LAB 1

Interfețe cu utilizatorul

Orice sistem de operare deţine o interfață prin intermediul căreia realizează comunicarea cu operatorul uman. Primele sisteme de operare aveau interfețe foarte simple, formate dintr-un set mic de comenzi de bază.

Spre exemplu, **CP/M**, un sistem destinat microcalculatoarelor cu procesoare pe 8 biţi (de exemplu Z80), avea aproximativ 5 comenzi. Odată cu trecerea timpului, interfeţele dintre sistemele de operare şi utilizator au devenit din ce in ce mai complexe, oferind mai multe facilităţi şi uşurând munca de configurare şi întreţinere a calculatoarelor pe care le deservesc.

Pentru a vedea care sunt avantajele şi dezavantajele diferitelor sisteme de operare în ceea ce priveşte interfaţa cu utilizatorul, să încercăm mai întâi să realizăm o clasificare a interfeţelor. În primul rând, trebuie remarcat faptul că *interfaţă* este orice instrument care permite comunicarea între un sistem de operare şi un operator, indiferent dacă acest instrument este de natură *hardware* sau *software*. Din această perspectivă, putem observa următoarele tipuri de interfeţe cu utilizatorul:

- **Interfeţe în linie de comandă (sau interfeţe text).** Acestea sunt reprezentate, în general, de un program numit *interpretor de comenzi*, care afişează pe ecran un prompter, primeşte comanda introdusă de operator şi o execută. Comenzile se scriu folosind tastatura şi pot fi însoţite de parametri. Aproape toate sistemele de operare includ o interfaţă în linie de comandă, unele foarte bine puse la punct (cazul sistemelor Unix), iar altele destul de primitive (MS-DOS şi Microsoft Windows). În ultimele versiuni ale sistemului de operare Windows (Windows 7, Windows 8, Windows 10, Windows Server 2012, etc) Microsoft a introdus o interfață cu utilizatorul mult mai evoluată: Windows PowerShell.
- **Interfeţe grafice.** Sunt cele mai populare interfeţe cu utilizatorul şi se prezintă sub forma unui set de obiecte grafice (de regulă suprafeţe rectangulare) prin intermediul cărora operatorul poate comunica cu sistemul de operare, lansând aplicaţii, setând diferite opţiuni contextuale, etc. Dispozitivul cel mai folosit în acest caz este mouse-ul, de aceea acest tip de interfaţă este utilă în primul rând utilizatorilor neexperimentaţi şi neprofesionişti lor.

	Interfață în linie de comandă	Interfață grafică
Avantaje	• Permite scrierea clară şi explicită a comenzilor, cu toţi parametrii bine definiţi • Oferă flexibilitate în utilizare • Comunicarea cu sistemul de operare se face rapid şi eficient	• Este intuitivă şi uşor de folosit • Poate fi utilizată şi de către neprofesionişti • Creează un mediu de lucru ordonat • Permite crearea şi utilizarea de aplicaţii de complexe, precum şi integrarea acestora în medii de lucru unitare
Dezavantaje	• Operatorul trebuie să cunoască bine comenzile şi efectele lor • Este mai greu de utilizat de către neprofesionişti	• Anumite operaţii legate, de exemplu, de configurarea sistemului pot să nu fie accesibile din meniurile şi ferestrele interfeţei grafice • Interfaţa ascunde anumite detalii legate de preluarea şi execuţia comenzilor • Foloseşte mai multe resurse şi este mai puţin flexibilă decât interfaţa în linie de comandă

Sistemul de fișiere UNIX

În sistemele UNIX datele sunt organizate pe disc sub formă de fișiere. Acestea sunt simple fluxuri de octeți, nemaiexistând alte forme de organizare a informației. Există totuși posibilitatea de organizare a fișierelor în directoare, dar acestea sunt la rândul lor tot fișiere ce nu pot fi scrise de utilizator. UNIX are un sistem de fişiere arborescent. Spre deosebire de sistemul de fisiere DOS/Windows, separatorul între componentele numelui de fişier (directoare şi numele propriu-zis) este caracterul `/` (slash) şi nu `\` (backslash). De asemenea, numele de fişiere nu conţin un identificator al discului fizic (A:, C:, etc.), ci întreaga ierarhie de fişiere porneşte de la o rădăcină unică, notată cu `'/'` (slash). Un exemplu de organizare a sistemului de fisier UNIX poate fi și următorul (extras de pe un sistem Linux Debian 9.8 - folosind comanda *tree*):

```
/
|--                                                                  bin
|--                                                                 boot
|--                                                                  dev
|--                                                                  etc
|--                                                                 home
|                                                  |--             student
|                                                 |--            student1
|--                                                                  lib
|--                                                                lib32
|--                                                                lib64
|--                                                                media
|--                                                                  mnt
|--                                                                  opt
|--                                                                 proc
|--                                                                 root
|--                                                                  run
|--                                                                 sbin
|--                                                                  srv
|--                                                                  sys
|--                                                                  tmp
|--                                                                  usr
|                                                    |--              bin
|                                                   |--             games
|                                                  |--            include
|                                                     |--              lib
|                                                   |--             lib32
|                                                   |--             local
|                                                    |--              sbin
|                                                   |--             share
|                                                     `--              src
|--                                                                  var
|--        initrd.img        ->        boot/initrd.img-4.9.0-8-amd64
|--      initrd.img.old       ->      boot/initrd.img-4.9.0-7-amd64
|--         vmlinuz          ->         boot/vmlinuz-4.9.0-8-amd64
`--        vmlinuz.old         ->        boot/vmlinuz-4.9.0-7-amd64
```

- /bin - prescurtarea de la "*binaries*" - acest director conține unele programe utilitare fundamentale (cum ar fi ls, cp,...)
- /boot - conține fișierele necesare pentru a porni cu succes un sistem UNIX: fișierul ce reprezintă kernel-ul (nucleul), fișierul initrd (initial ramdisk) ce reprezintă sistemul de fișiere inițial folosit la pornirea kernel-ului, fișierul cu opțiunile de configurare cu care a fost compilat kernel-ul, fissiere caracteristice bootloader-ului

- /dev - numele provine de la "*devices*"; conține fișiere speciale ce reprezintă echipementele hardware ale sistemului (ex: fisierul /dev/mem reprezintă întreaga memorie - spațiu de adrese - a sistemului, /dev/sda - reprezintă primul hard-disk al sistemului)
- /etc - este un director dedicat stocării fișierelor de configurare ale sistemului și ale programelor și serviciilor ce rulează în sistem
- /home - conține directoarele "personale" ale utilizatorilor. Fiecare utilizator are asignat un director personal în care are drepturi depline (poate să creeze, să șteargă, să editeze și eventual să execute fișiere), dar nu are dreptul să-și șteargă acest director. Această organizare a fost preluată ulterior de către sistemele Microsoft Windows (Windows 7, Windows 8, Windows 10 ...) prin directorul C:\Users
- /lib, /lib32, /lib64 - numele provine de la *libraries*; conține bibliotecile de bază ale sistemului, în principal cele folosite de programele din /bin. Variantele /lib32, /lib64 sunt prezente pentru a oferi suport pentru arhitecturi diferite. Bibliotecile conținute în acest director sunt de obicei biblioteci link-editate dinamic (shared libraries - cu extensia .so)
- /media - reprezinta un director folosit ca punct de montare implicit al dispozitivelor de stocare temporare (removable devices) precum stick-uri USB, HDD externe, etc.
- /mnt - numele provine de la *mount* și reprezinta directorul folosit în mod comun de administratorii sistemului pentru a monta diferite dispozitive de stocare, în principal hard-disk-urile
- /opt - numele provine de la *optional* și reprezintă un director folosit pentru a instala unele aplicații.
- /proc - acest director există pe disc, dar conținutul lui nu reprezintă fișiere propriu-zise. În acest director se montează sistemul de fișiere *procfs* ce conține, tot sub formă de fișiere, informații despre procesele ce rulează în sistem. Aceste fișiere nu ocupa spațiu efectiv pe disc ci reprezintă doar o interfață
- /root - directorul de home al utilizatorului *root*. Acest utilizator are drepturi depline în sistem fiind numit și *superuser*.
- /sbin - numele directorului provine de la "*system binaries*" și conține utilitare fundamentale necesare pornirii sistemului
- /srv - denumit și "*server data*"; conține anumite date ale unor servicii ce pot rula în sistem
- /sys - acest director există pe disc, dar conținutul lui nu reprezintă fișiere propriu-zise (analog cu /proc). În acest director se montează sistemul de fișiere *sysfs* ce conține fișiere pentru a accesa mai ușor anumite dispozitive din sistem
- /tmp - director folosit pentru fișiere temporare. Nu este definit dacă fișierele stocate aici rămân și după repornirea sistemului. În majoritatea cazurilor, conținutul acestui director este șters la pornirea sistemului. În principiu, orice utilizator are dreptul să scrie fișiere în acest director, dar nu are voie să execute fișiere din acest director.

- /usr - denumit și "*user filesystem*" și contine programe (fișiere binare - în /usr/bin), biblioteci statice sau dinamice (/usr/lib), fișiere header (/usr/include) ce nu sunt critice pentru sistem.
- /var - provine de la cuvântul "*variable*" și conține fișiere ce se modifică mai des față de restul sistemului. În această categorie pot intra fișiere de log, fișiere temporare create de server-ul de mail, etc.

Este important de mentionat faptul că sistemele UNIX/Linux nu folosesc extensia la fișiere, aceasta fiind folosită doar pentru a facilita utilizarea sistemului. Un fișier poate fi executabil indiferent de extensia lui (poate fi executabil un fișier fără extensie, cu extensia .jpg sau .txt). Extensiile sunt folosite de către utilizator pentru a identifica cu ușurință tipul de fișier. De asemenea, unele programe mai pot folosi extensia pentru a se adapta mai ușor. De exemplu, un editor de texte dacă deschide un fișier .c, .html, .java ar putea activa facilitatea de syntax highlight corespunzătoare.

Drepturi de acces

Sistemele UNIX sunt administrate folosind utilizatorul root (denumit și superuser). Acest utilizator are directorul de home în /root și nu în /home ca și celalalți utilizatori. Utilizatorul root are drepturi depline în sistemele UNIX și doar el poate crea sau șterge utilizatori. Dat fiind faptul că în sistemele UNIX toate activitățile se rezumă la accese de fișiere, restricționarea drepturilor utilizatorilor la componentele sistemului se realizează prin drepturi de acces la fișiere. Pentru a se putea explica drepturile de acces se va porni de la rezultatul comenzii *ls* cu argumentul *-al* (*ls -al*) rulat pe un director oarecare:

```
drwxr-xr-x  1 valy staffcs        4096 Mar 11 10:10 .
drwxr-xr-x  4 root root           4096 Mar 11 09:23 ..
-rwxr-xr-x  1 root root            584 Feb 21 10:35
#removeoffender.sh#
-rw-r--r--  1 valy staffcs        1028 Mar 13  2018 99-com.rules
drwxr-xr-x  2 valy staffcs        4096 Oct 25  2016 Desktop
-rwxr-xr-x  1 valy staffcs        7083 Dec 12  2017 a.out
-rwxr-xr--  1 valy staffcs         597 Oct  8 09:14 a.sh
drwxr-xr-x  3 valy staffcs        4096 Jan  3 21:04 apnd
drwxr-xr-x  2 valy staffcs        4096 May 24  2018 apt
drwxr-xr-x  2 valy staffcs        4096 May 24  2018 b
-rw-r--r--  1 valy staffcs         767 Nov 23  2017 big.c
-rw-r--r--  1 valy staffcs         770 Nov 14  2017 process.c
-rw-r--r--  1 valy staffcs         336 Nov 14  2017 process2.c
drwxr-xr-x  6 valy staffcs        4096 Nov  1 11:48 public_html
-rwxr-xr-x  1 valy staffcs         583 Nov 10 11:53
removeoffender.sh
drwxr-xr-x  6 valy staffcs        4096 Jan  8  2017 so
-rw-r--r--  1 valy staffcs        2630 Jul 28  2018
sshd_config_backup
```

```
drwxr-xr-x  3 valy staffcs        4096 Dec 12  2017 test3
drwxr-xr-x  2 valy staffcs        4096 Dec  5  2017 test_so
drwxr-xr-x  2 valy staffcs        4096 Nov 10  2017 testpid
drwxr-xr-x  2 valy staffcs        4096 Dec 14  2017 th
```

Comanda "ls" - denumită "list directory contents" are rolul de a printa la ieșirea standard (STDOUT) conținutul unui director. În cazul de față, s-a folosit comanda pe directorul curent al utilizatorului "valy" furnizându-se comenzii parametrul -l pentru a se opta pentru formatul lung. În acest format multicoloană (7 coloane), se printează, pe lângă numele fișierului/directorului reprezentat în ultima coloană, și unele informații suplimentare despre fișierele respective. Vom considera numerotarea coloanelor de la stânga la dreapta începând cu 1. Vom explica în continuare pe scurt informațiile din coloane, dar nu în ordinea apariției, ci într-o ordine mai ușor de înțeles

- coloana 7 - reprezintă numele fișierului/directorului
- coloana 6 - reprezintă data ultimei modificări a fișierului respectiv
- coloana 5 - reprezintă dimensiunea (în bytes) a fișierului respecitv
- coloana 2 - reprezintă numărul de legături hard-link
- coloana 3 - reprezintă utilizatorul proprietar al acestui fișier (owner)
- coloana 4 - reprezintă grupul din care face parte acest fișier
- coloana 1 - reprezintă drepturi de access

Formatul de reprezentare a drepturilor de acces este reprezentat prin 10 caractere (bytes):

	tip fișier	**drepturi pentru owner**			**drepturi pentru group**			**drepturi pentru others**		
număr caracter	1	2	3	4	5	6	7	8	9	10
exemplu	d	r	w	x	r	-	x	r	-	-

- primul byte reprezintă tipul fișierului. Acesta poate fi:
  - caracterul '-' (minus) în cazul în care este un fișier obișnuit (regular file)
  - caracterul 'd' în cazul în care fișierul respectiv este director
  - caracteru 'l' în cazul în care fișierul reprezintă o legătură simbolică
  - caracterul 'c' în cazul în care fișierul este un fișier de tip caracter

Următorii octeți pot constitui 3 grupe care se referă la drepturile asupra acestui fișier pentru:

- utilizatorul owner (prima grupă - octeții 2,3,4)
- utilizatorul care face parte din același grup ca și fișierul (a doua grupă - octeții 5,6,7)
- ceilalți utilizatori (cei ce nu sunt nici owner și nici nu fac parte din același grup din care face parte fișierul) - a treia grupă - octeții 8,9,10)

Astfel, dacă în oricare din cele 3 grupe, litera corespunzătoare este trecută atunci acel drept este valabil, iar daca litera este înlocuită cu '-' (minus) atunci acel drept nu este valabil.

Exemplu:

- pentru fișierul *a.sh* putem spune următoarele: este un fișier obișnuit (primul caracter din prima coloană este -), fișierul aparține utilizatorului "valy" și face parte din grupul "staffcs". Pentru owner (utilizatorul "valy") drepturile de acces sunt "rwx" - utilizatorul "valy" are drept de citire, scriere și execuție a acestui fișier. Pentru grup (utilizatorii ce fac parte din grupul "staffcs") drepturile de acces sunt "r-x" - utilizatorii ce fac parte din grupul "staffcs" vor avea drept de citire și executie, dar nu vor avea drept de scriere peste acest fișier. Pentru "*others*" adică ceilalți utilizatori (cei ce nu sunt nici utilizatorul "valy" și nici nu fac parte din grupul "staffcs") drepturile de acces sunt "r--" - acești utilizatori vor avea drept de citire pentru acel fișier, dar nu vor avea drept de scriere și execuție

Observație: Pentru un director, dreptul de execuție specifică dacă directorul respctiv este accesibil, adică se poate face **change directory** în acel director (se poate "intra" în director).

Drepturile de acces pot fi oricând modificate, dar numai de către utilizatorul owner sau utilizatorul root.

Interfața în linie de comandă UNIX. Comenzi UNIX

În modul linie de comandă, Unix prezintă mult mai multe facilităţi decât MS-DOS. Interpretorul de comenzi (care porneşte după introducerea numelui utilizatorului şi a parolei) poate fi ales de către utilizator. Există mai multe interpretoare clasice, fiecare răspunzând anumitor cerinţe. Interpretorul "standard" în Unix este **sh**, dar foarte folosite sunt şi **bash**, **tcsh**, **ksh**, **csh**. În sistemele Linux interpretorul de comenzi cel mai folosit este **bash**.

Comenzile Unix sunt de fapt programe executabile care pot fi găsite în directoarele **/bin**, **/usr/bin**. Diferenţele între interpretoarele de comenzi (numite şi ***shell***-uri) se văd mai ales în contextul fişierelor de comenzi. Practic, aceste interpretoare permit scrierea de adevărate programe, complexe, folosind comenzile Unix şi directivele speciale.

- comanda **man**

Din punct de vedere didactic, probabil una dintre cele mai importante comenzi este cea care afișează paginile de manual atât pentru comenzi de shell script și programe lansate de interpretorul de comenzi, cât și pentru prototipuri de funcții C. Se va discuta în continuare despre comanda **man**. Cea mai simplă formă de apel ar fi următoarea:

```
man [opțiuni] [secțiune] comandă
```

Argumentele dintre parantezele drepte [...] sunt opționale.

Paginile de manual se împart pe secțiuni. În momentul de față, în sistemele Linux există 9 secțiuni:

1. Secțiunea 1 - descrie comenzile standard (programe executabile și comenzi shell script)
2. Secțiunea 2 - apeluri sistem UNIX apelabile în limbajul C
3. Secțiunea 3 - funcțiile de bibliotecă C
4. Secțiunea 4 - informații despre fișierele speciale (în principal cele din /dev)
5. Secțiunea 5 - informații despre convențiile și formatele anumitor fișiere specifice sistemului
6. Secțiunea 6 - manuale de la jocurile din Linux
7. Secțiunea 7 - informații despre diverse teme ce nu pot fi incluse în alte secțiuni (spre exemplu man 7 signal)
8. Secțiunea 8 - comenzi de administrare a sistemului (de obicei doar pentru userul root)
9. Secțiunea 9 - rutine kernel

Argumentul *secțiune* este opțional. Astfel, în situația în care nu se specifică secțiunea pentru comanda/funcția pentru care se cere pagina de manual, programul man va căuta comanda/funcția în secțiuni în ordine crescătoare începând cu secțiunea 1. Programul man va afișa informațiile din prima secțiune în care funcția/comanda a fost găsită. În cazul în care există funcții/comenzi diferite, dar cu același nume și sunt în secțiuni diferite este necesar să se specifice secțiunile, în caz contrar se va furniza doar prima apariție.

În continuare se vor enumera și explica unele dintre cele mai importante și mai utilizate comenzi UNIX. Este **absolut necesară** consultarea paginii de manual.

- comanda **pwd** - tipărește numele directorului curent
- comanda **cd** (change directory) - schimbă directorul curent

Forma de apel pentru această comandă ar fi următoarea:

```
cd director
```

Argumentul director poate fi atât o cale *absolută* (cu referință directă către / - exemplu cd /home/student/Desktop) cât și o cale relativă (exemplu în cazul execuției din /home/student - cd Desktop). Dacă argumentul lipsește atunci cd va schimba directorul curent în directorul home al utilizatorului autentificat în sistem.

Pentru această comandă este necesar să se analizeze următoarele apeluri și răspunsuri ale interpretorului de comenzi executate din directorul home al utilizatorului *valy*:

```
valy@staff:~$ cd public_html
valy@staff:~/public_html$ cd

valy@staff:~$ cd public_html
valy@staff:~/public_html$ cd ..

valy@staff:~$ cd so/pipe/
valy@staff:~/so/pipe$ cd ..
valy@staff:~/so$ cd ..
valy@staff:~$
```

În primul rând este de remarcat prompt-ul și anume, în cazul de față: "valy@staff:~$".

Prompt-ul are următoarea sintaxă: "user@hostname:path$". În această sintaxă *hostname* reprezintă numele calculatorului și *user* numele utilizatorul autentificat. Se poate observa, în toate exemplele de mai sus că în câmpul pentru cale este trecut caracterul '~' (tilda). În Linux, acest caracter reprezintă o scurtătură pentru calea directorului *home* al utilizatorului autentificat. În cazul de față '~' (tilde) este o scurtătura către "*/home/valy*".

În primul exemplu se observă cum schimbarea în directorul "public_html" din rădăcina directorului home al utilizatorului *valy* duce și la schimbarea promptului indicând că acum directorul curent este "~/public_html" adică "/home/valy/public_html". Același lucru poate fi observat în toate exemplele. Puteți verifica aceste situații folosind comanda ***pwd***.

Pentru întoarcerea în directorul anterior (directorul părinte) s-a folosit forma "cd ..". Este important de analizat această formă. Argumentul ".." nu reprezintă un argument "special". Utilizând și exemplul de apel a comenzii "ls" de mai sus se poate constata că fiecare director are 2 subdirectoare implicite: directorul "." și directorul "..". Directorul ".." reprezintă o legătură

către directorul părinte al directorului curent, iar directorul "." reprezintă o legătură către însuși directorul curent.

- comanda **ls** - afișează conținutul directorului specificat ca și parametru.
- comanda **mkdir** - creează un director nou
- comanda **rmdir** - șterge directorul dat ca și parametru dacă acesta este gol
- comanda **cat** - afișează la fișierul standard de ieșire (STDOUT) conținutul fișierelor date ca și argument
- comanda **wc** - word cound - numără cuvinte, linii și caractere din fișiere (sau de la fișierul standard de intrare STDINT)
- comanda **find** - caută în directoarele date şi în subdirectoarele lor fişierele care satisfac condiţiile date ca și argumente
- comanda **od** - afişează conţinutul unui fişier în diferite formate: octal, zecimal, hexa, ASCII, etc.
- comanda **rm** - șterge fișiere
- comanda **mv** - schimbă numele fișierelor sau mută fișiere dintr-un director în altul
- comanda **cp** - copiază fișiere
- comanda **ln** - creează o legătură între fișiere
- comanda **chmod** - schimbă drepturile de acces pentru fișierul dat ca și argument
- comanda **df** - afişează informaţii despre un sistem de fişiere (disc, partiţie, etc.): spaţiul total, spaţiul liber şi altele. Sistemul de fişiere poate fi indicat prin directorul în care partiţia sau discul respectiv este montat, adică "rădăcina" sistemului de fişiere
- comanda **du** - afișează informaţii privind spaţiul ocupat de fişierele dintr-un director (inclusiv subdirectoarele sale). **du .** afişează aceste informaţii pentru directorul curent.
- comanda **ps** - afişează starea proceselor care rulează pe sistem la momentul curent
- comanda **kill** - oprirea unui proces în curs de execuţie
- comanda **date** - afişează data şi ora curente
- comanda **who** - afişează numele utilizatorilor conectaţi la sistem
- comanda **echo** - afișează la ieșirea standard argumentele date ca și parametru
- comanda **touch** - creează un fișier nou, gol (de dimensiune 0)

Editarea de fișiere text

În decursul acestui laborator se încurajează cât mai mult utilizarea sistemului in linie de comandă chiar și cu existența mediului grafic activat. Așadar se va descuraja deschiderea editorului de texte din meniurile interfeței grafice. Nu se vor impune la laborator editoarele de texte, studenții având libertatea să folosească editoarele preferate (dacă sunt instalate).

Este important de menționat diferența dintre un editor de documente și un editor de text. Editorul de documente nu este un editor de text, acesta salvând informația in fișiere binare (sau arhive, xml, etc). Exemple de editoare de documente: Microsoft Word, Libre Office Writer.

Un editor de texte va edita fișierul în mod text, iar rezultatul va fi un fișier text fără alte informații adiționale. Exemple de editoare de texte:

- Windows: Notepad, Notepad++
- Linux: Emacs, Gedit, vi, vim, mcedit, kate, etc.

Pentru apelarea corectă a unui editor de texte din linia de comandă în Linux se va folosi următoarea sintaxă:

```
<editor_de_texte> fișier_text &
```

Exemplu:

```
emacs fișier.c &
```

Semnul '&' (ampersant) are rolul de a "trece" editorul de texte invocat din terminal în background-ul acestuia. În această situație, după execuție, terminalul va rămâne "liber" și va mai putea fi folosit fără a se închide editorul de texte. În cazul în care nu se folosește semnul '&' editorul de texte va bloca terminalul respectiv și acesta nu va mai putea fi folosit decât după închiderea editorului.

Atenție! închiderea terminalului va duce la închiderea forțată a editorului ceea ce poate duce la pierderea datelor (dacă nu s-a salvat fișierul). Această metodă se aplică doar atunci când se folosește un mediu grafic... în mod de text absolut, în Linux, nu se recomandă această utilizare. La laborator se va folosi mediul grafic.

Pentru afișarea unui fișier text de mici dimensiuni se recomandș folosirea comenzii **cat** în loc de deschiderea acestuia într-un terminal.

LAB 2

Conținut teoretic

În Unix, dar şi, parţial, în DOS, majoritatea comenzilor folosesc aşa-numitele *fişiere standard de intrare* şi *fişiere standard de ieşire*. Acestea sunt concepte abstracte care reprezintă sursa din care comenzile îşi iau datele de intrare, respectiv destinaţia în care ele scriu rezultatele. Deci comenzile citesc din intrarea standard (**STDIN**) şi scriu în ieşirea standard (**STDOUT**). În mod normal, intrarea standard este reprezentată de tastatura calculatorului, iar ieşirea standard de către dispozitivul de afişare (monitorul). Adițional, există un fișier special de ieșire destinat pentru semnalarea erorilor ce apar în cursul execuției programelor, anume: *ieșirea standard de erori* (**STDERR**). Aceste fișiere speciale au si numere asociate (mai multe detalii în laboratorul intitulat *Apeluri sistem pentru lucrul cu fisiere*), anume 0 pentru STDIN, 1 pentru STDOUT și 2 pentru STDERR.

Atenție! Informația trimisă spre STDOUT este buffer-ată, pe când orice informație trimisă spre STDERR va fi afișată imediat. Pentru a evita conglomerarea informațiilor pe STDOUT se recomandă introducerea secvențelor speciale de final de linie la scriere, respectiv folosirea comenzilor de tip *flush*.

Exemplu: comanda **sort** (existentă atât în DOS cât şi în UNIX) funcţionează după principiul enunţat. Dacă este apelată fără nici un parametru, ea va aştepta introducerea liniilor de text de la tastatură (*intrarea standard*), până la introducerea caracterului **^Z** urmat de **Enter** în MS-DOS, sau a caracterului **^D**, în Unix, după care va sorta liniile şi le va afişa în ordine pe ecran (*ieşirea standard*).

Redirectări

Intrarea şi ieşirea standard pot fi schimbate folosind operatorii de ***redirectare***. Redirectarea "conectează" intrarea sau ieşirea comenzilor la un fişier dat. Pentru redirectarea intrării se foloseşte operatorul '<', iar pentru redirectarea ieşirii operatorul '>'.

Exemplu: comanda următoare preia liniile care trebuie sortate din fişierul **date.txt**, iar rezultatele vor fi afişate pe ecran. Se redirectează, deci, numai intrarea standard:

```
sort < date.txt
```

Pentru a redirecta numai ieşirea, adică liniile de text să fie citite de la tastatură, dar rezultatul să fie scris într-un fişier, se foloseşte următoarea formă:

```
sort > ordonat.txt
```

Redirectările se pot combina, astfel încât liniile să fie citite dintr-un fişier, iar rezultatul să fie scris în altul:

```
sort < date.txt > ordonat.txt
```

Pentru redirectarea ieșirii standard de erori însă trebuie să folosim operatorul într-o formă ușor modificată ce integrează și numărul asociat acestui fișier special:

```
sort 2> erori.txt
```

Fișierele spre care facem redirectări pot conține informații stocate anterior. Dacă se dorește păstrarea lor și adăugarea de informații adiționale prin rularea unei comenzi, se poate folosi operatorul '>>' (append). **Exemplu:**

```
sort >> ordonat.txt
```

În acest caz, informațiile furnizate de comanda *sort* se vor adăuga la finalul fișierului *ordonat.txt* fără a suprascrie informațiile deja stochate în fișier.

Atenție! Redirectările se fac înainte de executarea comenzilor.

Interpretoare de comenzi

În sistemul de operare Unix există mai multe interpretoare de comenzi, selectabile de către utilizator. Fiecare interpretor accepta un limbaj specific, astfel ca fișierele de comenzi care pot fi scrise diferă în funcție de acest limbaj. Interpretorul de comenzi "standard" este **sh**, dar foarte folosite sunt și **bash**, **tcsh**, **ksh**, **csh**. În sistemele Linux interpretorul de comenzi cel mai folosit este **bash**. În continuare, ne vom referi la comenzile și directivele specifice interpretoarelor *sh* și *bash*, pentru detalii referitoare la celelalte variante putând fi consultate paginile de manual corespunzătoare.

Ca terminologie, în limba engleză interpretorul de comenzi mai este numit ***shell***, iar un program (fișier de comenzi) scris în limbajul recunoscut de acesta se numește ***shell script***.

Lansarea în execuție a unui fișier de comenzi se face fie tastând direct numele acestuia (el trebuie să aibă dreptul de execuție setat):

```
petra@staff:~$ ls -l script.sh
-rw-r--r--  1 petra staffcs        1028 Mar 13  2018 script.sh

petra@staff:~$ chmod +x script.sh

petra@staff:~$ ls -l script.sh
-rwxr-xr-x  1 petra staffcs        1028 Mar 13  2018 script.sh


petra@staff:~$ ./script.sh
```

sau apelând interpretorul de comenzi cu un parametru reprezentând numele fișierului de comenzi:

```
petra@staff:~$ sh script.sh
```

sau

```
petra@staff:~$ bash script.sh
```

Atenție! În UNIX nu există o "extensie" dedicată care să identifice fișierele de comenzi, așa că numele lor pot fi alese liber.

O comandă poate fi lansată și în fundal (în *background*), adică execuția ei se va desfășura în paralel cu cea a interpretorului de comenzi, acesta afișând promptul imediat ce a lansat-o, fără să-i mai aștepte terminarea. Acest lucru se realizează adăugând caracterul '**&**' la sfârșitul liniei care conține comanda respectivă. **Exemplu:**

```
emacs fișier.c &
```

În mod convențional, scripturile încep cu o directivă de interpretor, care este practic un comentariu mai special, numit **shebang**. Acesta este folosit pentru selecția interpretorului dacă sistemul pune la dispoziția utilizatorilor mai multe. De exemplu, dacă dorim ca scriptul nostru să fie interpretat de bash, vom folosi:

```
#!/bin/bash
```

Înlănțuirea comenzilor

Comenzile se pot şi *înlănţui*, în sensul că ieşirea generată de una devine intrare pentru alta. Pentru aceasta, se foloseşte operatorul '|', numit uneori operatorul *pipe* (conductă).

Exemplu: Comanda **more** realizează afişarea pagină cu pagină a datelor citite din intrarea standard. O construcţie de forma:

```
ls | more
```

face ca ieşirea lui **ls** să fie legată la intrarea lui **more**, astfel încât, efectul va fi afişarea pagină cu pagină a fişierelor din directorul curent.

Se pot înlănţui oricâte comenzi şi, prin urmare, pentru afişarea pagină cu pagină, ordonate alfabetic, a numelor tuturor fişierelor din directorul curent, se foloseşte comanda:

```
ls | sort | more
```

Comenzile UNIX pot fi grupate în liste de comenzi trimise spre execuție intepretorului. Ele vor fi executate pe rând, o comandă fiind lansată în execuție numai după ce comanda anterioara s-a terminat. Listele se formează scriind un șir de comenzi separate prin caracterul '**;**'. **Exemplu:**

```
cd exemplu; ls -al
```

Dacă într-o listă, în loc de separatorul '**;**' se folosește separatorul '**&&**', atunci o comandă nu va fi executată decât în cazul în care precedenta s-a terminat cu cod de succes (codul 0).

```
ls -l dir && echo "mere"
```

În exemplul de mai sus, dacă utilizatorul are suficiente drepturi să obțină informații despre directorul *dir*, comanda *ls* va fi executată cu succes și se va trece la executarea următoarei comenzi, anume *echo*, ceea ce rezultă în afișarea cuvântului "mere" după informațiile despre conținutul directorului *dir*:

```
drwxr-xr-x  1 petra staffcs        4096 Mar 11 10:10 .

drwxr-xr-x  4 root root            4096 Mar 11 09:23 ..

drwxr-xr-x  2 petra staffcs        4096 Oct 25  2016 Desktop

-rwxr-xr-x  1 petra staffcs        7083 Dec 12  2017 a.out

-rwxr-xr--  1 petra staffcs         597 Oct  8 09:14 a.sh

mere
```

Dacă se folosește operatorul '**||**', atunci condiția este ca precedenta să se fi terminat cu cod de eroare (cod diferit de 0).

```
ls -l dir || echo "mere"
```

În acest exemplu, dacă utilizatorul are suficiente drepturi, iar comanda *ls* se execută cu succes, se vor afișa la ieșirea standard doar informațiile rezultate în urma execuției primei comenzi, a doua

fiind ignorată. Dacă în schimb, utilizatorul nu are suficiente drepturi, comanda *ls* nu se va termina cu un cod de succes și se va executa comanda *echo*, rezultând doar în afișarea cuvântului "mere".

```
true || echo aaa && echo bbb
```

Când avem o listă mai complexă ce include mai multe comenzi înlănțuite de mai mulți operatori, gruparea acestora se face de la dreapta la stânga.

```
(true || echo aaa) && (echo bbb)
```

Operatorul *'&&'* ne obligă să evaluăm prima dată partea stângă. Cele două comenzi fiind înlănțuite prin *'||'*, vom incerca să executăm prima comandă, anume *true*, ceea ce se întoarce imediat cu succes; prin urmare se va ignora instrucțiunea *echo aaa*. Prima paranteză se termină cu succes și se trece într-un final la evaluarea comenzii *echo bbb*, rezultând în afișarea cuvântului *bbb* la ieșirea standard.

Atenție! Atunci când comenzile se înlănțuie prin caracterul '|' (pipe) ele vor fi executate în paralel.

Variabile de mediu

Variabilele de mediu pot să conțină ca valoare un șir de caractere. Atribuirea de valori se face astfel:

```
variabilă=valoare
```

De exemplu:

```
var=ABCD
```

va asigna variabilei cu numele *var* șirul "ABCD". Dacă șirul asignat conține și spații, el trebuie încadrat între ghilimele.

Atenție! Nu se pune spațiu între numele variabilei și semnul egal (și nici după semnul egal)! Altfel, interpretorul de comenzi va considera că este vorba de o comandă numită *var* cu parametrii = și *ABCD* și nu de o atribuire.

Referirea unei variabile se face prin numele ei, precedat de *simbolul $*. De exemplu:

```
echo $var
```

va determina afișarea textului ABCD.

În UNIX există câteva variabile predefinite:

1. variabile *read-only*, actualizate de interpretor:

- **$?** - codul returnat de ultima comandă executată
- **$$** - identificatorul de proces al interpretorului de comenzi
- **$!** - identificatorul ultimului proces lansat în paralel
- **$#** - numărul de argumente cu care a fost apelat fișierul de comenzi curent
- **$0** - conține numele comenzii executate de interpretor
- **$1, $2** ... - argumentele cu care a fost apelat fișierul de comenzi care se află în execuție

2. variabile inițializate la intrarea în sesiune:

- **$HOME** - numele directorului "home" afectat utilizatorului;
- **$PATH** - căile de căutare a programelor;
- **$PS1** - prompter-ul pe care îl afișează interpretorul atunci când așteaptă o comandă;
- **$PS2** - al doilea prompter;
- **$TERM** - tipul terminalului pe care se lucrează.

Directive de control

Instrucțiuni de decizie

1. Instrucțiunea if

O comandă returnează o valoare la terminarea ei. În general, dacă o comandă s-a terminat cu succes, ea va returna 0, altfel va returna un cod de eroare nenul.

În prima formă a comenzii **if**, se execută *lista1*, iar dacă *și* ultima instructiune din listă returnează codul 0 (succes), se execută *lista2*, altfel se execută *lista3*.

```
if lista1
then lista2
else lista3
fi
```

În a doua formă se pot testa mai multe condiții: dacă *lista1* se termină cu succes, se va executa *lista2*, altfel se execută *lista3*. Dacă aceasta se termină cu succes se execută *lista4*, altfel se execută *lista5*.

```
if lista1
then lista2
elif lista3
then lista4
else lista5
fi
```

2. Instrucțiunea case

Această instrucțiune implementează decizia multiplă. Șablonul *tipar* este o construcție care poate conține simbolurile ? și *, similară celor folosite la specificarea generică a numelor de fișiere. Comanda expandează (evaluează) șirul *cuvânt* și încearcă să îl potrivească pe unul din tipare. Va fi executată lista de comenzi pentru care această potrivire poate fi făcută.

```
case cuvânt in
tipar1) lista1;;
tipar2) lista2;;

...
esac
```

Exemplu:

```
var="lorem ipsum"
case "$var" in
"lorem ipsum") echo "bla";;
"util") echo "whoooohooo";;
esac
```

3. Instrucțiuni de testare

O comandă ce se folosește adesea într-o construcție decizională este **test**. Se poate folosi pentru comparații aritmetice simple, comparații mai complexe între string-uri și verificarea unor atribute specifice fișierelor. În paragrafele următoare vom da câteva exemple din fiecare clasă. Pentru *informații adiționale* consultați *pagina de manual pentru comanda test.*

Din categoria operațiilor de comparație aritmetică ce ne sunt puse la dispoziție amintim: **-eq** (equal), **-ne** (not equal), **-lt** (less than), **-gt** (greater than), **-le** (less or equal) și **-ge** (greater or equal).

Exemplu:

```
if test "$var" -eq 2
then

    echo "var conține numărul 2"

else

    echo "var conține un alt număr"

fi
```

sau în mod echivalent:

```
test "$var" -eq 2 && echo "var conține numărul 2" || echo "var
conține un alt număr"
```

Pentru string-uri ne putem folosi de: **=** (equal), **!=** (not equal), \< și \> pentru comparații lexicografice, **-z** (zero - empty string) și **-n** (non-zero - not empty string).

Exemplul 1:

```
if test -z "$raspuns"
```

```
then

    echo "raspunsul este gol"

else

    echo "am primit raspuns :)"

fi
```

Exemplul 2:

```
if test "$var1" \< "$var2"

then

    echo "var1 este primul alfabetic"

else

    echo "var2 este primul alfabetic"

fi
```

Pentru a verifica diferite informații despre fișiere putem folosi: **-e** (exists), **-f** (fișier obișnuit - regular), **-d** (director), **-h** sau **-L** (legătură simbolică), **-r** (are permisiune de citire pentru user-ul curent), **-w** (are permisiune de scriere pentru user-ul curent) și **-x** (are permisiune de execuție pentru user-ul curent).

Exemplu:

```
if test -f "$cale"

then

    echo "Fișier obișnuit!"
```

```
fi
```

O metodă echivalentă este folosirea parantezelor drepte: **[** și **]** în jurul expresiilor, înlocuind cuvântul cheie *test,* în construcțiile decizionale:

```
if test expresie
```

este echivalent cu:

```
if [ expresie ]
```

Atenție! În jurul parantezelor drepte **[** și **]** trebuie pus spațiu; în caz contrar vor apărea erori la execuție!

Exemplu:

```
if [ -f "$cale" ]

then

    echo "Fișier obișnuit!"

fi
```

Atenție! În Bash există comenzi și construcții adiționale precum **[[...]], ((...))** și **let ... ,** însă acestea **nu sunt incluse în standardul POSIX** și prin urmare **nu există garanții legate de portabilitate.** Există o serie de diferențe între comenzile integrate prezentate anterior și cele adiționale din Bash; noi vom aminti doar câteva în cele ce urmează, iar pentru informații complete se recomandă consultarea paginilor de manual aferente.

Începând cu versiunea 2.02 pentru Bash, s-a adăugat **[[...]]**, numită și *comanda test extinsă*, introducând o sintaxă mai apropiată de alte limbaje de programare. Această construcție permite utilizarea operatorilor &&, ||, < și > în interiorul său fără a genera erori precum se întâmplă în cazul [...]. De asemenea, în cazul comparațiilor între string-uri, dacă se folosește [[...]] nu mai este

nevoie să citățm operatorii < și >. Nu în ultimul rând, această comandă se poate folosi și pentru a verifica expresii regulate.

Instrucțiuni de ciclare

1. Instrucțiunea while

Se execută comenzile din *lista2* în mod repetat, cât timp lista de comenzi *lista1* se încheie cu cod de succes.

```
while lista1
do lista2

done
```

Exemplu:

```
numar=1
while test $numar -le 5
do
    echo $numar
    numar=`expr $numar + 1`
done
```

2. Instrucțiunea until

Se execută comenzile din *lista2* în mod repetat, până când lista de comenzi *lista1* se încheie cu cod de succes.

```
until lista1
do lista2

done
```

Exemplu:

```
until test -r fisier

do sleep 5

done
```

3. Instrucțiunea for

Se execută lista de comenzi în mod repetat, variabila luând pe rând valorile *val1, val2,* ... Dacă lipsește cuvântul cheie **in**, valorile pe care le va lua pe rând *variabila* vor fi parametrii din linia de comandă pe care i-a primit fișierul de comenzi atunci când a fost lansat în execuție.

```
for variabila [in val1 val2 ...]
do lista

done
```

Exemplul 1:

```
for arg in "$@"
do
    echo "$arg"
done
```

Exemplul 2:

```
for element in abc def ghi jkl
do
  echo "Element: $element"
done
```

Alte instrucțiuni utile

- **break** - permite ieșirea din ciclu înainte de îndeplinirea condiției;
- **continue** - permite reluarea ciclului cu următoarea iterație, înainte de terminarea iterației curente;
- **exec** *cmd* - comenzile specificate ca argumente sunt executate de interpretorul de comenzi în loc să se creeze procese separate de execuție; dacă se dorește rularea comenzilor în procese separate ele se scriu direct, așa cum se scriu și în linia de comandă
- **shift** - realizează deplasarea argumentelor cu o pozitie la stânga ($2$1, $3$2, etc.);
- **wait** [*pid*] - permite sincronizarea unui proces cu sfârșitul procesului cu pid-ul indicat sau cu sfârșitul tuturor proceselor "fii";
- **expr** *expresie* - permite evaluarea unei expresii; această comandă așteaptă 3 argumente: primul operand, urmat de operator și la final cel de-al doilea operand (argumentele trebuie să fie separate de spații)

Substituții

Atunci când într-un *shell script* o comandă este încadrată de caractere ` (accent grav), interpretorul de comenzi va executa comanda, după care rezultatul acesteia (textul) va substitui locul comenzii în program. De exemplu, comanda:

```
director=`pwd`
```

va atribui variabilei *director* rezultatul execuției comenzii **pwd**, adică șirul de caractere ce conține numele directorului curent.

Un exemplu de utilizare a substituției este construirea de expresii aritmetice:

```
contor=1
contor=`expr $contor + 1`
```

Această secvență inițializează o variabilă *contor* la valoarea 1 (șir de caractere !) și apoi o "incrementează", în sensul că la sfârșit, ea va conține șirul de caractere "2".

Atenție! Operatorii și operanzii sunt argumente diferite ale comenzii, prin urmare, comanda:

```
expr 1+2
```

este greșită. Corect este:

```
expr 1 + 2
```

Parantezele duble **((...))**, o extensie regăsită în *ksh, zsh* și *bash*, ne permit să efectuăm operații aritmetice și manipulări de variabile folosind operatorii întâlniți în limbajul C (incrementare, decrementare, <=, >=, etc.).

Exemplul 1:

```
var=1
(( var++ ))
echo "Rezultat: $var"
```

Exemplul 2:

```
for (( c=5; c>0; c-- ))
do
   echo "$c secunde"
done
```

Atenție! Construcția ((...)) va efectua toate calculele în cadrul expresiei, însă nu va returna rezultatul final ale acestora. Pentru a obține rezultatul final, se recomandă folosirea construcției **$((...))**.

Exemplu:

```
var=$((1+2))
echo "Rezultat: $var"
```

Parantezele simple **(...)** ne permit să rulăm comenzile regăsite în interiorul lor într-un subshell. Când se termină execuția lor, se returnează un cod de ieșire, însă nu și informațiile trimise spre STDOUT. Pentru a capta aceste informații, se recomandă construcția **$(...)**, care se poate considera o versiune alternativă pentru **` ... `** . În general, se recurge la o astfel de construcție pentru a limita efectele secundare ale diferitelor comenzi (de exemplu: modificări în valorile variabilelor). Dacă o astfel de construcție se regăsește pe o linie cu mai multe comenzi, se va evalua prima dată construcția $(...) și apoi se va trece la evaluarea restului.

Exemplul 1:

```
echo "Azi: $(date) este o zi frumoasă!"
```

Exemplul 2:

```
var=2
(var=4)
echo $var
```

Acoladele **{ ... }** ne permit să tratăm un set de comenzi ca și un grup compact.

Exemplu:

```
true && { var=5; echo "bla"; echo "$var"; exit 1; }
```

Dacă acoladele sunt precedate de semnul **$**, construcția va returna valoarea variabilei cuprinse.

Exemplu:

```
var="./dir/dir2"
echo ${var}/fisier.txt
```

Atenție! Diferite mecanisme de citare duc la rezultate total diferite. Ghilimelele duble " **...** " păstrează valoarea literală a caracterelor, însă vor permite substituții în cazul elementelor precedate de **$,** ` și \. Ghilimelele simple ' **...** ' în schimb mențin nealterată informația regăsită în interior.

Exemplu:

```
var=5
echo "variabila var: $var"      -> variabila var: 5
echo 'variabila var: $var'      -> variabila var: $var
```

Atenție! Când se lucrează cu string-uri ce conțin spații, se recomandă încadrarea lor între ghilimele pentru a evita interpretarea lor ca și invocare de comandă. **Exemplu:**

```
var="ana are mere"

if test -n $var       -> se generează eroare: test: too many
arguments
then
    echo "$var"
fi

if test -n "$var"
then
    echo "$var"       -> se afișează: ana are mere

fi
```

Metode de citire

Există numeroase metode pentru a citi date de la tastatură sau dintr-un fișier transmis la runtime. În exemplele următoare, vom prezenta secvențe de cod ce ne permit procesări linie cu linie.

Exemplul 1:

```
while read linie

do

  echo $linie

done
```

Această secvență va încerca să preia informația introdusă de către utilizator linie cu linie. Informația poate proveni în urma redirectării unui fișier sau a introducerii directe de la tastatură, caz în care se va marca finalizarea prin apăsarea combinației *CTRL+D* în terminal.

Exemplul 2:

```
while read linie

do

  echo $linie

done < fisier.txt
```

În acest caz, observăm o *redirectare* la nivelul buclei *while*. Aceasta ne permite să preluăm din *fisier.txt* informația stocată, linie cu linie, și să o afișăm la *STDOUT*.

IFS (Internal Field Separator) este o variabilă specială folosită de către comanda *read* pentru a împărți informația pe linii, respectiv cuvinte. Valoarea implicită a variabilei este **spațiu ; tab ; linie-nouă**, însemnând că ori de câte ori se va întâlni unul dintre aceste caractere în cadrul textului, se va considera acel punct ca și început de linie nouă. Acest lucru creează probleme când avem de-a face cu fișiere text normale și dorim să îl procesăm linie cu linie folosind o comandă precum *cat*. Dacă fișierul de intrare conține:

```
ana are mere *
banane
```

cu valoarea implicită pentru IFS se va afișa:

```
ana
are
mere
fisier.txt
main.sh
banane
```

Se observă că se execută și interpretări ale informației citite, caracterul * fiind expandat la lista fișierelor din directorul curent (*glob expansion*). Pentru a evita acest lucru, vom șterge conținutul implicit al variabilei IFS. În acest caz, vom obține rezultatul așteptat, prevenind împărțiri si expansiuni nedorite:

```
IFS=""

for linie in `cat fisier.txt`
do
  echo $linie
done
```

Funcții

Se pot crea și funcții folosind sintaxa:

```
nume () { listă-de-instrucțiuni } [ redirectări ]
```

sau

```
function nume [()] { listă-de-instrucțiuni } [ redirectări ]
```

Entitățile trecute între paranteze drepte se consideră *opționale*. Dacă se folosește cuvântul cheie *function*, nu mai este obligatoriu să se pună parantezele rotunde după numele funcției.

Observăm că nu există o listă de parametri, precum în alte limbaje de programare, imediat după numele funcției. În cazul funcțiilor de shell, argumentele transmise după numele funcției la apel devin parametri poziționali ca și în cazul scriptului în sine și pot fi referențiati prin $1, $2, etc. La fel ca și în alte limbaje de programare există conceptul de vizibilitate pentru variabile: dacă există mai multe variabile cu același nume, cea referită va fi mereu cea cu vizibilitatea cea mai locală. Prin urmare când scriem $1 într-o funcție vom accesa mereu primul argument al funcției și nu primul argument al scriptului.

Exemplu:

```
#apel script: myscript ana are mere



#!/bin/sh

functiamea()
{
      echo "Primul arg: $1"        -> Primul arg: 1
      echo "Al doilea arg: $2"     -> Al doilea arg: 2
      var="altceva"
}

#scriptul se execută începând de aici
var="ceva"
functiamea 1 2
echo "Primul arg: $1"              -> Primul arg: ana
echo "Al doilea arg: $2"           -> Al doilea arg: are
echo $var                          -> altceva
```

De asemenea codul de ieșire pentru o funcție este codul returnat de ultima comandă din corpul funcției.

Recursivitate

Este posibil să apelăm scripturi din interiorul altor scripturi, inclusiv scriptul curent. Apelul scripturilor se poate face cum am descris anterior, ca și cum am face apelul din terminal. Putem interoga codul de ieșire al scriptului apelat folosind variabila **$?**. Pentru a seta un cod de ieșire putem folosi comanda **exit** urmată de valoare.

Exemplu:

```
factorial=1
if [ $1 -ne 0 -a $1 -ne 1 ]
then
      nou=`expr $1 - 1`
      bash main.sh $nou
      factorial=`expr $1 \* $?`
fi
exit $factorial
```

Scriptul de mai sus calculează valoarea factorialului pentru numărul transmis ca și prin argument. Pentru a vedea rezultatul final, din terminal putem invoca *echo* având ca și *argument $?*.

Alte comenzi utile

În continuare vom da o scurtă descriere pentru câteva comenzi mai des folosite pentru procesarea de informații:

- ***cat***
  scrie fiecare linie de la intrarea standard (sau din fișiere ale căror nume sunt date ca argumente) la ieșirea standard, fără modificări.

- ***head***
  scrie primele maxim 10 linii (10 este implicit; se poate specifica numărul de linii cu argumentul *-n*, sau se poate preciza un număr de octeți cu argumentul *-c*) de la intrare către ieșirea standard.

- ***tail***
  similară cu comanda ***head***, scrie ultimele 10 linii.
  Observație: Această comandă este obligată să citească întreg fișierul înainte de a scrie ceva la ieșire, astfel fiind un impediment în pipeline-izare. Se recomandă să fie folosită ultima, dacă este posibil, într-un lanț de comenzi.

- ***sort***
  afișează liniile citite în mod ordonat lexicografic, *implicit* ordinea fiind *crescătoare*. Comanda are o serie de argumente utile. Citiți pagina de manual **sort(1)**!

- ***uniq***
  elimină liniile succesive identice dintre cele de la intrarea standard. **Atenție**: două linii identice, dar care nu sunt citite una după alta nu vor fi depistate!

- ***cut***
  tipărește porțiuni din liniile citite. Aceste secțiuni pot fi intervale de octeți, caractere sau câmpuri, în funcție de unul din argumentele *-b*, *-c* respectiv *-f*. În ultimul caz se vor tipări acele câmpuri delimitate de un caracter (*implicit TAB*, se poate preciza cu argumentul *-d*) care sunt specificate folosind argumentul -f în forma*: lista[,lista]....* O *listă* poate fi un simplu număr reprezentând

câmpul dorit, sau poate fi de forma *N-M*, unde *N* și *M* sunt numere reprezentând primul, respectiv ultimul câmp ce trebuie afișat. Sau N sau M poate lipsi, în locul lor subînțelegându-se primul, respectiv ultimul câmp din linie. Aceeași notație folosită cu *-b* sau *-c* seminifică intervalul de octeți sau caractere ce se vor afișa. **Exemplu:**

```
ls -l | cut -f 1 -d ' '
```

va tipari doar lista de permisiuni a fișierelor din directorul curent.

- ***tr***
translatează sau șterge caractere. ***tr*** implicit translatează, caz în care trebuie date ca argumente două șiruri de caractere reprezentând două seturi. Caracterele din primul set vor fi translatate în caracterele din al doilea. Dacă numărul de caractere din seturile date nu este același, caracterele excedentare dintr-al doilea se ignoră dacă acesta e mai lung, sau se repetă ultimul caracter din al doilea set (dacă acesta e mai scurt) până la lungimea primului set. În cazul argumentului *-d* se dă un singur set de caractere, care vor fi eliminate la scrierea la ieșirea standard. Argumentul *-s* (*squeeze*) realizează "contracția" caracterelor din setul dat ca parametru: în cazul în care la intrare filtrul citește două sau mai multe caractere identice, din set, va fi tipărit la ieșire doar unul singur. **Exemplu:**

```
ps -x | tr -s ' ' | cut -f 2,6 -d ' '
```

Comanda *ps -x* afișează lista proceselor utilizatorului care o invocă. Cu ajutorul lui *tr -s ' '* se șterg spațiile dintre coloanele afișate, iar *cut -f 2,6 -d ' '* face să apară la ieșire doar coloana corespunzătoare identificatorilor de proces (coloana 2) și cea a numelui comenzii corespunzătoare procesului (coloana 6).

LAB 3

Expresii regulate

Expresiile regulate sunt niște șiruri de caractere ce reprezintă șabloane sau tipare (*pattern* în limba engleză). Ele se construiesc pe baza unei gramatici, la fel ca și un limbaj de programare. Aceste șabloane sunt folosite pentru "recunoașterea" și manipularea unor șiruri de caractere. Analog cu expresiile aritmetice, o expresie regulată este construită prin combinarea unor expresii mai mici cu ajutorul unor operatori.

O expresie regulată are 3 tipuri de componente principale:

- ancore (*anchors*) - folosite pentru a preciza poziționarea tiparelor relativ la textul analizat
- seturi de caractere (character sets) - simboluri ce se potrivesc cu una sau mai multe caractere din text
- modificatori (modifiers) - permit introducerea de repetiții în tipare

Ancore

Majoritatea comenzilor de procesare de text lucrează la nivelul liniilor. Prin urmare, metode de a referi începutul, respectiv finalul de linie devin utile în cazul expresiilor regulate. Pentru a căuta un anumit tipar la începutul liniei, se poate folosi caracterul **'^'**. Pentru a căuta anumite informații la finalul liniei, ne putem folosi de simbolul **'$'.**

Exemplul 1:

```
^turing$
```

se va potrivi doar cu șirul "turing" (nu și cu "featuring" sau "turing ").

Exemplul 2:

```
^joaca
```

se va potrivi cu șirul "joaca de copil", dar nu și cu "se joaca" sau "jocuri".

Seturi de caractere

Punctul (**.**) se potrivește cu orice caracter, unul singur (mai puțin caracterul *newline*, de obicei). Să începem cu un exemplu simplu:

```
a.z
```

Această expresie regulată se potrivește cu orice șir de caractere ce conține literele 'a' și 'z' între care se găsește orice caracter - dar unul singur (cu excepția caracterului newline, de obicei), cum ar fi: "*axz*", "*aaz*", "*barza*", dar nu "*abcz*".

Cele mai simple expresii regulate sunt cele care "se potrivesc" cu un singur caracter: majoritatea caracterelor (toate literele și cifrele) se potrivesc cu ele însele. Alte caractere însă au semnificație specială, și dacă dorim ca expresia regulată să se potrivească cu acel caracter, trebuie să îl citățm (*quote* în limba engleză). Aceasta se poate realiza prin plasarea unui *backslash* (**'\'**) în fața caracterului respectiv. Expresiile regulate mai complexe se vor forma fie prin concatenare (scriere una după alta), fie cu ajutorul operatorilor ce vor fi descriși mai jos.

Atenție! Prin **concatenarea** a două expresii regulate rezultă o expresie regulată ce se va potrivi cu șiruri de caractere ce conțin două subșiruri adiacente ce se vor potrivi cu prima respectiv a doua expresie regulată.

O altă construcție care potrivește un singur caracter este o listă de caractere închise între paranteze drepte **[...]**. Această expresie se va potrivi cu oricare din caracterele din listă. Astfel, expresia regulată:

```
compl[ei]ment
```

se va potrivi cu oricare din șirurile "*complement*", "*compliment*" sau "*mulțumesc pentru complimentul dumneavoastră*". Dacă o construcție cu paranteze drepte începe cu un **'^'**, atunci ea se va potrivi cu orice caracter ce **nu** este între paranteze. De **exemplu**:

```
3[^6890]
```

reprezintă o expresie regulată ce se potrivește cu orice șir ce conține cifra 3 și nu conține pe poziția următoare una din cifrele 6, 8, 9 sau 0 (*atenție*: dacă în șirul căruia i se aplică expresia regulată nu conține după 3 nici un alt caracter, expresia nu se va potrivi!). De asemenea se pot specifica intervale întregi (considerând ordinea ASCII a caracterelor) cu ajutorul **cratimei '-'**. De **exemplu**:

```
[A-Za-z]
```

reprezintă orice literă, mică sau mare.

Atenție! Caracterele care nu se potrivesc cu ele însele și de care aminteam mai sus sunt metacaractere și operatori. O parte dintre ele, printre cele mai des utilizate și implementate în diversele variante de expresii regulate le vom descrie mai jos, alături de alte construcții.

Modificatori

Sunt definiți o serie de operatori pentru a specifica repetițiile. Ei se aplică în dreapta unei expresii regulate, făcând-o să se potrivească repetitiv:

Operator	Modificare adusă
*****	Potrivește de **0** *sau* **mai multe** ori
+	Potrivește de **1** *sau* **mai multe** ori
?	Potrivește de **0** *sau* **1** ori

Parantezele rotunde **(...)** se folosesc pentru a grupa expresiile regulate. Astfel, dacă scriem:

```
ab*
```

aceasta semnifică un 'a' urmat de oricâte 'b'-uri (inclusiv nici unul); dar daca scriem

```
(ab)*
```

operatorul * se aplică grupului, semnificația fiind 0 sau mai multe repetiții ale șirului de caractere "ab".

Acoladele (pentru unele versiuni de interpretoare trebuie să fie precedate de backslash) **{ ..., ... }** ne permit să specificăm un număr minim, respectiv maxim de repetiții pentru un tipar. De **exemplu:**

```
[a-zA-Z]{4,8}
```

tiparul de mai sus va verifica dacă există cel puțin 4 litere mici sau mari, dar maxim 8 astfel de caractere. Din tipar poate lipsi limita minimă sau cea maximă; nu este obligatoriu să se specifice ambele.

Un alt operator util este **'|'**, operatorul de **alternare**. Rezultatul lui este că se va potrivi fie expresia regulata din stânga, fie cea din dreapta:

```
ion (pozitiv|negativ)
```

se va potrivi fie cu "ion pozitiv" fie cu "ion negativ".

Pentru mai multe detalii, consultați paginile de manual **grep(1)** și **perlre(1)**. Ultima prezintă expresiile regulate implementate în limbajul perl, și nu vor fi întotdeauna compatibile cu

comenzi precum grep și sed. În schimb găsiți acolo o descriere mai amănunțită. Expresiile regulate pot fi folosite și din limbaje de programare precum C, vedeți **regex(3)** și **regex(7)**.

Filtre

Filtrele sunt comenzi care citesc rând cu rând fișierul standard de intrare și afișează la ieșirea standard integral sau doar în parte rândurile citite, modificate sau nu, în funcție de semantica lor. Operația continuă până la întâlnirea marcajului de sfârșit de fișier. Aceste comenzi se înlănțuie des cu ajutorul pipe-urilor, pentru a le conjuga efectul. În sistemele UNIX există o serie de comenzi care se comportă ca filtre, multe dintre ele făcând parte din *standardul POSIX*. Majoritatea filtrelor pot însă citi date și din fișiere specificate în linia de comandă.

În continuare vom da o scurtă descriere pentru câteva filtre mai des folosite:

- ***cat***
  scrie fiecare linie de la intrarea standard (sau din fișiere ale căror nume sunt date ca argumente) la ieșirea standard, fără modificări.
- ***head***
  scrie primele maxim 10 linii (10 este implicit; se poate specifica numărul de linii cu argumentul *-n*, sau se poate preciza un număr de octeți cu argumentul *-c*) de la intrare către ieșirea standard.
- ***tail***
  similară cu comanda ***head***, scrie ultimele 10 linii.
  Observație: Această comandă este obligată să citească întreg fișierul înainte de a scrie ceva la ieșire, astfel fiind un impediment în pipeline-izare. Se recomandă să fie folosită ultima, dacă este posibil, într-un lanț de comenzi.
- ***sort***
  afișează liniile citite în mod ordonat lexicografic, *implicit* ordinea fiind *crescătoare*. Comanda are o serie de argumente utile. Citiți pagina de manual **sort(1)**!
- ***uniq***
  elimină liniile succesive identice dintre cele de la intrarea standard. **Atenție**: două linii identice, dar care nu sunt citite una după alta nu vor fi depistate!
- ***cut***
  tipărește porțiuni din liniile citite. Aceste secțiuni pot fi intervale de octeți, caractere sau câmpuri, în funcție de unul din argumentele *-b*, *-c* respectiv *-f*. În ultimul caz se vor

tipări acele câmpuri delimitate de un caracter (*implicit TAB*, se poate preciza cu argumentul *-d*) care sunt specificate folosind argumentul -f în forma*: lista[,lista]...*. O *listă* poate fi un simplu număr reprezentând câmpul dorit, sau poate fi de forma *N-M*, unde *N* și *M* sunt numere reprezentând primul, respectiv ultimul câmp ce trebuie afișat. Sau N sau M poate lipsi, în locul lor subînțelegându-se primul, respectiv ultimul câmp din linie. Aceeași notație folosită cu *-b* sau *-c* seminifică intervalul de octeți sau caractere ce se vor afișa. **Exemplu:**

```
ls -l | cut -f 1 -d ' '
```

va tipari doar lista de permisiuni a fișierelor din directorul curent.

- ***tr***
  traduce (*translate*) sau șterge caractere. ***tr*** implicit traduce, caz în care trebuie date ca argumente două șiruri de caractere reprezentând două seturi. Caracterele din primul set vor fi traduse în caracterele din al doilea. Dacă numărul de caractere din seturile date nu este același, caracterele excedentare dintr-al doilea se ignoră dacă acesta e mai lung, sau se repetă ultimul caracter din al doilea set (dacă acesta e mai scurt) până la lungimea primului set. În cazul argumentului *-d* se dă un singur set de caractere, care vor fi eliminate la scrierea la ieșirea standard. Argumentul *-s* (*squeeze*) realizează "contracția" caracterelor din setul dat ca parametru: în cazul în care la intrare filtrul citește două sau mai multe caractere identice, din set, va fi tipărit la ieșire doar unul singur. **Exemplu:**

```
ps -x | tr -s ' ' | cut -f 2,6 -d ' '
```

Comanda *ps -x* afișează lista proceselor utilizatorului care o invocă. Cu ajutorul lui *tr -s ' '* se șterg spațiile dintre coloanele afișate, iar *cut -f 2,6 -d ' '* face să apară la ieșire doar coloana corespunzătoare identificatorilor de proces (coloana 2) și cea a numelui comenzii corespunzătoare procesului (coloana 6).

- ***grep***
  este un filtru mai complex. Are cel puțin un parametru, care va fi interpretat ca expresie regulată. Liniile citite de la intrarea standard (sau din fișierele date ca parametri) care se potrivesc cu expresia regulată dată vor fi afișate. Există o serie de opțiuni care modifică modul de afișare. O parte le găsiți în lista de mai jos:
  - *-i* face comanda insensibila la diferențele dintre literele mari și cele mici
  - *-v* tipărește liniile cu care tiparul dat *nu* se potrivește

- *-n* tipărește numărul liniei urmat de caracterul **:** urmat de linia în sine
- *-E* forțează folosirea expresiilor regulate extinse (în mod normal se folosesc niște expresii regulate cu sintaxa foarte simplă); există comanda **egrep** care e echivalentă cu **grep -E**
- *-x* tipărește doar liniile pentru care tiparul dat se potrivește cu întreaga linie, nu cu un subșir din aceasta
- *-c* tipărește doar numărul de linii ce s-ar fi afișat în mod normal

Exemplu:

```
ps -x | grep lab9
```

Comenzile afișeaza liniile generate de **ps** corespunzătoare proceselor ce conțin în numele lor (sau în parametri) șirul "lab9".

- ***tee***
  scrie la ieșirea standard ceea ce citește de la intrarea standard, în plus scrie și în fișierele ale căror nume sunt date ca parametri.

- ***wc***
  afișează numărul de caractere (parametrul *-m*), de octeți (*-c*), de linii (*-l*) sau de cuvinte (*-w*), sau lungimea celei mai lungi linii întâlnite (parametrul *-L*). De **exemplu:**

```
cat /etc/passwd | wc -l
```

va afișa numărul de intrări (linii) din fișierul de parole.

- ***xargs***
  citește argumente de la intrarea standard, separate prin spații sau caractere *newline* și execută comanda dată ca parametru (sau **echo** implicit) cu o listă de argumente specificate ca parametri urmate de argumentele citite de la intrare. Liniile goale vor fi ignorate. Parametrul *-i șir* va modifica comportamentul lui xargs astfel încât comanda dată se va executa cu argumentele precizate ca parametri în care se înlocuiește șirul *șir* cu argumentele citite de la intrarea standard. **Exemplu:**

```
echo "1 2 3" | xargs mkdir -v
```

va lansa de trei ori comanda mkdir, cu parametrii -v și câte una din cifrele 1, 2 și 3.

Comanda sed

Comanda **sed** derivă din editorul **ed** care este orientat pe linie (spre deosebire de chiar cele mai simple editoare actuale care sunt orientate pe ecran). Spre deosebire de acesta din urmă, sed nu este interactiv, ci aplică tuturor sau unui grup de linii din fișierul prelucrat (care poate fi intrarea standard dacă nu se specifică un nume de fișier ca parametru) o anumită comandă. Sintaxa (simplificată) este:

```
sed [-n] [-e script] [-f script-file] [script-if-no-other-script]
[file...]
```

Optiunile *-e* (după care urmează o comandă) și *-f* (dupa care urmează un nume de fișier care conține câte o comandă pe fiecare rând) adaugă comenzi în lista celor ce vor fi aplicate asupra liniilor de text. Dacă nici -e și nici -f nu sunt găsite, primul parametru care nu este opțiune (nu începe cu cratima) este considerat comandă. Dacă după ce s-au prelucrat parametrii așa cum am descris mai rămân și alți parametri în linia de comandă, aceștia se vor considera ca nume de fișiere care trebuie procesate (în care caz sed nu se mai comportă ca filtru).

Sintaxa comenzilor **sed** este destul de complexă. Descrierea completă o găsiți în pagina *texinfo* a comenzii sed (în pagina de manual descrierea nu este foarte clară) pe care o puteți citi cu comanda **info sed**. În general o comandă **sed** este compusă dintr-o adresă sau interval de adrese de linii, urmat de un caracter ce reprezintă o acțiune de efectuat, urmat eventual de un șir de caractere a cărui semnificație depinde de acțiune.

Intervalele de adrese se specifică sub forma a doua adrese despărțite prin virgulă. O adresă poate fi un număr și atunci reprezintă numărul liniei, sau o expresie regulată încadrată de caractere **'/'**, caz în care reprezintă prima linie (începând cu linia curentă) ce corespunde (se potrivește) expresiei regulate. O adresă mai poate fi și caracterul **$**, care semnifică "ultima linie a ultimului fișier de intrare".

Acțiunile corespunzătoare comenzilor nu se vor aplica, în cazul în care avem adrese, decât liniilor corespunzătoare adresei, respectiv liniilor cuprinse în intervalul dat. Cea mai des folosită acțiune (sau comanda) este cea de **substituție**. Ea are forma:

```
s/regexp/replacement/flags
```

unde *regexp* este o expresie regulată, *replacement* este un șir de caractere cu care se va înlocui acea parte a liniei prelucrate care corespunde expresiei regulate, iar *flags* este o listă de zero sau mai multe caractere dintre următoarele:

- *g* - înlocuiește toate porțiunile care se potrivesc cu *regexp*, nu doar prima astfel de porțiune.
- *p* - dacă substituția a fost făcută, tipărește linia corespunzător modificată (utilă cu argumentul *-n*).
- *numărul N* - înlocuieste doar a N-a potrivire a expresiei regulate.

De **exemplu:**

```
sed 's/abc/def/g' <fisier.intrare >fisier.iesire
```

va prelua informațiile din fișierul *fisier.intrare,* va înlocui fiecare apariție a secvenței de caractere "abc", cu "def", iar la final va scrie rezultatele în fișierul *fisier.iesire*.

Atenție! Forwardslash-ul **'/'** are o semnificație specială, prin urmare dacă trebuie introdus în construcția lui sed avem 2 alternative: îl precedăm cu un backslash sau recurgem la o a doua formă a lui sed pentru subtituții. De **exemplu**, dacă dorim să filtrăm după */usr/local* pe o listă de path-uri și să înlocuim *prima* apariție cu */bin* putem folosi:

```
sed 's/\/usr\/local/\/bin/' <fisier.intrare >fisier.iesire
```

sau apelăm la a doua formă, având ca și separator '**|**':

```
sed 's|/usr/local|/bin|' <fisier.intrare >fisier.iesire
```

Simbolul **'&'** ne permite să *referim* porțiunea de text ce s-a potrivit cu expresia regulată din câmpul 2. De **exemplu**, dacă dorim să găsim un prenume pe o linie și să îi adăugam un ! la final, putem să folosim:

```
sed 's/[A-Z][a-z]*/&!/' <fisier.intrare >fisier.iesire
```

Atenție! Pentru a putea folosi operatorii de regex din setul *extins*, este nevoie să specificăm *opțiunea -r* imediat după cuvântul cheie *sed*.

Dacă dorim să ținem minte mai multe porțiuni de text ce se potrivesc cu anumite tipare, putem include tiparele între șiși în câmpul 2, și să le referim în câmpul 3 folosind **\1**, **\2**, **\3**, etc. în ordinea în care apar. De **exemplu**, dacă dorim să inversăm 2 cuvinte, putem folosi:

```
echo "lorem ipsum" | sed 's/[a-z]*[a-z]* [a-z]*[a-z]*/\2 \1/'
```

Sau dacă dorim să păstrăm ultimul cuvânt dintr-o propoziție, putem folosi:

```
echo "Lorem ipsum dolor sit amet." | sed 's/[A-Za-z ]*
[a-z]*[a-z]*\./\1/'
```

LAB 4

Organizarea unităților de stocare fixe (HDD, SSD)

Fiecare sistem de operare are un mod propriu de organizare și exploatare a informației stocate pe suporturile de memorare fizice. Principiile, regulile și structurile care realizează acest lucru compun *sistemul de fișiere* caracteristic sistemului de operare respectiv.

In general, din punctul de vedere al utilizatorului, sistemele de fisiere prezinta o organizare bazata pe conceptele de *fisier* si *director (catalog)*. Fisierele sunt entitati care incapsuleaza informatia de un anumit tip, iar directoarele grupeaza in interiorul lor fisiere si alte directoare. Orice fisier sau director poate fi identificat prin numele sau, indicat in mod absolut, ca nume de cale sau relativ, fata de directorul curent.

In cazul discurilor fixe (*hard-disk*-uri) informatia se memoreaza folosind proprietatile magnetice ale acestora. In cazul unitatilor fixe de tip SSD (Solid State Drive) informatiile se stocheaza folosind diferite tehnologii de memorie FLASH.

Hard-disk-ul contine in interior mai multe platane ce pot memora informatie, iar discheta este formata dintr-un singur disc flexibil (cu ambele fete magnetizate). O fata a unui disc este impartita in *piste*, care sunt cercuri concentrice in care poate fi memorata informatia. Pistele sunt impartite la randul lor in *sectoare*, un sector memorand o cantitate fixa de informatie (de obicei 512 octeti). Citirea si scrierea informatiei pe un disc se face la nivel de *blocuri de date*. Un bloc (*cluster)* poate fi format dintr-un singur sector (cum se intampla la dischete) sau din mai multe (ca la hard-disk-uri). In cazul unitatilor SSD notiunile de piste si sectoare dispar dar in continuare citirea si scrierea informatiei se face la nivel de bloc.

Un hard-disk poate fi impartit de utilizator in *partitii*, fiecare partitie comportandu-se, la nivel utilizator, ca un disc de sine statator. Partitia memoreaza sistemul de fisiere, de unde rezulta ca pe acelasi disc fizic pot fi intalnite mai multe sisteme de fisiere. Pentru calculatoarele personale obisnuite (PC), informatiile referitoare la partitii se memoreaza la inceputul discului, in asa-numita *tabela de partitii*. Aceasta contine 4 intrari in care memoreaza pozitiile, dimensiunile si tipurile partitiilor de pe disc. Partitiile memorate tabela de la inceputul discului se numesc *partitii primare*, care pot fi, evident, cel mult 4 la numar. Este posibil, insa, ca in interiorul oricarei partitii primare sa se creeze cate o noua tabela de partitii, referind partitii care fizic se afla in interiorul partitiei curente si care se numesc *partitii extinse*.

Sistemul de fisiere UNIX

Spatiul fiecarei partitii Unix contine urmatoarele zone:

Boot block (block incarcare)	Superblock	Zona noduri index	Swapping	Continut

- **Blocul de incarcare** (*boot block*)contine programele care realizeaza incarcarea partii rezidente a sistemului de operare Unix.
- **Superblocul** contine informatii generale despre sistemul de fisiere de pe disc: inceputul zonelor urmatoare, inceputul zonelor libere de pe disc.
- **Zona de noduri index** are o dimensiune fixata la creareea sistemului de fisiere si contine cate o intrare pentru fiecare fisier ce poate fi creat pe acest suport
- **Zona pentru swapping** (daca exista) este rezervata pentru pastrarea imaginilor proceselor atunci cand sunt eliminate temporar din memorie pentru a face loc altor procese. De obicei, insa, pentru zona de swap se folosesc partitii distincte.
- Ultima zona contine blocurile care memoreaza fisierele propriu-zise.

In UNIX directoarele sunt implementate tot prin intermediul fisierelor cu proprietatea ca fisierele de tip director nu pot fi scrise de utilizator. Astfel, *intrarile dintr-un director*, au o structura foarte simpla, continand doar doua compuri: numele fisierului si numarul nodului index asociat fisierului

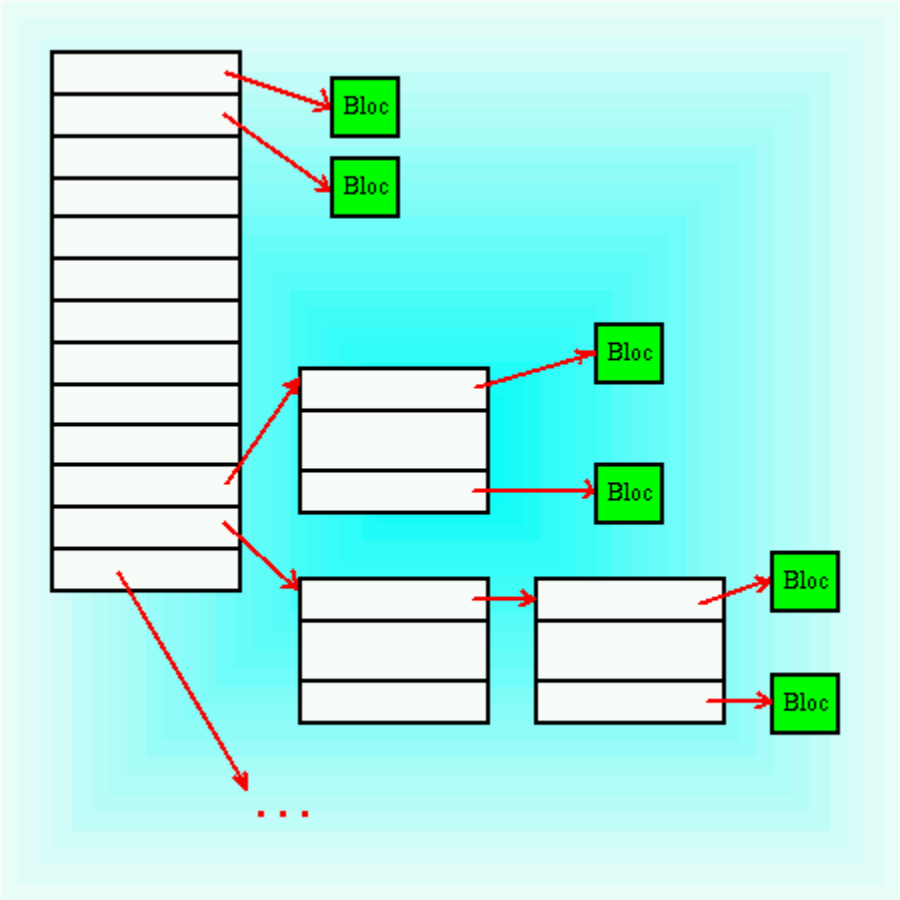
Un nod index (*i-node*) contine informatiile esentiale despre fisierul caruia ii corespunde. Exista cate un singur nod index pentru fiecare fisier. Este posibil sa intalnim mai multe intrari in director indicand acelasi nod index (sistemul de fisiere din Unix accepta crearea de legaturi multiple).

Informatia din nodul index cuprinde:

- **identificatorul utilizatorului**: *uid (user-id.)*. Identifica proprietarul fisierului
- **identificatorul de grup al utilizatorului**
- **drepturile de acces la fisier.** Drepturile sunt de trei tipuri (***r****-read,* ***w****-write,* ***x****-execute)* si sunt grupate pe trei categorii:
  - *user* - drepturile proprietarului fisierului
  - *group* - drepturile utilizatorilor din grupul proprietarului
  - *others* - drepturile tuturor celorlalti utilizatori
- **timpul ultimului acces la fisier**
- **timpul ultimei actualizari a fisierului**
- **timpul ultimului acces pentru actualizarea nodului index**

- **codul fisierului (tipul fisierului).** Fisierele pot fi: fisiere obisnuite (-), directoare (d), periferice (c) etc.
- **lungimea fisierului (in octeti)**
- **contorul de legaturi al fisierului.** Reprezinta numarul de legaturi existente spre acest nod index. Este utilizat la operatia de stergere a nodului index.
- **lista de blocuri** care contin fisierul

Lista de blocuri de pe disc care contin fisierul se realizeaza printr-un tablou cu 13 intrari. Primele 10 intrari contin direct adresele de bloc (cluster) pentru primele 10 blocuri ale fisierului. A unsprezecea intrare din aceasta lista este adresa unui bloc, rezervat fisierului, al carui continut este, insa, interpretat ca lista de adrese de blocuri. Se spune ca aceste blocuri sunt adresate prin *indirectare simpla*. Intrarea a 12-a contine un bloc al carui continut consta in adrese de blocuri, care *acestea* contin adrese de blocuri de date (*indirectare dubla)*. In mod analog, intrarea cu numarul 13 determina o *indirectare tripla*.



Sistemul de fisiere din UNIX permite crearea asa-numitelor legaturi la fisiere. O asemenea legatura (*link*) este vazuta de catre utilizator ca un fisier cu un nume propriu, dar care in realitate refera un alt fisier de pe disc. Orice operatie care se executa asupra fisierului legatura

(mai putin stergerea) isi va avea efectul de fapt asupra fisierului indicat de legatura. Daca este solicitata stergerea, efectul depinde de tipul legaturii respective.

Legaturile sunt de doua tipuri:

- **fizice (*hard links*)**
- **simbolice (*symbolic links*)**

Legaturile din prima categorie se realizeaza prin introducerea de intrari in director care pointeaza spre acelasi nod index, si anume cel al fisierului indicat. Cand spre fisier este stearsa si ultima intrare in director care il indica, fisierul in sine va fi sters si el. Legaturile de acest tip au dezavantajul ca nu pot indica nume de directoare si nici fisiere din alte partitii decat cea pe care se afla.

Legaturile simbolice sunt de fapt fisiere distincte, marcate cu un cod special, care au ca si continut numele complet al fisierului indicat. Stergerea lor nu afecteaza fisierul. Pot referi directoare, precum si fisiere si directoare din alta partitie sau alt disc, dar au dezavantajul ca pentru ele (fiind fisiere) trebuie creat un nod index separat si, in plus, ocupa spatiu pe disc prin continutul lor.

Crearea legaturilor spre fisiere sau directoare se face cu ajutorul comenzii **ln**.

- **ln *fisier_indicat nume_legatura*** - creeaza o legatura "fizica"
- **ln -s *fisier_indicat nume_legatura*** - creeaza o legatura simbolica

Pe langa legaturi, in Unix exista si alte fisiere speciale. Tipul acestora poate fi observat citind primul caracter afisat de comanda **ls -l.** Astfel, pot exista urmatoarele tipuri de fisiere:

1. Fisiere obisnuite
2. Directoare. Dupa cum am vazut, sunt fisiere care, avand un format special, grupeaza fisiere
3. Fisiere speciale care corespund unor dispozitive orientate pe caractere
4. Fisiere speciale care corespund unor dispozitive orientate pe blocuri
5. Fisiere FIFO
6. Legaturi simbolice

Fisierele speciale evidentiate la punctele 3 si 4 reprezinta metoda prin care sistemul Unix abstractizeaza dispozitivele de intrare-iesire si alte dipozitive din sistemul de calcul. Toate aceste fisiere se gasesc in directorul **/dev**.

Spre exemplu, fiecarei unitati de disc ii corespunde cate un fisier in directorul **/dev**. In Linux, primul hard-disk pe interfata SATA (sau SCSI) din system ii corespunde fisierul specia **/dev/sda**, iar primei partitii din acest disc ii corespunde fisierul **/dev/sda1**. A doua

partitie de pe primul disc are ca si corespondent fisierul **/dev/sda2**, al doilea hard-disk SATA (sau SCSI) are ca si corespondent fisierul **/dev/sdb**, ...etc.

Fisierele speciale care indica unitati de disc sau partitii sunt folosite in operatia numita *montare* a sistemelor de fisiere. Sistemul de operare Unix permite montarea intr-un director a unui sistem de fisiere aflat pe un disc sau o partitie. Aceasta inseamna ca, dupa montare, in directorul respectiv se va afla intreaga structura de fisiere si directoare de pe sistemul de fisiere respectiv. Mecanismul este deosebit de puternic, deoarece ofera posibilitatea de a avea o structura de directoare unitara, care grupeaza fisiere de pe mai multe partitii sau discuri. Daca se adauga si sistemul de fisiere **NFS *(Network File System)***, aceasta structura de directoare va putea contine si sisteme de fisiere montate de la distanta (de pe alta masina)

Montarea unui sistem de fisiere se face cu comanda **mount**. Data fara nici un parametru, ea afiseaza sistemele de fisiere montate in momentul respectiv in sistem. O alta forma a ei este urmatoarea:

mount *fisier-special director*

care monteaza un disc sau o partitie intr-un director dat; sau

mount -t *tip fisier-special director*

cu acelasi efect, doar ca se specifica in clar tipul sistemului de fisiere care se monteaza. Diferitele variante de Unix cunosc mai multe sau mai putine tipuri de sisteme de fisiere. Spre exemplu, Linux cunoaste, printre altele, urmatoarele:

- ext2, ext3, **ext4** - sistemele caracteristice Linux
- msdos - sistemul de fisiere DOS FAT16 sau FAT32
- iso9660 - sistem de fisiere pentru unitatile CD-ROM
- smbfs - sistem de fisiere montat la distanta prin protocol SAMBA
- sshfs - sistem de fisiere montat la distanta prin protocolul SSH

Pentru a afla ce sisteme de fisiere cunoaste un sistem de operare Linux instalat se poate consulta continutul fisierului /proc/filesystem.

Orice sistem de fisiere montat de pe o unitate de disc care permite inlaturarea discului respectiv trebuie *demontat* inainte de a scoate discul. De asemenea, inainte de inchiderea sau repornirea calculatorului, trebuie de-montate si sistemele de fisiere de pe discurile fixe (in Linux, aceasta din urma operatie se efectueaza automat la restartarea sistemului prin apasarea simultana a tastelor Ctrl+Alt+Del). De-montarea fisierelor se face cu comanda

umount *fisier-special*

sau

umount *director*

(unde *director* este numele directorului in care a fost montat sistemul de fisiere).

Apeluri sistem si functii de biblioteca pentru lucrul cu fisiere

Orice sistem de operare pune la dispozitia programatorilor o serie de *servicii* prin intermediul carora acestora li se ofera acces la resursele hardware si software gestionate de sistem: lucrul cu tastatura, cu discurile, cu dispozitivul de afisare, gestionarea fisierelor si directoarelor etc. Aceste servicii se numesc *apeluri sistem*. De cele mai multe ori, operatiile pe care ele le pot face asupra resurselor gestionate sunt operatii simple, cu destul de putine facilitati. De aceea, frecvent, se pot intalni in bibliotecile specifice limbajelor de programare colectii de functii mai complicate care gestioneaza resursele respective, dar oferind programatorului niveluri suplimentare de abstractizare a operatiilor efectuate, precum si importante facilitati in plus. Acestea sunt *functiile de biblioteca*. Trebuie subliniat faptul ca functiile de biblioteca cu ajutorul carora se poate gestiona o anumita resursa sunt implementate folosind chiar functiile sistem corespunzatoare, specifice sistemului de operare.

Apelurile sistem, dar si functiile de biblioteca, de obicei returneaza anumite valori prin care se comunica apelantului daca acel apel s-a executat corect sau nu. Valoarea efectiv returnata se poate afla in pagina de manual a functiei respective, fie ca este vorba de apel sistem sau functie de biblioteca. In cazul in care un apel sistem sau functie de biblioteca s-a terminat cu eroare este posibil ca toate operatiunile urmatoare efectuate de programator sa fie compromise. Din aceste motive, este absolut necear sa se testeze.

De asemenea, in majoritatea situatilor aceste apeluri, in caz de eroare, vor seta variabila globala **errno** (declara in errno.h), mostenita de fiecare program. Aceasta varibila errno, in urma aparitiei unei erori este setata cu o valoare menita sa explice situatia eronata aparuta (mai multe informatii despre errno si valorile posibile se pot gasi in pagina de manual pentru errno, sectiunea 3 - man 3 errno). Aceste valori, de multe ori, sunt greu de a fi folosite in varianta lor initiala. Pentru a facilita obtinerea erorilor aparute se pot folosi urmatoarele functii:

```
void perror(const char *s);
```

Functia perror afiseaza la iesirea standard de eroare un mesaj menit sa descrie ultima eroare aparuta. Practic aceasta functie afiseaza un mesaj ce "transcrie" valoarea numerica a lui *errno*.

String-ul asteptat ca si parametru este afisat inainte mesajului de eroare. Se poate furniza si string-ul nul "". Pagina de manual a functiei se afla in sectiunea 3 *(man 3 perror)*.

```
char *strerror(int errnum);
```

Functia strerror este asemanatoare cu functia perror dar aceasta nu afiseaza ci returneaza string-ul ce l-ar afisa functia perror. Practic aceasta functie returneaza un string ce descrie valoarea erorii descrise de obicei de errno dar codul numeric trebuie dat ca si parametru. Pagina de manual a functiei se afla in sectiunea 3 (man 3 strerror). Un exemplu de utilizare ar putea fi urmatorul:

```
#include                                                    <string.h>
#include                                                     <errno.h>
.......
char *error_string = strerror(errno);
```

Intr-un mod foare simplificat, o metoda de testare a unui apel sistem (sau functie de biblioteca) ar putea fi urmatoarea:

```
#include                                                      <stdio.h>

int                                                          main(void)
{
   .....
   int           return_value          =           system_call(...);
   if                (return_value               <                0)
   {
      perror("mesaj");
      exit(-1);
   }
   .....
   return                                                            0;
}
```

Apelurile sistem care vor fi discutate aici sunt caracteristice sistemului de operare UNIX. Principalele operatii urmarite vor fi: deschiderea si inchiderea fisierului, citirea din fisier, scrierea in fisier operatiuni de modificare a pozitiei curente in fisier, operatii de obtinere de informatii suplimentare despre fisiere (structura unui i-node).

Pentru a putea actiona asupra unui fisier, este nevoie inainte de toate de o metoda de a identifica in mod unic fisierul. In cazul functiilor ce vor fi discutate, identificarea fisierului se face printr-un asa-numit *descriptor de fisier (file descriptor)*. Acesta este un numar intreg care este asociat fisierului in momentul deschiderii acestuia. Este important de mentionat ca un descriptor de fisier asignat la deschiderea fisierului identifica in mod unic acel fisier in cadrul programului respectiv. Valoarea intreaga a acestui descriptor nu are nici o semnificatie pentru un alt program. Descriptorul este asignat de catre nucleu pentru programul apelant.

Apeluri sistem pentru deschiderea si inchiderea fisierelor

Deschiderea unui fisier este operatia prin care fisierul este pregatit pentru a putea fi prelucrat in continuare. Prin aceasta operatiune, programul apelant primeste de la nucleu descriptorul unic pentru acel fisier. Aceasta operatie se realizeaza prin intermediul functiei **open**:

```
int open(const char *pathname, int oflag, [, mode_t mode]);
```

Pagina de manual a functiei se afla in sectiunea 2 (apeluri sistem): *man 2 open*

Functia returneaza -1 in caz de eroare si seteaza valoarea corespunzatoare a variabilei *errno*.In caz contrar, functia returneaza descriptorul de fisier asociat fisierului deschis. Acest descriptor va fi apoi folosit pe tot parcursul programului pentru a identifica in mod unic fisierul.

Argumentele functiei:

- **pathname** - calea catre fisier (poate fi aboluta sau relativa la fisierul executabil al programului)
- **oflag** - optiunile de deschidere a fisierului. Acest camp este in realitate un sir de biti (bitfield) in care fiecare bit sau grupa de biti are o anumita semnificatie. Pentru fiecare astfel de semnificatie exista definite in fisierul *header* C **fcntl.h** cate o constanta. Constantele se pot combina folosind operatorul '|' (*sau logic pe biti*) din C, pentru a seta mai multi biti (deci a alege mai multe optiuni) in parametrul intreg **oflag**. Iata cateva din aceste constante:
  - O_RDONLY - deschidere numai pentru citire
  - O_WRONLY - deschidere numai pentru scriere
  - O_RDWR - deschidere pentru citire si scriere
  - O_APPEND - deschidere pentru adaugare la sfarsitul fisierului
  - O_CREAT - crearea fisierului, daca el nu exista deja; daca e folosita cu aceasta optiune, functia **open** trebuie sa primeasca si parametrul *mode*.
  - O_EXCL - creare "exclusiva" a fisierului: daca s-a folosit O_CREAT si fisierul exista deja, functia **open** va returna eroare
  - O_TRUNC - daca fisierul exista, continutul lui este sters

- **mode** - acest argument este tot un camp de biti (bitfield) si se foloseste si are sens doar in cazul in care fisierul este creat si specifica drepturile de acces asociate fisierului. Acestea se obtin prin combinarea unor constante folosind operatorul *sau* ('|'), la fel ca si la optiunea precedenta. Constantele pot fi:
  - S_IRUSR - drept de citire pentru proprietarul fisierului (*user*)
  - S_IWUSR - drept de scriere pentru proprietarul fisierului (*user*)
  - S_IXUSR - drept de executie pentru proprietarul fisierului (*user*)
  - S_IRGRP - drept de citire pentru grupul proprietar al fisierului
  - S_IWGRP - drept de scriere pentru grupul proprietar al fisierului
  - S_IXGRP - drept de executie pentru grupul proprietar al fisierului
  - S_IROTH - drept de citire pentru ceilalti utilizatori
  - S_IWOTH - drept de scriere pentru ceilalti utilizatori
  - S_IROTH - drept de executie pentru ceilalti utilizatori

Pentru crearea fisierelor poate fi folosita si functia *creat* care este de fapt un caz particular de apel a functiei *open*:

```
creat (const char *pathname, mode_t mode);
```

echivalent cu:

```
open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode)
```

Dupa utilizarea fisierului, acesta trebuie *inchis*, folosind functia *close*:

```
int close (int filedes);
```

Functia returneaza -1 in caz de eroare si seteaza variabila errno corespunzator. In caz de succes functia returneaza 0. Pagina de manual pentru aceasta functie se afla tot in sectiunea 2: *man 2 close*. Argumentul filedes specifica descriptorul catre fisierul ce se doreste a fi inchis.

Este absolut necear ca un fisier odata deschis sa fie inchis de catre programator. In cazul in care fisierul ramane deschis, la terminarea executiei programului, sistemul de operare va inchide fortat toate fisierele deschise si este posibil sa apare corupere si pierdere de date.

Un aspect important de mentionat este ca in momentul in care un fisier se inchide, sistemul de operare executa si operatiunea de *flush*. Prin aceasta operatiune sistemul de operare scrie

efectiv pe mediul de stocarea datele scrise anterior de utilizator/programator. In lipsa apelului de *close* este posibil ca operatiunea de flush sa nu mai fie executata si astfel datele scrie prin eventualele apeluri precedente sa nu ajunga si in mediul fizic de stocare. In caz concet, in momentul in care programatorul/utilizatorul scrie ceva intr-un fisier, sistemul de operare nu scrie efectiv in fisier acele date in acel moment ci acele date sunt trecute printr-o memorie tampon (prin buffer-e). In momentul in care buffer-ele interne sunt pline doar atunci sistemul de operare initiaza o scriere efectia in fisier. Motivele din spatele acestor operatiuni tin de eficientizarea scrierilor. Prin apelul sistem close, sistemul de operare va initia scrierea tuturor datelor fizic in fisier indiferend de starea buffer-elor.

Apeluri sistem pentru citirea si scrierea fisierelor

Citirea datelor dintr-un fisier deschis se face cu apelul sistem *read*:

```
ssize_t read(int fd, void *buff, size_t nbytes);
```

Functia citeste un numar de maxim *nbytes* octeti de la pozitia curenta in fisierul al carui descriptor este *fd* si ii pune in zona de memorie indicata de pointerul *buff*. Functia returneaza numarul de octeti cititi. Nu este neaparat ca functia sa citeasca din fisier numarul de octeti specificati prin *nbytes*. Este posibil ca in fisier sa fie de citit la un moment dat mai putin de *nbytes* octeti (de exemplu daca s-a ajuns spre sfarsitul fisierului), astfel ca functia *read* va pune in buffer doar atatia octeti cati poate citi. In cazul in care s-a ajuns la capatul fisierului functia *read* va returna 0. In caz de eroare functia *read* va returna valoarea -1 si se va seta corespunzator valoarea lui *errno*. Dupa fiecare citire functia incrementeaza pozitia curenta a indicatorului fisierului cu numarul de octeti cititi.

Este foarte important de mentionat faptul ca aceasta functie nu face nici o prelucrare a datelor citite ci efectiv citeste in mod binar un numar specificati de octeti. O greseala frecventa este aceea de a se considera ca daca fisierul este text si se citeste un numar de octeti, acestia vor forma un string. Aceasta presupunere este total gresita din motivul ca in string este un sir de caracter terminat cu octetul 0x00 ('\0'). Functia read nu adauga acest octet la sfarsit deoarece ea doar citeste din fisier.

De asemenea, functia este absolut gresit a se apela o singura data functia read pentru intregul fisier (mai ales daca este vorba de un fisier de mari dimensiuni). De cele mai multe ori se recomanda un apel ciclic al acestei functii prin care se va citi un numar limitat de octeti pana cand se ajunge la sfarsitul fisierului.

Pagina de manual a apelului *read* se gaseste in sectiunea 2 (*man 2 read*).

Scrierea datelor in fisier se face cu apelul sistem *write*:

```
ssize_t write(int fd, void *buff, size_t nbytes);
```

Functia scrie in fisier primii *nbytes* octeti, la pozitia curenta, din bufferul indicat de *buff* in fisierul a carui descriptor este *fd*. Functia returneaza numarul de octeti ce au fost scrisi in fisier. In caz de eroare functia returneaza -1 si seteaza corespunzator valoarea lui errno. Dupa fiecare scriere, functia incrementeaza pozitia curenta a indicatorului fisierului cu numarul de octeti scrisi.

Pagina de manual a apelului write se gaseste in sectiunea 2 (*man 2 write*).

Apeluri sistem pentru modificarea indicatorului pozitiei curente

Operatiile de scriere si citire in si din fisier se fac la o anumita pozitie in fisier, considerata pozitia curenta. Fiecare operatie de citire, de exemplu, va actualiza indicatorul pozitiei curente incrementand-o cu numarul de octeti cititi. Indicatorul pozitiei curente poate fi setat si in mod explicit, cu ajutorul functiei **lseek**:

```
off_t lseek(int fd, off_t offset, int pos);
```

Functia pozitioneaza indicatorul la deplasamentul *offset* in fisier, astfel:

- daca parametrul pos ia valoarea **SEEK_SET**, pozitionarea se face relativ la inceputul fisierului
- daca parametrul pos ia valoarea **SEEK_CUR**, pozitionarea se face relativ la pozitia curenta
- daca parametrul pos ia valoarea **SEEK_END**, pozitionarea se face relativ la sfarsitul fisierului

Parametrul *offset* poate lua si valori negative si reprezinta deplasamentul, calculat in octeti.
In caz de eroare functia va retruna -1 si va seta corespunzator valoarea lui *errno*.

Apeluri sistem pentru aflarea atributelor fisierelor

Atributele unui fisier reprezinta niste informatii aditionale ce se pot afla despre acestea, informatii ce reprezinta de fapt structura i-node-ului. Pentru a se obtine atributele unui fisier se poate folosi apelul sistem *stat*:

```
int stat(const char *file_name, struct stat *buf);
```

Acest apel sistem primeste ca prim parametru o cale abosoluta sau relativa la programul executabil a unui fisier de pe disc. Al doilea parametru, ce poate fi considerat un parametru de iesire, reprezinta un pointer spre o zona de memorie ce contine o variabila de tip *struct stat*. Aceasta zona de memorie trebuie sa existe (sa fi fost alocata **static** sau dinamic) deorece functia stat va scrie in aceasta locatie. Apelul stat retuneaza 0 daca apelul a decurs cu succes sau -1 in caz de eroare cu setarea corespunzatoare a variabilei *errno*. Pagina de manual a acestei functii se afla in sectiunea 2 (*man 2 stat*).

Structura *struct stat* are urmatoarea forma:

```
    struct stat {
               dev_t     st_dev;         /* ID of device containing
file */
               ino_t     st_ino;         /* inode number */
               mode_t    st_mode;        /* file type and mode */
               nlink_t   st_nlink;       /* number of hard links */
               uid_t     st_uid;         /* user ID of owner */
               gid_t     st_gid;         /* group ID of owner */
               dev_t     st_rdev;        /* device ID (if special
file) */
               off_t     st_size;        /* total size, in bytes */
               blksize_t st_blksize;     /* blocksize for
filesystem I/O */
               blkcnt_t  st_blocks;      /* number of 512B blocks
allocated */

               /* Since Linux 2.6, the kernel supports nanosecond
                  precision for the following timestamp fields.
                  For the details before Linux 2.6, see NOTES. */

               struct timespec st_atim;  /* time of last access */
               struct timespec st_mtim;  /* time of last
```

```
modification */
               struct timespec st_ctim;  /* time of last status
change */

           #define st_atime st_atim.tv_sec      /* Backward
compatibility */
           #define st_mtime st_mtim.tv_sec
           #define st_ctime st_ctim.tv_sec
           };
```

Elementele din aceasta structura au urmatoarea semnificatie (se vor descrie doar cele mai utilizate):

- st_mode - reprezinta un bitfield ce contine tipul fisierului si drepturi de acces. Pentru a extrage informatiile despre tipul fisierului din acest bitfield se rcomanda a se folosi unele macro-uri ce sunt puse la dispozitie: S_ISREG(m), S_ISDIR(m), S_ISLNK(m) unde m reprezinta de fapt st_mode (pentru mai multe informatii despre aceste macro-uri se recomanda consultarea paginii de manual). Pentru extragerea infromatiilor despre drepturile de acces din acest bitfield se pot folosi definitiile ce contin anumite masti de biti precum S_IRWXU, S_IRUSR, S_IROTH, ... etc (pentru mai multe informatii despre aceste macro-uri se recomanda consultarea paginii de manual: *man 7 inode*)
- st_uid - numar intreg ce reprezinta identificatorul utilizatorului owner al acestui fisier (analog cu coloana 3 din iesirea comenzii ls -l)
- st_gid - numar intreg ce reprezinta identificatorul grupului din care face parte fisierul (analog cu coloana 4 din iesirea comenzii ls -l)
- st_size - numar intreg ce reprezinta dimensiunea in bytes a fisierului

Este important de mentionat faptul ca faca aceasta functie este apelata pentru o legatura simbolica informatiile vor fi oferite pentru fisierul referentiat de legatura simbolica (target) si nu pentru insusi fisierul legatura simbolica. Astfel, functia *lstat* este asemanatoare cu *stat*, cu diferenta ca, daca este aplicata unei legaturi simbolice, informatiile returnate se vor referi la legatura, si nu la fisierul indicat. De asemenea functia *lstat* funcioneaza identic cu functia *stat* daca se aplica oricarui tip de fisier ce nu este legatura simbolica.

```
int lstat(const char *file_name, struct stat *buf);
```

Pagina de manual a acestei functii se afla in sectiunea 2 (*man 2 stat*) si in aceeasi pagina de manual cu functia *stat*.

```
int fstat(int filedes, struct stat *buf);
```

Functia *fstat* are acelasi efect, cu deosebirea ca ea primeste ca argument un descriptor de fisier, si nu numele acestuia, deci se poate aplica doar fisierelor in prealabil deschise. Pagina de manual a acestei functii se afla in sectiunea 2 (*man 2 stat*) si in aceeasi pagina de manual cu functia *stat*.

Apeluri sistem pentru gestionarea legaturilor

Asemanator comenzii ln prezentata in lucrarile de laborator anterioare, exista apeluri sistem ce pot fi folosite din programe scrise in limbajul C ce pot fi folosite pentru a crea sau a sterge legaturi fizice (hard links) sau legaturi simbolice (symbolic link - symlinks).

```
int link(const char *oldpath, const char *newpath); // creeaza
legaturi fixe spre fisiere
int symlink(const char *oldpath, const char *newpath); //  creeaza
legaturi simbolice spre fisiere sau directoare
int unlink(const char *pathname); // sterge o intrare in director
(legatura, fisier sau director)
```

Paginile de manual pentru aceste apeluri se gasesc toate in sectiunea 2, acestea fiind apeluri sistem.

LAB 5

Functii pentru gestionarea directoarelor

Directoarele sunt, in esenta, fisiere cu un format special. Sistemul de operare UNIX pune la dispozitia programatorului un set de apeluri sistem care ofera posibilitatea de a citi continutul directoarelor, accesand astfel informatii despre fisierele si directoarele continute. Folosind aceste apeluri sistem, biblioteca standard C defineste un set de functii care se conformeaza standardului POSIX si care ofera aceleasi facilitati. Fiind recomandabil sa se utilizeze, in locul apelarii directe a functiilor sistem, functiile de biblioteca , in continuare vor fi prezentate numai acestea din urma. Principalele operatiuni ce se pot realiza asupra directoarelor ar fi: deschiderea directoarelor, parcurgerea fisierelor si directoarelor dintr-un director si inchiderea directoarelor.

Pentru a putea fi pracurs, un director este necesar ca in prealabil sa fie deschis. In urma deschiderii unui director, similar ca si la fisiere, se va primi ca rezultat o variabila de un anumit tip care va referentia in mod unic acel director pe tot parcursul programului pana la inchiderea directorului. Functia responsabila de deschiderea unui director este:

```
DIR *opendir(const char *name);
```

Functia opendir primeste ca si parametru o cale absoluta sau relativa la programul executabil a unui director. In urma procesului de deschidere a directorului, functia va returna o referinta catre o structura dedicata de tip DIR. Variabila referinta catre aceasta structura va identifica in mod unic acel director. In caz de eroare opendir va returna pointer-ul NULL si se va seta valoarea corespunzatoarea a variabilei *errno*. Pagina de manual pentru functia opendir se va gasi in sectiunea 3 (man 3 opendir).

Pentru a se putea parcurge director deschis folosind functia opendir se vor folosi succesiv apeluri ale functiei *readdir*.

```
struct dirent *readdir(DIR *dirp);
```

Functia readdir primeste ca parametru un pointer (o referinta) catre un tip de date DIR obtinut prin apelul *opendir*. Functia readdir va returna un pointer catre o structura de tip *struct dirent*:

```
struct                          dirent                          {
   ino_t                  d_ino;             /*  Inode  number  */
   off_t             d_off;          /* Not an offset; see below */
   unsigned  short  d_reclen;        /*  Length  of  this  record  */
   unsigned char  d_type;      /* Type of file; not supported  by all
filesystem                          types                          */
   char               d_name[256];  /*  Null-terminated  filename  */
};
```

Functia *readdir* va returna, la fiecare apel succesiv cate un pointer la o structura de tip *struct dirent* ce va referentia intrarea curenta din director. La fiecare apel va fi furnizata urmatoarea intrare in director. In momentul in care s-au epuizat toate intrarile in directorul respectiv, functia *readdir* va returna NULL. Asadar este necesar ca aceasta functie sa fie apelata ciclic pana cand se ajunge la ultima inregistrare din director.

Cele mai importante campuri din structura *struct dirent* ar fi urmatoarele:

- d_name - reprezinta un string (terminat cu caracterul 0x00) ce contine numele intrarii curente din director. Atentie ! acest camp contine doar numele intrarii, si nu calea fie ea relativa sau absoluta
- d_type - reperzinta o valoare ce identifica tipul intrarii: DT_DIR - director, DT_LNK - legatura simbolica, DT_REG - fisier obisnuit (regular file). Atentie! Conform documentatiei, acest camp nu este suportat de toate tipurile de sisteme de fisiere. Nu se recomanda folosirea acestui camp pentru identificarea tipului de fisier ci mai degraba se recomanda folosirea apelului sistem *stat*.

Pagina de manual pentru functia readdir se va gasi in sectiunea 3 (man 3 readdir).

Ca si in cazul fisierelor un director odata deschis va trebui sa fie inchis si sa nu ramana in sarcina sistemului de operare inchiderea fortata a acestuia la temrinarea programului. Functia pentru inchiderea unui director este closedir:

```
int closedir(DIR *dir);
```

Functia *closedir* trebuie sa primeasca ca si parametru un pointer catre un tip de date DIR primit in urma deschiderii directorului cu *opendir*. Functia closedir returneaza 0 daca s-a terminat cu

succes. In caz de eroare functia returneaza -1 si seteaza corespunzator variabila *errno*. Pagina de manual pentru functia *closedir* se afla in sectiunea 3 (man 3 closedir);

Metodologie pentru parcurgerea directoarelor

Pentru a explica metodologia de parcurgere a unui director se considera urmatorul caz de test:

```
dir
|--                                                                  .
|--                                                                 ..
|--                                                               a.c
|--                                                               b.c
|--                                                              dir1
       |--                                                           .
       |--                                                          ..
|                                                  |--              a.c
|                                          `--             lab_signal.c
`--                                                              dir2
       |--                                                           .
       |-- ..
```

In acest caz de test se face din nou observatie asupra faptului ca orice director va avea 2 subdirectoare obligatorii: subdirectorul "." ce reprezinta o legatura catre directorul curent si subdirectorul ".." ce reprezinta o legatura catre directorul parinte

Pentru parcurgerea arborelui descris mai sus se procedeaza astfel:

- se deschide directorul cu functia opendir
- se citeste pe rand cate o intrare din director, apeland functia readdir, Fiecare apel al acestei functii va returna un pointer la o structura struct dirent in care se vor gasi informatii despre intrarea in director citita. Intrarile vor fi parcurse una dupa alta pana cand se ajunge la ultima inregistrare. In momentul in care nu mai exista inregistrari in directorul citit, functia readdir va returna NULL. Dupa cum a fost aratat mai sus, singura informatie care poate fi extrasa (conform POSIX) din structura **dirent** este numele intrarii in director. Toate celelalte informatii despre intrarea citita se pot afla apeland in continuare functiile *stat, fstat* sau *lstat*.
- pentru a se extrage informatii sau pentru a se prelucra intrarile dintr-un subdirector folosind functiile din familia stat este necesar sa se creeze calea relativa (sau absoluta) catre acea inregistrare. Daca programul principal este pe acelasi nivel cu directorul "dir" atunci "dir" va fi argumentul pentru functia opendir. La fiecare apel a functiei readdir se vor furniza intrarile "a.c", "b.c", dir1, etc.... Asadar pentru a accesa una dintre

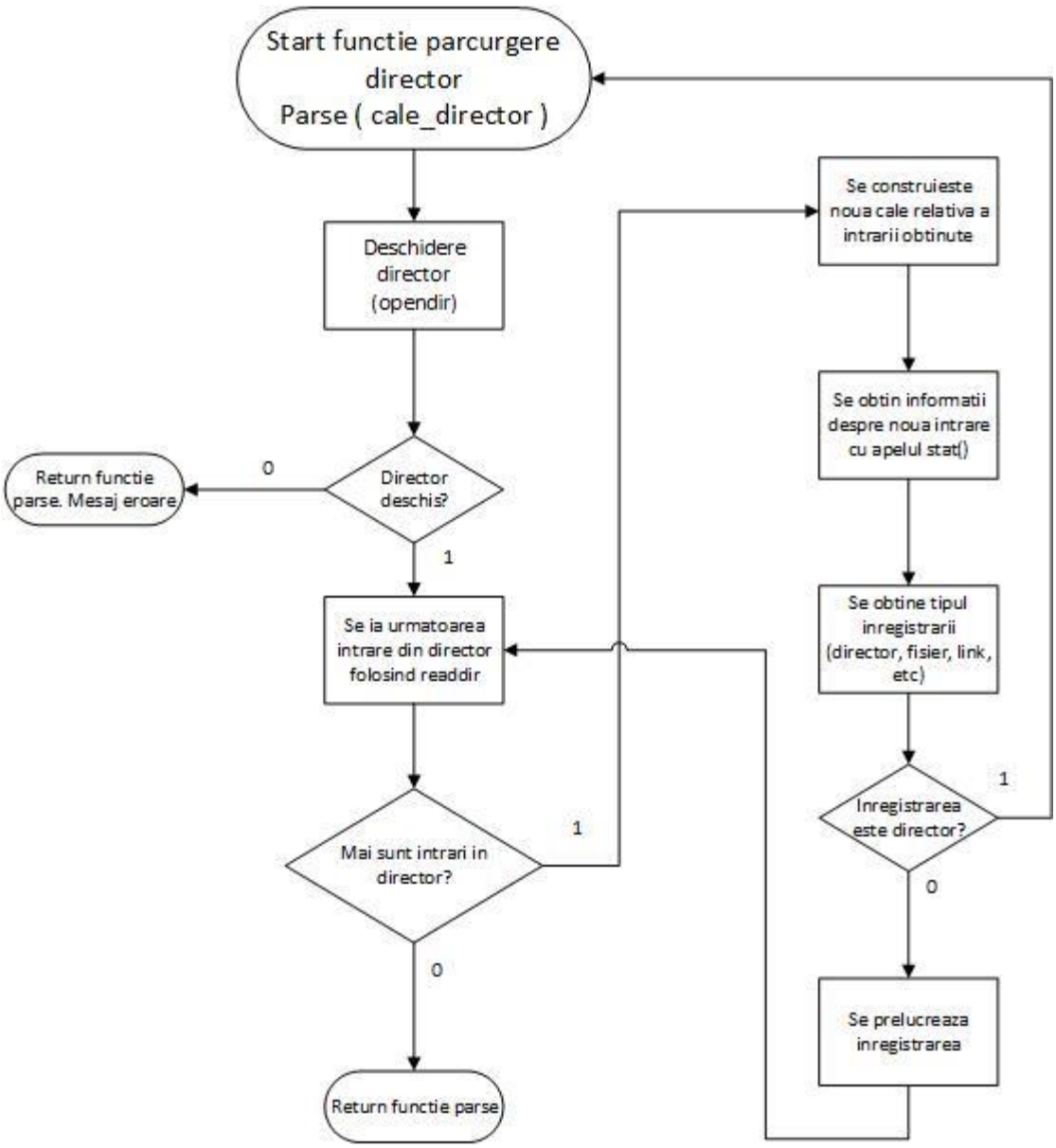
aceste inregistrari este necesar sa se creeze calea relativa prin adaugarea directorului parinte si sa se obtina astfel "dir/a.c", "dir/b.c", "dir/dir1". In caz contrar, daca nu se creaza calea completa relativa functii precum cele din familia stat vor returna eroare deoarece spre exemplu nu exista intrarea "a.c" ci doar "dir/a.c".
- dupa ce a fost creata calea relativa completa se pot apela functiile din familia stat pentru a se afla tipul inregistrarii. In cazul in care inregistrarea este director se poate apela recursiv intregul algoritm.
- in final, directorul este inchis, folosind functia closedir

Pentru a se crea calea relativa se recomanda folosirea functiei sprintf:

```
int sprintf(char *str, const char *format, ...);
```

Functia sprintf se afla in pagina de manual din sectiunea 3 (man 3 sprintf sau man sprintf). Aceasta functie are aceeasi functionalitate ca si functia printf cu exceptia ca iesirea functiei nu este afisarea la STDOUT ci intr-un string furnizat ca si prim argument. Acest prim argument este necesar sa fie alocat in prealabil (static sau dinamic) - se recomanda alocare statica.

O propunere de implementare a unui algoritm de parcurgere recursiva a unui director este prezentata in urmatoarea diagrama:


Start functie parcurgere director
Parse (cale_director)
Deschidere director (opendir)
Director deschis?
0
Return functie parse. Mesaj eroare
1
Se ia urmatoarea intrare din director folosind readdir
Mai sunt intrari in director?
1
0
Return functie parse
Se construieste noua cale relativa a intrarii obtinute
Se obtin informatii despre noua intrare cu apelul stat()
Se obtine tipul inregistrarii (director, fisier, link, etc)
Inregistrarea este director?
1
0
Se prelucreaza inregistrarea

LAB 6

Concepte de baza

Orice sistem de calcul modern este capabil sa execute mai multe programe in acelasi timp. Cu toate acestea, in cele mai multe cazuri, unitatea centrala de prelucrare (CPU) nu poate executa la un moment dat decat un singur program. De aceea, sarcina de a rula mai multe programe in acelasi timp revine sistemului de operare, care trebuie sa introduca un model prin intermediul caruia executia programelor, privita din perspectiva utilizatorului, sa se desfasoare in paralel. Se realizeaza, de fapt, un *pseudoparalelism*, prin care procesorul este alocat pe rand programelor care trebuie rulate, cate o cuanta de timp pentru fiecare, astfel incat din exterior ele par ca ruleaza efectiv in acelasi timp.

Cel mai raspandit model care introduce paralelismul in executia programelor este modelul bazat pe **procese**. Acest model este cel adoptat de sistemul de operare Unix si va face obiectul acestei lucrari.

Un *proces* este un program secvential aflat *in executie*, impreuna cu zona sa de date, stiva si numaratorul de instructiuni (program counter). Trebuie facuta inca de la inceput distinctia dintre *proces* si *program*. Un program este, in fond, un sir de instructiuni care trebuie executate de catre calculator, in vreme ce un proces este o abstractizare a programului, specifica sistemelor de operare. Se poate spune ca un proces executa un program si ca sistemul de operare lucreaza cu procese, iar nu cu programe. Procesul include in plus fata de program informatiile de stare legate de executia programului respectiv (stiva, valorile registrilor CPU etc.). De asemenea, este important de subliniat faptul ca un program (ca aplicatie software) poate fi format din *mai multe procese* care sa ruleze sau nu in paralel.

Unitatea elementara de executie intr-un proces este thread-ul. Asadar, dupa cum s-a spus anterior, fiecarui thread sistemul de operare ii acorda o cuanta de timp procesor. Componenta din nucleul sistemelor de operare Linux (si nu numai) care acorda aceste cuante de timp thread-urilor se numeste planificator (scheduler). In Linux, fiecare thread are asociata o politica de planificare (scheduling policy) si o prioritate statica de planificare (static sched policy) iar in functie de acesti parametrii, planificatorul decide, pe baza unor algoritmi, cand si cat i se acorda cuanta de timp procesor fiecarui thread.

Politicile de planificare (sau algoritmii de planificare), in functie de constrangerile de timp se pot imparti in:

- hard real-time: aplicatiile din aceasta categorie au conditii stricte de timp si se considera ca daca aplicatia si-a depasit timpul alocat (deadline-ul) atunci consecintele asupra sistemului sunt catastrofale. Exemplu de aplicatii hard real-time: verificarea starii pedalei de frana la masina, verificarea starilor comenzilor la avioane, verificarea parametrilor intr-un reactor nuclear. In oricare din aceste exemple daca o aplicatie si-a depasit deadline-ul si astfel nu si-a indeplinit scopul, urmarile pot duce la dezastre.

- firm real-time: la aplicatiile din aceasta categorie depasirea unui deadline poate fi tolerata dar depasirile repetate ale deadline-ului pot duce la nefunctionarea totala a sistemului. Exemplu: achizitia de semnal audio - daca se pierde un esantion audio foarte rar este aproape insesizabil dar pierderea repetata de esantioane audio poate duce la pierderea totala a informatiei.
- soft real-time: la aplicatiile din aceasta categorie se considera ca depasirea unui deadline poate afecta doar performanta sistemului fara sa-i afecteze functionalitatea de baza. Exemplu: aplicatie de streaming video
- non real-time: la aceasta categorie de aplicatii nu exista nici un fel de constrangeri de timp

In Linux exista mai multe politici de planificare disponibile:

- SCHED_OTHER (SCHED_NORMAL) - aceasta reprezinta politica implicita de planificare a thread-urilor in Linux. Thread-urile vor avea in acest caz toate prioritate 0. Thread-urile vor fi alese pentru a fi executate dintr-o lista, avand fiecare drept egal de executie. Exista totusi un mecanism pentru a favoriza sau defavoriza anumite thread-uri, reprezentat printr-o prioritate dinamica numita *nice-level*. Aceasta prioritate dinamica se aplica doar pentru prioritatea statica 0.
- SCHED_BATCH - similar ca si SCHED_NORMAL, dar in plus planificatorul va considera in algoritmul de planificare aspectul ca un astfel de thread poate folosi procesorul in mod intensiv
- SCHED_IDLE - politica de planificare folosita pentru thread-uri cu o prioritate extrem de mica
- SCHED_RR, SCHED_FIFO - aceste 2 politici de planificare sunt considerate soft real-time si se folosesc cu prioritate statica mai mare decat 0.
- SCHED_DEADLINE - aceasta politica de planificare a fost introdusa in nucleul de Linux incepand cu versiunea 3.14 in urma unui proiect desfasurat la Universitatea Scuola Sant'Anna din Pisa si inclus in nucleul de Linux in 2014. Este o politica de scheduling hard real-time bazata pe algoritmul de planificare EDF (Earliest Deadline First). Prin aceasta politica de planificare sistemele Linux devin astfel utilizabile si in domenii cu constrangeri stranse de timp

Sistemul de planificare a thread-urilor in Linux este preemptiv adica un thread de o anumita prioritate statica poate fi oricand intrerupt pentru a lansa in executie un thread de o prioritate mai mare.

Pentru a se schimba politica de planificare a unui thread exista apeluri sistem dedicate. Trebuie mentionat doar ca un thread lansat in executie (un program, un proces... etc) fara a se specifica nimic legat de politica de planificare este lansat de catre nucleu folosind politica implicita SCHED_OTHER.

Orice proces este executat secvential, iar mai multe procese pot sa ruleze in paralel. De cele mai multe ori, executia in paralel se realizeaza alocand pe rand procesorul cate unui proces. Desi la un moment dat se executa un singur proces, in decurs de o secunda, de exemplu, pot fi executate portiuni din mai multe procese. Din aceasta schema rezulta ca un proces se poate gasi, la un moment dat, in una din urmatoarele stari:

- in executie
- pregatit pentru executie
- blocat

Procesul se gaseste *in executie* atunci cand procesorul ii executa instructiunile. *Pregatit de executie* este un proces care, desi ar fi gata sa isi continue executia, este lasat in asteptare din cauza ca un alt proces este in executie la momentul respectiv. De asemenea, un proces poate fi *blocat* din doua motive: el isi suspenda executia in mod voit sau procesul efectueaza o operatie in afara procesorului, mare consumatoare de timp (cum e cazul operatiilor de intrare-iesire - acestea sunt mai lente si intre timp procesorul ar putea executa parti din alte procese).

Pentru mai multe informatii despre sistemul de planificare din Linux se recomanda a se citi pagina de manual sched(7): *man 7 sched*.

Gestionarea proceselor in UNIX

Apelul sistem fork()

Din perspectiva programatorului, sistemul de operare UNIX pune la dispozitie un mecanism elegant si simplu pentru crearea si utilizarea proceselor.

Orice proces trebuie creat de catre un alt proces. Procesul creator este numit *proces parinte*, iar procesul creat *proces fiu*. Exista o singura exceptie de la aceasta regula, si anume procesul *init*, care este procesul initial, creat la pornirea sistemului de operare si care este responsabil pentru crearea urmatoarelor procese. Interpretorul de comenzi, de exemplu, ruleaza si el in interiorul unui proces.

Fiecare proces are un identificator numeric, numit *identificator de proces (process identifier - PID)*. Acest identificator este folosit atunci cand se face referire la procesul respectiv, din interiorul programelor sau prin intermediul interpretorului de comenzi.

Un proces trebuie creat folosind apelul sistem fork()

```
pid_t fork()
```

Prin aceasta functie sistem, procesul apelant (parintele) creeaza un nou proces (fiul) care va fi o *copie fidela* a parintelui (se copiaza din parinte catre fiu continutul zonelor de memorie - stiva, heap, zona statica - , program counter-ul, descriptorii). Noul proces va avea propria lui zona de date, propria lui stiva, propriul lui cod executabil, toate fiind copiate de la parinte in cele mai mici detalii. Rezulta ca variabilele fiului vor avea **valorile** variabilelor parintelui in momentul apelului functiei *fork( )*, iar executia fiului va continua cu instructiunile care urmeaza imediat acestui apel, codul fiului fiind identic cu cel al parintelui. Cu toate acestea, in sistem vor exista din acest moment doua procese independente, (desi identice), cu zone de date si stiva distincte. Orice modificare facuta, prin urmare, asupra unei variabile din procesul fiu, va ramane invizibila procesului parinte si invers. Practic intre cele doua procese nu va mai exista nicio legatura, doar cea de parinte-fiu.

Procesul fiu va mosteni de la parinte toti descriptorii de fisier deschisi de catre acesta, asa ca orice prelucrari ulterioare in fisiere vor fi efectuate in punctul in care le-a lasat parintele.

Deoarece codul parintelui si codul fiului sunt identice si pentru ca aceste procese vor rula in continuare in paralel, trebuie facuta clar distinctia, in interiorul programului, intre actiunile ce vor fi executate de fiu si cele ale parintelui. Cu alte cuvinte, este nevoie de o metoda care sa indice care este portiunea de cod a parintelui si care a fiului.
Acest lucru se poate face simplu, folosind valoarea returnata de functia *fork( )*. Ea returneaza:

- -1, daca operatia nu s-a putut efectua (eroare) si se seteaza corespunzator valoarea variabilei globale *errno*.
- 0, in codul fiului
- *pid*, in codul parintelui, unde *pid* este identificatorul de proces al fiului nou-creat.

Prin urmare, o posibila schema de apelare a functiei *fork( )* ar fi:

```
if( ( pid=fork() ) < 0)
  {
    perror("Eroare");
    exit(1);
  }
if(pid==0)
  {
    /* codul fiului - doar codul fiului poate ajunge aici (doar in
fiu fork() returneaza 0)*/
    ...
    exit(0); // apel necesar pentru a se opri codul fiului astfel
incat acesta sa nu execute si codul parintelui
  }
/* codul parintelui */
```

Principalele cazuri de eroare ce pot fi semnalate de apelul sistem fork, ceea ce reprezinta faptul ca nu se mai pot crea procese noi, pot fi:

- memorie insuficienta
- s-au epuizat toate valorile de PID disponibile (numarul maxim de valori pentru PID la un moment dat este specificat in fisierul special /proc/sys/kernel/pid_max)
- s-a ajuns la limita maxima de thread-uri ce se pot crea (numarul maxim de thread-uri din sistem se poate afla consultand fisierul special /proc/sys/kernel/threads-max)

Pentru mai multe informatii se recomanda consultarea paginii de manual. Pagina de manual a apelului sistem fork() se afla in sectiunea 2 (man 2 fork).

Apeluri sistem wait() si waitpid()

In momentul in care procesul isi termina executia (in general prin apelul sistem exit()) el nu este complet sters de catre nucleu ci este trecut intr-o stare de proces terminat sau *zombie*. In aceasta stare nucleul nu sterge procesul din tabela de procese ci asteapta ca procesul parinte sa ii citeasca starea cu care si-a terminat executia. Cat timp procesul este in starea zombie, nucleul nu face nici o modificare asupra lui, astfel ca toate componentele lui de memorie raman alocate (inclusiv PID-ul rezervat in momentul in care a fost creat). Un proces parinte poate sa obtina starea cu care s-a terminat un proces fiu folosind functiile din familia *wait*. Abia dupa un apel al acestor functii nucleul sterge complet procesul fiu ramas in starea zombie:

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

Functia *wait( )* este folosita pentru asteptarea terminarii fiului si preluarea valorii returnate de acesta. Parametrul *status* este folosit pentru evaluarea valorii returnate, folosind cateva macro-uri definite special (vezi paginile de manual corespunzatoare functiilor *wait( )* si *waitpid( )*). Functia *waitpid( )* este asemanatoare cu *wait( )*, dar asteapta terminarea unui anumit proces dat, in vreme ce *wait( )* asteapta terminarea oricarui fiu al procesului curent. Este obligatoriu ca starea proceselor sa fie preluata dupa terminarea acestora, astfel ca functiile din aceasta categorie nu sunt optionale. Conform paginii de manual, functiile wait si waitpid returneaza valoarea PID-ului procesului fiu in starea zombie (ce si-a terminat executia) si -1 in caz de eroare. Este important de mentionat ca printre erori se gaseste si situatia in care nu mai exista procese fiu ale procesului parinte apelant care sa se termine, in acest caz functiile *wait* si *waitpid* returnand tot -1. Pentru a se face distrinctie intre erori se recomanda si inspectarea valorii *errno*. Pagina de manual pentru functiile *wait* si *waitpid* se gasesc in sectiunea 2 (man 2 wait).

Functii din familia exec()

Functia *fork( )* creeaza un proces identic cu procesul parinte. Pentru a crea un nou proces care sa ruleze un program diferit de cel al parintelui, aceasta functie se va folosi impreuna cu unul din apelurile sistem de tipul *exec( )*: *execl( ), execlp( ), execv( ), execvp( ), execle( ), execve( )*.

Toate aceste functii primesc ca parametru un nume de fisier care reprezinta un program executabil si realizeaza lansarea in executie a programului. Programul va fi lansat atfel incat se vor suprascrie codul, datele si stiva procesului care apeleaza exec( ). Imediat dupa acest apel codul programului initial (a procesului apelant) nu va mai exista in memorie. Procesul va ramane, insa, identificat prin acelasi numar (PID) si va mosteni toate eventualele redirectari facute in prealabil asupra descriptorilor de fisiere (de exemplu intrarea si iesirea standard). De asemenea, el va pastra relatia parinte-fiu cu procesul care a apelat fork( ). Toate functiile din familia *exec()* in principiu fac acelasi lucru dar pun la dispozitia programatorului mai multe forme de apel. In final toate aceste functii ajung in a face apelul sistem *execve()*. Putem deci considera ca functiile din familia exec() sunt doar niste functii de tip wrapper peste apelul sistem *execve()*:

Functiile din familia exec() sunt functii de biblioteca si se gasesc astfel in pagina de manual din sectiunea 3 (ex. man 3 exec):

```
int execl(const char *path, const char *arg, ... /* (char  *) NULL
*/);
int execlp(const char *file, const char *arg, ... /* (char  *) NULL
*/);
int execle(const char *path, const char *arg, ... /*, (char *)
NULL, char * const envp[] */);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const
envp[]);
```

Apelul sistem execve() care sta la baza functiilor de biblioteca din familia exec se afla in pagina de manual din sectiunea 2 (man 2 execve):

```
int execve(const char *filename, char *const argv[], char *const
envp[]);
```

Exemplu de utilizare a unei functii din familia exec():

```
int main(void)
{
       int pid;
       int status;
       if ((pid = fork()) < 0)
       {
              ... tratare erori  ...
       }
       if (pid == 0)
       {
              /* cod fiu */
              execlp ("ls","ls","-l",NULL);
              // daca codul a ajuns la aceasta linie -> execlp a
esuat
              ... (tratare erori etc) ...
       }
       /* cod parinte */
```

```
        ... operatii normale de continuare/finalizare a procesului
principal ...
}
```

Functia system()

O varianta extrem de simplificata a celor expuse in paragraful anterior, despre functiile din familia exec(), este folosirea functiei system(). Aceasta functie lanseaza in executie un program de pe disc, folosind in acest scop un apel *fork( )*, urmat de *exec( )*, impreuna cu *waitpid( )* in parinte:

```
int system(const char *cmd);
```

Pagina de manual pentru functia system, fiind o functie de biblioteca, se afla in sectiunea 3 (*man 3 system*).

Apelul sistem vfork()

O varianta mai rapida dar si incompleta a apelului sistem fork() este apelul sistem vfork(). Acesta creeaza un nou proces, la fel ca *fork( )*, dar nu copiaza in intregime spatiul de adrese al parintelui in fiu. De obicei este folosit in conjunctie cu functiile *exec( )*, si are avantajul ca nu se mai consuma timpul necesar operatiilor de copiere care oricum ar fi inutile daca imediat dupa aceea se apeleaza *exec( )* (oricum, procesul fiu va fi supascris cu programul luat de pe disc).

Alte apeluri sistem pentru gestiunea proceselor

```
pid_t getpid();  // returneaza PID-ul procesului curent
pid_t getppid(); // returneaza PID-ul parintelui procesului curent
uid_t getuid();  // returneaza identificatorul utilizatorului care
a lansat procesul curent
gid_t getgid();  // returneaza identificatorul grupului
utilizatorului care a lansat procesul curent
```

Pentru mai multe detalii despre aceste apeluri sistem se recomanda cautarea lor in pagina de manual aferenta (fiind apeluri sistem se vor gasi in sectiunea 2).

Gestionarea proceselor din linia de comanda

Sistemul de operare UNIX are cateva comenzi foarte utile care se refera la procese:

- comanda **ps** - afiseaza informatii despre procesele care ruleaza in mod curent pe sistem. Comanda este foarte utila pentru a afla PID-ul unui proces ce ruleaza deja

- comanda **kill** - trimite un semnal catre un proces identificat prin PID: kill -*semnal pid.* Semnalul poate fi dat atat ca si numar cat si ca si denumire
- comanda **kilall** - trimite un semnal catre un proces identificat prin nume: kill -*semnal nume*.

LAB 7

Concepte de baza

Semnalele sunt o modalitate de exprimare a evenimentelor care apar asincron in sistem. Un proces oarecare poate atat sa genereze, cat si sa primeasca semnale. In cazul in care un proces primeste un semnal, el poate alege sa reactioneze la semnalul respectiv intr-unul din urmatoarele trei moduri:

- Sa le capteze, executand o actiune oarecare, prin intermediul unei **functii de tratare a semnalului** (**signal handler**)
- Sa le ignore
- Sa execute actiunea implicita la primirea unui semnal, care poate fi, dupa caz, terminarea procesului sau ignorarea semnalului respectiv.

Semnalele pot fi de mai multe tipuri, care corespund in general unor actiuni specifice. Fiecare semnal are asociat un numar, iar acestor numere le corespund unele constante simbolice definite in bibliotecile sistemului de operare. Standardul POSIX.1 defineste cateva semnale care trebuie sa existe in orice sistem UNIX.Cele mai importante si mai folosite semnale ar fi urmatoarele:

Semnal	**ID semnal**	**Descriere**
SIGHUP	1	Hangup - terminalul folosit de proces a fost inchis
SIGINT	2	Interrupt - intrerupere de la tastatura (in general prin CTRL+C)
SIGQUIT	3	Quit - cerere de iesire din program de la tastatura (CTRL+\)
SIGILL	4	Illegal Instruction - se genereaza atunci cand procesul a executat o instructiune al carei *opcode* nu are corespondent in setul de instructiuni sau pentru care nu exista privilegii suficiente

Semnal	ID semnal	Descriere
SIGABRT	6	Abort - semnal de terminare anormala a procesului generat de functia *abort*.
SIGFPE	8	Floating Point Exception - semnal generat atunci cand in executia procesului a aparut o eroare la o operatie in virtual flotanta (ex. impartire la 0)
SIGKILL	9	Kill - Semnalul are ca efect distrugerea imediata a procesului. Acest semnal nu poate fi ignorat.
SIGUSR1	10	User defined 1 - Semnal fara semnificatie lasat pentru a putea fi folosit de catre utilizator
SIGUSR2	12	User defined 2 - Semnal fara semnificatie lasat pentru a putea fi folosit de catre utilizator
SIGSEGV	11	Segmentation fault - Semnalul apare atunci cand procesul a facut un acces ilegal la memorie
SIGPIPE	13	Broken pipe - se genereaza atunci cand s-a incercat scrierea intr-un *pipe* care are descriptorul de la capatul de citire inchis
SIGALRM	14	Timer alarm - semnal general in urma expirarii timer-ului setat de apelul *alarm*.
SIGTERM	15	Terminate - specifica o cerere de terminare a programului. Utilizatorul poate implementa cum doreste acest semnal (poate fi si ignorat)
SIGCONT	18	Continue - are ca efect continuarea unui proces suspendat prin SIGSTOP

Semnal	ID semnal	Descriere
SIGSTOP	19	Stop - Are ca rezultat suspendarea executiei procesului pana cand aceasta va fi reluata prin primirea unui semnal SIGCONT
SIGCHLD	17	Child terminated - Semnalul este primit de procesul parinte atunci cand un proces fiu si-a terminat executia

Este important de analizat diferenta dintre semnalele SIGKILL si SIGTERM. Semnalul SIGKILL opreste executie procesului destinar si il distruge fortat. Procesul nu poate ignora acest semnal si nici nu poate sa ia vreo masura impotriva acestui comportament. In acest caz procesul se inchide fortat si este posibil ca unele operatiuni prevazute pentru o inchidere normala sa nu se execute.

Semnalul SIGTERM este un semnal care poate fi ignorat de catre un proces si poate fi dat orice comportament pentru tratarea lui. In general, acest semnal a fost gandit pentru a fi folosit in scopul inchiderii normale a unui proces. In momentul in care se doreste ca un proces sa isi termine executia, i se trimite semnalul SIGTERM. La receptia acestui semnal procesul ar trebui sa isi incheie executia intr-un mod corect si normal (sa isi inchida descriptorii utilizati, sa elibereze memoria folosita, etc). Acest comportament nu este obligatoriu dar este recomandat.

In general, in sistemul de operare Linux, cand se doreste ca un serviciu sau prorgram sa isi termine executia, sistemul ii trimite intai semnalul SIGTERM. Daca procesul respectiv nu isi incheie executia intr-un timp dat, sistemul este posibil sa ii trimita un semnal SIGKILL pentru a forta inchiderea acestuia.

Gestionarea semnalelor

Apelul sistem sigaction()

Apelul sistem sigaction() are menirea de a defini comportarea procesului la primirea unui semnal. Acest apel primeste ca parametri numarul semnalului (signum), si doua structuri de tip *struct sigaction* * (act si oldact). Executia sa va avea ca efect instalarea noii actiuni pentru semnalul specificat din act (daca acest parametru este nenul), si salvarea actiunii curente in oldact (la fel, daca parametrul este nenul).

```
int sigaction(int signum, const struct sigaction *act, struct
sigaction *oldact);
```

Structura de tip *struct sigaction* este definita in felul urmator:

```
struct sigaction {
       void     (*sa_handler)(int);
       void     (*sa_sigaction)(int, siginfo_t *, void *);
       sigset_t   sa_mask;
       int        sa_flags;
       void     (*sa_restorer)(void);
};
```

Parametrul *sa_handler* reprezinta noua rutina de tratare a semnalului specificat. Alternativ, daca in sa_flags este setat indicatorul **SA_SIGINFO** (prezentat mai jos), se poate defini o rutina care primeste trei parametri in loc de unul singur, si in acest caz ea se specifica prin parametrul sa_sigaction. Pentru mai multe detalii legate de folosirea acestei din urma modalitati, consultati pagina de manual *sigaction(2)*. In unele cazuri, acesti doi parametri sunt prinsi intr-un **union**, si deci se recomanda sa se specifice doar unul din ei.

Parametrul sa_mask va specifica setul de semnale care vor fi blocate in timpul executiei rutinei de tratare a semnalului dat. Acest parametru este de tipul sigset_t, care este de fapt o masca de biti, cu cate un bit pentru fiecare semnal definit in sistem. Operatiile asupra acestui tip de masca se fac folosind functiile din familia sigsetops(3). Sintaxa si functionarea acestor apeluri este foarte simpla, consultati pagina de manual pentru a vedea exact care sunt parametrii lor si modul de functionare.

Parametrul *sa_flags* va specifica un set de indicatori care afecteaza comportarea procesului de tratare a semnalelor. Acest parametru se formeaza prin efectuarea unei operatii de SAU pe biti folosind una sau mai multe din urmatoarele valori:

- **SA_NOCLDSTOP** - daca signum este **SIGCHLD**, procesul nu va primi un semnal **SIGCHLD** atunci cand procesul fiu este suspendat (de exemplu cu **SIGSTOP**), ci numai cand acesta isi termina executia;
- **SA_ONESHOT** sau **SA_RESETHAND** - va avea ca efect resetarea rutinei de tratare a semnalului la **SIG_DFL** dupa prima rulare a rutinei, asemanator cu comportamentul implementarii originale a apelului **`signal()`**;
- **SA_ONSTACK** - executia rutinei de tratare va avea loc folosind alta stiva;
- **SA_RESTART** - ofera compatibilitate cu comportamentul semnalelor in sistemele din familia 4.3BSD;

- **SA_NOMASK** sau **SA_NODEFER** - semnalul in discutie nu va fi inclus in mod automat in *sa_mask* (comportamentul implicit este acela de a impiedica aparitia unui semnal in timpul executiei rutinei de tratare a semnalului respectiv);
- **SA_SIGINFO** - se specifica atunci cand se doreste utilizarea lui *sa_siginfo* in loc de *sa_handler*. Pentru mai multe detalii, consultati pagina de manual **sigaction(2)**.

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

Apelul sigprocmask() este folosit pentru a modifica lista semnalelor care sunt blocate la un moment dat. Acest lucru se face in functie de valoarea parametrului how, in felul urmator:

- **SIG_BLOCK** - adauga la lista semnalelor blocate semnalele din lista set data ca parametru;
- **SIG_UNBLOCK** - sterge din lista semnalelor blocate semnalele aflate in lista set;
- **SIG_SETMASK** - face ca doar semnalele din lista *set* sa se regaseasca in lista semnalelor blocate.

Daca parametrul oldset este nenul, in el se va memora valoarea listei semnalelor blocate anterioara executiei lui sigprocmask().

```
int sigpending(sigset_t *set);
```

Apelul sistem sigpending() permite examinarea semnalelor care au aparut in timpul in care ele au fost blocate, prin returnarea acestor semnale in masca set data ca parametru.

```
int sigsuspend(const sigset_t *mask);
```

Apelul sigsuspend() inlocuieste temporar masca semnalelor blocate a procesului cu cea data in parametrul mask si suspenda procesul pana la primirea unui semnal.

Apelurile sigaction(), sigprocmask() si sigpending()() returneaza valoarea 0 in caz de succes si -1 in caz de eroare. Apelul sigsuspend() retruneaza intotdeauna -1, iar in mod normal variabila errno este setata la valoarea EINTR.

In urma executiei cu eroare a unuia din apelurile de mai sus, variabila **errno** poate sa ia una din urmatoarele valori:

- **EINVAL** - atunci cand a fost specificat un semnal invalid, adica nedefinit in implementarea curenta, sau unul din semnalele **SIGKILL** sau **SIGSTOP**;

- **EFAULT** - atunci cand una din variabilele date ca parametru indica spre o zona de memorie care nu face parte din spatiul de adrese al procesului;
- **EINTR** - atunci cand apelul a fost intrerupt.

Fiind apeluri sistem, toate aceste functii se afla in pagina de manual aferenta din sectiunea 2.

Apelul sistem kill()

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

Acest apel sistem este folosit pentru a trimite un semnal unui anumit proces sau grup de procese. In functie de valoarea parametrului pid, executia apelului va avea unul din urmatoarele efecte:

- Daca pid > 0, semnalul va fi trimis procesului care are PID-ul egal cu pid;
- Daca pid == 0, semnalul va fi trimis tuturor proceselor din acelasi grup de procese cu procesul curent;
- Daca pid == -1, semnalul va fi trimis tuturor proceselor care ruleaza in sistem (de notat faptul ca, in majoritatea implementarilor, nu se poate trimite in acest fel catre procesul **init** un semnal pentru care acesta nu are prevazuta o rutina de tratare, si de asemenea, faptul ca de obicei in acest fel procesului curent nu i se trimite semnalul respectiv);
- Daca pid < -1, se va trimite semnalul catre toate procesele din grupul de procese cu numarul -pid.

Daca valoarea lui sig este zero, nu se va trimite nici un semnal, dar apelul va executa verificarile de eroare. Acest lucru este util in cazul in care se doreste sa se stie, de exemplu, daca avem suficiente permisiuni pentru a trimite un semnal catre un proces dat.

Acest apel returneaza 0 in caz de succes si -1 in caz de eroare, setand variabila **errno** la una din urmatoarele valori in cazul executiei cu eroare:

- **EINVAL** - in cazul specificarii unui semnal invalid;
- **ESRCH** - in cazul in care procesul sau grupul de procese specificat nu exista;
- **EPERM** - in cazul in care nu se dispune de permisiuni suficiente pentru a trimite semnalul respectiv procesului specificat.

Functia raise()

```
int raise(int sig);
```

Aceasta functie este folosita pentru a trimite un semnal catre procesul curent. Executia ei este similara cu executia urmatorului apel:

```
kill(getpid(), sig);
```

Functia returneaza 0 in caz de succes, si o valoare diferita de zero in caz de eroare.

Functia abort()

```
void abort(void);
```

Aceasta functie are ca efect trimiterea catre procesul curent a unui semnal SIGABRT, care are ca efect terminarea anormala a procesului, mai putin daca semnalul este tratat de o rutina care nu se termina. Daca executia lui abort() are ca efect terminarea procesului, toate fisierele deschise in interiorul acestuia vor fi inchise.Este important de notat ca daca semnalul SIGABRT este ignorat sau blocat, executia acestei functii nu va tine cont de acest lucru si procesul va fi terminat in mod anormal.

Apelul sistem alarm()

```
unsigned int alarm(int seconds);
```

Aceasta functie are ca efect faptul ca nucleul instantiaza un timer initializat cu numarul de secunde specificat ca parametru. Dupa expirarea timpului, nucleul trimite procesului apelant un semnal SIGALRM.Daca o alarma a fost deja programata, ea este anulata in momentul executiei ultimului apel, iar daca valoarea lui *seconds* este zero, nu va fi programata o alarma noua.In urma executiei, se returneaza numarul de secunde ramase din alarma precedenta, sau 0 daca nu era programata nici o alarma.

Functiile sleep() si usleep()

```
unsigned int sleep(unsigned int seconds);
int usleep(useconds_t usec);
```

Cele 2 functii au ambele rolul de a suspenda executia procesului (sau a thread-ului) apelant pentru o perioada de timp specificata ca parametru. Functia *sleep()* suspenda executia pentru un numar de secunde iar functia *usleep()* pentru un numar de microsecunde. Procesul (sau thread-ul) nu va fi suspendat exact cat a fost specificat ci este posibil sa fie

suspendat putin mai mult in functie de gradul de incarcare a sistemului si de granularitatea timer-elor din sistem.

LAB 8

O metoda foarte des utilizata in UNIX pentru comunicarea intre procese este folosirea primitivei numita *pipe* (conducta). "Conducta" este o cale de legatura care poate fi stabilita intre doua procese inrudite (au un stramos comun sau sunt in relatia stramos-urmas). Ea are doua capete, unul prin care se pot scrie date si altul prin care datele pot fi citite, permitand o comunicare intr-o singura directie. In general, sistemul de operare permite conectarea a unuia sau mai multor procese la fiecare din capetele unui *pipe*, astfel incat, la un moment dat este posibil sa existe mai multe procese care scriu, respectiv mai multe procese care citesc din *pipe*. Se realizeaza, astfel, comunicarea unidirectionala intre procesele care scriu si procesele care citesc.

Apelul sistem pipe()

Crearea "conductelor de date", pipe-urilor, se face in UNIX folosind apelul sistem *pipe( )*:

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

Functia creeaza un *pipe*, precum si o pereche de descriptori de fisier care refera cele doua capete ale acestuia. Descriptorii sunt returnati catre programul apelant completandu-se cele doua pozitii ale tabloului *filedes* trimis ca parametru apelului sistem. Pe prima pozitie va fi memorat descriptorul care indica extremitatea prin care se pot citi date (capatul de citire), iar pe a doua pozitie va fi memorat descriptorul capatului de scriere in *pipe*.

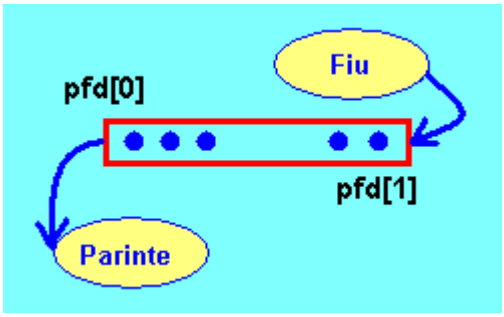
Cei doi descriptori sunt descriptori de fisier obisnuiti, asemanatori celor returnati de apelul sistem *open( )*. Mai mult, *pipe*-ul poate fi folosit in mod similar folosirii fisierelor, adica in el pot fi scrise date folosind functia *write( )* (aplicata capatului de scriere) si pot fi citite date prin functia *read( )* (aplicata capatului de citire).

Fiind implicati descriptori de fisier obisnuiti, daca un *pipe* este creat intr-un proces parinte, fiii acestuia vor mosteni cei doi descriptori (asa cum, in general, ei mostenesc orice descriptor de fisier deschis de parinte). Prin urmare, atat parintele cat si fiii vor putea scrie sau citi din *pipe*. In acest mod se justifica afirmatia facuta la inceputul acestui document prin care se spunea ca *pipe*-urile sunt folosite la comunicarea intre procese *inrudite*. Pentru ca legatura dintre procese sa se faca corect, fiecare proces trebuie sa declare daca va folosi *pipe*-ul pentru a scrie in el (transmitand informatii altor procese) sau il va folosi doar pentru citire. In acest scop, fiecare proces trebuie sa inchida capatul *pipe*-ului pe care nu il foloseste: procesele care scriu

in *pipe* vor inchide capatul de citire, iar procesele care citesc vor inchide capatul de scriere, folosind functia *close( )*.

Functia returneaza 0 daca operatia de creare s-a efectuat cu succes si -1 in caz de eroare setandu-se corspunzator valoarea variabilei errno. Pagina de manual a acestui apel sistem se va gasi in sectiunea 2.

Primitiva *pipe* se comporta in mod asemanator cu o structura de date coada: scrierea introduce elemente in coada, iar citirea le extrage pe la capatul opus.



Cantitatea de date care poate fi scrisa la un moment dat intr-un *pipe* este limitata. Numarul de octeti pe care un *pipe* ii poate pastra fara ca ei sa fie extrasi prin citire de catre un proces este dependenta de sistem (de implementare). Standardul POSIX specifica limita minima a capacitatii unui *pipe*: 512 octeti.

Atunci cind un *pipe* este "plin", operatia *write()* se va bloca pina cind un alt proces citeste suficienti octeti din *pipe*. Un proces care citeste din pipe va primi valoarea **0** ca valoare returnata de *read( )* in momentul in care toate procesele care scriau in *pipe* au inchis capatul de scriere si nu mai exista date in *pipe*. In cazul in care un proces vrea sa citeasca dintr-un pipe valid dar nu exista date disponibile in pipe (si exista capete de scriere deschise) atunci apelul read() se blocheaza pana cand un proces va scrie date in pipe.

Daca pentru un pipe sunt conectate procese doar la capatul de scriere (cele de la capatul opus au inchis toate conexiunea) operatiile *write* efectuate de procesele ramase vor returna eroare. Intern, in aceasta situatie va fi generat semnalul SIG_PIPE care va intrerupe apelul sistem *write* respectiv. Codul de eroare (setat in variabila globala *errno*) rezultat este cel corespunzator mesajului de eroare "Broken pipe".

Operatia de scriere in *pipe* este atomica doar in cazul in care numarul de octeti scrisi este mai mic decit constanta PIPE_BUF. Altfel, in sirul de octeti scrisi pot fi intercalate datele scrise de un alt proces in *pipe*.

Pentru mai multe informatii despre procesul de scriere/citire din pipe precum si despre limitarile existente se va consulta pagina de manual pipe(7): *man 7 pipe*

Un posibil scenariu pentru crearea unui sistem format din doua procese care comunica prin pipe este urmatorul:

- procesul parinte creeaza un *pipe*
- parintele apeleaza *fork( )* pentru a crea fiul
- fiul inchide unul din capete (ex: capatul de citire)
- parintele inchide celalalt capat al *pipe*-ului (cel de scriere)
- fiul scrie date in *pipe* folosind descriptorul ramas deschis (capatul de scriere)
- parintele citeste date din *pipe* prin capatul de citire.

Acest scenariu poate fi implementat astfel:

```
int main(void)
{
       int pfd[2];
       int pid;

       ...
       if(pipe(pfd)<0)
       {
         perror("Eroare la crearea pipe-ului\n");
         exit(1);
       }
       ...
       if((pid=fork())<0)
       {
         perror("Eroare la fork\n");
         exit(1);
       }
       if(pid==0)
       {
          /* procesul fiu */
         close(pfd[0]); /* inchide capatul de citire; */
                        /* procesul va scrie in pipe  */
         ...
         write(pfd[1],buff,len); /* operatie de scriere in pipe */
         ...
         close(pfd[1]); /* la sfarsit inchide si capatul utilizat */
         exit(0);
       }
       /* procesul parinte */
       close(pfd[1]); /* inchide capatul de scriere; */
                      /* procesul va citi din pipe  */
       ...
       read(pfd[0],buff,len); /* operatie de citire din pipe */
       ...
       close(pfd[0]); /* la sfarsit inchide si capatul utilizat */
```

```
        return 0;
}
```

Este important de mentionat faptul ca se vor aplica aceleasi metodologii de citire/scriere folosind apeluri sistem read/write ca si la fisiere. Si in cazul pipe-urilor se va citi si scrie ciclic pentru a efectua operatiuni corecte si complete. Practic, din punct de vedere al programatorului, exista foarte putine diferente in a efectua operatiuni de citire/scriere asupra fisierelor sau a pipe-urilor.

Functia fdopen()

```
FILE *fdopen(int fd, const char *mode);
```

Functia fdopen() asociaza un descriptor deja deschis unui flux de date (stream) de tip FILE. Dupa apelul acestei functii, se poate scrie si citi formatat dintr-un descriptor folosind functii din biblioteca stdio precum fprintf, fscanf, fread, fwrite, etc... De asemenea, chiar daca s-a facut asocierea cu stream de tip FILE, se mai pot folosi in continuare apelurile sistem de baza peste descriptorul *fd*. Aceasta functie se poate folosi pentru orice tip de descriptor ce este deja deshis, fie el descriptor de fisier obtinut in urma unui apel sistem *open()* fie un descriptor de pipe obtinut in urma unui apel *pipe()*. (practic nu exista nicio diferenta intre ei).

Dupa asocierea cu un stream de tip FILE, cand descriptorul nu mai este folosit acesta trebuie inchis. Acest lucru se poate realiza atat cu functia *fclose()* asupra stream-ului de tip FILE cat si folosind apelul sistem *close()* asupra descriptorului initial.

Redirectarea descriptorilor de fisier

Se stie ca functia *open( )* returneaza un descriptor de fisier. Acest descriptor va indica fisierul deschis cu *open( )* pana la terminarea programului sau pana la inchiderea fisierului. Sistemul de operare UNIX ofera, insa, posibilitatea ca un descriptor oarecare sa indice un alt fisier decat cel obisnuit. Operatia se numeste *redirectare* si se foloseste cel mai des in cazul descriptorilor de fisier cu valorile 0, 1 si 2 care reprezinta intrarea standard, iesirea standard si, respectiv, iesirea standard de eroare. De asemenea, este folosita si operatia de *duplicare* a descriptorilor de fisier, care determina existenta a mai mult de un descriptor pentru acelasi fisier. De fapt, redirectarea poate fi vazuta ca un caz particular de duplicare.

Duplicarea si redirectarea se fac, in functie de cerinte, folosind unul din urmatoarele apeluri sistem:

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Functia *dup( )* realizeaza duplicarea descriptorului *oldfd*, returnand noul descriptor. Aceasta inseamna ca descriptorul returnat va indica acelasi fisier ca si *oldfd*, atat noul cat si vechiul descriptor folosind in comun pointerul de pozitie in fisier, flag-urile fisierului etc. Daca pozitia in fisier e modificata prin intermediul functiei *lseek( )* folosind unul dintre descriptori, efectul va fi observat si pentru operatiile facute folosind celalalt descriptor. Descriptorul nou alocat de *dup( )* este cel mai mic descriptor liber (inchis) disponibil.

Functia *dup2( )* se comporta in mod asemanator cu *dup( )*, cu deosebirea ca poate fi indicat explicit care sa fie noul descriptor. Dupa apelul *dup2( )*, descriptorul *newfd* va indica acelasi fisier ca si *oldfd*. Daca inainte de operatie descriptorul *newfd* era deschis, fisierul indicat este mai intai inchis, dupa care se face duplicarea.

Ambele functii returneaza descriptorul nou creat (in cazul lui *dup2( )*, egal cu *newfd*) sau -1 in caz de eroare cu setarea corespunzatoare a valorii variabilei errno.

Urmatoarea secventa de cod realizeaza redirectarea iesirii standard spre un fisier deschis, cu descriptorul corespunzator *fd*:

```
...
fd=open("Fisier.txt", O_WRONLY);
...
if((newfd=dup2(fd,1))<0)
  {
       perror("Eroare la dup2\n");
       exit(1);
  }
...
printf("ABCD");
...
```

In urma redirectarii, textul "ABCD" tiparit cu *printf( )* nu va fi scris pe ecran, ci in fisierul cu numele "Fisier.txt".

Redirectarile de fisiere se pastreaza chiar si dupa apelarea unei functii de tip *exec( )* (care suprascrie procesul curent cu programul luat de pe disc). Folosind aceasta facilitate, este posibila, de exemplu, conectarea prin *pipe* a doua procese, unul din ele ruland un program executabil citit de pe disc. Secventa de cod care realizeaza acest lucru este data mai jos. Se

considera ca parintele deschide un pipe din care va citi date, iar fiul este un proces care executa un program de pe disc. Tot ce afiseaza la iesirea standard procesul luat de pe disc, va fi redirectat spre capatul de scriere al *pipe*-ului, astfel incat parintele poate citi datele produse de acesta. Aceasta facilitate este exemplificata in urmatoarea secventa de cod in care iesirea la stdout a comenzii ls va fi redirectata intr-un pipe (mai exact spre capatul de scriere a pipe-ului):

```
int                                                     main(void)
{
       int                                                 pfd[2];
       int                                                    pid;
       FILE                                               *stream;


              ...
       if(pipe(pfd)<0)
       {
              printf("Eroare      la      crearea      pipe-ului\n");
              exit(1);
       }
       ...
       if((pid=fork())<0)
       {
              printf("Eroare               la               fork\n");
              exit(1);
       }
       if(pid==0)         /*         procesul         fiu         */
       {
              close(pfd[0]);  /*  inchide  capatul  de  citire;  */
                                /* procesul va scrie in pipe  */
              ...
              dup2(pfd[1],1); /* redirecteaza iesirea standard spre
pipe                                                           */
              ...
              execlp("ls","ls","-l",NULL);   /*   procesul   va   rula
comanda                           ls                           */
              printf("Eroare               la               exec\n);
              /* Daca execlp s-a intors, inseamna ca programul nu a
putut         fi         lansat         in         executie         */
       }
       /*              procesul              parinte              */

       close(pfd[1]);   /*   inchide   capatul   de   scriere;   */
       /*     procesul     va     citi     din     pipe          */
       ...
       stream=fdopen(pfd[0],"r");
       /* deschide un stream (FILE *) pentru capatul de citire */
       fscanf(stream,"%s",string);
```

```
        /*  citire  din  pipe,  folosind  stream-ul  asociat  */
        ...
        close(pfd[0]); /* la sfarsit inchide si capatul utilizat */
          ...
          return                                                0;
}
```

LAB 9

Concepte de baza

Intr-o lucrare de laborator anterioara au fost introduse elementele de baza referitoare la *procese*. Recapituland pe scurt, un proces era vazut ca fiind format dintr-o zona de cod, o zona de date, stiva si registri ai procesorului (Program Counter si altii). In consecinta, fiecare proces aparea ca o entitate distincta, independenta de celelalte procese aflate in executie la un moment dat. De asemenea, a fost remarcat faptul ca, avand in vedere ca procesorul poate rula la un moment dat un singur proces, procesele sunt executate pe rand, dupa un anumit algoritm de planificare, astfel incat, la nivelul aplicatiilor, acestea par ca se executa in paralel.

Se desprind, astfel, doua idei importante referitoare la procese:

- ruleaza independent, avand zone de cod, stiva si date distincte
- trebuie planificate la executie, astfel incat ele sa ruleze aparent in paralel

Executia planificata a proceselor presupune ca, la momente de timp determinate de algoritmul folosit, procesorul sa fie "luat" de la procesul care tocmai se executa si sa fie "dat" unui alt proces. Aceasta comutare intre procese (*process switching*) este o operatie consumatoare de timp, deoarece trebuie "comutate" toate resursele care apartin proceselor: trebuie salvati si restaurati toti registrii procesor, trebuie (re)mapate zonele de memorie care apartin de noul proces etc.

Un concept interesant care se regaseste in toate sistemele de operare moderne este acela de *fir de executie* (*thread*) in interiorul unui proces. Firele de executie sunt uneori numite *procese usoare* (*lightweight processes*), sugerandu-se asemanarea lor cu procesele, dar si, intr-un anume sens, deosebirile dintre ele.

Un fir de executie trebuie vazut ca un flux de instructiuni care se executa *in interiorul unui proces*. Un proces poate sa fie format din mai multe asemenea fire, care se executa in paralel, *avand, insa, in comun toate resursele principale caracteristice procesului*. Prin urmare, in interiorul unui proces, firele de executie sunt entitati care ruleaza in paralel, impartind intre ele zona de date si executand portiuni distincte din acelasi cod. Deoarece zona de date este comuna, toate variabilele procesului vor fi vazute la fel de catre toate firele de executie, orice modificare facuta de catre un fir devenind vizibila pentru toate celelalte. Generalizand, un proces, asa cum era el perceput in lucrarile de laborator precedente, este de fapt un proces format dintr-un singur fir de executie.

La nivelul sistemului de operare, executia in paralel a firelor de executie este obtinuta in mod asemanator cu cea a proceselor, realizandu-se o comutare intre fire, conform unui algoritm de planificare. Spre deosebire de cazul proceselor, insa, aici comutarea poate fi facuta mult mai rapid, deoarece informatiile memorate de catre sistem pentru fiecare fir de executie sunt mult mai putine decat in cazul proceselor, datorita faptului ca firele de executie au foarte putine resurse proprii. Practic, un fir de executie poate fi vazut ca un numarator de program, o stiva si un set de registri,

toate celelalte resurse (zona de date, identificatori de fisier etc) apartinand procesului in care ruleaza si fiind exploatate in comun.

Implementarea firelor de executie in Linux

Linux implementeaza firele de executie oferind, la nivel scazut, apelul sistem *clone( )*:

```
pid_t clone(void *sp, unsigned long flags);
```

Apelul sistem *clone( )* este o interfata alternativa la functia sistem *fork( )*, ea avand ca efect crearea unui proces fiu, oferind, insa, mai multe optiuni la creare.

Daca *sp* este diferit de zero, procesul fiu va folosi sp ca indicator al stivei sale, permitandu-se astfel programatorului sa aleaga stiva noului proces.

Argumentul *flags* este un sir de biti continand diferite optiuni pentru crearea procesului fiu. Octetul inferior din *flags* contine semnalul care va fi trimis la parinte in momentul terminarii fiului nou creat. Alte optiuni care pot fi introduse in cuvantul *flags* sunt: COPYVM si COPYFD. Daca este setat COPYVM, paginile de memorie al fiului vor fi copii fidele ale paginilor de memori ale parintelui, ca la functia *fork( )*. Daca COPYVM nu este setat, fiul va imparti cu parintele paginile de memorie ale acestuia. Cand COPYFD este setat, fiul va primi descriptorii de fisier ai parintelui ca si copii distincte, iar daca nu este setat, fiul va imparti descriptorii de fisier cu parintele.

Functia returneaza PID-ul fiului in parinte si zero in fiu. Pentru mai multe detalii ale acestui apel se recomanda consultarea paginii de manual din sectiunea 2 (man 2 clone)

Prin urmare, apelul sistem *fork( )* este echivalent cu:

```
clone(0, SIGCLD | COPYVM);
```

Se observa ca functia *clone( )* ofera suficiente facilitati pentru a putea crea primitive de tip fire de executie.

Pentru un exemplu concret se recomanda consultarea fisierului clone.c (autor Linux Torvalds) pus la dispozitie pe pagina principala a cursului la sectiunea Fire de executie (in aceeasi sectiune cu acesta pagina).

Utilizarea firelor de executie folosind bilioteca LinuxThreads

In programe este indicat sa nu se foloseasca direct functia *clone( ),* in primul rand din cauza ca ea nu este portabila (fiind specifica Linux) si apoi pentru ca utilizarea ei este intrucatva greoaie.

Standardul POSIX 1003.1c, adoptat de catre IEEE ca parte a standardelor POSIX, defineste o interfata de programare pentru utilizarea firelor de executie, numita *pthread*. Interfata este implementata pe multe arhitecturi; mai mult, sistemele de operare care contineau biblioteci proprii de fire de executie (cum este SOLARIS) introduc suport pentru acest standard.

In Linux exista o biblioteca numita *LinuxThreads*, care implementeaza versiunea finala a standardului POSIX 1003.1c si utilizeaza functia *clone()* ca instrument de creare a firelor de executie. In continuare, ne vom referi la aceasta biblioteca de functii si vom trece in revista o parte din primitivele introduse de catre ea.

Aspecte de compilare privind biblioteca Linux Threads

Biblioteca LinuxThreads este disponibila automat in toate distributiile moderne de Linux (presupunand ca au fost instalate pachetele de development).

Pentru a putea utiliza fire de executie, este necesar ca programele sa fie compilate folosind comanda:

```
gcc -Wall -o <executabil> -D_REENTRANT -lpthread <fisier.c>
```

Se observa ca este necesara definirea constantei _REENTRANT (din considerente legate de executia paralela a firelor) si ca trebuie inclusa explicit biblioteca *pthread* (numele sub care se regaseste LinuxThreads, conform POSIX). Biblioteca *pthread* este un *shared object* in Linux si va fi link-editata dinamic cu programul ce o foloseste. Ea exista in sistem in general intr-un fisier de tip *shared object* cu numele *libpthread.so*.

Observatie: in Windows Subsystem for Linux, compilarea se face astfel:

```
gcc -Wall -o <executabil> -pthread <fisier.c>
```

Crearea firelor de executie

Un fir de executie se creeaza folosind functia:

```
int  pthread_create(pthread_t *thread, pthread_attr_t *attr, void
*(*start_routine)(void *), void *arg);
```

Functia creeaza un *thread* care se va executa in paralel cu *thread*-ul creator. Noul fir de executie va fi format de functia *start_routine* care trebuie definita in program avand un singur argument de tip **(void *)**. Parametrul *arg* este argumentul care va fi transmis acestei functii. Parametrul *attr* este un cuvant care specifica diferite optiuni de creare a firului de executie. In mod obisnuit, acesta este dat ca NULL, acceptand optiunile implicite. Firul de executie creat va primi un identificator care va fi returnat in variabila indicata de parametrul *thread*. Functia returneaza 0 daca crearea a avut succes si un numar diferit de zero in caz contrar ce reprezinta eroarea (cu aceleasi valori ca si cele ale variabilei *errno*). Atentie, in acest caz **nu** se seteaza corespunzator valoarea lui *errno* ci se returneaza aceasta valoare. Pentru a se obtine un text ce reprezinta eroarea se recomanda folosirea functiei *strerror()*.

Thread-ul va consta in executia functiei date ca argument, iar terminarea lui se va face ori apeland explicit functia *pthread_exit( )*, ori implicit, prin iesirea din functia *start_routine*.

Identificatorul de thread returnat de functie prin primul argument se numeste Thread ID (TID). Un proces ce are un singur fir de executia va avea PID-ul sau egal cu TID-ul thread-ului principal.

Terminarea firelor de executie

Un fir de executie se poate termina apeland:

```
void pthread_exit(void *retval);
```

De asemenea, un thread se poate termina si prin iesirea normala din functie referentiata de *start_routine* (valoare returnata de tip void*).

Valoarea *retval* este valoarea pe care *thread*-ul o returneaza la terminare. Starea returnata de firele de executie poate fi preluata de catre *oricare* din *thread*-urile aceluiasi proces (ce au acces la identificatorul thread-ului), folosind functia :

```
int pthread_join(pthread_t th, void **thread_return);
```

Aceasta intrerupe firul de executie care o apeleaza pana cand firul de executie cu identificatorul *th* se termina, moment in care starea lui va fi returnata la adresa data de parametrul *thread_return*.

Demn de observat este faptul ca *nu pentru toate firele de executie poate fi preluata starea de iesire*. De fapt, conform standardului POSIX, firele de executie se impart in doua categorii:

- joinable - ale caror stari pot fi preluate de catre celelalte fire din proces
- detached - ale caror stari nu pot fi preluate

In cazul *thread*-urilor *joinable*, in momentul terminarii acestora, resursele lor nu sunt complet dezalocate, asteptandu-se un viitor *pthread_join* pentru ele. Firele de executie *detached* se dezaloca in intregime, starea lor devenind nedisponibila pentru alte fire de executie.

Tipul unui fir de executie poate fi specificat la crearea acestuia, folosind optiunile din argumentul *attr* (implicit este *joinable*). De asemenea, un fir de executie *joinable* poate fi "detasat" mai tarziu, folosind functia *pthread_detach( )*.

Observatii:

- Un proces, imediat ce a fost creat, este format dintr-un singur fir de executie, numit fir de executie principal (initial).
- Toate firele de executie din cadrul unui proces se vor executa in paralel.
- Datorita faptului ca impart aceeasi zona de date, firele de executie ale unui proces vor folosi in comun toate variabilele globale. De aceea, se recomanda ca in programe firele de executie sa utilizeze numai variabilele locale, definite in functiile care implementeaza firul, in afara de cazurile in care se doreste partajarea explicita a unor resurse.
- Daca un proces format din mai multe fire de executie se termina, toate firele de executie ale sale se vor termina.
- Daca un fir de executie apeleaza functia *exit( )*, efectul va fi terminarea *intregului* proces, cu toate firele de executie din interior.
- Functiile *sleep()* si *usleep()* vor afecta doar thread-ul apelant.

Sincronizarea firelor de executie

Dupa cum s-a explicat anterior, toate firele de executie din cadrul unui proces au acces la aceeasi zona de memorie globala. Asadar, orice fir de executie va putea accesa oricand prin scriere sau citire orice variabila globala. TInand cont ca, din punct de vedere al programatorului, firele de executie ruleaza in paralel, accesarea datelor fara nici un mecanism de protectie si sincronizarea poate duce la hazarde de date.

Hazardele de date pot fi (si se considera 2 threaduri th1, si th2):

- read after write (RAW) - apare atunci cand thread-ul th2 incearca sa citeasca o resursa inainte ca th1 sa o scrie. In acest caz este posibil ca thread-ul th2 sa citeasca o resursa "veche"
- write after read (WAR) - apare atunci cand thread-ul th2 incearca sa scrie o resursa inainte sa fie citita de thread-ul th1)
- write after write (WAW) - apare atunci cand thread-ul th2 incearca sa scrie o resursa inainte sa fie scrisa de thread-ul th1.

In momentul aparitiei oricarui unui astfel de hazard datele pot fi corupte sau thread-urile pot citi date "prea vechi" sau "prea noi" astfel pierzand valori/ evenimente.

Unul din mecanismele puse la dispozitie de bilioteca pthread pentru sincronizarea firelor de executie este reprezentat de mutex (lacat)

Lacatele (locks sau mutexes) sunt un mecanism de sincronizare asemanator semafoarelor. Ele sunt folosite pentru excludere mutuala - de unde numele de mutex. Lacatele au o stare cu doua valori posibile (locked si unlocked) si doua operatii (lock sau acquire si unlock sau release).

La creare, un lacat este in starea "deschis". Operatia *lock* pe un lacat deschis va face ca lacatul sa treaca in starea "inchis" iar firul care inchide lacatul se spune ca il detine. Operatia *lock* efectuata asupra unui lacat inchis va bloca firul apelant.

Operatia *unlock* efectuata asupra unui lacat inchis, **apelata de firul ce detine lacatul**, va debloca unul din firele de executie blocate in asteptare in urma unui *lock* pe acel lacat. Firul deblocat va fi cel care detine lacatul in continuare. Daca nu exista fire in asteptare, lacatul va fi trecut in starea *unlocked* si nici un fir nu va mai detine lacatul.

Daca un fir de executie efectueaza operatia *unlock* pe un lacat deschis sau detinut de alt fir, ea va returna eroare. De asemenea, operatia *lock* efectuata asupra unui lacat deja inchis si detinut de firul apelant va returna eroare.

In Linux (si alte citeva sisteme asemanatoare UNIX), exista trei tipuri de lacate: "fast", "error checking" si "recursive". Descrierea de mai sus se refera la tipul "error checking" pe care il vom utiliza. Cititi paginile de manual pentru o descriere completa.

Principalele functii din biblioteca pthread ce gestioneaza un mutex ar fi urmatoarele:

```
int      pthread_mutex_init(pthread_mutex_t      *mutex,      const
pthread_mutexattr_t                                    *mutexattr);
int           pthread_mutex_lock(pthread_mutex_t           *mutex);
int          pthread_mutex_unlock(pthread_mutex_t          *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Pentru informatii despre cum functioneaza aceste functii se recomanda consultarea paginilor de manual din sectiunea 3.

Un exemplu de sincronizare a proceselor folosind mutex se afla in sectiunea Fire de executie de pagina acestui curs.