiere ce pot fi modificate. La primele versiuni de Unix conținea și fișierele utilizatorilor. În prezent este punctul de montare pentru partitia ce conține usr. El conține programe executabile dependente și independente de sistemul de fișiere, precum și fișiere necesare acestora, dar care nu cresc dinamic. Asupra conținutului de subdirectoare ale lui `/usr` vom reveni puțin mai târziu.

Directorul `/bin` conține programele principalelor comenzi standard Unix: compilatoare, asamblare, editoare de texte, utilitare etc. Versiunile mai noi de Unix plasează acest director în `/usr/bin`.

Directorul `/sbin` (super-utilizator bin) conține comenzi critice pentru procedura de încărare a sistemului. Orice comenzi administrative care necesită lucru mono-utilizator sunt în acest director. Copii ale acestor comenzi se află în directoarele `/usr/bin` și `/usr/sbin`, astfel încât el (`/sbin`) poate fi refăcut dacă este necesar.

Directorul `/lib` conține diverse biblioteci și baze de date necesare apelurilor sistem. Doar versiunile mai vechi de Unix plasează acest director în `/`, cele actuale îl plasează în `/usr/lib`.

Directorul `/dev` este folosit pentru memorarea fișierelor speciale (fișierele devices). Practic, fiecare tip de terminal și fiecare tip de unitate de disc trebuie să aibă asociat un astfel de fișier special. Începând cu *SVR4* se permite ca în `dev` să existe și subdirectoare care să grupeze astfel de device-uri.

Directorul `/stand` conține informațiile necesare încărcării sistemului.

Directorul `/mnt` este folosit pentru a monta un sistem de fișiere temporar. De exemplu, un sistem de fișiere de pe un disc flexibil poate fi montat în `/mnt` pentru a verifica fișierele de pe această dischetă.

Directorul `/spool` este plasat în `/` doar în versiunile mai vechi de Unix. În el sunt memorate fișierele tampon temporare destinate prelucrărilor asincrone: listări asincrone de fișiere (`/spool/lpd`) și execuția la termen a unor comenzi (`/spool/at`).

Directorul `/export` este folosit ca punct implicit de montare pentru un arbore de sistem de fișiere exportat, pentru fișierele în rețea gestionate prin pachetul NFS (Network File System).

Directorul `/var` este folosit pentru memorarea fișierelor care cresc dinamic. În particular, multe dintre versiunile de Unix țin în acest director fișierele `INBOX` cu căsuțele poștale ale utilizatorilor. Structura de subdirectoare a lui `/var` o vom descrie ceva mai încolo.

Directorul `/tmp` este folosit pentru a memora fișiere temporare pentru aplicații. În mod normal, aceste fișiere nu sunt salvate și sunt șterse după o perioadă de timp.

În continuare vom descrie pe scurt conținutul directorului `/usr`. După cum se poate vedea, unele dintre subdirectoare sunt plasate (și au fost descrise) la nivel de rădăcină `/`. Versiunile mai noi de Unix, începând cu *SVR4*, le-au coborât din rădăcină ca subdirectoare ale lui `/usr`. Este cazul directoarelor `bin`, `sbin`, `lib`.

Directorul `/usr/include` conține fișierele header (`*.h`) standard ale limbajului C de sub Unix.

Directorul `/usr/share` conține o serie de directoare partajabile în rețea. În multe dintre sistemele noi, în el se află directoarele: `man` cu manualele Unix, `src` cu sursele C ale nucleului Unix și `lib` mai sus prezentat.

Directorul `/usr/ucb` conține programele executabile compatibile Unix BSD.

Directorul `/usr/src` conține textele sursă C ale nucleului Unix de pe mașina respectivă.

Directorul `/usr/ccs` conține instrumentele de dezvoltare a programelor C oferite de Unix: `cc`, `gcc`, `dbx`, `cb`, `indent`, etc.

În continuare vom descrie conținutul directorului `/var`. Ca și mai sus, unele subdirectoare de la nivelele superioare au fost mutate aici de către versiunile mai noi de Unix. Este vorba de directoarele `spool` și `tmp`.

Directorul `/var/saf` conține fișiere jurnal și de contabilizare a serviciilor oferite.

Directorul `/var/uucp` conține programele necesare efectuării de copieri de fișiere între sisteme Unix (Unix to Unix CoPy). Acest gen de servicii este primul pachet de comunicații instalat pe sisteme Unix, este operațional încă din 1978 și este utilizat și astăzi atunci când nu există alt mijloc mai modern de comunicații. Un astfel de sistem permite, de exemplu, apelul telefonic între două sisteme Unix, iar după luarea contactului cele două sisteme își schimbă între ele o serie de fișiere, ca de exemplu mesajele de poștă electronică ce le sunt destinate.

Directorul `/var/news` conține fișierele necesare serviciului de (știri) noutăți (news) care poate fi instalat pe mașinile Unix.

Directorul `/var/mail` conține căsuțele poștale implicate ale utilizatorilor (`INBOX`). Pe unele sisteme, ca de exemplu pe Linux, acestea se află în `/var/spool/mail`.

Directorul `/var/cron` conține fișierele jurnal necesare serviciilor executate la termen.

Directorul `/var/opt` constituie un punct de montare pentru diferite pachete de aplicații.

Directorul `/var/preserve` conține, la unele implementări Unix (SVR4) fișiere jurnal destinate refacerii stării editoarelor de texte “picate” ca urmare a unor incidente.

Directorul `/var/adm` conține fișiere jurnal (log-uri) de contabilizare și administrare a sistemului. La versiunile mai noi acestea sunt în `/var/log`.

După cum se poate vedea ușor, structura de directori Unix începând de la rădăcină este relativ dependentă de tipul și versiunea de Unix. De fapt, este vorba de “Unix vechi” și “Unix noi”. De asemenea, multe dintre directoare au fost înlocuite sau li s-a schimbat poziția în structura de directori. Tabelul de mai jos prezintă câteva corespondențe între vechile și noile plasări de fișiere.

Nume vechi	Nume nou
<code>/bin</code>	<code>/usr/bin</code>
<code>/lib</code>	<code>/usr/lib</code>
<code>/usr/adm</code>	<code>/var/adm</code>
<code>/usr/spool</code>	<code>/var/spool</code>
<code>/usr/tmp</code>	<code>/var/tmp</code>
<code>/etc/termcap</code>	<code>/usr/share/lib/termcap</code>
<code>/usr/lib/terminfo</code>	<code>/usr/share/lib/terminfo</code>
<code>/usr/lib/cron</code>	<code>/etc/cron.d</code>
<code>/usr/man</code>	<code>/usr/share/man</code>
<code>/etc/<programe></code>	<code>/usr/bin/<programe></code>
<code>/etc/<programe></code>	<code>/sbin/<programe></code>

4.2 Structura internă a discului Unix

4.2.1 Partiții și blocuri

Un sistem de fișiere Unix este găzduit fie pe un periferic oarecare (hard-disc, CD, dischetă etc.), fie pe o *partiție* a unui hard-disc. Partiționarea unui hard-disc este o operație (relativ) independentă de sistemul de operare ce va fi găzduit în partiția respectivă. De aceea, atât partiților, cât și suporturilor fizice reale le vom spune generic, *discuri Unix*.

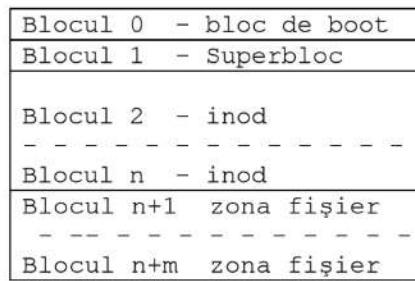


Figura 4.4 Structura unui disc Unix

Un fișier Unix este o succesiune de octeți, fiecare octet putând fi adresat în mod individual. Este permis atât accesul secvențial, cât și cel direct.

Unitatea de schimb dintre disc și memorie este *blocul*. La sistemele mai vechi acesta are 512 octeți, iar la cele mai noi poate avea și 1Ko sau 4Ko, pentru o mai eficientă gestiune a spațiului.

Un sistem de fișiere Unix este o structură de date rezidentă pe disc. Așa după cum se vede din fig. 4.4, un disc este compus din patru categorii de blocuri.

Blocul 0 conține programul de încărcare al **SO**. Acest program este dependent de mașina sub care se lucrează. Acțiunea lui se declanșează la pornirea sistemului - apăsarea butonului <**Reset**> sau <**Power**>. La acest moment, intră în lucru un mic program din memoria ROM, așa-numitul *cod startup din BIOS*. Acesta știe să citească primii 512 octeți de pe disc, să îi depună undeva în memoria RAM și să lanseze în execuție secvența de octeți citită. Evident, în primii 512 octeți de pe disc va fi un program, *bootprogramul* sau *programul de pornire a încărcării sistemului de operare*. Principala sarcină a *bootprogramului* este aceea de a încărca în RAM partea din *kernel*-ul Unix (sau a altui sistem de operare) special destinată încărcării complete a sistemului de operare.

Nu este obligatoriu ca întregul program de boot să se găsească în blocul 0, ci doar partea care odată executată să permită sistemului lucrul cu alte tipuri de memorie. Odată acest punct atins, execuția programului poate continua cu părți care se găsesc pe alte medii de stocare nevolatile, cum ar fi un harddisk, o dischetă, un CR-ROM sau chiar pe un server de boot în cazul stațiilor fără harddisk.

In majoritatea cazurilor în care nu este vorba de o primă instalare a sistemului, restul programului de boot se află stocat pe harddisk într-o zonă specială a acestuia numită *partiție* sau *segment de boot*. Această partiție are o structură de date extrem de simplă (mult mai simplă decât a sistemului de fișiere) și poate fi accesată foarte ușor.

Blocul 1 este numit și *superbloc*. În el sunt trecute o serie de informații prin care se definește sistemul de fișiere de pe disc. Printre aceste informații amintim:

- numărul *n* de *inoduri* (detaliem imediat);
- numărul de zone definite pe disc;
- pointeri spre harta de biți a alocării inodurilor;
- pointeri spre harta de biți a spațiului liber disc;
- dimensiunile zonelor disc, etc.

Blocurile 2 la n, unde *n* este o constantă a formatării discului, se constituie în *zona de inoduri*. Un *inod* (sau *i-nod*) este numele, în terminologia Unix, a *descriptorului* unui fișier. Inodurile sunt memorate pe disc sub forma unei liste (numită *i-listă*). Numărul de ordine al unui inod în cadrul i-listei se reprezintă pe doi octeți și se numește *i-număr*. Acest i-număr constituie legătura dintre fișier și programele utilizator.

Blocurile n+1 la n+m reprezintă *zona fișierelor*. Este partea cea mai mare din cadrul discului. Alocarea spațiului pentru fișiere se face printr-o variantă elegantă de indexare. Informațiile de plecare pentru alocare sunt fixate în inoduri. La discurile Unix actuale există, de regulă, mai multe zone de inoduri intercalate cu mai multe zone de fișiere.

4.2.2 Directori și inoduri

Structura unei intrări într-un fișier director este ilustrată în fig. 4.5.

Numele fișierului (practic oricât de lung)	inumăr
--	--------

Figura 4.5 Structura unei intrări în director

Deci, în director se află numele fișierului și referința spre inodul descriptor al fișierului.

Un *inod* are, de regulă, între 64 și 128 de octeți și el conține informațiile din tabelul următor:

mode	Drepturile de acces și tipul fișierului.
link count	Numărul de directoare care conțin referiri la acest inod, adică numărul de legături spre acest fișier.
user ID	Numărul (UID) de identificare a proprietarului.
group ID	Numărul (GID) de identificare a grupului.
Size	Numărul de octeți (lungimea) fișierului.
access time	Momentul ultimului acces la fișier.
mod time	Momentul ultimei modificări a fișierului.
inode time	Momentul ultimei modificări a structurii inodului.
block list	Lista adreselor disc pentru primele blocuri care aparțin fișierului.
indirect list	Referințe către celelalte blocuri care aparțin fișierului.

4.2.3 Schema de alocare a blocurilor disc pentru un fișier

Fiecare sistem de fișiere Unix are câteva constante proprii, printre care amintim:

- lungimea unui inod,
- lungimea unui bloc,
- lungimea unei adrese disc (implicit câte adrese disc încap într-un bloc),
- câte adrese de prime blocuri se înregistrează direct în inod,
- câte referințe se trec în lista de referințe indirecte.

Indiferent de valorile acestor constante, principiile de înregistrare / regăsire sunt aceleași și le vom prezenta în cele ce urmează. Pentru fixarea ideilor, vom alege aceste constante cu valorile întâlnite mai frecvent la sistemele de fișiere deja consacrate. Cu aceste constante, în fig. 4.6 este prezentată structura pointerilor spre blocurile atașate unui fișier Unix. Aceste constante sunt:

- un inod se reprezintă pe 64 octeți,
- un bloc are lungimea de 512 octeți,
- adresa disc se reprezintă pe 4 octeți, deci încap 128 adrese disc într-un bloc,
- în inod trec direct primele 10 adrese de blocuri,
- lista de adrese indirecte are 3 elemente.

În inodul fișierului se află o listă cu 13 intrări, care desemnează blocurile fizice aparținând fișierului.

- Primele 10 intrări conțin *adresele primelor 10 blocuri* de câte 512 octeți care aparțin fișierului.
- Intrarea nr. 11 conține adresa unui bloc, numit *bloc de indirectare simplă*. El conține adresele următoarelor 128 blocuri de câte 512 octeți, care aparțin fișierului.
- Intrarea nr. 12 conține adresa unui bloc, numit *bloc de indirectare dublă*. El conține adresele a 128 blocuri de indirectare simplă, care la rândul lor conțin, fiecare, adresele a câte 128 blocuri, de 512 octeți fiecare, cu informații aparținând fișierului.

- Intrarea nr. 13 conține adresa unui bloc, numit *bloc de indirectare triplă*. În acest bloc sunt conținute adresele a 128 blocuri de indirectare dublă, fiecare dintre acestea conținând adresele a câte 128 blocuri de indirectare simplă, iar fiecare dintre acestea conține adresele a câte 128 blocuri, de către 512 octeți, cu informații ale fișierului.

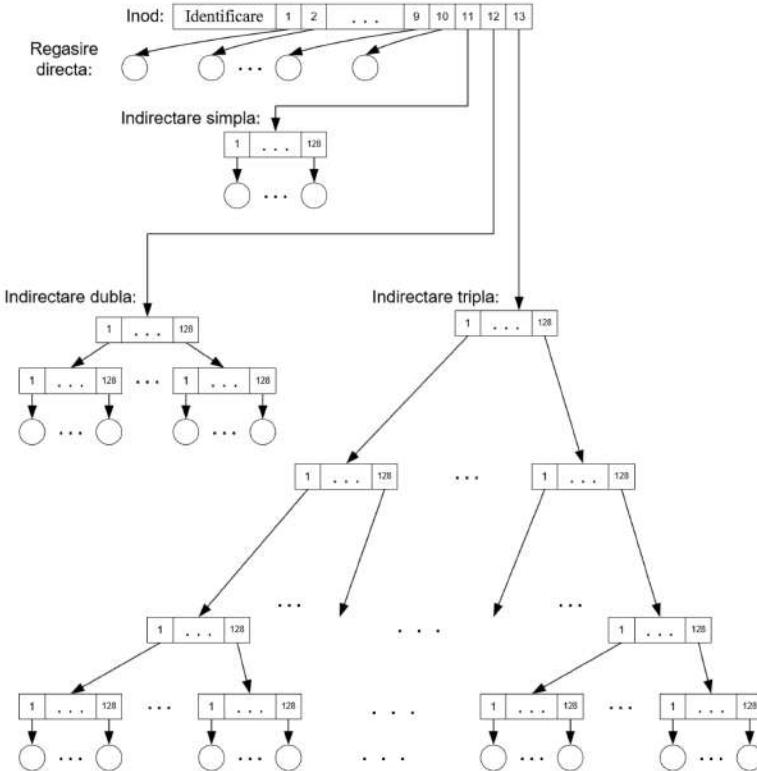


Figura 4.6 Structura unui inod și accesul la blocurile unui fișier

In fig. 4.6 am ilustrat prin cercuri blocurile de informație care aparțin fișierului, iar prin dreptunghiuri blocurile de referințe, în interiorul acestora marcând referințele.

Numărul de accese necesare pentru a obține direct un octet oarecare este cel mult 4. Pentru fișiere mici acest număr este și mai mic. Atât timp cât fișierul este deschis, inodul lui este prezent în memoria internă. Tabelul următor dă numărul maxim de accese la disc pentru a obține, în acces direct orice octet dintr-un fișier, în funcție de lungimea fișierului.

Lungime maximă (blocuri)	Lungime maximă (octeți)	Accese indirecte	Accese la informație	Total accese
10	5120	-	1	1
$10+128 = 138$	70656	1	1	2
$10+128+128^2 = 16522$	8459264	2	1	3
$10+128+128^2+128^3= 2113674$	1082201088	3	1	4

La sistemele Unix actuale lungimea unui bloc este de 4096 octeți care poate înregistra 1024 adrese, iar în inod se înregistrează direct adresele primelor 12 blocuri. În aceste condiții, tabelul de mai sus se transformă în:

Lungime maximă (blocuri)	Lungime maximă (octeți)	Accese indirecte	Accese la informație	Total accese
12	49152	–	1	1
$12+1024 = 1036$	4243456	1	1	2
$12+1024+1024^2 = 1049612$	4299210752	2	1	3
$12+1024+1024^2+1024^3 = 1073741824$	4398046511104 (peste 5000Go)	3	1	4

Practic, orice fișier actual, indiferent de mărimea lui, poate fi reprezentat printr-o astfel de schemă.

4.2.4 Accesul proceselor la fișiere

Unix privește conceptul de fișier într-un sens ceva mai larg decât o fac alte sisteme de operare. Așa cum am mai arătat mai sus, există opt tipuri de fișiere:

- normale,
- directori,
- legături hard,
- legături simbolice,
- FIFO, (pipe cu nume),
- socketuri,
- periferice caracter,
- periferice bloc.

Pe lângă acestea, nucleul mai gestionează, într-o sintaxă similară fișierelor, următoarele patru tipuri de comunicații între procese:

- pipe anonte (a se deosebi de FIFO - pipe cu nume);
- segmente de memorie partajată;
- cozi de mesaje;
- semafoare.

Suporturile fizice pentru aceste 12 tipuri de fișiere sunt, în ultimă instanță:

- *Zona fișierelor pe disc*, pentru fișierele normale, directori, legături hard și simbolice, FIFO și socket din familia Unix.
- *Perifericul respectiv* pentru perifericele caracter și bloc.
- *Zone rezervate de nucleu în memoria internă*, pentru pipe, memorie partajată, cozi de mesaje și semafoare.
- *Interfața de comunicație prin rețea*, pentru socket din familia Internet.

Pentru a putea asigura o tratare uniformă, traseul accesului unui proces la un fișier trece prin mai multe nivele: proces, sistem, inod, fișier, așa cum se vede în fig. 4.7. În fig. 4.7 prezentăm un exemplu în care există trei procese: **A**, **B**, **C** și patru fișiere **F1**, **F2**, **F3**, **F4**. Intrările

pentru legături la diversele nivele le-am notat cu litere mici **a – w**. In cele ce urmează descriem cele patru nivele de legătură.

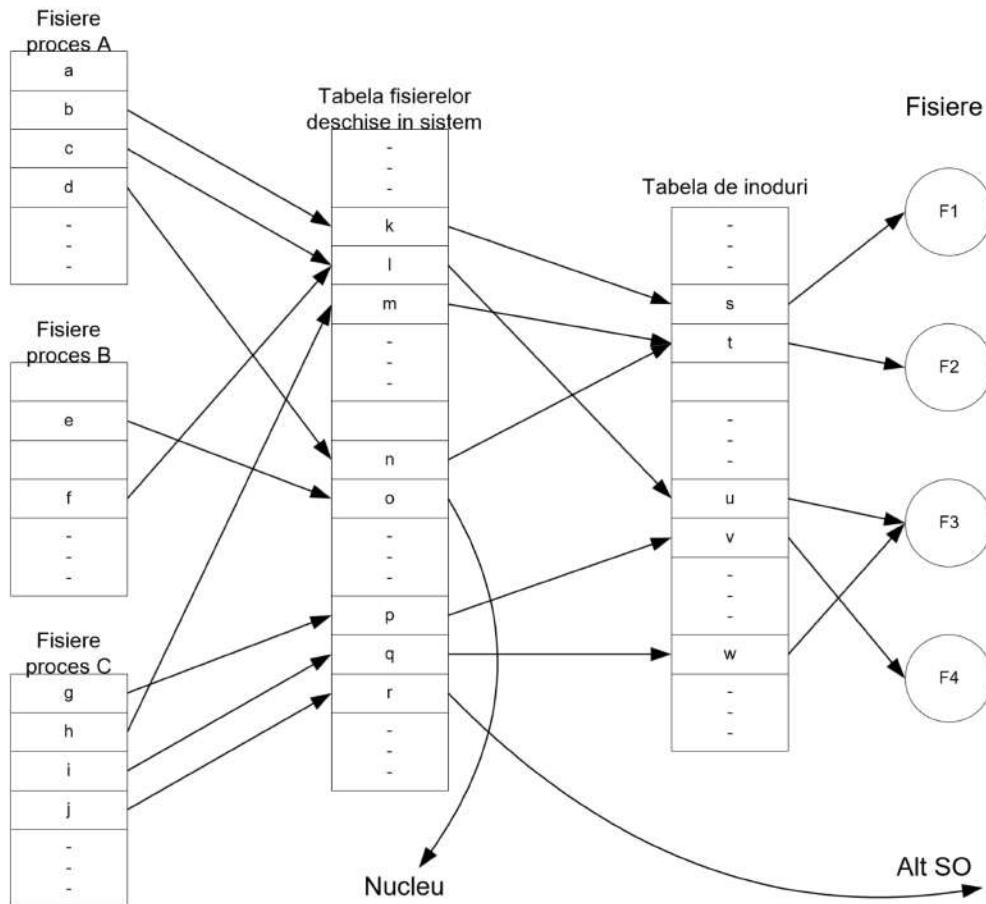


Figura 4.7 Corespondența Unix între procese și fișiere

Nivelul proces. Fiecare proces își întreține o tabelă proprie în care înregistrează toate fișierele lui deschise. In fig. 4.7 am notat cu **a-j** câteva intrări de pe acest nivel.

Nivelul sistem întreține o tabelă unică cu toate fișierele deschise de către toate procesele din sistem. In fig. 4.7 am notat prin **k-r** câteva intrări din această tabelă.

Nivelul inod este de fapt zona (zonele) de inoduri de pe disc. Pentru fișierele deschise, se păstrează în memoria internă copii ale inodurilor corespunzătoare. In fig. 4.7 am notat prin **s-w** câteva astfel de intrări.

Nivelul fișier este reprezentat de blocurile disc ce aparțin fișierului. In fig. 4.7 am notat **F1-F4** astfel de fișiere.

Tabela de fișiere la nivel proces are intrările numerotate începând de la 0. Primele trei intrări sunt rezervate astfel:

- intrarea 0 este rezervată intrării standard a procesului (vezi în fig. 4.7 intrările **a** din procesul **A** și **g** din procesul **C**);
- intrarea 1 este rezervată ieșirii standard (vezi intrările **b**, **e**, **h** din fig. 4.7);
- intrarea 2 este rezervată fișierului standard de eroare (unde sistemul afișează mesajele de eroare pentru proces - vezi intrările **c**, **i** din fig. 4.7).

După cum se va vedea în secțiunile următoare, toate apelurile sistem de lucru cu fișiere folosesc pentru identificarea fișierului un număr întreg numit *handle* sau *descriptor de fișier*. Acest întreg este chiar indexul intrării fișierului în tabela procesului respectiv.

In gestiunea fișierelor deschise pe aceste patru nivele, marea majoritate a fișierelor au exact câte o singură intrare pe fiecare nivel, corespunzătoare fișierului respectiv. De exemplu, procesul **A** din fig. 4.7 vede fișierul **F1** prin intermediul intrărilor **b**, **k**, **s** din cele trei tabele. De asemenea, procesul C vede fișierul F4 prin intermediul intrărilor **g**, **p**, **v**.

In Unix este posibil ca mai multe proceze să deschidă, în același timp, un același fișier - *multiacces la fișiere*. Acest lucru este posibil prin faptul că două intrări de fișiere din două proceze pot puncta spre aceeași intrare din tabela sistem. In fig. 4.7 intrarea **c** de la procesul **A** și intrarea **f** de la procesul **B** folosesc în comun același fișier, cel localizat de intrarea **1** din tabela sistem.

De regulă, fiecare intrare din tabela sistem punctează spre un inod, iar intrări diferite punctează spre inoduri diferite. Așa sunt, de exemplu, legăturile **k-s**, **l-u**, **p-v**, **q-w**.

Sunt interesante abaterile de la regula de mai sus. Astfel, spre exemplu avem legăturile **m-t** și **n-t**. Această corespondență este posibilă în cazul în care cel puțin una dintre intrările **m** sau **n** se referă la o *legătură simbolică*.

Legăturile din **o** și din **r** nu punctează spre nici un inod. Legătura **o** este un canal de tip pipe, memorie partajată, coadă de mesaje sau semafor, acestea fiind găzduite în nucleu. Legătura **r** este un socket, prin care se realizează legătura prin rețea cu un alt sistem de operare.

In fine, în marea majoritate a cazurilor există o corespondență biunivocă între inoduri și fișierele corespunzătoare. In fig. 4.7 avem corespondențele **s-F1**, **t-F2**, **v-F4**.

Abaterea de la această regulă este făcută doar de *legăturile hard*. Fiecare astfel de legătură creează un nou inod pentru același fișier. In fig. 4.7 avem corespondențele **u-F3** și **w-F3**, cel puțin una dintre **u** și **w** fiind o legătură hard.

4.3 Apeluri sistem pentru lucrul cu fișiere

4.3.1 Operații I/O

Există două posibilități de efectuare a operațiilor I/O din programe C:

- Prin funcțiile standard C (`fopen`, `fclose`, `fgets`, `fprintf`, `fread`, `fwrite`, `fseek`, etc.) existente în bibliotecile standard C; prototipurile acestora se află în fișierul header `<stdio.h>` (*nivelul superior de prelucrare al fișierelor*).
- Prin funcții standardizate POSIX (`open`, `close`, `read`, `write`, `lseek`, `dup`, `dup2`, `fcntl`, etc.) care reprezintă puncte de intrare în nucleul Unix și ale căror prototipuri se află de regulă în fișierul header `<unistd.h>`, dar uneori se pot afla și în `<sys/types.h>`, `<sys/stat.h>` sau `<fcntl.h>` (*nivelul inferior de prelucrare al fișierelor*).

Prima categorie de funcții o presupunem cunoscută deoarece face parte din standardul C (ANSI). Funcțiile din această categorie reperează orice fișier printr-o structură `FILE *`, pe care o vom numi *descriptor de fișier*.

Funcțiile din a doua categorie constituie apeluri sistem Unix pentru lucru cu fișiere și fac obiectul secțiunii care urmează. Ele (antetul lor) sunt cuprinse în standardul POSIX. Funcțiile din această categorie reperează orice fișier printr-un întreg nenegativ, numit *handle*, dar atunci când confuzia nu este posibilă îl vom numi tot *descriptor de fișier*. Fiecare astfel de descriptor este index în tabela de fișiere deschise a procesului. De exemplu, să considerăm procesul **C** din fig. 4.7. Aici, aceste handle sau descriptori sunt indecsi cu valorile **0, 1, 2, 3 ...** care punctează la intrările **g, h, i, j, ...** din tabela de fișiere a procesului **C**.

4.3.2 Apelul sistem `open`

Prototipul funcției sistem este:

```
int open (char *nume, int flag [, unsigned int drepturi ]);
```

Funcția `open` întoarce un întreg - *handle* sau *descriptor de fișier*, folosit ca prim argument de către celealte funcții POSIX de acces la fișier. În caz de eșec `open` întoarce valoarea `-1` și poziționează corespunzător variabila `errno`. În cele ce urmează vom numi `descr` acest număr.

`nume` - specifică printr-un string C, calea și numele fișierului în conformitate cu standardul Unix.

Modul de deschidere este precizat de parametrul de deschidere `flag`. Principalele lui valori posibile sunt:

- `O_RDONLY` deschide fișierul numai pentru citire
- `O_WRONLY` deschide fișierul numai pentru scriere
- `O_RDWR` deschide fișierul atât pentru citire și pentru scriere
- `O_APPEND` deschide pentru adaugarea - scrierea la sfîrșitul fișierului
- `O_CREAT` creează un fișier nou dacă acesta nu există, sau nu are efect dacă fișierul deja există; următoarele două constante completează crearea unui fișier
- `O_TRUNC` asociat cu `O_CREAT` (și exclus `O_EXCL` vezi mai jos) indică crearea necondiționată, indiferent dacă fișierul există sau nu
- `O_EXCL` asociat cu `O_CREAT` (și exclus `O_TRUNC`), în cazul în care fișierul există deja, `open` eșuează și semnalează eroare

- O_NDELAY este valabil doar pentru fișiere de tip pipe sau FIFO și vom reveni asupra lui când vom vorbi despre pipe și FIFO.

In cazul în care se folosesc mai multe constante, acestea se leagă prin operatorul |, ca de exemplu: O_CREAT | O_TRUNC | O_WRONLY.

Parametrul drepturi este necesar doar la crearea fișierului și indică drepturile de acces la fișier (prin cei 9 biți de protecție) și acționează în concordanță cu specificarea umask.

4.3.3 Apelul sistem close

Inchiderea unui fișier se face prin apelul sistem:

```
int close (int descr);
```

Parametrul descr este cel întors de apelul open cu care s-a deschis fișierul. Funcția întoarce 0 la succes sau -1 în caz de eșec.

4.3.4 Apelurile sistem read și write

Acestea sunt cele mai importante funcții sistem de acces la conținutul fișierului. Prototipurile lor sunt:

```
int read (int descr, void *mem, unsigned int noct);
int write (int descr, const void *mem, unsigned int noct);
```

Efectul lor este acela de a citi (scrie) din (în) fișierul indicat de descr un număr de noct octeți, depunând (luând) informațiile în (din) zona de memorie aflată la adresa mem. În majoritatea cazurilor de utilizare, mem referă un sir de caractere (char* mem).

Cele două funcții întorc numărul de octeți efectiv transferați între memorie și suportul fizic al fișierului. De regulă, acest număr coincide cu noct, dar sunt situații în care se transferă mai puțin de noct octeți.

Aceste două operații sunt atomice - indivizibile și neintreruptibile de către alte procese. Deci, dacă un proces a început un transfer între memorie și suportul fișierului, atunci nici un alt proces nu mai are acces la fișier până la terminarea operației read sau write curente.

Operația se consideră terminată dacă s-a transferat cel puțin un octet, dar nu mai mult de maximul dintre noct și ceea ce permite suportul fișierului. Astfel, dacă în fișier există doar n < noct octeți rămași necititi atunci se vor transfera în memorie doar n octeți și read va întoarce valoarea n. Dacă în suportul fișierului există cel mult n < noct octeți disponibili, atunci se depun din memorie în fișier doar n octeți și write va întoarce valoarea n. În ambele situații, pointerul curent al fișierului avansează cu numărul de octeți transferați. Dacă la lansarea unui read nu mai există nici un octet necitit din fișier - s-a întâlnit marcajul de sfârșit de fișier, atunci funcția întoarce valoarea 0.

Dacă operația de citire sau de scriere nu se poate termina - a apărut o eroare în timpul transferului - funcția întoarce valoarea -1 și se poziționează corespunzător variabila `errno`.

Ca exemplu de folosire a apelurilor sistem `open`, `close`, `read` și `write` vom scrie un program care face același lucru ca și comanda shell `cp`, adică copiază conținutul unui fișier într-altul. Cele două fișiere se vor da ca parametrii în linia de comandă. Dacă al doilea fișier (fișierul destinație) nu există, el se va crea, iar dacă există deja, el se va suprascrie. Sursa programului este prezentată în fig. 4.8.

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

main(int argc, char* argv[]) {
    int fd_sursa, fd_dest, n, i;
    char buf[100], *p;
    if (argc!=3) {
        fprintf(stderr, "Eroare: trebuie dati 2 parametrii.\n");
        exit(1);
    }
    //deschidem primul fișier în modul read-only
    fd_sursa = open(argv[1], O_RDONLY);
    if (fd_sursa<0) {
        fprintf(stderr, "Eroare: %s nu poate fi deschis.\n", argv[1]);
        exit(1);
    }
    /* deschidem al doilea fișier în modul write-only. Dacă el nu
     * există, se va crea, sau dacă există va fi trunciat. 0755
     * specifică drepturile de acces ale fișierului nou creat
     * (citire+scriere+execuție pentru proprietar, citire+execuție
     * pentru grup și alții).
     */
    fd_dest = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0755);
    if (fd_dest<0) {
        fprintf(stderr, "Eroare: fișierul %s nu poate fi deschis.\n",
                argv[2]);
        exit(1);
    }
    //citim din fișierul fd_sursa bucăți de maxim 100 octeți și le
    //scriem în fișierul fd_dest, până când nu mai avem ce citi.
    for ( ; ; ) {
        n=read(fd_sursa, buf, sizeof(buf));
        if ((n == 0) break; // S-a terminat de citit fișierul
        p = buf;
        for( ; n > 0 ; ) { // Poate nu se poate scrie odata n octeti
            i = write(fd_dest, p, n);
            if (i == n) break;
            p += i;
            n -= i;
        }
    }
    //închidem cele două fișiere
    close(fd_sursa);
    close(fd_dest);
}
```

Figura 4.8 Copierea unui fișier cu apelurile sistem `read` și `write`

4.3.5 Apelul sistem lseek

Funcția sistem `lseek` facilitează accesul direct la orice octet din fișier. Evident, pentru aceasta trebuie ca suportul fișierului să fie unul adresabil. Prototipul acestei funcții sistem este:

```
long lseek (int descr, long noct, int deUnde);
```

Se modifică pointerul curent în fișierul indicat de `descr` cu un număr de `noct` octeți. Punctul de unde începe numărarea celor `noct` octeți este indicat de către valoarea parametrului `deUnde`, astfel:

- de la începutul fișierului, dacă are valoarea `SEEK_SET` (valoarea 0)
- de la poziția curentă dacă are valoarea `SEEK_CUR` (valoarea 1)
- de la sfârșitul fișierului dacă are valoarea `SEEK_END` (valoarea 2)

4.3.6 Apelurile sistem dup și dup2

Aceste două funcții sistem permit ca un același fișier să fie accesibil prin doi descriptori diferiți. Presupunem că `descrvechi` este o intrare în tabela de fișiere deschise a unui proces, care punctează, așa cum am văzut în fig. 4.7 din 4.2.4, spre o intrare în tabela de fișiere deschise a sistemului. În urma apelului sistem, prin `dup` sau `dup2` se ocupă o nouă intrare `descrnou` din tabela de fișiere a procesului, care va puncta spre același fișier în tabela de fișiere deschise a sistemului. Această duplicare are loc în următoarele condiții:

- `descrvechi` și `descripnou` referă același fișier fizic
- ambele păstrează modul de acces la fișier stabilit la deschidere
- ambii descriptori partajează același pointer curent în fișier

Prototipurile celor două apeluri sistem sunt:

```
int dup (int descrvechi);
int dup2 (int descrvechi, int descripnou);
```

Apelul sistem `dup` face o copie a `descrvechi` în primul (cel mai mic număr) descriptor liber din tabela de fișiere a procesului.

Apelul sistem `dup2` face o copie a `descrvechi` în `descripnou`, închizând, dacă este cazul, fișierul către care punctă înainte `descripnou`.

Ambele apeluri întorc noul descriptor (`descripnou`) care duplică accesul la fișierul reperat prin `descrvechi`. În caz de eșec, ambele întorc -1 și pozitionează corespunzător variabila `errno`.

De exemplu, dacă se dorește ca dintr-un program C mesajele de eroare să fie trecute într-un fișier de pe disc numit "ERORI", se poate proceda ca în programul din fig. 4.9.

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

main () {
    int descrvechi, descripnou;
```

```

descrvechi = open("ERORI", O_CREAT|O_WRONLY, 0755);
if (descrvechi < 0) {
    fprintf(stderr,"Pe terminal prin stderr: open ERORI imposibil\n");
    fprintf(stdout,"Pe terminal prin stdout: open ERORI imposibil\n");
    exit(1);
}
descrnou = dup2(descrvechi, 2);
if (descrnou != 2) {
    fprintf(stderr,"Pe terminal prin stderr: dup2 imposibil\n");
    fprintf(stdout,"Pe terminal prin stdout: dup2 imposibil\n");
    exit(1);
}
fprintf(stderr,"Mesaj in ERORI prin stderr: dup2 REUSIT\n");
fprintf(stdout,"Mesaj pe terminal prin stdout: dup2 REUSIT\n");
} //main

```

Figura 4.9 Sursa **testdup2.c**

Presupunem că programul se lansează fără nici o redirectare pentru I/O standard. Partea esențială a programului este `descrnou = dup2(descrvechi, 2);`; Prin aceasta, se închide automat fișierul cu descriptorul 2 care referă fișierul standard `stderr` și noul fișier standard de erori va fi fișierul `ERORI`. Dacă deschiderea sau `dup2` nu pot fi executate, atunci pe terminal va apărea un mesaj în dublu exemplar: trimis prin `stderr` și prin `stdout`. În caz de succes, pe terminal va apărea o linie trimisă prin `stdout` iar în fișierul `ERORI` linia trimisă prin `stderr`.

4.3.7 Apelul sistem `fcntl`

Această funcție sistem furnizează sau schimbă proprietăți ale unui fișier deschis indicat printr-un descriptor. Această funcție poate fi apelată prin unul dintre următoarele trei prototipuri:

```

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);

```

Primul parametru, `fd`, referă fișierul deschis asupra căruia se dorește aplicarea controlului.

Parametrul `cmd` este o constantă care specifică operația dorită și în funcție de ea mai urmează sau nu încă un argument. Vom prezenta doar o parte dintr-o parte posibilitățile oferite de `fcntl`, urmând ca în secțiunile următoare să revenim cu prezentarea de noi facilități:

- `fcntl(fd, F_DUPFD, arg)` are aproximativ același efect ca și `dup(fd)`, doar că `fcntl` copiază descriptorul `fd` în cel mai mic descriptor, mai mare sau egal cu `arg`.
- `fcntl(fd, F_SETFL, arg)` modifică flagurile de tratare a fișierului față de cum au fost fixate prin `open` sau printr-un `fcntl` precedent. Parametrul `arg` are aceleași valoari ca și parametrul `flag` de la `open`: una sau mai multe constante folosite pentru a specifica modul de deschidere a fișierului.
- `fcntl(fd, F_GETFL)` întoarce o configurație de biți reprezentând reuniunea valorilor curente ale flagurilor de tratare a fișierului, aşa cum au fost ele fixate prin `open` sau printr-un `fcntl(... F_SETFL ...)`.

Ce-a de a treia formă va fi tratată în secțiunea 4.6.3.

4.4 Gestiunea fișierelor

Unix permite efectuarea din C a principalelor operații asupra sistemului de fișiere, oferind în acest sens câteva apeluri sistem. Ele sunt echivalente cu comenziile Shell care efectuează aceleași operații. Aceste operații sunt definite prin specificări POSIX corespunzătoare.

De regulă, aceste funcții au prototipurile într-unul dintre fișierele header:

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

4.4.1 Manevrarea fișierelor în sistemul de fișiere

Iată prototipurile celor mai importante dintre aceste apeluri sistem:

```
int chdir (const char *nume);
char *getcwd(char *mem, int dimensiune);
int mkdir (const char *nume, unsigned int drepturi);
int rmdir (const char *nume);
int unlink(const char *nume);
int link(const char *numevechi, const char *numenou);
int symlink(const char *numevechi, const char *numenou);
int chmod (const char *nume, unsigned int drepturi);
int stat  (const char *nume, struct stat *stare);
int mknod(const char *nume, unsigned int mod, dev_t dev);
int chown(const char *nume, unsigned int proprietar,
          unsigned int grup);
int access(const char *nume, int permisiuni);
int rename(const char *numevechi, const char *numenou);
```

- `chdir` schimbă directorul curent în cel specificat prin `nume`. Pe sistemele BSD și Unix System V release 4 mai este prezent și apelul `fchdir` care face același lucru ca și `chdir`, doar că directorul este specificat printr-un descriptor de fișier deschis, nu prin `nume`.
- `getcwd` copiază în zona de memorie indicată de `mem`, de lungime `dimensiune` octeți, calea absolută a directorului curent. Dacă calea absolută a directorului are lungimea mai mare decât `dimensiune`, se returnează NULL și `errno` primește valoarea ERANGE.
- `mkdir` creează un nou director, având calea și numele specificate prin `nume` și drepturile indicate prin întregul `drepturi`, din care se rețin numai primii 9 biți.
- `rmdir` șterge directorul specificat prin `nume` (acest director trebuie să fie gol).
- `unlink` șterge fișierul specificat prin `nume`.
- `link` creează o legătură hard cu numele `numenou` spre un fișier existent, `numevechi`. Numele nou se poate folosi în locul celui vechi în toate operațiile și nu se poate spune care dintre cele două nume este cel inițial.
- `symlink` creează o legătură simbolică (soft) cu numele `numenou` spre un fișier existent, `numevechi`. O legătură soft este doar o referință la un fișier existent sau

inexistent. Dacă șterg o legătură soft se șterge doar legătura, nu și fișierul original, pe când dacă șterg o legătură hard, se șterge fișierul original.

- `chmod` atribuie drepturile de acces drepturi la fișierul specificat prin nume. Pe sistemele BSD și Unix System V release 4 mai este prezent și apelul `fchmod` care face același lucru ca și `chmod`, doar că fișierul este specificat printr-un descriptor de fișier deschis, nu prin nume.
- `stat` depune la adresa `stare` informații privind fișierul specificat prin nume (inode-ul fișierului, drepturile de acces, id-ul proprietarului, id-ul grupului, lungimea în octeți, numărul de blocuri ale fișierului, data ultimului acces la fișier, etc. – vezi exemplul de mai jos). Sistemele BSD și Unix System V release 4 au și apelul `fstat` care face același lucru ca și `stat`, doar că fișierul este specificat printr-un descriptor de fișier deschis, nu prin nume.
- `mknod` creează un fișier simplu sau un fișier special desemnat prin nume. Parametrul `mod` specifică printr-o combinație de constante simbolice legate prin simbolul ‘|’ atât drepturile de acces la fișierul nou creat, cât și tipul fișierului care poate fi unul dintre următoarele:
 - `S_IFREG` (fișier normal)
 - `S_IFCHR` (fișier special de tip caracter)
 - `S_IFBLK` (fișier special de tip bloc)
 - `S_IFIFO` (*pipe* cu nume sau FIFO – vezi capitolul 5)
 - `S_IFSOCK` (Unix domain socket – folosit pentru comunicarea locală între procese)

Dacă tipul fișierului este `S_IFCHR` sau `S_IFBLK`, atunci `dev` conține numărul minor și numărul major al fișierului special nou creat; altfel, acest parametru se ignoră.

- `chown` schimbă proprietarul și grupul din care face parte proprietarul unui fișier specificat prin nume. Noul proprietar al fișierului va fi cel indicat de parametrul `proprietar`, iar noul grup al fișierului va fi cel indicat de parametrul `grup`. Pe sistemele BSD și Unix System V release 4 mai este prezent și apelul `fcchown` care face același lucru ca și `chown`, doar că fișierul este specificat printr-un descriptor de fișier deschis, nu prin nume.
- `access` verifică dacă procesul curent are dreptul specificat de permisiuni relativ la fișierul specificat prin nume. `permisiuni` va conține una sau mai multe valori legate prin ‘|’ dintre următoarele: `R_OK` - citire, `W_OK` - scriere, `X_OK` – execuție și `F_OK` – existență fișier. Verificarea se face cu `UID`-ul și `GID`-ul reale ale procesului, nu cele efective (vezi capitolul 5).
- `rename` redenumește fișierul specificat de `numeVechi` în `numenou`.

4.4.2 *Creat, truncate, readdir*

Următoarele trei apeluri nu au corespondent direct printre comenziile Shell:

```
int creat(const char *pathname, unsigned int mod);
int truncate(const char *nume, long int lung);

#include <dirent.h>
struct dirent *readdir(DIR *dir);
```

Apelul sistem `creat` este echivalent cu următorul apel sistem:

```
open (const char *nume, O_CREAT|O_WRONLY|O_TRUNC);
```

Apelul sistem `truncate` trunchiază fișierul specificat prin nume la exact lung octeți.

Funcția `readdir` nu este o funcție sistem, ci este funcția POSIX pentru parcurgerea subdirectoarelor și fișierelor dintr-un director. Ea întoarce într-o structură de tipul `dirent` următorul subdirector sau fișier din directorul dat de parametrul `dir`. Această funcție încapsulează de fapt apelul sistem `getdents`. Alte funcții în legătură cu `readdir` și specificate de standardul POSIX sunt: `opendir`, `closedir`, `rewinddir`, `scandir`, `seekdir` și `telldir`. Nu intrăm în detalierea acestor funcții deoarece ele nu sunt funcții sistem.

Majoritatea apelurilor de mai sus întorc valoarea 0 în caz de succes și -1 (plus setarea corespunzătoare a variabilei `errno`) în caz de eroare. Excepție de la această regulă fac apelurile: `getcwd` care returnează NULL în caz de eroare și setează corespunzător `errno` și `readdir` care întoarce NULL în caz de eroare.

4.4.3 Un exemplu: obținerea tipului de fișier prin apelul sistem `stat`

In această secțiune vom da un exemplu de utilizare a lui `stat`. Exemplul se referă la afișarea tipului de fișier pentru fișierele ale căror nume sunt date ca argumente la linia de comandă.

Programul `tipfis.c` din fig. 4.10 exemplifică folosirea apelului sistem `stat` pentru determinarea tipului de fișier recunoscut de către sistem. Tipurile de fișiere le-am prezentat într-o secțiune precedentă.

```
/* tipfis.c:
 * Tipareste tipurile fisierelor date in linia de comanda
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>

main (int argc, char *argv[]) {
    int i;
    struct stat statbuff;
    char tip[40];
    strcpy(tip, "");
    for (i = 1; i < argc; i++) {
        printf ("%s: ", argv[i]);
        if (stat (argv[i], &statbuff) < 0)
            fprintf (stderr, "Eroare stat");
        switch (statbuff.st_mode & S_IFMT) {
            case S_IFDIR:
                strcpy(tip, "Director");
                break;
            case S_IFCHR:
                strcpy(tip, "Special de tip caracter");
                break;
            case S_IFBLK:
                strcpy(tip, "Special de tip bloc");
                break;
            case S_IFREG:
                strcpy(tip, "Obisnuit");
                break;
            case S_IFLNK:
                break;
        }
    }
}
```

```

/*
 * acest test nu va fi adevărat niciodată deoarece stat
 * verifică în cazul unei legături simbolice, fișierul
 * pe care îl referă legătura și nu legătura în sine.
 */
    * scriem aici pentru completitudine. Ca să verificăm
    * tipul unei legături simbolice putem folosi funcția
    * lstat asemănătoare cu stat și disponibilă pe
    * versiunile BSD și Unix System V.
    */
    strcpy(tip, "Legatura simbolica");
    break;
case S_IFSOCK:
    strcpy(tip, "Socket");
    break;
case S_IFIFO:
    strcpy(tip, "FIFO");
    break;
default:
    strcpy(tip, "Tip necunoscut");
}//switch
printf ("%s\n", tip);
}//for
}//main

```

Figura 4.10 Sursa tipfis.c

4.5 Alte apeluri sistem

In acest subcapitol, vom prezenta câteva apeluri sistem care pot fi utile programatorului Unix și care nu au mai fost prezentate până aici. Înainte de prezentarea fiecărui prototip vom scrie și directiva #include necesară la începutul sursei programului C. În unele cazuri apar două astfel de directive, iar programatorul o va alege, încercând, pe cea care este potrivită pentru varianta lui de Unix.

4.5.1 Time

```
#include <time.h>
time_t time(time_t *t);
```

Apelul sistem `time` returnează timpul scurs de la 1 Ianuarie 1970 00:00 UTC, măsurat în număr de secunde. Dacă `t` este nenul, numărul de secunde va fi salvat la adresa referită de `t`.

4.5.2 Umask

```
#include <sys/types.h>
#include <sys/stat.h>
unsigned int umask(unsigned int masca);
```

Apelul sistem `umask` setează masca drepturilor pentru fișierele nou create la valoarea dată de `masca & 0777`. Această mască de biți indică drepturile de acces care nu sunt setate implicit la crearea unui fișier. Valoarea anterioară a lui `umask` este returnată.

4.5.3 *Gethostname*

```
#include <unistd.h>
int gethostname(char *name, int lung);
```

Apelul sistem *gethostname* pune la adresa dată de *name*, pe lungime de maxim *lung* octeți, numele mașinii (calculatorului).

4.5.4 *Gettimeofday*

```
#include <sys/time.h>
int gettimeofday(struct timeval *tv, 0);
```

Apelul sistem *gettimeofday* pune în structura *tv*, de tip *timeval*, valoarea timpului curent. Structura *timeval* este definită în felul următor:

```
struct timeval {
    long      tv_sec;   /* numărul de secunde */
    long      tv_usec;  /* numărul de microsecunde */
};
```

4.5.5 *Mmap și munmap*

```
#include <sys/mman.h>
void* mmap(void *start, size_t lung, int prot,
           int atribut, int fd, off_t offset);
int munmap(void *start, size_t lung);
```

Apelul sistem *mmap* mapează lung octeți începând de la deplasamentul *offset* din fișierul (sau alt obiect) indicat de descriptorul de fișier *fd* în memorie, preferabil la adresa *start*. În general, *start* are valoarea 0 și adresa unde are loc maparea este returnată de *mmap*. Apelul sistem *munmap* șterge maparea de la adresa *start*, pe lungime *lung*. Parametrul *prot* specifică protecția de memorie dorită și poate avea valorile:

- PROT_EXEC - paginile de memorie pot fi executate
- PROT_READ - paginile de memorie pot fi citite
- PROT_WRITE - paginile de memorie pot fi modificate
- PROT_NONE - paginile de memorie nu pot fi accesate

Parametrul *atribut* specifică tipul de obiect mapat, opțiuni de mapare și dacă modificări ale conținutului memoriei unde a avut loc maparea sunt vizibile numai procesului curent sau și altor procese. Cele mai importante valori ale sale sunt: MAP_FIXED, MAP_SHARED, MAP_PRIVATE.

4.5.6 *Fsync și fdatasync*

```
#include <unistd.h>
int fsync(int fd);
int fdatasync(int fd);
```

Apelul sistem `fsync` scrie pe disc datele fișierului `fd` modificate în buffer-ele nucleului, dar nesalvate încă pe disc. Diferența dintre `fsync` și `fdatasync` este că `fdatasync` nu salvează și date de control despre fișier ca timpul ultimului acces la fișier, ci doar conținutul fișierului.

4.5.7 *Uname*

```
#include <sys/utsname.h>
int uname(struct utsname *buf);
```

Apelul sistem `uname` returnează în structura `buf` de tip `utsname` date despre nucleul sistemului de operare. Tipul `utsname` are următoarele câmpuri:

```
struct utsname {
    char sysname[];
    char nodename[];
    char release[];
    char version[];
    char machine[];
#define _GNU_SOURCE
    char domainname[];
#endif
};
```

Aceste câmpuri au aceleași sensuri ca și câmpurile afișare de comanda shell `uname`.

4.5.8 *Select și seturi de descriptori*

```
#include <sys/select.h>
int select(int n, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

Apelul sistem `select` monitorizează într-un interval de timp dat de `timeout` starea descriptorilor de fișiere din seturile `readfds`, `writefds` și `exceptfds` și raportează (returnează) câte dintre ele și-au modificat starea. Descriptorii din `readfds` vor fi monitorizați pentru a vedea dacă au apărut caractere noi disponibile pentru citire, cei din `writefds` pentru a vedea dacă o nouă operație de scriere nu se va bloca, iar cei din `exceptfds` vor fi monitorizați pentru excepții. Parametrul `n` trebuie să aibă ca valoare cel mai mare număr de descriptor din cele trei seturi plus 1.

Standardul POSIX specifică și patru macrouri pentru a manipula cele trei seturi de descriptori. Prototipurile lor sunt date mai jos, unde prin `fd` am notat un descriptor de fișiere, iar prin `set` unul dintre seturile de descriptori din `select`:

```
FD_CLR(int fd, fd_set *set); //sterge fd din set
FD_ISSET(int fd, fd_set *set); //este fd in set?
FD_SET(int fd, fd_set *set); //adauga fd la set
FD_ZERO(fd_set *set); //goiese set
```

Apelul sistem `select` multiplează de fapt mai multe canale de intrare-iesire sincrone.

4.6 Blocarea fișierelor

4.6.1 Un (contra)exemplu

Una din problemele ce apar frecvent în medii concurente este partajarea resurselor. Să considerăm următoarea situație: se dă un fișier cu numele `secv` care conține pe prima linie un număr întreg de 5 cifre, reprezentate în ASCII. Să considerăm că fiecare proces (deocamdată prin proces vom înțelege program aflat în execuție) face următoarele acțiuni:

- Citește acest număr din `secv`.
- Mărește numărul cu o unitate.
- Rescrie numărul rezultat în `secv`.

Programul `lockfile.c` din fig. 4.11 realizează aceste acțiuni. Funcțiile `my_lock(fd)` și `my_unlock(fd)` au sarcina de a asigura accesul exclusiv la fișierul `secv` pe durata acțiunii. Într-o primă variantă, comentăm apelurile celor două funcții.

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

main () {
    int fd, i, nrsecv;
    char buf[6];
    fd = open ("secv", O_RDWR);
    for (i = 0; i < 10000; i++) {

        // my_lock(fd);           Blocheaza fisierul

        lseek (fd, 0L, 0);      //pozitionare la inceput
        n = read (fd, buf, 5);  //citim numarul
        buf[n] = '\0';          //zeroul terminal
        sscanf (buf, "%d\n", &nrsecv); //convertim stringul citit in numar
        printf ("pid = %d, secventa = %d\n", getpid(), nrsecv);
        nrsecv++;               //incrementeaza secventa
        sprintf (buf, "%5d\n", nrsecv); //convertim numarul in string
        lseek (fd, 0L, 0);      //revenire inainte de scriere
        write (fd, buf, strlen(buf)); //scriem numarul inapoi

        // my_unlock(fd);         Deblocheaza fisierul
    }
}
```

Figura 4.11 Sursa `lockfile.c`

Inaintea rulării programului, se va crea cu un editor de texte fișierul `secv` în care se va introduce o singură linie conținând numărul 1. Se compilează programul și să presupunem că fișierul executabil are numele `a.out`. Se rulează apoi programul, simultan în două procese, folosind comanda:

```
$ a.out & a.out &
```

Cele două procese vor afișa, intercalat, linii pe ieșirea standard. Ultima linie afișată va fi ceva de forma:

```
pid = 1265 secventa = 11953
```

Rezultatul pare ciudat, cel puțin la prima vedere. În mod normal, valoarea numărului de secvență ar trebui să fie 20000! De ce nu se întâmplă aşa? Deoarece prin comentarea apelului funcțiilor `my_lock` și `my_unlock` secvența dintre ele poate fi executată, total sau parțial, în același moment de către cele două procese! Frecvent se va întâmpla că ambele vor citi din fișier aceeași valoare, o vor incrementa și o vor scrie. În consecință valoarea va fi mărită doar cu o unitate, deși au acționat asupra ei două procese!

4.6.2 Tipuri de blocare

Exemplul din secțiunea precedentă ilustrează faptul că în anumite condiții se impune ca asupra unui fișier sau asupra unei porțiuni din fișier, să aibă dreptul să acționeze doar un singur proces. Această restricție este cunoscută sub numele de *blocarea* unui fișier. Sub Unix sunt utilizate două feluri de blocare: conciliantă și obligatorie.

Blocare conciliantă (advisory) este atunci când sistemul știe care fișiere au fost blocate și de către cine, iar procesele cooperează, prin funcții de tip `my_lock` și `my_unlock`, la accesarea fișierului. Nucleul sistemului de operare nu previne situația în care un proces indisiplinat scrie în fișierul blocat.

Blocarea obligatorie (mandatory) are loc atunci când sistemul verifică la fiecare scriere și citire dacă fișierul este blocat sau nu. Pentru a permite blocarea obligatorie, trebuie invalidat dreptul de execuție al grupului și bitul set-group-ID să fie 1 pentru fișierul respectiv, iar la montarea sistemului de fișiere (prin `mount`) să se permită blocare obligatorie.

Blocarea unui fișier înseamnă blocarea accesului la orice octet din fișier.

Blocarea unui articol înseamnă blocarea accesului la un număr de octeți consecutivi din fișier.

Mecanismele de blocare a fișierelor sub Unix specificate de standardul POSIX (și implementate de biblioteca C de la GNU) permit, în esență, două tipuri de blocări, fiecare dintre ele putând fi conciliante sau obligatorii (implicit conciliante) în funcție de îndeplinirea sau nu a condițiilor de mai sus:

- blocare *exclusivă*, atunci când un singur proces are acces la fișier sau porțiunea din fișier. De obicei, este vorba aici de o operație de scriere în fișier, iar pe durata pregătirii operației și a serierii propriu-zise, nici un alt proces nu poate nici scrie, nici citi porțiunea rezervată.
- blocare *partajată*, atunci când porțiunea rezervată poate fi citită, simultan, de către mai multe procese, dar nici un proces nu scrie.

Acste tipuri de blocare corespund rezolvării unei probleme celebre de programare concurrentă - problema cititorilor și a scriitorilor.

4.6.3 Blocarea conciliantă prin `fcntl`

Nucleele Unix oferă mai multe apeluri sistem destinate blocării. De exemplu, sub Linux și FreeBSD există o funcție sistem `flock` pentru blocare. De asemenea, biblioteca C de la GNU oferă funcțiile `flock` și `lockf`. Toate fac uz de serviciile apelului sistem `fcntl` pentru realizarea blocării. Reamintim cea de-a treia formă a prototipului `fcntl`, folosită pentru blocare:

```
int fcntl(int fd, int cmd, struct flock *lock);
```

Parametrul `fd` este descriptorul fișierului supus blocării.

Parametrul `lock` este un pointer la o structură de tip `flock`, prin care sunt indicați parametrii operației de blocare. Această structură ce conține cel puțin următoarele câmpuri:

```
struct flock {
    - - -
    short l_type;
    short l_whence; /* cum se interpretează l_start */
    off_t l_start;  /* offset-ul de început */
    off_t l_len;    /* numărul de octeți */
    pid_t l_pid;   /* PID-ul procesului care blocheaza blocajul
                      curent */
    - - -
};
```

Câmpul `l_type` indică tipul operației de blocare:

- `F_WRLCK` indică faptul că este vorba de o blocare exclusivă - blocare la scriere. Un singur proces poate efectua scriere în porțiunea rezervată, iar toate celelalte procese solicitante sunt în așteptare.
- `F_RDLCK` indică faptul că este vorba de o blocare partajată, procesele ce solicită porțiunea rezervată pentru citire își pot efectua operațiile, în timp ce procesele scriitorii stau în așteptare.
- `F_UNLCK` indică ridicarea stării de blocare, lăsând eventualele alte procese ce doresc blocarea să o facă.

Câmpul `l_whence` indică locul începând de unde se determină localizarea porțiunii rezervate. Valorile lui sunt aceleași cu cele de la funcția `lseek`:

- `SEEK_SET` indică reperarea porțiunii rezervate față de începutul fișierului.
- `SEEK_CUR` indică reperarea porțiunii rezervate față de poziția curentă a pointerului fișierului.
- `SEEK_END` indică reperarea porțiunii rezervate față de sfârșitul fișierului.

Câmpul `l_start` indică începutul porțiunii rezervate, numărat în octeți față de locul indicat prin `l_whence`. Acest câmp poate avea atât valori pozitive, cât și valori negative. În consecință, începutul zonei se specifică prin combinația parametrilor `l_whence` și `l_start`.

Câmpul `l_len` indică lungimea în octeți a zonei rezervate, un număr nenegativ. Dacă valoarea este strict pozitivă atunci sunt rezervați pentru blocare un număr de `l_len` octeți de la începutul zonei rezervate. Dacă `l_len` are valoarea zero, atunci sunt rezervați toți octeții de la începutul zonei până la sfârșitul fișierului.

Câmpul `l_pid` conține PID-ul procesului care ține blocată regiunea (are sens numai în cazul comenzi `F_GETLK`).

Parametrul `cmd`, în acțiunile de blocare, poate avea următoarele valori:

- `F_SETLKW` indică faptul că procesul cooperează la blocarea fișierului. Dacă fișierul este blocat exclusiv (la scriere) de către un alt proces, atunci procesul curent intră în aşteptare până la deblocarea fișierului. Acțiunea de blocare pe care o execută, după eventuala ieșire din starea de aşteptare, este cea dictată de parametrul `lock`.
- `F_SETLK` este similar cu `F_SETLKW`, numai că în cazul unei blocări exclusive procesul nu mai intră în aşteptare, ci întoarce `-1` și setează `errno` la `EACCES` sau `EAGAIN`.
- `F_GETLK` întoarce la adresa `lock` starea de blocare în care se află fișierul.

In fig. 4.12 sunt prezentate sursele funcțiilor `my_lock` și `my_unlock` în care se folosește blocarea și deblocarea prin `fcntl`.

```
void my_lock(int fd) {
    struct flock lock;
    lock.l_type = F_WRLCK; //blocat exclusiv (la scriere)
    lock.l_whence = SEEK_SET; //baza blocarii (inceputul fisierului)
    lock.l_start = 0; //offset blocare fata de bază
    lock.l_len = 0; //lungimea blocarii, tot fisierul
    fcntl(fd, F_SETLK, &lock); //comanda blocarea
} //my_lock

void my_unlock(int fd) {
    struct flock lock;
    lock.l_type = F_UNLCK; //deblocheaza
    lock.l_whence = SEEK_SET; //incepand de la inceputul fisierului
    lock.l_start = 0; //pozitia 0 fata de whence
    lock.l_len = 0; //tot fisierul
    fcntl(fd, F_SETLK, &lock); //comanda deblocarea
} //my_unlock
```

Figura 4.12 funcțiile `my_lock` și `my_unlock` folosind `fcntl`

4.6.4 Blocare prin `lockf` și `flock`

Prototipurile celor două funcții sunt:

```
int lockf(int fd, int actiune, long lungime);
int flock(int fd, int actiune);
```

Apelul `lockf` realizează, în funcție de condițiile specificate mai sus, fie blocare conciliantă, fie obligatorie (implicit, blocare conciliantă). De asemenea, poate bloca un fișier întreg sau numai o parte din el. Apelul `flock` blochează numai conciliant și numai întregul fișier.

actiune pentru `lockf` este una dintre următoarele:

- `F_ULOCK` deblocarea unei regiuni blocate
- `F_LOCK` blocarea unei regiuni
- `F_TLOCK` testarea și blocare dacă nu este deja blocată
- `F_TEST` testarea unei regiuni dacă este blocată sau nu

actiune pentru `flock` este una dintre următoarele:

- `LOCK_SH` face blocare partajată (acces din mai multe procese)

- `LOCK_EX` face blocare exclusivă
- `LOCK_UN` face deblocare
- `LOCK_NB` în general, dacă se cere blocarea și fișierul este deja blocat, atunci programul așteaptă la apelul `flock` până când poate realiza blocarea cerută. Legarea acestui atribut, cu *sau* (), de una dintre acțiunile anterioare, face ca în această situație să nu se mai aștepte.

`fd` este descriptorul de fișier.

`lungime` spune câți octeți vor fi blocați în fișier începând cu poziția curentă. Dacă `lungime=0` atunci se blochează întregul fișier.

In caz de erori, funcția întoarce -1 și poziționează variabila globală sistem `errno`.

Reamintim că dintre cele două funcții, numai `flock` reprezintă o funcție sistem implementată de nucleu (pe care o apelăm prin intermediu funcției POSIX cu același nume implementată de biblioteca C de la GNU – menționăm din nou că funcția `flock` a bibliotecii C de la GNU este o funcție *wrapper* pentru funcția sistem `flock` implementată de nucleu), apelul funcției `lockf` traducându-se de fapt într-un apel sistem `fcntl`.

Folosirea lui `F_TEST` este relativ nesigură, din cauză că secvența:

```
if (lockf (fd, F_TEST, size) == 0)
    rc = lockf (fd, F_LOCK, size)
```

nu este echivalentă cu secvența:

```
rc = lockf (fd, F_TLOCK, size)
```

deoarece între `F_TEST` și `F_LOCK` ar putea interveni un alt proces care să blocheze.

In fig. 4.13 sunt prezentate sursele `my_lock` și `my_unlock` realizate folosind `lockf`.

```
void my_lock (int fd) {
    lseek (fd, 0L, 0);
    lockf (fd, F_LOCK, 0L);
}//my_lock

void my_unlock (int fd) {
    lseek (fd, 0L, 0);
    lockf (fd, F_UNLOCK, 0L);
}//my_unlock
```

Figura 4.13 funcțiile `my_lock` și `my_unlock` folosind `lockf`