

Utilizarea sistemelor de operare

Curs

Drăgulici Dumitru Daniel

Facultatea de matematică și informatică,
Universitatea București

2021

Cuprins

- 1 Sisteme de calcul, sisteme de operare
 - Organizarea unui sistem de calcul
 - Funcțiile sistemului de operare
- 2 Sisteme de operare uzuale
 - Incișiune în sistemul Windows
 - Incișiune în sistemul Linux
- 3 Interfața de programare C
 - Obiecte software și apeluri sistem
 - Utilizatori și grupuri
 - Fișiere și directoare
 - Procese
 - Semnale
 - Tuburi

- Biblioteca standard C (tipul 'FILE')
- 4 Interfață interactivă linie de comandă
 - Interfața Command Prompt (Windows)
 - Interfața Shell (Linux)
- 5 Interfață de scripting linie de comandă
 - Interfața Batch Scripting (Windows)
 - Interfața Shell Scripting (Linux)
- 6 Teme de laborator
 - Utilizatori și grupuri
 - Fișiere, directoare
 - Procese, semnale, tuburi

Bibliografie:

- Andrew S. Tanenbaum: Sisteme de operare moderne, Editia a 2-a, Editura Byblos, 2004
- Richard Petersen: Linux: The Complete Reference, Sixth Edition, The McGraw-Hill Companies, 2008
- Michael Kerrisk: The Linux Programming Interface, No Starch Press, 2010
- Joan Lambert: Windows 10 Step by Step, Second Edition, Pearson Education, Inc., 2018

Resurse web:

- <https://tldp.org/guides.html>
- <https://www.gnu.org/software/libc/manual/pdf/libc.pdf>
- <https://zetcode.com/gui/winapi/>
- https://en.wikibooks.org/wiki/Windows_Batch_Scripting

Cuprins

1 Sisteme de calcul, sisteme de operare

Organizarea unui sistem de calcul
Funcțiile sistemului de operare

2 Sisteme de operare uzuale

Incursiune în sistemul Windows
Incursiune în sistemul Linux

3 Interfața de programare C

Obiecte software și apeluri sistem
Utilizatori și grupuri
Fișiere și directoare
Procese
Semnale
Tuburi

Biblioteca standard C (tipul 'FILE')

4 Interfață interactivă linie de comandă

Interfața Command Prompt
(Windows)
Interfața Shell (Linux)

5 Interfață de scripting linie de comandă

Interfața Batch Scripting (Windows)
Interfața Shell Scripting (Linux)

6 Teme de laborator

Utilizatori și grupuri
Fișiere, directoare
Procese, semnale, tuburi

Cuprins

1 Sisteme de calcul, sisteme de operare

Organizarea unui sistem de calcul

Funcțiile sistemului de operare

2 Sisteme de operare uzuale

Incursiune în sistemul Windows

Incursiune în sistemul Linux

3 Interfața de programare C

Obiecte software și apeluri sistem

Utilizatori și grupuri

Fișiere și directoare

Procese

Semnale

Tuburi

Biblioteca standard C (tipul 'FILE')

4 Interfață interactivă linie de comandă

Interfața Command Prompt
(Windows)

Interfața Shell (Linux)

5 Interfață de scripting linie de comandă

Interfața Batch Scripting (Windows)

Interfața Shell Scripting (Linux)

6 Teme de laborator

Utilizatori și grupuri

Fișiere, directoare

Procese, semnale, tuburi

Sistem de calcul (SC): ansamblu de componente hardware și software care furnizează o formă definită de servicii unui tip de utilizatori.

Un SC este format din:

- Partea de hard (hardware): totalitatea componentelor fizice (carcasă, procesor, tastatură, etc.);
- Partea de soft (software): totalitatea componentelor logice (programe, date, etc.).

În general, un SC are o organizare stratificată, începând cu stratul hardware și continuând cu mai multe straturi de software, primul dintre ele fiind sistemul de operare (SO):



De exemplu, un program Java este interpretat de mașina virtuală Java (JVM), care rulează deasupra SO, efectuând cereri către acesta (apeluri sistem) iar instrucțiunile mașina ale SO și JVM sunt executate de hardware.

Dacă programul Java este un browser web, atunci o pagina web sau un script Javascript interpretate de acesta ar fi obiecte software din stratul al 5-lea.

Delimitarea între straturi este discutabilă - anumite aplicații sunt livrate odată cu SO ca utilitare (fac parte din distribuție): editoare de text, compilatoare uzuale, etc.; alte aplicații sunt vândute separat și nu sunt prezente în orice instalare a SO respectiv.

Stratul "SO" din desenul anterior este constituit de fapt din aşa-numitul kernel (nucleu), care implementează funcționalitățile de bază ale SO și care rulează într-un mod special, privilegiat, numit kernel mode (sau mod supervisor). Celelalte programe rulează în user mode (sau mod utilizator), mai puțin privilegiat, și sunt considerate în stratul "Aplicații".

Cuprins

1 Sisteme de calcul, sisteme de operare

Organizarea unui sistem de calcul

Funcțiile sistemului de operare

2 Sisteme de operare uzuale

Incișiune în sistemul Windows

Incișiune în sistemul Linux

3 Interfața de programare C

Obiecte software și apeluri sistem

Utilizatori și grupuri

Fișiere și directoare

Procese

Semnale

Tuburi

Biblioteca standard C (tipul 'FILE')

4 Interfață interactivă linie de comandă

Interfața Command Prompt
(Windows)

Interfața Shell (Linux)

5 Interfață de scripting linie de comandă

Interfața Batch Scripting (Windows)

Interfața Shell Scripting (Linux)

6 Teme de laborator

Utilizatori și grupuri

Fișiere, directoare

Procese, semnale, tuburi

Există mai multe puncte de vedere privind setul minimal de funcții pe care trebuie să le ofere un sistem de operare. Un asemenea set ar fi:

- Gestionația utilizatorilor și securitatea sistemului:

Există sisteme:

- monouser: nu fac distincție între utilizatorii fizici care se pot conecta; orice persoană care operează în sistem are acces necondiționat la toate resursele oferite de SO, ca și când ar exista un singur utilizator care lucrează mereu - de fapt, conceptul de utilizator logic nu este implementat.

Exemplu: MS-DOS.

- multiuser: fac distincție între utilizatorii fizici care se pot conecta, prin intermediu conturilor; în sistem sunt definite un set de conturi, având un nume de cont (username), o parolă (password), niște drepturi, etc., și orice persoană (utilizator fizic), dacă vrea să se conecteze la sistem (logare), trebuie să indice mai întâi un cont și parola corespunzătoare; odată acceptat, sistemul îl asimilează cu persoana pentru care s-a creat contul respectiv (chiar dacă nu este, dar îcunoaște contul și parola); conturile sunt utilizatorii logici ai sistemului - în cele ce urmează, prin "utilizator" vom înțelege de fapt un cont.

Exemple: UNIX, Linux, Windows, Novell Netware, etc.

Într-un sistem multisuer, orice proces, fișier, etc., are printre caracteristicile sale un proprietar (un cont) și niște drepturi, iar sistemul împiedică o acțiune a unui utilizator să acceseze un obiect al altui utilizator, dacă proprietarul obiectului nu a marcat acolo permisiunile necesare - astfel se asigură securitatea datelor fiecărui utilizator.

De exemplu, un proces având ca proprietar utilizatorul U1 nu poate citi dintr-un fișier ce are ca proprietar utilizatorul U2 dacă U2 nu a setat pentru acel fișier permisiunea de citire pentru categoria de utilizatori din care face parte U1 (când procesul va încerca să deschidă fișierul în citire cu apelul sistem "open", acest apel va eșua și va returna procesului un cod de eroare).

În general există însă și utilizatori privilegiași (ca de exemplu "root" în sistemele UNIX și Linux) care au automat toate drepturile. Aceștia pot, de exemplu, crea și distrugă alți utilizatori. Un utilizator obișnuit poate face doar modificări minime și, de exemplu schimarea parolei proprii.

- Gestiunea fișierelor:

Fișierele sunt resursa abstractă a dispozitivelor de stocare.

Ele au un nume și diverse caracteristici:

- atribute, în sistemele MS-DOS și Windows: Read-only, Archive, System, Hidden;
- proprietar și drepturi de acces, în sistemele multiuser.

Fișierele figurează în directoare - structuri de date ce conțin informații despre diverse fișiere și alte directoare; în UNIX, Linux, directoarele sunt implementate tot ca niște fișiere, de un tip special.

Sistemul de fișiere și directoare, legate prin relația de apartenență la un director, formează arborescențe. Un dispozitiv de stocare este văzut ca un ansamblu de discuri logice, fiecare conținând câte o asemenea arborescență.

Întrucât numele fișierelor dintr-un sistem nu sunt neapărat unice, un fișier este, în general, desemnat cu ajutorul unui specifikator format din disc, calea către el în arborescența discului respectiv și apoi numele fișierului.

- Gestiunea proceselor:

Procesul este un concept cheie, fundamental, în sistemele de operare.

Proces: execuție a unui program.

Nu trebuie confundată noțiunea de proces cu cea de fișier executabil sau de program.

Mai multe proceze diferite pot executa același fișier - atunci, deși procesele execută același program, ele pot avea la un moment dat alte valori pentru variabilele declarate în program, pot fi la o altă instrucțiune curentă, etc.

Există sisteme:

- monotasking: pot rula doar un singur proces la un moment dat; deci procesele trebuie rulate pe rând.

Exemplu: MS-DOS.

- multitasking: pot rula mai multe proceze simultan (multiprogramare); procesele pot rula pe procesoare diferite, sau mai multe pe un același procesor, în mod intercalat (alternativ, câte o porțiune din fiecare).

Exemple: UNIX, Linux, Windows, Novell Netware.

Caracteristici ale proceselor:

- fișierul pe care îl execută;
- spațiul de adresare - o zonă de memorie pe care o poate accesa cu instrucțiunile obișnuite din fișierul (programul) executat; aici se rezervă locații pentru valorile proprii curente ale variabilelor din program și pentru stiva proprie;
- diverse informații asociate de SO: proprietarul (un cont), prioritatea, adresa instrucțiunii curente (reținută în registrul IP (arhitectura Intel) sau PC (arhitectura MIPS)), adresa vârfului curent al stivei (reținută în registrul SP), etc.

SO poate întrerupe temporar un proces (de exemplu pentru a executa o parte din alt proces în cadrul multitasking); atunci informațiile folosite la gestionarea lui (inclusiv valorile acelor registri) sunt salvate în tabela proceselor (un vector de structuri, câte una pentru fiecare proces).

De asemenea, SO gestionează comunicarea între proceze și arbitrează accesul concurrent al acestora la resurse, pentru a evita interblocajele.

Procesele pot lansa procese copil, organizându-se după această filiație în arborescențe; pentru aceasta, folosesc niște apeluri sistem (de exemplu "fork" în UNIX, Linux).

În general, SO oferă instrumente pentru crearea, distrugerea, blocarea, rularea proceselor.

- Gestiunea memoriei:

O alocare a memoriei la nivel de octet sau cuvânt este ineficientă; SO implementează algoritmi de gestionare și livrare a memoriei la nivel de blocuri, a căror evidență e menținută în niște liste.

SO arbitrează cererile diverselor procese pentru o aceeași zonă de memorie și asigură că un proces nu poate accesa zonele de memorie alocate altui proces (este un aspect de securitate).

SO implementează, de asemenea, extensiile ale memoriei principale sub formă de memorie virtuală, folosind capacitatele de memorare ale altor dispozitive (de obicei memoria principală este RAM iar cea virtuală este stocată pe disc) - astfel încât memoria principală să pară mai mare.

Memoria virtuală este văzută logic de procese ca și când ar face parte din memoria principală. În funcție de locul unde se află informațiile de care are nevoie un proces, SO swap-ează (interschimbă) blocuri din memoria RAM cu blocuri de pe disc.

SO poate permite unui proces să partajeze memoria folosită cu alte procese de pe aceeași mașină sau mașini diferite (prin rețea).

- Gestiunea dispozitivelor de I/O:

De obicei, SO privesc toate dispozitivele externe (discuri, benzi, terminale, imprimante, etc.) într-un același fel, generic.

Ei se ocupă de alocarea, izolarea și acordarea de dispozitive în funcție de o anumită politică, ținând cont de anumite priorități.

Când adăugăm la sistemul de calcul un dispozitiv nou, trebuie adăugat la SO driverul necesar; dacă nu avem codurile sursă, nu putem recompila SO a.î. să includă noul driver; această limitare a dus la dezvoltarea unor drivere reconfigurabile.

- Interfața cu aplicațiile:

SO (kernelul) implementează diverse tipuri de obiecte software (unele corespund unor obiecte hardware - de exemplu, fișierele bloc = discuri, fisierele caracter = terminale, altele sunt doar colectii de informații - de exemplu, fișierele obișnuite, directoarele, utilizatorii) și niște operații cu ele, sub forma unor rutine numite apele sisteme (AS), apelabile din aplicații.

Dintr-o aplicație, rulată ca proces, se pot cere servicii SO apelând AS cu obiecte software indicate ca parametri.

Obiectele software și AS pentru ele formează interfața SO cu aplicațiile (interfață de programare a SO, API - Application Programming Interface).

SO moderne împiedică aplicațiile (procesele) să ocolească interfața sa de programare, accesând resursele hardware sau software direct (de exemplu, echipamentele hardware prin instrucțiuni mașină speciale, de citire/scriere la porturile acestora, sau structurile de informații ale kernelului sau altor procese, prin accesarea directă, prin pointeri, a memoriei fizice).

În acest scop, sunt folosite facilități hardware speciale ale calculatoarelor moderne, cum ar fi nivelurile de privilegiu ale procesorului (rings) și adresarea protejată a memoriei.

- Interfața utilizator:

Poate fi:

- în mod linie de comandă;
- în mod grafic (GUI - Graphical User Interface).

Ea se poate implementa și cu ajutorul unor aplicații, programe utilitare, instalate în sistem.

MS-DOS are în mod nativ o interfață linie de comandă; îi se poate adăuga ca aplicație o interfață grafică (de exemplu, Windows 3.11 sau mai vechi).

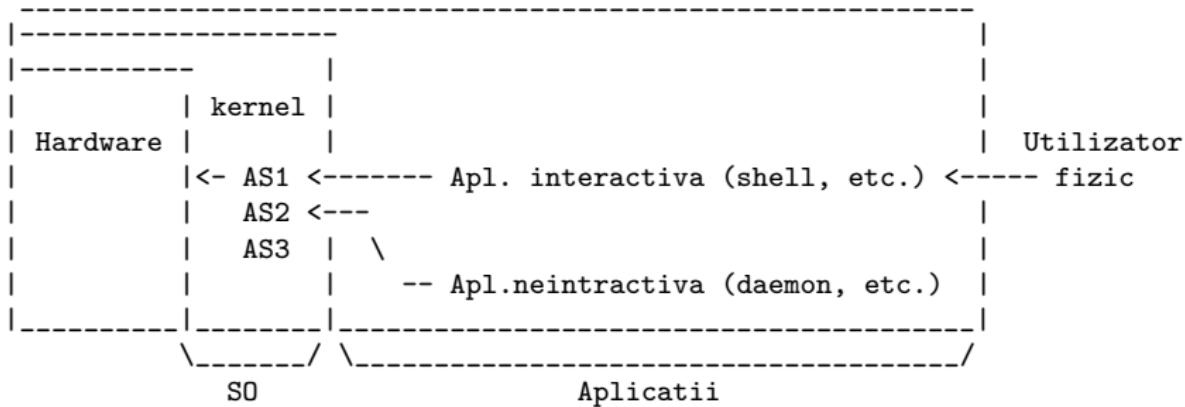
Windows are interfață grafică, dar are și una linie de comandă, prin "Command prompt".

UNIX, Linux implementează ambele tipuri de interfețe ca aplicații (interactive):

- pentru modul linie de comandă utilizează programele shell;
- pentru modul grafic utilizează subsistemul X-Windows.

Așadar, kernelul sistemelor UNIX, Linux are doar interfață cu aplicațiile (obiecte software, apeleuri sistem) iar aplicațiile pot fi interactive sau nu. Interfața cu utilizatorul fizic este realizată prin aplicații interactive (programe shell, gestionare de fișiere din subsistemul X-Windows, etc.).

SC cu SO UNIX, Linux



Alte funcții ale SO:

- Facilități privind lucrul în rețea.

Etc.

Cuprins

1 Sisteme de calcul, sisteme de operare

Organizarea unui sistem de calcul
Funcțiile sistemului de operare

2 Sisteme de operare uzuale

Incursiune în sistemul Windows
Incursiune în sistemul Linux

3 Interfața de programare C

Obiecte software și apeluri sistem
Utilizatori și grupuri
Fișiere și directoare
Procese
Semnale
Tuburi

Biblioteca standard C (tipul 'FILE')

4 Interfață interactivă linie de comandă

Interfața Command Prompt
(Windows)
Interfața Shell (Linux)

5 Interfață de scripting linie de comandă

Interfața Batch Scripting (Windows)
Interfața Shell Scripting (Linux)

6 Teme de laborator

Utilizatori și grupuri
Fișiere, directoare
Procese, semnale, tuburi

Într-un SC, pot fi instalate mai multe SO iar la pornirea acestuia (boot-are) se poate selecta dintr-un meniu (cu taste Up, Down, Enter) SO dorit.
În continuare, se lanseaza SO respectiv.

În exemplele de mai jos, în SC sunt instalate SO MX Linux și Windows 10.

Cuprins

1 Sisteme de calcul, sisteme de operare

Organizarea unui sistem de calcul
Funcțiile sistemului de operare

2 Sisteme de operare uzuale

Incișiune în sistemul Windows

Incișiune în sistemul Linux

3 Interfață de programare C

Obiecte software și apeluri sistem

Utilizatori și grupuri

Fișiere și directoare

Procese

Semnale

Tuburi

Biblioteca standard C (tipul 'FILE')

4 Interfață interactivă linie de comandă

Interfața Command Prompt
(Windows)

Interfața Shell (Linux)

5 Interfață de scripting linie de comandă

Interfața Batch Scripting (Windows)

Interfața Shell Scripting (Linux)

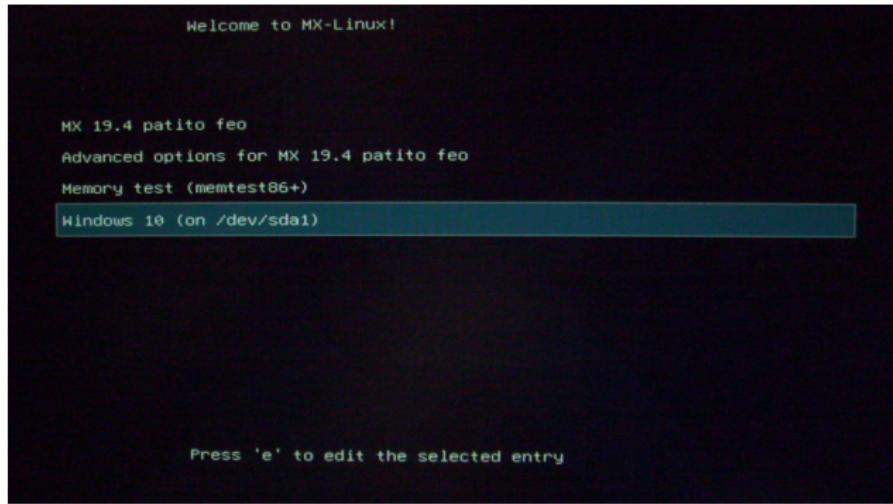
6 Teme de laborator

Utilizatori și grupuri

Fișiere, directoare

Procese, semnale, tuburi

La boot-are, alegem Windows 10:

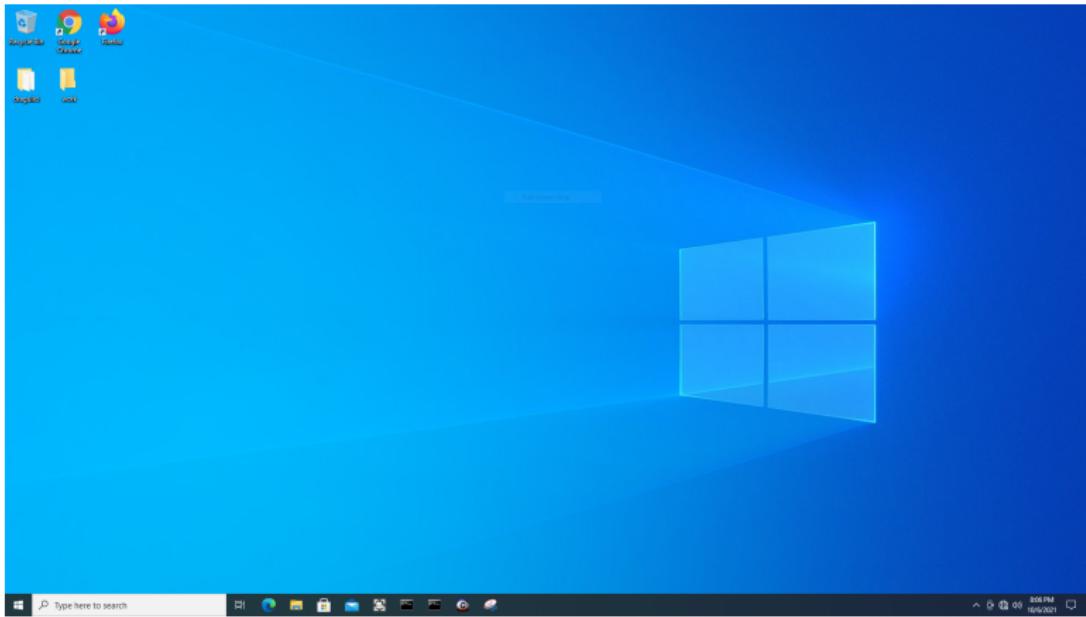


Odată lansat, Windows afisaza o interfață grafică (GUI) construită din ferestre - ferestre obișnuite, meniuri, etc. - care conțin elemente de interacțiune (widget) - etichete, butoane, casete de input text, etc. - ele sunt vizualizate ca liste de nume sau array de imagini (pictograme).

Ferestrele se pot suprapune sau pot fi conținute unele în altele, formând o ierarhie.

Întregul ansamblu de elemente afișate pe ecran (desktop) formează el însuși o fereastră, care este rădăcina ierarhiei.

În funcție de setări, la lansare, Windows poate cere într-o casetă contul și parola (autentificare), sau poate face autentificare automata, după care este afisat Desktop-ul pentru utilizare obișnuită.



Obs: pe desktop sunt 5 pictograme, dedesubt se află bara de sarcini (taskbar) unde sunt prezente alte pictograme, caseta de input text pentru căutări rapide (Search), butonul de start (stânga jos), alte informații (dreapta jos).

Taskbar-ul arată activitățile în desfășurare și conține comenzi rapide pentru activitățile frecvente. Aspectul grafic (imaginăria de fundal, amplasamentul pictogramelor, conținutul taskbar, etc.) este configurabil.

Asupra ferestrelor sau widget-urilor se poate acționa cu click sau dublu click un buton mouse, apăsare tasta (dacă obiectul deține focus-ul), click + tastă, click susținut + deplasare cursor mouse, etc., declanșând diverse comportamente.

De obicei:

- dublu click stânga lansează o activitate;
- click stânga mută focusul pe fereastră sau widget (fereastra este mutată în prim plan, widget-ul este desenat cu un fundal diferit); ulterior, tastele apăsate lansează activități asociate deținătorului focusului;
- click dreapta pe un widget sau într-o zona liberă deschide un meniu contextual (care este tot o fereastră), în care comenziile se pot da prin simplu click stânga;
- click stânga susținut, urmat de deplasare cursor, urmat de eliberare buton stânga, are efect de redimensionare sau mutare a unui obiect (drag-and-drop); pentru a redimensiona o fereastră, pornim acțiunea de la o margine sau colț al ei; pentru a muta o fereastră, pornim acțiunea din interiorul barei de titlu (aflată deasupra);

- click stânga susținut, deplasare, eliberare (ca mai sus) dar pornită dintr-o zonă liberă, desenează un dreptunghi și marchează obiectele interioare; ulterior, cu drag-and-drop de la un obiect interior, grupul se poate muta cu totul, iar dacă deplasarea se termină deasupra unui folder, obiectele din grup sunt mutate în folder;

odată marcate niște obiecte ca mai sus, cu **ctrl + click** stânga pe un obiect, el se poate adăuga/exclude din grupul de obiecte marcate, iar cu **click** într-un spațiu gol exterior, grupul se demarchează.

În cele ce urmează, prin click sau dublu click fără a preciza butonul vom subînțelege butonul stâng.

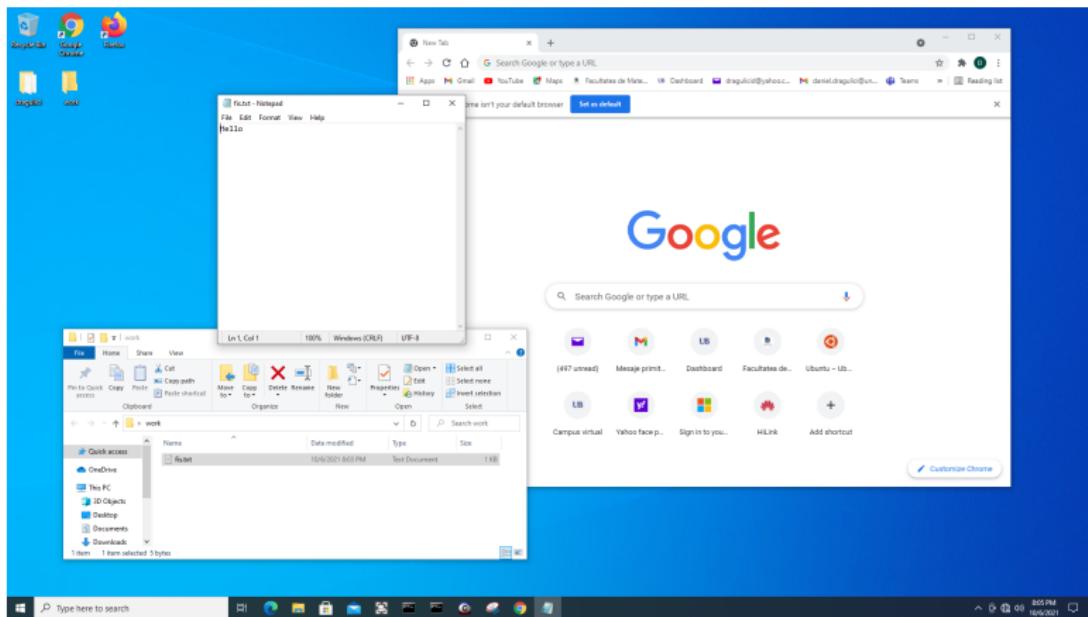
Din meniurile contextuale se pot da, de exemplu, comenziile 'New', cu care se poate crea un obiect nou și 'Delete', cu care se poate șterge obiectul curent - de fapt, el este mutat în folder-ul 'Recycle Bin', care poate fi golit cu click dreapta și 'Empty Recycle Bin'.

De obicei, activitățile asociate unei pictograme (lansate cu dublu click) sunt:

- vizualizarea unui folder asociat (colecție de obiecte, de exemplu un director din sistemul de fișiere), într-o nouă fereastră; obiectele din folder sunt afișate ca listă de nume sau array de pictograme;
- lansarea ca proces a unei aplicații asociate, cu maparea interfeței sale grafice într-o nouă fereastră;
- lansarea într-o nouă fereastră a unei aplicații care deschide un document asociat, pentru a opera asupra lui (de exemplu, se lansează un editor de texte care deschide documentul asociat pictogramei pentru editare).

Asocierile și comenziile aferente pot fi (re)configurate.

Exemplu:

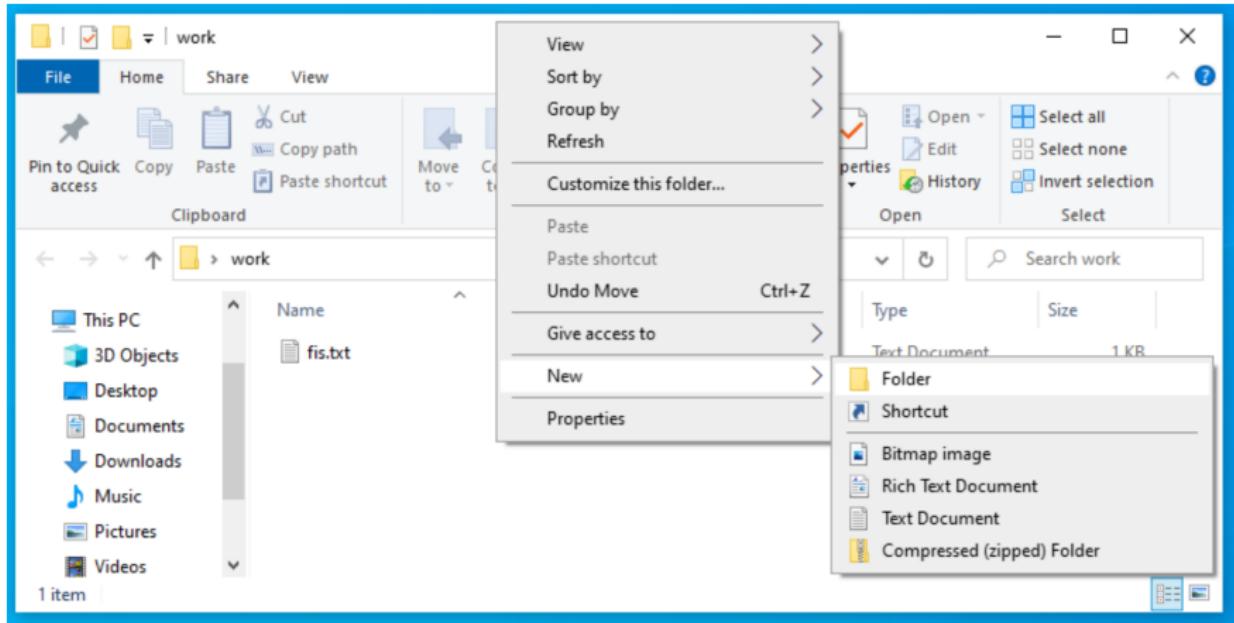


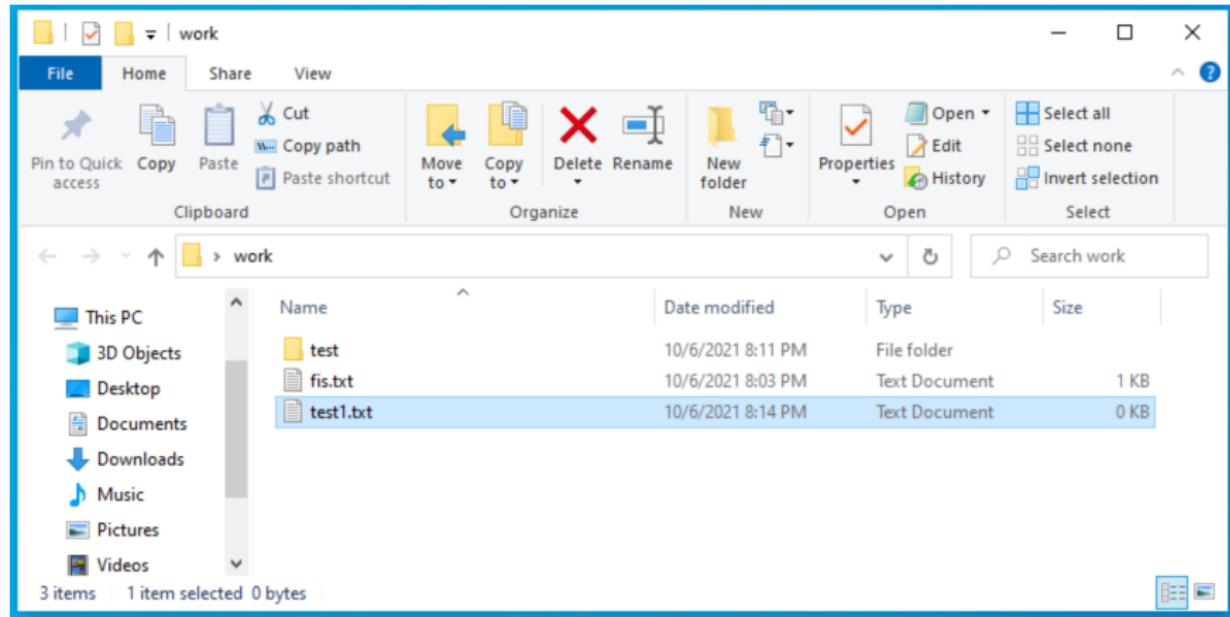
Prin dublu click, am deschis folderul (directorul) 'work', am lansat aplicația 'Chrome' (browser web) și am deschis spre editare cu editorul 'Notepad' fișierul 'fis.txt' din directorul 'work'.

Ulterior, cu click pe butoanele din dreapta sus ale ferestrei '-', '[]', 'X' putem minimiza fereastra la o pictogramă în taskbar (de unde cu click o putem restaura), respectiv maximiza la tot ecranul/restaura, respectiv închide.

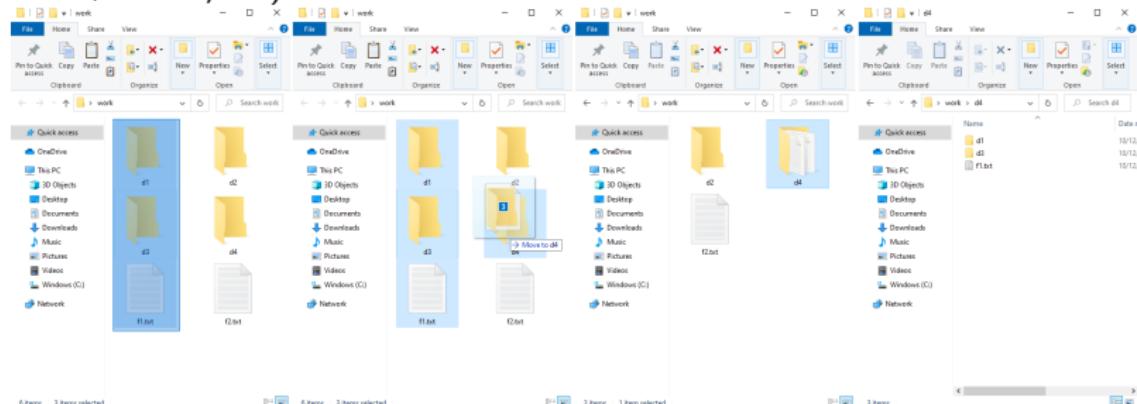
În exemplul următor, în fereastra/directorul 'work', cu click dreapta într-un spațiu liber și click în meniurile contextuale apărute, creăm un subdirector 'test1' și un fișier text 'test2.txt'.

Ulterior, cu click dreapta pe ele și 'Delete', sau cu click pe ele (capătă focus) și tasta 'Delete', le putem șterge.





În exemplul următor, în directorul 'work', am creat subdirectoarele 'd1', 'd2', 'd3', 'd4' și fișierele 'f1.txt', 'f2.txt', apoi am marcat grupul 'd1', 'd3', 'f1.txt' și l-am mutat în directorul 'd4', apoi am vizualizat 'd4' într-o fereastră (după aceea, le-am șters):



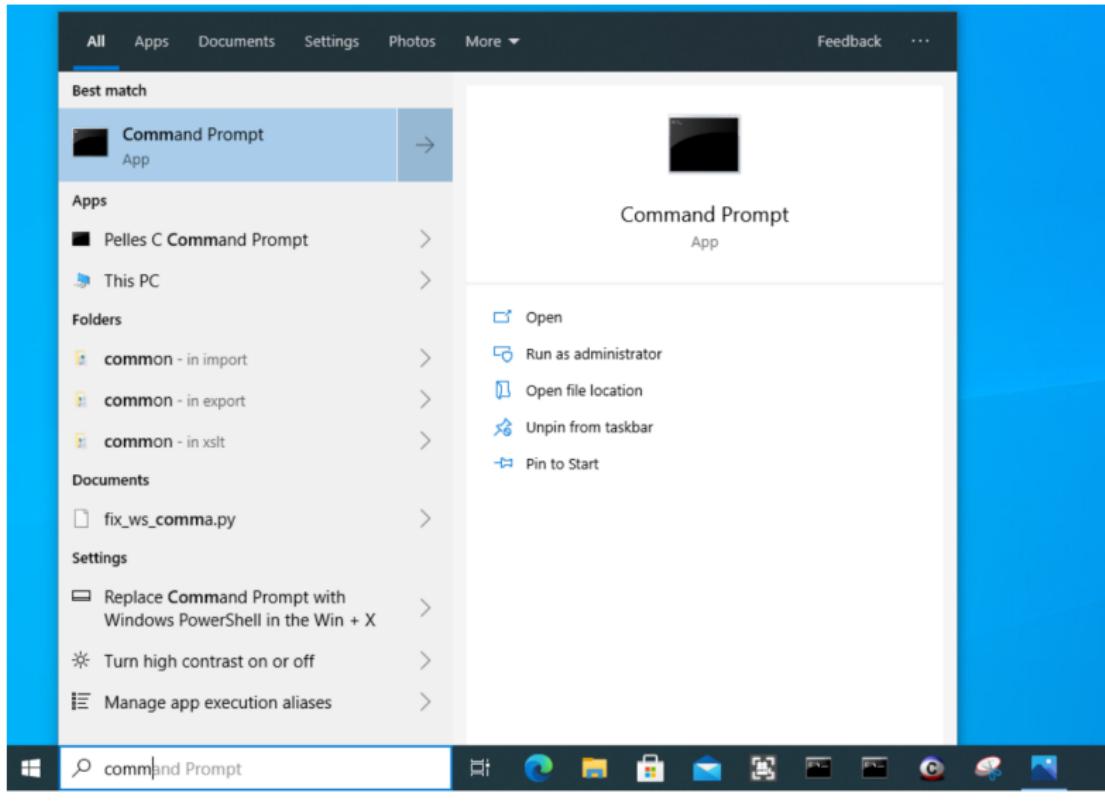
Cu click pe butonul start (stânga jos) apare un meniu din care se pot lansa diverse programe instalate sau subprograme ale sistemului de operare.

De exemplu, cu succesiunea 'Start - Windows System - Command Prompt' (am descris succesiunea de comenzi și meniuri care apar) se lansează un interpretor de linii de comandă (shell), cu care se poate opera asupra sistemului prin linii de comandă, în mod text.

Se pot lansa mai multe instanțe de execuție ale 'Command Prompt' - ele vor funcționa în paralel, în ferestre separate, independente una de alta (fiecare reține propriile setări curente, propriul istoric de comenzi, etc.).

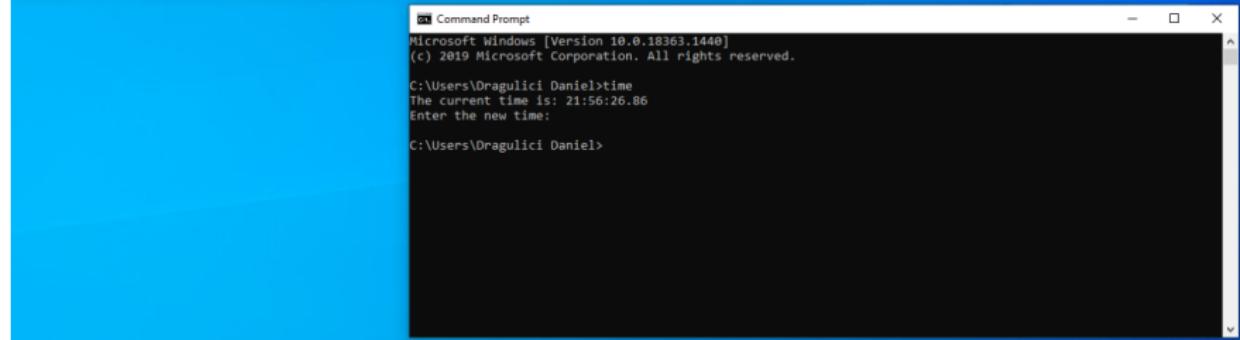
Întrucât meniul start este mare iar comenzi sunt greu de găsit, se pot culege în caseta Search alăturată butonului de start primele litere din comandă și vor apărea într-un meniu doar comenzi care încep cu literele respective.

În exemplul următor, am căutat cu ajutorul casetei Search literele 'comm', am găsit 'Command Prompt' și l-am lansat de 2 ori (2 instanțe), am redimensionat și mutat ferestrele acestora ca să nu se suprapună, apoi unei instanțe i-am dat comanda 'whoami', care afișază mașina și utilizatorul curent, iar celeilalte i-am dat comanda 'time' cu care se poate afla și modifica ora curentă.



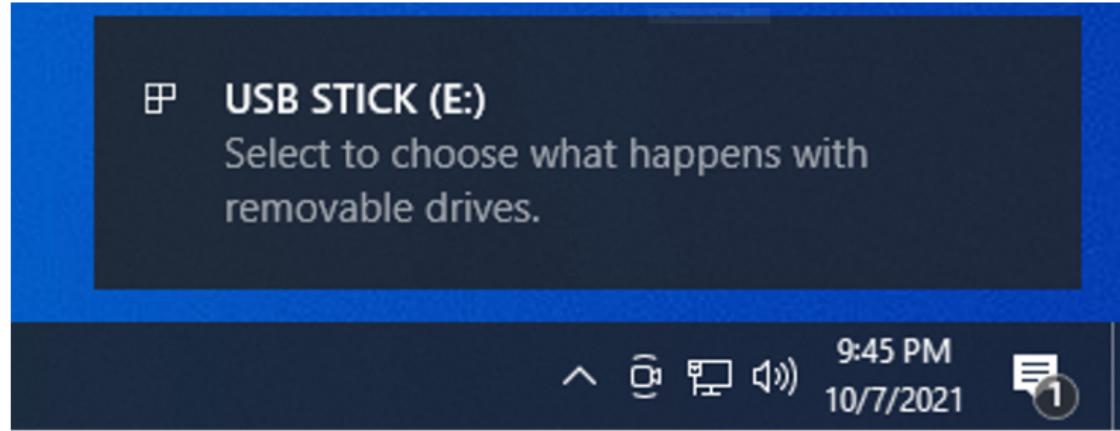


A screenshot of a Windows 10 desktop environment. In the center, there is a Command Prompt window titled "Command Prompt". The window shows the following text:
Microsoft Windows [Version 10.0.18363.1440]
(c) 2019 Microsoft Corporation. All rights reserved.
C:\Users\Dragulici Daniel>whoami
desktop-c23d3m1\dragulici daniel
C:\Users\Dragulici Daniel>

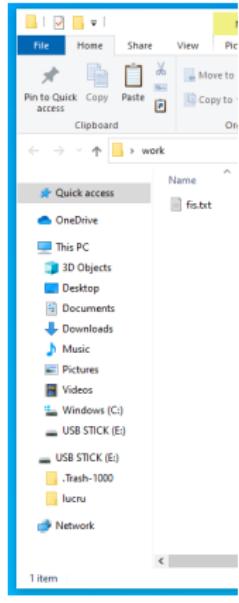


A screenshot of a Windows 10 desktop environment. In the center, there is a Command Prompt window titled "Command Prompt". The window shows the following text:
Microsoft Windows [Version 10.0.18363.1440]
(c) 2019 Microsoft Corporation. All rights reserved.
C:\Users\Dragulici Daniel>time
The current time is: 21:56:26.86
Enter the new time:
C:\Users\Dragulici Daniel>

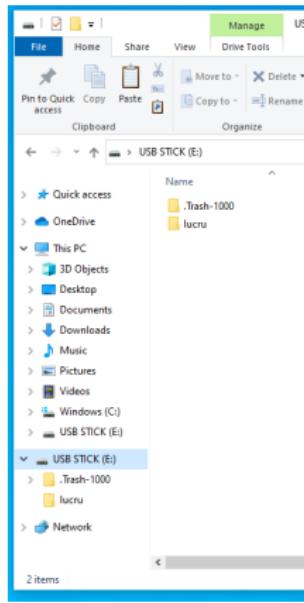
La conectarea unui memory-stick la un port USB, este afișată o notificare în colțul din dreapta jos:



iar în ferestrele care permit navigarea în sistemul de fișiere (subfereastra din stânga a ferestrelor care afișază directoare) este afișat stick-ul, ca un nou disc, și arborescenta sa:

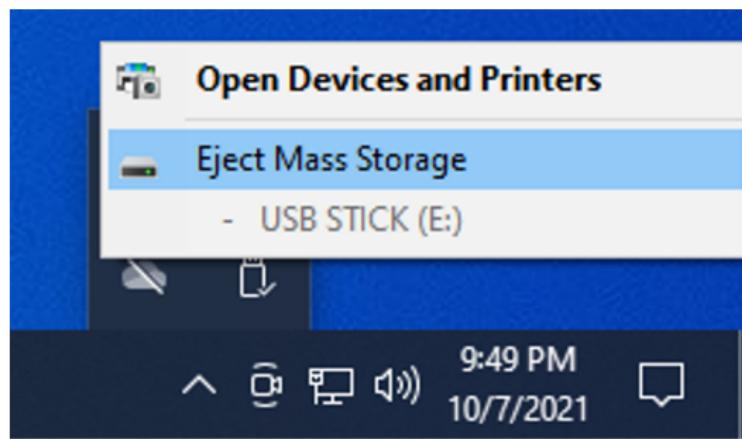


În continuare, se poate naviga pe stick în același fel ca printre directoarele de pe disc - de exemplu, cu click pe simbolul-stick se afișază în fereastra conținutul directorului rădăcina al acestuia:

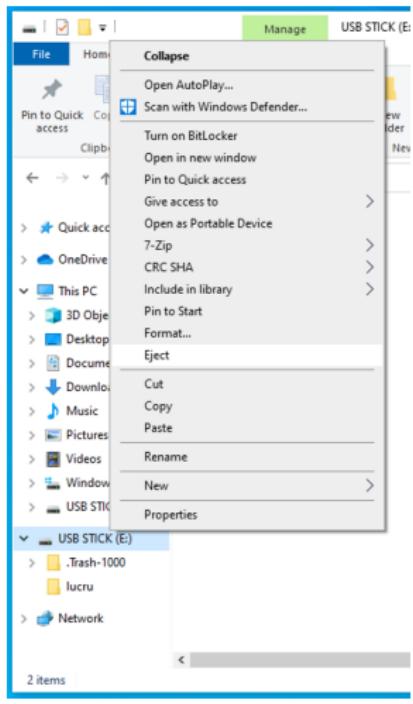


Pentru a extrage stick-ul, el trebuie demontat logic, apoi fizic (altfel, dacă sunt operațiuni cu el în desfășurare, se pot pierde sau corupe informații).

Pentru demontare logică, se poate folosi taskbar: click pe simbolul '^', apoi pe simbolul - stick, apoi pe 'Eject Mass storage':



sau se poate folosi o fereastră de navigare în sistemul de fișiere: click dreapta pe simbolul - stick, apoi click în meniu contextual pe 'Eject':



În final, se va afișa o notificare care informează că stick-ul se poate extrage fizic în siguranță:



Windows poate gestiona mai multe discuri logice, care sunt partiții ale unor discuri fizice, fiecare având câte o arborescență de directoare și fișiere. Pentru a putea fi identificat precis, fiecare obiect al sistemului de fișiere trebuie indicat printr-un specificator, format din disc, cale (drum de directoare în arborescență), nume (care se poate termina cu o extensie, precedată de ".") , separate prin '\'.

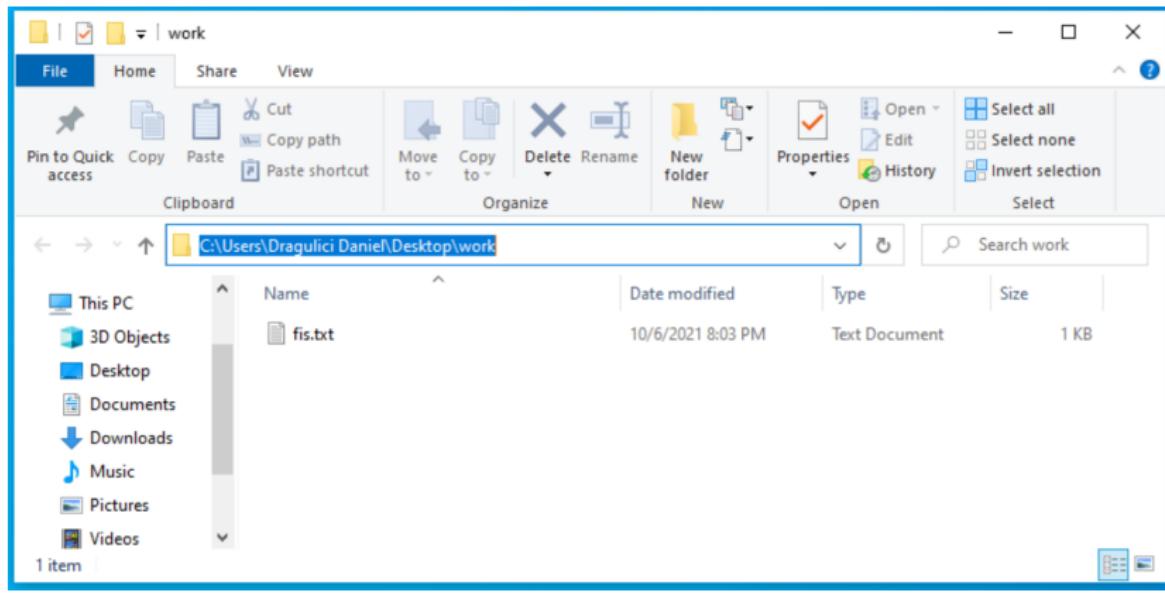
Specificatorul directorului vizualizat într-o fereastră apare în caseta de adrese situată în partea de sus a ferestrei.

Exemplu:

Directorul 'work' de pe desktop (vizualizat în fereastră) are specifikatorul 'C:\Users\Dragulici Daniel\Desktop\work'.

Fișierul 'fis.txt' din el are specifikatorul

'C:\Users\Dragulici Daniel\Desktop\work\fis.txt'.



În comenziile shell care operează cu obiecte ale sistemului de fișiere (fișiere și directoare), acestea trebuie indicate prin specifatori.

Întrucât specifatorii pot fi prea lungi, pentru productivitate, sunt implementate conceptele de disc curent și director curent al unui disc.

Fiecare proces reține ca atribut al său un disc ca fiind curent și câte un director de pe fiecare disc ca fiind directorul curent al discului respectiv.

Atunci, el poate identifica obiectele sistemului de fișiere și dacă sunt desemnate prin specifatori mai scurți, anume:

- dacă lipsește discul, se subînțelege cel curent;
- dacă lipsește calea, se subînțelege directorul curent al discului specificat;
- dacă lipsește discul și calea (este prezent doar numele cu extensie), se subînțelege discul curent și directorul său curent.

O cale poate începe cu:

- '\' și înseamnă că începe cu directorul rădăcină al discului (specificat sau curent);
- '.' și înseamnă că începe cu directorul curent al discului;
- '..' și înseamnă că începe cu directorul părinte al celui curent al discului;
- un nume și înseamnă că începe cu un subdirector (director copil) al celui curent al discului.

În ultimul caz, numele poate fi chiar al obiectului la care se referă specificatorul.

O cale poate conține '.' sau '..' și în interior, având semnificațiile:

- '.' în interior înseamnă că se trece din directorul anterior în el însuși;
- '..' în interior înseamnă că se trece din directorul anterior în parintele său (urcare un nivel).

Cările care încep cu disc și '\' (de exemplu, 'c:\') sunt căi absolute - ale au aceeași semnificație indiferent care este discul sau directorul curent.

Cările fără disc sau care încep cu '.', '..', sau nume, sunt căi relative - semnificația lor depinde de discul și directorul curent.

Shell-ul are comenzi prin care își poate afișa sau seta discul sau directorul curent iar comenziile referitoare la fișiere și directoare pot fi date folosind căi relative.

De exemplu (prin 'fișier' și 'director' înțelegem specicatori pentru un fișier, respectiv director, iar prin 'disc' înțelegem o specificare de disc, ex. 'c:'):

cd

- afișază discul și directorul curent

cd disc

- afișază directorul curent al discului specificat

cd director

- schimbă directorul curent pe discul specificat în 'director'

disc

- schimbă discul curent în cel specificat

dir

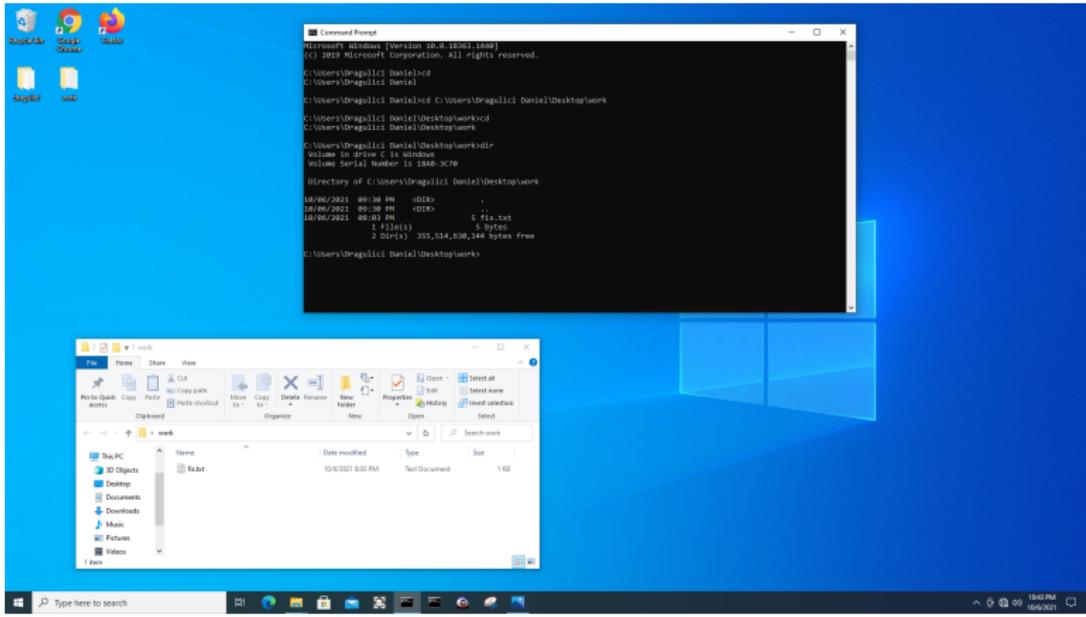
- afișază conținutul directorului curent al discului curent

dir director

- afișază conținutul directorului specificat

În exemplul următor, am efectuat, pe rând, următoarele:

- am deschis într-o fereastră folderul 'work' (care conține fișierul 'fis.txt');
- am lansat o consola shell 'Command Prompt';
- am dat în consolă comanda 'cd' și am văzut că discul și directorul curent sunt 'C:\Users\Dragulici Daniel';
- am scris în consolă 'cd ', fără ENTER;
- cu click în caseta de adrese a ferestrei 'work' a fost vizualizat și marcat specificatorul lui 'work');
- cu click dreapta în caseta de adrese, 'Copy' din meniul contextual, click dreapta în fereastra shell, s-a efectuat copy + paste a specificatorului în continuarea liniei de comanda shell;
- cu ENTER se execută comanda shell de schimbare a directorului curent pe discul 'c:' (care apare în specificator);
- cu altă comanda 'cd', am observat noul director curent al shell-ului;
- cu comanda 'dir', am afișat conținutul directorului curent de pe discul curent al shell-ului, anume 'C:\Users\Dragulici Daniel\work', iar acolo am observat fișierul 'fis.txt'.



Command Prompt

Microsoft Windows [Version 10.0.18363.1440]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Dragulici Daniel>cd
C:\Users\Dragulici Daniel

C:\Users\Dragulici Daniel>cd C:\Users\Dragulici Daniel\Desktop\work

C:\Users\Dragulici Daniel\Desktop\work>cd
C:\Users\Dragulici Daniel\Desktop\work

C:\Users\Dragulici Daniel\Desktop\work>dir
Volume in drive C is Windows
Volume Serial Number is 18A8-3C70

Directory of C:\Users\Dragulici Daniel\Desktop\work

10/06/2021	09:30 PM	<DIR>	.
10/06/2021	09:30 PM	<DIR>	..
10/06/2021	08:03 PM		5 fis.txt
		1 File(s)	5 bytes
		2 Dir(s)	355,514,630,144 bytes free

C:\Users\Dragulici Daniel\Desktop\work>

În exemplul anterior, putem vizualiza același director 'work' cu oricare dintre comenziile următoare:

```
dir C:\Users\Dragulici" "Daniel\Desktop\work  
dir "C:\Users\Dragulici Daniel\Desktop\work"
```

(ghilimelele înseamnă despecializare - shell-ul va trata caracterele speciale dintre ele ca fiind caractere obișnuite, de exemplu, spațiul din al doilea nume de director nu va fi considerat sfârșitul specificatorului ci parte a unui nume lung 'Dragulici Daniel')

```
dir C:.  
dir C:..\work  
dir C:..\work\..\..\..\..\..\..\Desktop\work
```

sau variantele fară disc (deoarece 'c:' este curent):

```
dir \Users\Dragulici" "Daniel\Desktop\work  
dir "\Users\Dragulici Daniel\Desktop\work"  
dir .  
dir ..\work  
dir ..\work\..\..\..\..\..\..\..\Desktop\work
```

Observăm că 'dir .' este echivalent cu 'dir' (simplu).

Prin comenzi shell putem crea sau șterge directoare:

mkdir director

- crează directorul specificat

rmdir director

- șterge directorul specificat; se poate doar dacă este gol; deci, pentru a șterge o arborescență, trebuie să ștergem de la periferie spre rădăcină.

Prezentăm câteva exemple (am apreluat cu copy + paste conținutul din fereastra shell):

```
C:\Users\Dragulici Daniel\Desktop\work>mkdir a
```

```
C:\Users\Dragulici Daniel\Desktop\work>dir
```

```
Volume in drive C is Windows
```

```
Volume Serial Number is 18A8-3C70
```

```
Directory of C:\Users\Dragulici Daniel\Desktop\work
```

10/06/2021	11:06 PM	<DIR>	.
10/06/2021	11:06 PM	<DIR>	..
10/06/2021	11:06 PM	<DIR>	a
10/06/2021	08:03 PM		5 fis.txt
		1 File(s)	5 bytes
		3 Dir(s)	355,517,603,840 bytes free

```
C:\Users\Dragulici Daniel\Desktop\work>mkdir a\b
```

```
C:\Users\Dragulici Daniel\Desktop\work>dir
```

```
Volume in drive C is Windows
```

```
Volume Serial Number is 18A8-3C70
```

```
Directory of C:\Users\Dragulici Daniel\Desktop\work
```

10/06/2021	11:06 PM	<DIR>	.
10/06/2021	11:06 PM	<DIR>	..
10/06/2021	11:06 PM	<DIR>	a
10/06/2021	08:03 PM		5 fis.txt
		1 File(s)	5 bytes
		3 Dir(s)	355,517,599,744 bytes free

```
C:\Users\Dragulici Daniel\Desktop\work>dir a
Volume in drive C is Windows
Volume Serial Number is 18A8-3C70
```

```
Directory of C:\Users\Dragulici Daniel\Desktop\work\a
```

```
10/06/2021  11:06 PM    <DIR>          .
10/06/2021  11:06 PM    <DIR>          ..
10/06/2021  11:06 PM    <DIR>          b
                           0 File(s)           0 bytes
                           3 Dir(s)   355,517,599,744 bytes free
```

```
C:\Users\Dragulici Daniel\Desktop\work>rmdir a  
The directory is not empty.
```

```
C:\Users\Dragulici Daniel\Desktop\work>rmdir a\b
```

```
C:\Users\Dragulici Daniel\Desktop\work>rmdir a
```

```
C:\Users\Dragulici Daniel\Desktop\work>dir  
Volume in drive C is Windows  
Volume Serial Number is 18A8-3C70
```

```
Directory of C:\Users\Dragulici Daniel\Desktop\work
```

10/06/2021	11:06 PM	<DIR>	.
10/06/2021	11:06 PM	<DIR>	..
10/06/2021	08:03 PM		5 fis.txt
		1 File(s)	5 bytes
		2 Dir(s)	355,517,599,744 bytes free

Exercițiu: refaceti exemplul, urmărind evoluția conținutului directoarelor și în ferestre.

Câteva comenzi shell pentru fișiere:

type fișier

- se afișază conținutul fișierului

copy fișier1 fișier2

- se copiază fișierul 'fișier1' în 'fișier2'

copy fișier1 + fișier2 + ... + fișier n fișier

- se copiază în fișierul 'fișier' o concatenare a conținuturilor fișierelor 'fișier1', ..., 'fișier n'

move fișier1 fișier2

- se redenumește și/sau mută 'fișier1' în 'fișier2' (dacă specificatorul 'fișier2' diferă de 'fișier1' doar prin nume și extensie, este doar o redenumire, iar dacă diferă calea sau discul, este o mutare (și, eventual, o redenumire))

del fișier

- se șterge fișierul

fc /b fișier1 fișier2

- compară binar conținuturile celor două fișiere și afișază diferențele (dacă conținuturile sunt identice, afișază un mesaj adecvat)

Prezentăm în continuare câteva exemple:

```
C:\Users\Dragulici Daniel\Desktop\work>dir  
Volume in drive C is Windows  
Volume Serial Number is 18A8-3C70
```

```
Directory of C:\Users\Dragulici Daniel\Desktop\work
```

```
10/06/2021  11:06 PM    <DIR>          .  
10/06/2021  11:06 PM    <DIR>          ..  
10/06/2021  08:03 PM           5 fis.txt  
                           1 File(s)      5 bytes  
                           2 Dir(s)   355,516,882,944 bytes free
```

```
C:\Users\Dragulici Daniel\Desktop\work>type fis.txt
```

```
Hello
```

```
C:\Users\Dragulici Daniel\Desktop\work>type c...\\work\fis.txt
```

```
Hello
```

```
C:\Users\Dragulici Daniel\Desktop\work>copy fis.txt fis1.txt  
1 file(s) copied.
```

```
C:\Users\Dragulici Daniel\Desktop\work>dir  
Volume in drive C is Windows  
Volume Serial Number is 18A8-3C70
```

```
Directory of C:\Users\Dragulici Daniel\Desktop\work
```

10/06/2021	11:14 PM	<DIR>	.
10/06/2021	11:14 PM	<DIR>	..
10/06/2021	08:03 PM		5 fis.txt
10/06/2021	08:03 PM		5 fis1.txt
		2 File(s)	10 bytes
		2 Dir(s)	355,516,817,408 bytes free

```
C:\Users\Dragulici Daniel\Desktop\work>type fis1.txt  
Hello
```

```
C:\Users\Dragulici Daniel\Desktop\work>move fis1.txt fis2.txt  
1 file(s) moved.
```

```
C:\Users\Dragulici Daniel\Desktop\work>dir  
Volume in drive C is Windows  
Volume Serial Number is 18A8-3C70
```

```
Directory of C:\Users\Dragulici Daniel\Desktop\work
```

```
10/06/2021  11:14 PM    <DIR>          .  
10/06/2021  11:14 PM    <DIR>          ..  
10/06/2021  08:03 PM           5 fis.txt  
10/06/2021  08:03 PM           5 fis2.txt  
                           2 File(s)        10 bytes  
                           2 Dir(s)   355,516,809,216 bytes free
```

```
C:\Users\Dragulici Daniel\Desktop\work>type fis2.txt  
Hello
```

```
C:\Users\Dragulici Daniel\Desktop\work>del fis2.txt
```

```
C:\Users\Dragulici Daniel\Desktop\work>dir
Volume in drive C is Windows
Volume Serial Number is 18A8-3C70
```

```
Directory of C:\Users\Dragulici Daniel\Desktop\work
```

10/06/2021 11:15 PM	<DIR>	.
10/06/2021 11:15 PM	<DIR>	..
10/06/2021 08:03 PM		5 fis.txt
	1 File(s)	5 bytes
	2 Dir(s)	355,516,809,216 bytes free

```
C:\Users\Dragulici Daniel\Desktop\work>copy fis.txt + fis.txt dest.txt  
fis.txt  
fis.txt  
1 file(s) copied.
```

```
C:\Users\Dragulici Daniel\Desktop\work>type dest.txt  
HelloHello
```

```
C:\Users\Dragulici Daniel\Desktop\work>copy fis.txt fisnou.txt  
1 file(s) copied.
```

```
C:\Users\Dragulici Daniel\Desktop\work>fc /b fis.txt fisnou.txt  
Comparing files fis.txt and FISNOU.TXT  
FC: no differences encountered
```

```
C:\Users\Dragulici Daniel\Desktop\work>fc /b fis.txt dest.txt  
Comparing files fis.txt and DEST.TXT  
FC: DEST.TXT longer than fis.txt
```

Pot fi specificate generic grupuri de fișiere, cu ajutorul unor măști (expresii regulate), care au forma unor specifiicatori (scriși după toate regulile anterioare referitoare la căi de mai devreme) în care ultima componentă (numele obiectului specificat) conține și caractere speciale, ca '*' sau '?'.

O mască va desemna toate fișierelor al căror specifiicator se poate obține din ea înlocuind caracterele speciale după anumite reguli.

De exemplu, '*' înseamnă un sir oarecare, eventual vid, iar '?' înseamnă un singur caracter, oarecare.

Un specifiicator poate fi considerat un caz particular de mască, fără caractere speciale (dacă are sens, desemnează un singur obiect).

Putem folosi măști în comenziile shell dinainte:

dir mască

- afisază doar fișierele care se potrivesc cu masca

copy mască director

- copiază fișierele care se potrivesc cu masca în director, păstrându-le același nume

move mască director

- mută fișierele care se potrivesc cu masca în director, păstrându-le același nume

del mască

- șterge fișierele care se potrivesc cu masca

Mai prezentăm câteva comenzi/opțiuni, care facilitează exemplificările:

`cls`

- șterge ecranul (conținutul ferestrei 'Command Prompt')

`prompt $g`

- în continuare, prompterul shell se reduce la semnul '>'

`dir /b argumente`

- 'dir' afișază informațiile în format minimal (bare format), anume fără header și sumar

Putem afla informații despre opțiunile unei comenzi shell dacă o lansăm cu opțiunea '/?' (de exemplu, 'dir /?').

Exemplu:

Adaugăm în directorul 'work' de mai devreme fișierele:

abc.txt

abcd.txt

ab.txt

ac.tex

c.tex

În continuare, în consola shell dăm următoare comenzi și obținem:

```
C:\Users\Dragulici Daniel\Desktop\work>prompt $g
```

```
>dir
```

```
Volume in drive C is Windows  
Volume Serial Number is 18A8-3C70
```

```
Directory of C:\Users\Dragulici Daniel\Desktop\work
```

10/07/2021	01:59 PM	<DIR>	.
10/07/2021	01:59 PM	<DIR>	..
10/07/2021	01:51 PM		0 ab.txt
10/07/2021	01:47 PM		0 abc.txt
10/07/2021	01:48 PM		0 abcd.txt
10/07/2021	01:52 PM		0 ac.tex
10/07/2021	01:52 PM		0 c.tex
10/07/2021	01:53 PM		7 fis.txt
		6 File(s)	7 bytes
		2 Dir(s)	355,845,816,320 bytes free

```
>dir /b  
ab.txt  
abc.txt  
abcd.txt  
ac.tex  
c.tex  
fis.txt
```

```
>dir /b abc*.txt  
abc.txt  
abcd.txt
```

```
>dir /b ..\work\abc*.txt  
abc.txt  
abcd.txt
```

```
>dir /b ?c.*  
ac.tex
```

```
>dir /b *.*
```

ab.txt

abc.txt

abcd.txt

ac.tex

c.tex

fis.txt

```
>dir /b c.txt
```

File Not Found

```
>dir /b c.tex
```

c.tex

```
>mkdir x
```

```
>dir /b
```

```
ab.txt
```

```
abc.txt
```

```
abcd.txt
```

```
ac.tex
```

```
c.tex
```

```
fis.txt
```

```
x
```

```
>copy abc*.txt x
```

```
abc.txt
```

```
abcd.txt
```

```
2 file(s) copied.
```

```
>dir /b  
ab.txt  
abc.txt  
abcd.txt  
ac.tex  
c.tex  
fis.txt  
x
```

```
>dir /b x  
abc.txt  
abcd.txt
```

```
>del x\*.*  
C:\Users\Dragulici Daniel\Desktop\work\x\*.* , Are you sure (Y/N)? y
```

```
>dir /b x

>move abc*.txt x
C:\Users\Dragulici Daniel\Desktop\work\abc.txt
C:\Users\Dragulici Daniel\Desktop\work\abcd.txt
    2 file(s) moved.
```

```
>dir /b
ab.txt
ac.tex
c.tex
fis.txt
x
```

```
>dir /b x
abc.txt
abcd.txt
```

Obs: Dacă adăugăm comenzi 'del' opțiunea '/q', șterge automat fără să mai ceară interactiv confirmarea.

Pentru a dezvolta programe C, trebuie instalat un compilator al acestui limbaj.

Sunt mai multe compilatoare:

- în mod linie de comandă: fișierul sursă trebuie editat cu un editor separat, de exemplu 'Notepad', apoi compilat invocând utilitarul de compilare din linia de comanda a unui shell;
- IDE (Integrated Development Environment, Mediu integrat de dezvoltare): pachet de programe de sine stătător, cu interfață grafică - ferestre, care permite efectuarea tuturor operațiilor necesare dezvoltării programelor: editare, compilare, rulare, depanare.

De asemenea, compilatoarele pot fi:

- portabile, nu necesită instalare în sistem;
- care necesită instalare.

Un compilator de C pentru Windows în mod linie de comanda portabil este 'tcc' (Tiny C Compiler, se poate găsi la '<https://bellard.org/tcc/>')

Exemplu de folosire:

- Deschidem într-o fereastră directorul 'work' de mai devreme, creăm în el fișierul text 'prog.c', îl deschidem cu 'Notepad' (Click dreapta pe el + Edit) și scriem programul C următor (apoi salvăm cu Ctrl - S sau din meniu File - Save):

```
#include <stdio.h>
int main() {
    printf("Hello world !");
    return 0;
}
```

- Deschidem o consolă shell 'Command Prompt', schimbăm directorul său curent în 'work' (am vazut cum se face) și dăm comenziile shell:

```
C:\Users\Dragulici Daniel\Desktop\work>tcc prog.c
```

```
C:\Users\Dragulici Daniel\Desktop\work>dir /b
prog.c
prog.exe
```

```
C:\Users\Dragulici Daniel\Desktop\work>prog.exe
Hello world !
C:\Users\Dragulici Daniel\Desktop\work>prog
Hello world !
```

Observații:

1. La crearea fișierului 'prog.c', cu Click dreapta + 'New' + 'Text Document' în fereastra directorului, Windows crează un fișier cu extensia '.txt' iar această extensie poate fi invizibilă în fereastră, la fel și extensia '.exe', dacă în meniul 'View' al ferestrei nu este bifat 'File name extensions'. Atunci, va rezulta fișierul 'prog.c.txt', deși în fereastră se vede 'prog.c', iar 'tcc prog.c' nu va găsi fișierul. Dacă este bifată vizualizarea extensiilor, extensia adăugată automat '.txt' se poate șterge, a.î. să rămână doar '.c'.
2. Cu comanda 'tcc prog.c' am compilat fișierul 'prog.c' și a rezultat fișierul executabil 'prog.exe' (dacă ar fi existat erori, s-ar fi afișat erorile și nu s-ar fi generat fișierul executabil), apoi cu comanda 'prog.exe' sau 'prog' (extensiile 'exe', 'com' sau 'bat' sunt subînțelese de 'Command Prompt') s-a executat.
3. Fișierul executabil putea fi specificat și cu o cale mai lungă, folosind convențiile anterioare, de exemplu:

```
C:\Users\Dragulici Daniel\Desktop\work>..\work\prog  
Hello world !
```

Comenzile shell pot fi:

- comenzi interne - le executa shell-ul prin instrucțiunile sale;
- comenzi externe - sunt specificatori ai altor programe, însoțiti, eventual de argumente în linia de comandă (programele le primesc prin parametrii lui 'main()': 'argc', 'argv'), pe care shell-ul le lansează ca procese copil.

Numele comenzilor externe pot fi specificatori scriși după toate convențiile dinainte.

Procesele prin care se execută aplicațiile au printre atributele proprii și niște variabile de mediu (environment) pe care le pot crea, distrugе, seta/modifica, consulta și transmite prin moștenire proceselor pe care le lansează acestea.

Prin variabilele de mediu se poate realiza o comunicare între procese de la părinte la copil - de exemplu, părintele transmite copilului informații despre mediul în care trebuie să lucreze copilul.

Variabilele de mediu au valori string.

Shell-ul are comenzi prin care își poate consulta/modifica variabilele de mediu moștenite de la sistem:

`echo %VARIABILA%`

– afișază valoarea variabilei cu numele 'VARIABILA'

`set VARIABILA=sir`

– asignează variabila cu numele 'VARIABILA' cu stringul 'sir'; variabila setată astfel va putea fi transmisă prin moștenire proceselor copil create de comenziile shell externe date aici

Se poate specifica o valoare asignată care să conțină valoarea unei variabile de mediu existente, de exemplu:

`set VARIABILA1=abc%VARIABILA2%def`

– dacă 'VARIABILA2' conține 'xyz', în final 'VARIABILA1' va conține 'abcxyzdef'

Exemplu (lucrăm în continuare în consola shell deschisă):

```
C:\Users\Dragulici Daniel\Desktop\work>set a=abc
```

```
C:\Users\Dragulici Daniel\Desktop\work>echo %a%  
abc
```

```
C:\Users\Dragulici Daniel\Desktop\work>set b=123%a%xyz
```

```
C:\Users\Dragulici Daniel\Desktop\work>echo %b%  
123abctxyz
```

```
C:\Users\Dragulici Daniel\Desktop\work>set b=%b%/%b%ijk
```

```
C:\Users\Dragulici Daniel\Desktop\work>echo %b%  
123abctxyz123abctxyzijk
```

Printre variabilele de mediu ale shell-ului este și variabila 'PATH' care reține o listă de căi separate prin ';'. Dacă este dată o comandă shell externă (numele unui program executabil) fără cale, el este căutat în directorul curent dar și în directoarele din această listă.

Exemplu (lucrăm în continuare în consola shell deschisă):

```
C:\Users\Dragulici Daniel\Desktop\work>echo %PATH%
C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Windows\System32\OpenSSH\;C:\Users\Dragulici Daniel\AppData\Local\Microsoft\WindowsApps
```

```
C:\Users\Dragulici Daniel\Desktop\work>cd ..
```

```
C:\Users\Dragulici Daniel\Desktop>dir
Volume in drive C is Windows
Volume Serial Number is 18A8-3C70
```

Directory of C:\Users\Dragulici Daniel\Desktop

10/06/2021	09:01 PM	<DIR>	.
10/06/2021	09:01 PM	<DIR>	..
10/06/2021	07:59 PM	<DIR>	dragulici
10/06/2021	11:45 PM	<DIR>	work
		0 File(s)	0 bytes
		4 Dir(s)	355,496,906,752 bytes free

```
C:\Users\Dragulici Daniel\Desktop>set PATH=%PATH%;C:\Users\"Dragulici Daniel"\Desktop\work
```

```
C:\Users\Dragulici Daniel\Desktop>echo %PATH%
```

```
C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Windows\System32\OpenSSH\;C:\Users\Dragulici Daniel\AppData\Local\Microsoft\WindowsApps;C:\Users\"Dragulici Daniel"\Desktop\work
```

```
C:\Users\Dragulici Daniel\Desktop>prog  
Hello world !
```

Așadar: am văzut lista curentă de directoare de căutare, am schimbat directorul curent în părintele 'Desktop', am încercat să lansăm programul cu 'prog' și nu a mers, deoarece fișierul 'prog.exe' nu se afla în actualul director curent 'Desktop' și nici în lista de directoare reținută de 'PATH', apoi am concatenat la listă directorul 'C:\Users\Dragulici Daniel\Desktop\work' (unde se află fișierul 'prog.exe'), apoi am reușit să dăm comanda 'prog', deși directorul curent a rămas 'Desktop'.

Variabilele de mediu care vor fi transmise de sistem proceselor (inclusiv shell) se pot seta și din interfața grafică: în caseta de căutare (Search) de lângă butonul Start, culegem 'env' apoi alegem 'Edit environment variables for your account'.

Pentru a avea 'tcc' în sistem, se parcurg următorii pași:

- accesăm site-ul '<https://bellard.org/tcc/>'
- descarcăm arhiva cu compilatorul (versiunea cea mai recentă);
- dezarchivăm și mutăm directorul rezultat într-un loc convenabil (de exemplu, un director cu toate programele portabile)
- adaugăm la variabila 'PATH' (cu 'Edit environment variables for your account') specificatorul directorului în care se află executabilul 'tcc.exe' - astfel, putem da comanda shell externă 'tcc' fără cale, indiferent care este directorul curent al shell-ului (de obicei îl setăm să fie directorul cu programele sursă)

Un compilator de C pentru Windows de tip IDE este 'Pelles C', care poate fi descarcat de pe site-ul <http://www.smorgasbordet.com/pellesc/>

Cuprins

1 Sisteme de calcul, sisteme de operare

Organizarea unui sistem de calcul
Funcțiile sistemului de operare

2 Sisteme de operare uzuale

Incursiune în sistemul Windows
Incursiune în sistemul Linux

3 Interfața de programare C

Obiecte software și apeluri sistem
Utilizatori și grupuri
Fișiere și directoare
Procese
Semnale
Tuburi

Biblioteca standard C (tipul 'FILE')

4 Interfață interactivă linie de comandă

Interfața Command Prompt
(Windows)
Interfața Shell (Linux)

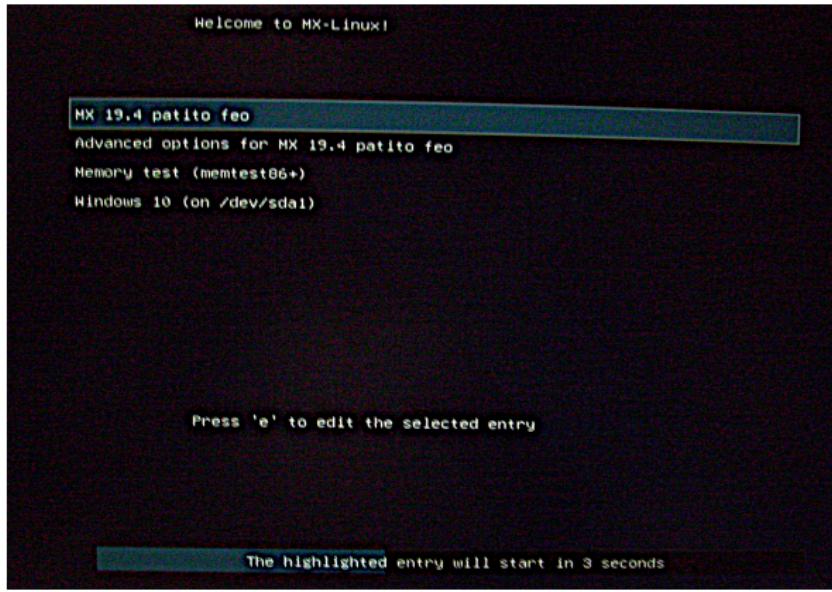
5 Interfață de scripting linie de comandă

Interfața Batch Scripting (Windows)
Interfața Shell Scripting (Linux)

6 Teme de laborator

Utilizatori și grupuri
Fișiere, directoare
Procese, semnale, tuburi

La boot-are, alegem Linux MX:



Odată lansat, Linux oferă mai multe terminale logice care se pot mapa alternativ pe terminalul fizic, folosind combinații de taste Ctrl - Alt - Fn (Fn este una din tastele funcționale F1, F2, ...).

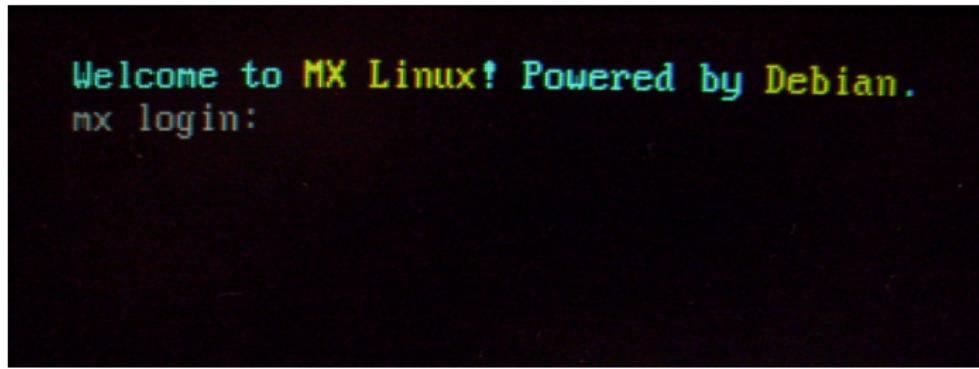
În fiecare terminal logic se poate face autentificare și se poate opera în mod independent (cu același cont sau conturi diferite).

Primele terminale logice operează în mod text - interfață linie de comandă, ultimul terminal operează în mod grafic - GUI, sistem de ferestre.

La un terminal text, utilizatorul este invitat să introducă un cont și o parolă. Dacă acestea sunt corecte, este lansat la terminalul respectiv un proces shell, care afișază un prompter și așteaptă linii de comandă.

Prompterul este, de obicei, \$ dar conținutul său este configurabil).

Exemplu de întâmpinare pe un terminal text:



Comenzile shell pot fi:

- comenzi interne: shell-ul le execută prin propriile sale instrucțiuni;
- comenzi externe: specificatori ai altor programe, pe care shell-ul le lansează ca procese - copil.

În absența unei mențiuni speciale, la introducerea unei comenzi externe, shell-ul va astepta terminarea procesului - copil, apoi își continuă executarea, afișând un nou prompter. Spunem că procesul - copil este executat în foreground iar el poate citi de la terminal.

Dacă adăugăm la sfârșitul liniei de comandă &, shell-ul nu va astepta terminarea procesului - copil ci va afișa un nou prompter imediat și i se pot da noi comenzi. Practic, shell-ul se va executa în paralel cu procesul - copil. Spunem că procesul - copil este executat în background iar el nu poate citi de la terminal.

Câteva comenzi shell:

whoami

- afișază user-ul logat (proprietarul procesului shell)

tty

- afișază terminalul logic curent (terminalul de contrul al procesului shell);
el este asimilat cu un fișier și se afișază specificatorul lui

pwd

- afisaza directorul curent (al procesului shell)

env

- afisază variabilele de mediu (environment) ale procesului shell

exit

- procesul shell se termină; utilizatorul este delegat de la terminalul logic respectiv iar procesele care se ocupă cu autentificarea la acel terminal sunt reluate și afișază din nou invitația de login.

Exemplu de operare pe un terminal text:

```
Welcome to MX Linux! Powered by Debian.  
mx login: dragulici  
Password:  
Last login: Wed Oct 13 22:12:21 EEST 2021 on tty2  
No mail.  
dragulici@mx:~  
$ whoami  
dragulici  
dragulici@mx:~  
$ export PS1=''$'  
$whoami  
dragulici  
$tty  
/dev/tty2  
$pwd  
/home/dragulici  
$
```

Observații:

- Ne-am logat cu user-ul 'dragulici'.
- Prompterul shell primit inițial a fost:

```
dragulici@mx:~  
$
```

- Cu comanda 'whoami' am afișat user-ul curent ('dragulici').
- Cu comanda ' export PS1=''\$' ' am setat ca prompterul să fie doar un simplu '\$' .

Setarea este locală procesului shell curent, ea se va pierde la terminarea acestuia; pentru a se face automat la fiecare lansare shell-ului, ea trebuie scrisă într-un fișier de configurare din directorul home al user-ului care îl lansează.

Dacă programul shell folosit este 'bash' și user-ul este 'dragulici', fișierul de configurare este '/home/dragulici/.bashrc'.

- Am dat din nou comanda 'whoami' pentru a observa noul modul de afișare a shell-ului.

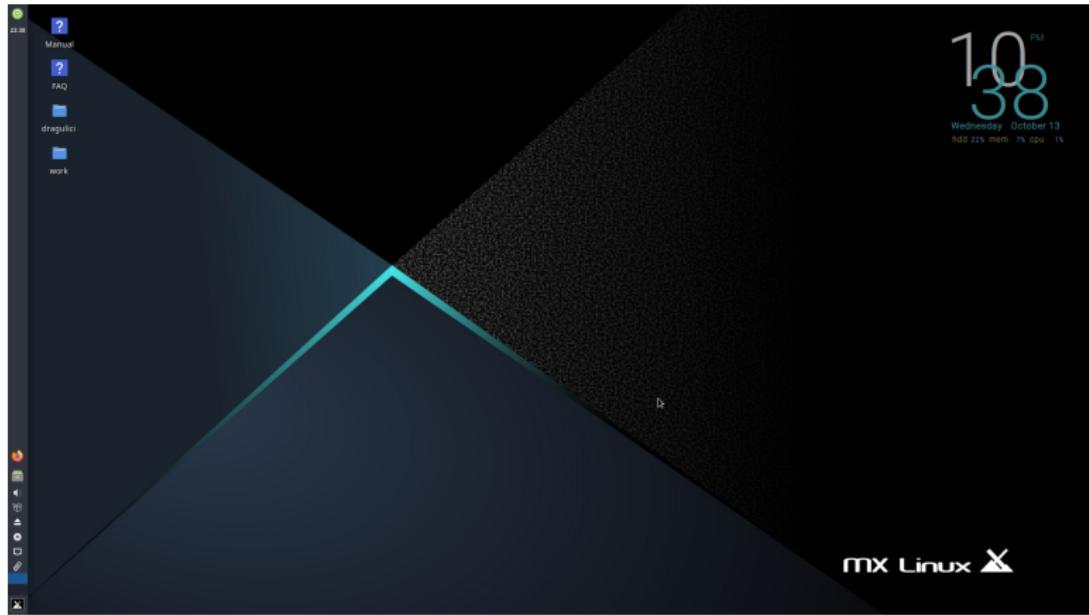
- Cu comanda 'tty' am afișat terminalul curent '/dev/tty2' (terminalul de control al procesului shell); terminalele (logice) sunt un tip special de fișiere și sunt specificate ca atare.
- Cu comanda 'pwd' am afișat directorul curent (al procesului shell).
- Cu comanda 'env' am fi afișat lista variabilelor de mediu (environment) ale procesului shell iar cu comanda 'exit' am fi terminat procesul shell - atunci, la același terminal logic '/dev/tty2' vor fi lansate din nou procesele care afișază invitația de login.
- Linux este case sensitive - face distincție între literele mari și mici.
- Componentele specificatorilor de fișier în Linux sunt separate cu '/' (nu cu '\', ca în Windows).
- Sistemul Linux nu face distincție între nume și extensie în cazul fișierelor; numirea fișierelor se face printr-un singur string iar caracterul '.' este un caracter obișnuit, care poate să facă parte sau nu din nume, chiar de mai multe ori.

Distincție între nume și extensie este doar din perspectiva umană - sensul pe care îl asociază omul numelor, iar acest sens poate fi implementat în aplicații (să trateze diferit fișierele în funcție de extensia lor).

- Numele care incep cu '.' sunt date, de obicei, unor fișiere de configurare prezente în directoarele home ale user-ilor; întrucât asupra lor nu se operează des, diverse aplicații le omit atunci cand caută fișiere pentru a opera asupra lor - le consideră 'ascunse'; dacă aplicațiile respective sunt lansate cu opțiuni speciale, nu le vor mai omite; subliniem că numele cu '.' sunt speciale doar pentru anumite aplicații, nu pentru nucleul Linux.

Ultima combinație Ctrl-Alt-Fn corespunde terminalului grafic.

Aici, are loc o interacțiune GUI, printr-un sistem de ferestre, asemănător cu Windows. La început, se poate cere într-o casetă un cont și o parolă; se poate seta, însă, autentificarea automată. Apoi, se afișază Desktop-ul:

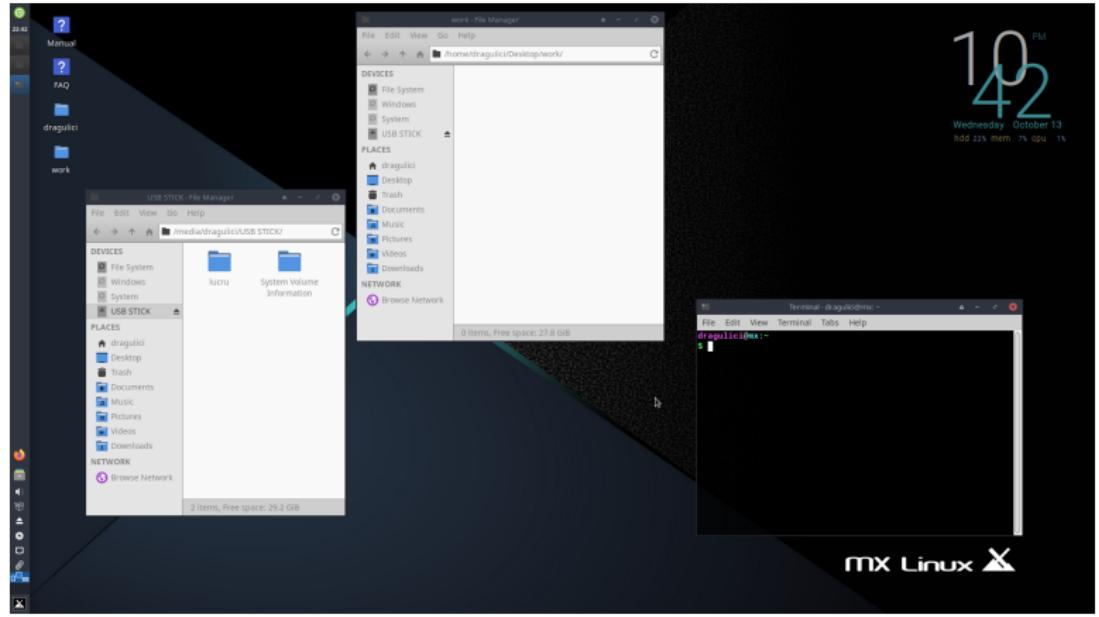


Aspectul grafic (ferestre și pictograme, widget-uri, taskbar, butonul start, meniuri, etc.) și modul de operare (click sau dublu click butoane mouse, drag-and-drop, taste interceptate de elementele care dețin focus-ul, marcaje, etc.) sunt asemănătoare celor de la Windows (sunt implementate aceleași principii generale de interfață grafică).

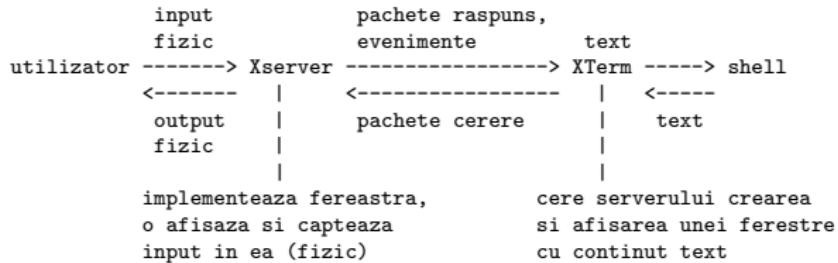
De asemenea, aspectul grafic este configurabil. În exemplul anterior, taskbar-ul este în dreapta (la Windows, era jos).

În exemplul următor:

- Am introdus un memory stick; sistemul l-a montat logic automat și a afișat într-o fereastră conținutul lui; cu click dreapta pe simbolul - stick din dreapta ferestrei și 'Eject' din meniul contextual, se poate demonta logic, apoi se poate extrage fizic.
- Am afișat într-o fereastră directorul 'work' de pe Desktop.
- Am lansat într-o fereastră un proces shell; în acest scop, am folosit meniul de start (butonul Start este în stânga jos).



Programele shell au interfață text iar aceasta nu se poate mapea acum direct pe terminalul fizic, care este setat în modul grafic; de aceea, între procesul shell și terminalul fizic se interpun anumite procese ale subsistemului grafic al Linux (numit XWindows): un proces X server și un proces Terminal (XTerm, Gnome Terminal, etc.); procesul Terminal solicită o fereastră, care este afișată de X server, lansează un proces copil shell (bash, ksh, csh, etc.) și îi mapează interfața text în fereastra respectivă:



Utilizatorul interacționează fizic cu serverul X dar logic (dpv. al problemei rezolvate) cu shell-ul - de aceea, identifică fereastra cu procesul shell.

Subliniem că în sistem pot fi prezente mai multe programe de tip Terminal și mai multe programe de tip shell, care pot fi asociate în diverse moduri.

Regulile de specificare a obiectelor sistemului de fișiere în Linux sunt asemănătoare celor din Windows (inclusiv cele referitoare la directorul curent al unui proces, semnificația lui '.' și '..', utilizarea de măști), cu câteva mențiuni:

- Într-o comandă shell, caracterul '~' scris la începutul unei căi este înlocuit de shell cu directorul home al proprietarului shell-ului (directorul home este un atribut al utilizatorilor); de exemplu, '~/Desktop/work' se expandează la '/home/dragulici/Desktop/work'.
- Linux este case sensitive, numirea obiectelor din sistemul de fișiere se face printr-un singur string (nu se implementează distinct numele și extensia), '.' este un caracter obișnuit, componentele unui specificator se separă prin '\' - am văzut mai devreme.
- Instanța Linux poate opera cu mai multe discuri / partiții, fiecare având arborescență sa de directoare și fișiere, dar integrează logic (montează) toate aceste arborescențe într-una singură; astfel, instanța Linux expune utilizatorilor o singură arborescență de directoare și fișiere în care pot fi parcuse căi.
De aceea, în Linux specificatorii au doar cale și nume (nu mai este specificat discul) și nu este implementat conceptul de disc curent.
- Diverse programe shell pot avea diverse limbi ale liniei de comandă, inclusiv diverse reguli de a scrie măști.

Comenzi shell discutate mai devreme sunt valabile și acum și afectează procesul shell unde sunt date. Dacă avem mai multe procese shell aflate simultan în execuție (în ferestre separate), ele sunt independente între ele și au propriul lor istoric de comenzi. Acum, comanda 'tty' afișază, ca terminal de control al shell-ului, un pseudoterminal mapat pe fereastra procesului Terminal în care rulează shell-ul.

Exemplu:

```
$tty  
/dev/pts/0  
$
```

În cele ce urmează, dacă nu se specifică altceva, exemplificările vor fi efectuate în ferestre shell pe Desktop iar rezultatul copiat în acest document cu copy/paste din interfața grafică.

De asemenea, prompterul setat va fi '\$'.

Prezentăm câteva comenzi prin care un proces shell își poate afișa sau schimba directorul curent sau poate afișa conținutul unui director (prin 'fișier', 'director' sau 'mască' înțelegem specificări ce pot conține și căi):

pwd

- se afișază directorul curent al procesului shell

cd director

- se schimbă directorul curent al procesului shell

ls

ls director

ls mască

ls fișier

- se afișază conținutul directorului curent al procesului shell,
respectiv conținutul directorului specificat,
respectiv obiectele desemnate de mască,
respectiv fișierul specificat.

Lista conține doar numele obiectelor, mai multe pe o linie.

Dacă dorim ca 'ls' să afișeze și alte informații, putem adăuga opțiuni:

- l
 - afișază obiectele cu detalii și câte unul pe o linie; sunt afișate: tipul obiectului ('-' = fișier obișnuit, 'd' = director), drepturile de acces, numărul de legături fizice (nume) ale obiectului, proprietarul și grupul proprietar, dimensiunea în octeți, momentul ultimei modificări a conținutului, numele obiectului din directorul afișat (detalii vom prezenta mai târziu)
- a
 - afișază și obiectele al căror nume începe cu '.' (ascunse)
- d
 - tratează o comandă de tip 'ls director' ca o comandă de tip 'ls fișier', adică nu afișază obiectele - copil ale directorului ci chiar obiectul director specificat

Opțiunile de mai sus pot fi prezente mai multe o dată, în orice ordine, scrise separat (de exemplu, '-l -a', '-a -l -d', etc.) sau concatenate și cu un singur '-' (de exemplu '-la', '-ald', etc.). Comanda 'ls' este externă (shell-ul lansează un program utilitar ca proces copil) iar acest tip de scriere a opțiunilor (separate, concatenate, în diverse ordini) este specific și altor programe utilitare.

Exemplu:

```
$pwd  
/home/dragulici/Desktop/work  
$cd ../../..  
$pwd  
/home/dragulici  
$cd Desktop/../../dragulici/./Desktop/..  
$pwd  
/home/dragulici  
$ls Desktop/work  
dir1  fis1.txt  
$ls -l Desktop/work  
total 8  
drwxrwxr-x 3 dragulici dragulici 4096 Oct 14 23:03 dir1  
-rw-rw-r-- 1 dragulici dragulici    9 Oct 14 23:03 fis1.txt  
$ls -la Desktop/work  
total 16  
drwxrwxr-x  3 dragulici dragulici 4096 Oct 14 23:03 .  
drwxr-xr-x 12 dragulici dragulici 4096 Oct 12 13:46 ..  
drwxrwxr-x  3 dragulici dragulici 4096 Oct 14 23:03 dir1  
-rw-rw-r--  1 dragulici dragulici    9 Oct 14 23:03 fis1.txt
```

```
$ls -la Desktop/work/dir1
total 24
drwxrwxr-x 3 dragulici dragulici 4096 Oct 14 23:10 .
drwxrwxr-x 3 dragulici dragulici 4096 Oct 14 23:03 ..
drwxrwxr-x 2 dragulici dragulici 4096 Oct 14 23:04 dir2
-rw-rw-r-- 1 dragulici dragulici     4 Oct 14 23:10 file
-rw-rw-r-- 1 dragulici dragulici     9 Oct 14 23:03 fis2.txt
-rw-rw-r-- 1 dragulici dragulici    6 Oct 14 23:10 fisier.c
$ls -la Desktop/work/dir1/fi*
-rw-rw-r-- 1 dragulici dragulici 4 Oct 14 23:10 Desktop/work/dir1/file
-rw-rw-r-- 1 dragulici dragulici 9 Oct 14 23:03 Desktop/work/dir1/fis2.txt
-rw-rw-r-- 1 dragulici dragulici 6 Oct 14 23:10 Desktop/work/dir1/fisier.c
$ls -la Desktop/work/dir1/fi*.*
-rw-rw-r-- 1 dragulici dragulici 9 Oct 14 23:03 Desktop/work/dir1/fis2.txt
-rw-rw-r-- 1 dragulici dragulici 6 Oct 14 23:10 Desktop/work/dir1/fisier.c
$ls -la Desktop/work/dir1/fi*.?
-rw-rw-r-- 1 dragulici dragulici 6 Oct 14 23:10 Desktop/work/dir1/fisier.c
```

```
$ls -la Desktop/work/dir1/fis2.txt
-rw-rw-r-- 1 dragulici dragulici 9 Oct 14 23:03 Desktop/work/dir1/fis2.txt
$ls -la Desktop/work/dir1/dir2
total 12
drwxrwxr-x 2 dragulici dragulici 4096 Oct 14 23:04 .
drwxrwxr-x 3 dragulici dragulici 4096 Oct 14 23:10 ..
-rw-rw-r-- 1 dragulici dragulici 13 Oct 14 23:04 fis3.txt
$ls -lad Desktop/work/dir1/dir2
drwxrwxr-x 2 dragulici dragulici 4096 Oct 14 23:04 Desktop/work/dir1/dir2
```

Comenzi shell pentru crearea și ștergerea directoarelor:

`mkdir director`

- crează directorul specificat

`rmdir director`

– șterge directorul specificat; se poate doar dacă este gol; deci, pentru a șterge o arborescență, trebuie să ștergem de la periferie spre rădăcină

Se poate șterge o arborescență printr-o singură comandă, de forma:

`rm -r director`

Exemplu:

```
$pwd  
/home/dragulici/Desktop/work  
$ls  
dir1  fis1.txt  
$mkdir a  
$ls  
a  dir1  fis1.txt  
$mkdir a/b  
$ls  
a  dir1  fis1.txt  
$ls a  
b  
$rmdir a  
rmdir: failed to remove 'a': Directory not empty  
$rmdir a/b  
$rmdir a  
$ls  
dir1  fis1.txt
```

```
$mkdir a  
$mkdir a/b  
$rm -r a  
$ls  
dir1  fis1.txt
```

Câteva comenzi shell pentru fișiere:

`cat fișier`

- se afișază conținutul fișierului

`cat fișier1 ... fișierN`

- se afișază succesiv (concatenat) conținuturile fișierelor
'fișier1', ..., 'fișierN'

`cp fișier1 fișier2`

- se copiază fișierul 'fișier1' în 'fișier2' (dacă 'fișier2' nu există,
se crează, altfel se suprascrie)

`cp fișier director`

- se crează o copie a fișierului 'fișier' în directorul (existent) 'director',
având același nume

`cp mască director`

`cp -r mască director`

- se crează copii ale fișierelor specificate de 'mască' în directorul (existent)
'director', având același nume; cu opțiunea '-r' se copiază și directoarele
(cu tot cu arborescențele având originea în ele)

`cp -r director1 director2`

se copiază arborescență cu originea în 'director1' într-o arborescență cu
originea în 'director2', dacă 'director2' nu există, sau într-un subarbore
plasat în 'director2', cu același nume al originii, dacă 'director2' există.

`mv fișier1 fișier2`

- se redenumește și/sau mută 'fișier1' în 'fișier2' (dacă 'fișier2' diferă de 'fișier1' doar prin nume, este doar o redenumire, iar dacă diferă calea, este o mutare (și, eventual, o redenumire))

`mv mască director`

- se mută toate obiectele specificate de mască în directorul (existent) 'director', cu păstrarea numelor; dacă vreunul dintre obiecte este director, se mută toată arborescența cu originea în el; observăm că nu este necesară opțiunea '-r'

`mv director1 director2`

- se redenumește / mută arborescența cu originea în 'director1' într-o arborescență cu originea în 'director2', dacă 'director2' nu există, sau ca un subarbore plasat în 'director2', cu același nume al originii, dacă 'director2' există; observăm că nu este necesară opțiunea '-r'

rm fișier

rm mască

rm -r mască

rm -r director

- se șterg obiectele specificate de sursă; dacă vreunul dintre obiecte este director, se șterge doar dacă este prezentă opțiunea '-r' și atunci se șterge împreună cu toata arborescența cu originea în el

Obs: Vom vedea că obiectele sistemului de fișiere Linux pot avea mai multe nume (legături fizice) iar comenziile 'mv', 'rm' înlocuiesc / șterg legături fizice; obiectul este șters doar atunci când nu mai are legături fizice și nu este deschis de vreun proces.

Exemplu:

```
$ls -l
total 12
drwxrwxr-x 2 dragulici dragulici 4096 Oct 16 22:02 dir
-rw-rw-r-- 1 dragulici dragulici     9 Oct 16 21:43 fis1.txt
-rw-rw-r-- 1 dragulici dragulici     9 Oct 16 21:43 fis2.txt
$cat fis1.txt
Hello !

$cat fis2.txt
Salut !

$ls dir
$cat fis1.txt fis1.txt fis2.txt fis1.txt
Hello !

Hello !

Salut !

Hello !
```

```
$cp fis1.txt dir/file.c
$ls dir
file.c
$cp fis1.txt dir
$ls dir
file.c  fis1.txt
$mkdir dirnou
$ls
dir  dirnou  fis1.txt  fis2.txt
$cp * dirnou
cp: -r not specified; omitting directory 'dir'
cp: -r not specified; omitting directory 'dirnou'
$ls dirnou
fis1.txt  fis2.txt
```

```
$mkdir fold  
$cp -r dir* fold  
$ls fold  
dir  dirnou  
$ls fold/dir  
file.c  fis1.txt  
$ls fold/dirnou  
fis1.txt  fis2.txt
```

```
$ls
dir dirnou fis1.txt fis2.txt fold
$mv fis1.txt doc.txt
$ls
dir dirnou doc.txt fis2.txt fold
$mkdir foldnou
$mv d* foldnou
$ls
fis2.txt fold foldnou
$ls foldnou
dir dirnou doc.txt
$ls foldnou/dir
file.c fis1.txt
$ls foldnou/dirnou
fis1.txt fis2.txt
```

```
$ls fold
dir dirnou
$ls fold/dir
file.c fis1.txt
$ls fold/dirnou
fis1.txt fis2.txt
$rm -r fold*
$ls
fis2.txt
```

Vom vedea că procesele care execută diverse aplicații pot accesa obiectele sistemului de fișiere (fișiere obișnuite, directoare, dar și terminale, care sunt implementate ca obiecte software de tip fișier special caracter) prin descriptori numerici.

Descriptorii sunt alocați de apelurile sistem care deschid fișierele la operații pentru procesul respectiv ('open()') și sunt eliberați de apelurile care închid fișierele pentru acest proces ('close()').

Descriptorii au sens la nivel de proces - de exemplu, descriptorul 5 poate duce la un fișier pentru un proces, la alt fișier pentru alt proces, sau poate să nu aibă sens (să fie închis) pentru un alt proces.

În general, când un proces lansează procese copil, aceștia moștenesc de la procesul părinte aceiași descriptori către aceleași fișiere dar, ulterior, și-i pot redirecta sau închide fără să afecteze descriptorii altor procese (inclusiv ai părintelui).

Descriptorii 0, 1, 2 s.n. standard input, standard output, resp. standard error.

Există funcții C predefinite care operează cu acești descriptori:

'scanf()' citește de la standard input

'printf()' scrie la standard output

'perror()' scrie la standard error

Descriptorii 0, 1, 2 pot duce la diverse fișiere sau pot fi închiși.

La shell, ei duc la terminalul său de control - deci, pentru shell, funcțiile 'scanf()', 'printf()', 'perror()' lucrează cu terminalul.

La executarea obișnuită a unei comenzi shell externe, procesul copil lansat va moșteni același terminal de control și descriptorii 0, 1, 2 ducând către el.

Astfel, pentru procesul care execută comanda, 'scanf()', 'printf()', 'perror()' vor lucra cu același terminal de unde s-a dat comanda.

Vom vedea că putem da comenziile shell externe a.î. procesele copil să fie create cu descriptorii standard ducând către alte obiecte ale sistemului de fișiere sau către alte procese (via niște fișiere tub) - atunci, pentru ele, funcțiile 'scanf()', 'printf()', 'perror()' nu vor lucra cu terminalul.

Multe programe utilitare afișază rezultatele procesării pe standard output - de exemplu: 'whoami', 'tty', 'pwd', 'env', 'ls', 'cat' (inclusiv variantele cu argumente și opțiuni). Comanda 'cp fișier1 fișier2' operează cu cele două fișiere prin alți descriptori decât 0, 1, 2.

De obicei, programele utilitare afișază mesajele de eroare pe standard error.

Întrucât, de obicei, standard output și standard error sunt terminalul, rezultatele și erorile sunt afișate pe terminal.

Unele programe utilitare care citesc dintr-un fișier de intrare pot fi lansate fără specificarea acestui fișier ca argument și atunci ele vor citi de la standard input. De obicei, acesta este terminalul iar procesul va aștepta tastarea fiecărei linii iar la ENTER o va citi și prelucra. Se pot seta combinații de taste pentru a specifica sfârșitul de fișier de la tastatură, de obicei este Ctrl-d.

De exemplu, 'cat' fără argumentele - fișier citește de la standard input (dar întotdeauna afișază la standard output):

```
$cat  
abc  
abc  
defg  
defg  
$
```

(după ce am tastat 'cat' și ENTER, el a așteptat linii de input, am tastat 'abc' și ENTER, el a citit și afișat 'abc' și linie nouă, apoi a așteptat alte linii de input, am tastat 'defg' și ENTER, el a citit și afișat 'defg' și linie nouă, apoi a așteptat alte linii de input, am tastat Ctrl-d iar procesul 'cat', detectând EOF în fișierul de intrare, s-a terminat și am repremit prompterul shell).

Mai prezentăm câteva comenzi shell utile.

Specificarea '...' (de exemplu 'opțiuni...') arată că pot fi mai multe.

Parantezele '()' arată că este optional.

În general, utilitarele de mai jos permit lipirea opțiunilor de o literă: de exemplu, în loc de '-i -v' să scriem '-iv'. De asemenea, afișarea lor este pe standard output.

more fișier

- se afișază conținutul fișierului, în mod paginat:
 - se afișază până se umple spațiul de afișare disponibil (fereastra),
 - apoi se afișază procentul din fișier parcurs (sub forma '-More-(nr%)')
 - și se așteaptă comenzi (taste); se pot da următoarele comenzi:
 - ENTER - se mai afișaza un rând
 - SPACE - se mai afișază o pagină
 - q - se termină procesul (se revine la shell)

less fișier

- se afișază conținutul fișierului, în mod paginat, dar se poate naviga în fișier cu UP, DOWN și se ieșe cu 'q'

```
diff fișier1 fișier2
```

```
diff -r director1 director2
```

– în primul caz, se compară conținuturile celor două fișiere;

în al doilea caz, se compară arborescențele cu originile în cele două directoare;

în ambele cazuri, se afișază diferențele;

dacă nu se afișaza nimic, fișierele, resp. arborescențele, au conținuturi identice

grep [opțiuni...] *șir* [fișier]

– afișaza liniile din 'fișier' care includ (ca subșir) *șirul 'șir'*

grep [opțiuni...] -f fișier1 [fișier]

– afișaza liniile din 'fișier' care includ (ca subșir) vreo linie din 'fișier1'

În ambele cazuri, dacă 'fișier' lipsește, conținutul este citit de la standard input (de obicei terminalul); se pot folosi opțiunile:

-i

ignoră case-ul (distincția între litere mari și mici)

-v

selectează liniile de input care nu conțin *șirul*

-x

selectează liniile de input care coincid cu (nu doar includ) *șirul*

-c

nu afișaza liniile selectate ci numărul lor

-m nr

citirea se oprește după 'nr' liniii selectate;

Obs: dacă folosim opțiunea '-r' iar în loc de 'fișier' este un director, sunt procesate recursiv fișierele din arborescența cu originea în el; cu '-r' dar fără director, este considerat directorul curent.

head [-n nr] [fișier]

tail [-n nr] [fișier]

– afișază primele, respectiv ultimele, 'nr' linii din 'fișier' (dacă în fișier sunt mai puține linii, se afișază doar acelea); dacă 'nr' lipsește, se consideră 10; dacă 'fișier' lipsește, se consideră standard input

sort [opțiuni] [fișier]

– afișază liniile din 'fișier', sortate lexicografic (ținând cont și de opțiuni); dacă 'fișier' lipsește, se consideră standard input;

opțiuni:

-f

ignoră case-ul, asimilând literele mici cu litere mari

-r

sortează lexicografic descrescător

uniq [opțiuni] [fișier]

– afișază liniile din 'fișier', eliminând repetițiile succesive; dacă 'fișier' lipsește, se consideră standard input;

opțiuni:

-i

ignoră case-ul

-w nr

compară doar primele 'nr' caractere din fiecare linie

wc [opțiuni] [fișier]

– calculează numărul de linii, numărul de cuvinte (șiruri nevide maximale de caractere ne-albe) și numărul de caractere din 'fișier'; fără opțiuni, afișază toate aceste numere; cu opțiuni, afișază doar numerele care corespund opțiunilor prezente; dacă 'fișier' lipsește, se consideră standard input; opțiuni:

-l

afișază numărul de linii

-w

afișază numărul de cuvinte

-c

afișază numărul de octeți

-m

afișază numărul de caractere (în funcție de codarea locală, un caracter poate ocupa mai mulți octeți)

`echo [opțiuni...] [șiruri...]`

– afișază șirurile

opțiuni:

`-e`

secvențele escape cu backslash din șiruri sunt interpretate

`-n`

nu afișază capul de linie de la sfârșit (prompterul shell apare pe aceeași linie, după ultimul șir)

Obs: comanda este utilă în combinație cu alte facilități; de exemplu, dacă șirurile conțin construcții de forma '\$variabilă', unde 'variabilă' este o variabilă de environment a shell-ului, shell-ul înlocuiește această construcție din linia de comandă cu valoarea variabilei (care este un șir) și apoi lansează comanda - astfel, cu 'echo \$variabilă' vom afișa valoarea variabilei 'variabilă'

export variabilă=șir

- setează variabila de environment a shell-ului 'variabilă' cu sirul 'șir' (poate fi și sirul vid), făcând-o și exportabilă (moștenibilă de către procesele copil ale shell-ului); numele varabilelor pot conține litere mari sau mici, cifre sau '_'; sirul poate conține și construcții de forma '\$nume', unde 'nume' corespunde unei variabile de environment a shell-ului existentă, iar construcția va fi înlocuită cu valoarea variabilei înainte de executarea comenzi

export PS1=șir

- caz particular al comenzi dinainte, care setează prompterul shell să fie sirul respectiv; pentru a putea include în sir caractere speciale shell, cum ar fi '\$' (la întâlnirea lui, shell-ul face o substituție, am văzut), este util ca sirul să fie pus între caractere apostrof - de exemplu: `export PS1='$'`

unset variabilă

- elimină din environment-ul shell-ului variabila 'variabilă'

clear

- șterge ecranul

du [opțiuni...] [fișiere...]

- calculează și afișază dimensiunile fișierelor specificate; dacă printre ele este vreun director, calculează recursiv dimensiunea arborescenței cu originea în el; dacă 'fișiere...' lipsește, se consideră directorul curent
- opțiuni:
 - b

afișază dimensiunea în octeți (altfel, afișază în blocuri de o dimensiune implicită)

- a
 - afișază dimensiunea pentru fiecare fișier în parte (altfel, afișază doar pentru directoare)

- c
 - afișază un total general

- h
 - afișază în format "human readable" (ex: 1K 234M 2G)

df

- afișază gradul de utilizare a spațiului pe fiecare partitie montată logic în sistem (partitiile sunt specificate ca fișiere speciale bloc)

`man [opțiuni] nume`

`info [opțiuni] nume`

– afișază informații despre 'nume'; poate fi numele unei comenzi shell sau funcții C; în conținutul afișat se poate naviga cu UP, DOWN și se poate ieși cu 'q';

putem da ca opțiune un număr de la 1 la 9, pentru a specifica căutarea numelui în secțiunea respectivă de manual (unelte nume pot apărea în mai multe secțiuni); secțiunile sunt:

- 1 Executable programs or shell commands
- 2 System calls (functions provided by the kernel)
- 3 Library calls (functions within program libraries)
- 4 Special files (usually found in /dev)
- 5 File formats and conventions, e.g. /etc/passwd
- 6 Games
- 7 Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)
- 8 System administration commands (usually only for root)
- 9 Kernel routines [Non standard]

Exemple: 'man ls', 'man strcpy', 'man man', 'man printf' (se referă la o comandă shell), 'man 3 printf' (se referă la o funcție C)

Exemplu:

```
$cat f1
Hello !
$cat f2
Hello again !
$diff f1 f1
$cp f1 f3
$diff f1 f3
$diff f1 f2
1c1
< Hello !
---
> Hello again !
```

```
$ls a
f  f1  f2  x
$cat a/f
Salut !
$cat a/f1
Hello !
$cat a/f2
Hello again !
$ls a/x
$ls b
f  f2
$cat b/f
Salut din nou !
$cat b/f2
Hello again !
$diff -r a b
diff -r a/f b/f
1c1
< Salut !
---
> Salut din nou !
Only in a: f1
Only in a: x
```

Exemplu:

```
$cat f1
abc12
3def
abc
def56
ghij
ghi
$cat f2
abc
def
$grep abc f1
abc12
abc
$grep -v abc f1
3def
def56
ghij
ghi
```

```
$grep -x abc f1
abc
$grep -xc abc f1
1
$grep Abc f1
$grep -i Abc f1
abc12
abc
$grep -f f2 f1
abc12
3def
abc
def56
$grep -m 2 -f f2 f1
abc12
3def
```

Exemplu:

```
$head f1
abc12
3def
abc
def56
ghij
ghi
$head -n 3 f1
abc12
3def
abc
$tail -n 3 f1
def56
ghij
ghi
```

Exemplu:

```
$cat fis
abc12
3def
3de
abc
abc
3def
ghij
ghi
```

```
$sort fis
3de
3def
3def
abc
abc
abc12
ghi
ghij
$sort -r fis
ghij
ghi
abc12
abc
abc
3def
3def
3de
```

```
$uniq fis
abc12
3def
3de
abc
3def
ghij
ghi
$uniq -w 3 fis
abc12
3def
abc
3def
ghij
```

Exemplu:

```
$cat fis1  
Ana    are multe  
      mere
```

.

```
$wc fis1  
4 5 27 fis1  
$wc -l fis1  
4 fis1  
$wc -cl fis1  
4 27 fis1
```

Exemplu:

```
$echo  
  
$echo abc      def gh  
abc def gh  
$echo -n abc      def gh  
abc def gh$echo Lista este: $PATH  
Lista este: /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:  
/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

Exemplu:

```
$export a=123xyz
$echo $a
123xyz
$export a=$a$a.ijk
$echo $a
123xyz123xyz.ijk
$export a=
$echo $a

$unset a
```

(în final, dacă dăm 'env', nu vom mai vedea 'a' intre variabilele de environment)

Exemplu:

```
$ls
a b fis
$cat fis
Hello !
$ls a
c f1
$cat a/f1
123
$ls a/c
f2
$cat a/c/f2
abcdef
$ls b
f3
$cat b/f3
ijkl
```

```
$du
8 ./b
8 ./a/c
16 ./a
32 .
$du -b
4101 ./b
4103 ./a/c
8203 ./a
16408 .
$du -b a b fis
4103 a/c
8203 a
4101 b
8 fis
```

```
$du -ba
5 ./b/f3
4101 ./b
4 ./a/f1
7 ./a/c/f2
4103 ./a/c
8203 ./a
8 ./fis
16408 .
$du -bc
4101 ./b
4103 ./a/c
8203 ./a
16408 .
16408 total
$du -bch
4,1K ./b
4,1K ./a/c
8,1K ./a
17K .
17K total
```

Putem lansa un shell din shell tastând comanda 'sh' sau specificatorul unui program shell din sistem (de exemplu, '/bin/bash'). Atunci, vom avea un proces shell copil al procesului shell parinte, care va rula în foreground la același terminal, în timp ce parintele va aștepta terminarea lui. Setările pe care îl le vom face vor fi locale acestui proces (de exemplu, forma prompterului) iar la terminarea lui (cu comanda 'exit'), shell-ul parinte își va continua rularea în foreground la același terminal, va afișa un nou prompter și va aștepta noi comenzi.

Exemplu (am lansat 3 shell - în - shell):

```
$sh
$export PS1=':'
:sh
:export PS1='>'
>sh
>export PS1='-'-
-tty
/dev/pts/0
-exit
>exit
:exit
$
```

Putem afla informații despre procesele curente cu comanda:

`ps [opțiuni]`

fără opțiunile '-e', '-t', '-U', '-u' de mai jos, se afișază procesele de la terminalul de unde s-a dat comanda; câteva opțiuni:

`-e`

afișază toate procesele din instanța Linux (indiferent de terminal)

`-t terminale...`

afișază toate procesele al căror terminal de control este în lista de terminale (listă de specificatori de fișier - terminal)

`-U utilizatori...`

afișază toate procesele al căror proprietar real este în lista de utilizatori (listă de nume sau UID)

`-u utilizatori...`

afișază toate procesele al căror proprietar efectiv este în lista de utilizatori (listă de nume sau UID)

`-f`

afișază informații complete despre procese

`f`

afișază arborescența de procese (ASCII art)

-o format

afișază informațiile care corespund formatului; acesta este o listă de specificări separate prin virgulă sau spațiu, de exemplu: 'ps -o uid,pid,ppid'; vor fi afișate doar coloanele care corespund acestor informații; câteva specificări:

pid = identificatorul procesului

ppid = identificatorul procesului părinte

pri = prioritatea procesului (număr mare înseamnă prioritate mică)

tid = identificatorul thread-ului kernel-ului

s = starea (pentru procese: O=Nonexistent, A=Active, W=Swapped,

I=Idle (waiting for startup), Z=Canceled, T=Stopped; pentru thread-uri

kernel: O=Nonexistent, R=Running, S=Sleeping, W=Swapped,

Z=Canceled, T=Stopped)

stime = momentul lansării procesului

sz = dimensiunea imaginii core a procesului în unități de 1K

time = timpul total de execuție a procesului

tty = terminalul de control al procesului (- = nu are)

uid = proprietarul real al procesului

euid = proprietarul efectiv al procesului

wchan = evenimentul așteptat de procesul sau thread-ul nucleu blocat

Exemplu:

```
$ps
 PID TTY          TIME CMD
 3785 pts/1    00:00:00 bash
 4405 pts/1    00:00:00 ps
```

Obs: la terminalul unde am dat comanda, rulează shell-ul căruia i-am dat comanda și procesul 'ps' lansat de comandă (la momentul căutării proceselor, încă mai rula și 'ps').

Exemplu:

```
$ps -u dragulici
 935 ?      00:00:00 systemd
...
 3645 pts/0    00:00:00 bash
 3656 pts/0    00:00:00 man
 3666 pts/0    00:00:00 pager
 3785 pts/1    00:00:00 bash
 4098 ?      00:00:05 chrome
 4111 ?      00:00:00 chrome
 4447 pts/1    00:00:00 ps
```

Obs: am omis unele procese afișate; 'systemd' și 'chrome' nu au termini de control; pe desktop, aveam 2 ferestre Terminal + shell, într-o fereastră rula nefinalizat 'man' (procese 'bash', 'man', 'pager'), în altă fereastră am dat comanda 'ps' (procese 'bash', 'ps').

Exemplu:

```
$ps -t pts/0
    PID TTY          TIME CMD
 3645 pts/0    00:00:00 bash
 3656 pts/0    00:00:00 man
 3666 pts/0    00:00:00 pager
```

Obs: am dat comanda din fereastra corespunzătoare lui 'pts/1'.

Exemplu:

```
$ps -o uid,pid,ppid
   UID     PID     PPID
 1000    3785    3638
 1000    4540    3785
```

Obs: sunt afișate procesele 'bash' și 'ps' de la terminalul (fereastra) de unde am dat comanda.

Prezentăm în continuare câteva facilități shell utile:

- Istoricul comenziilor: fiecare proces shell reține istoricul comenziilor pe care le-a primit; cu UP, DOWN, putem aduce pe prompter comenzi din acest istoric, le putem edita (modifica), apoi cu ENTER le putem da ca și comenzi noi (se vor adăuga la istoric); după ce am editat o comandă veche, nu este necesar să ducem cursorul la sfârșitul liniei ca să tastăm ENTER.
- Autocompletarea liniei de comandă: la introducerea unei comenzi shell, dacă trebuie scris un specificator (de fișir, director, mască), putem tasta primele caractere ale numelui curent de pe cale și apoi TAB - atunci, shell-ul va completa numele până la sfârșit sau până întâlnește o ambiguitate (între numele care figurează în directorul către care duce partea deja scrisă a căii); în caz de ambiguitate, putem tasta în continuare câteva caractere și apoi TAB - shell-ul va completa până la sfârșit sau până la următoarea ambiguitate. Astfel, putem tasta specificatorii mai repede.

- Despecializarea caracterelor speciale: shell-ul consideră unele caractere din linia de comandă ca fiind speciale: SPACE, \$, ', ", \, #, >, <, &, |, etc., - la întâlnirea lor, le elimină și efectuează o anumită prelucrare (intropolare) liniei de comandă înainte de a o executa (de exemplu, construcțiile '\$ume' sunt înlocuite cu valoarea variabilei de environment 'ume'); dacă dorim ca aceste caractere să fie considerate ca atare (cu valoarea lor literală), le putem despecializa, în felul următor:

pentru a despecializa un caracter, îl precedăm de backslash;

pentru a despecializa un sir, îl includem între caractere apostrof; sirul nu poate conține apostroful, nici măcar precedat de backslash;

dacă includem sirul între caractere ghilimele, sunt despecializate caracterele lui, cu excepția \$, ' , \, !; \ își păstrează semnificația specială doar dacă este urmat de unul dintre \$, ' , " , \ sau NEWLINE.

Exemplu:

```
$echo \\  
\\  
$echo \$  
$  
$echo \a a  
a a  
$ls  
$cat abc def ghi  
cat: abc: No such file or directory  
cat: def: No such file or directory  
cat: ghi: No such file or directory  
$cat 'abc def' ghi  
cat: 'abc def': No such file or directory  
cat: ghi: No such file or directory  
$cat abc' 'def ghi  
cat: 'abc def': No such file or directory  
cat: ghi: No such file or directory
```

```
$export a=def
$cat abc $a ghi
cat: abc: No such file or directory
cat: def: No such file or directory
cat: ghi: No such file or directory
$cat 'abc $a' ghi
cat: 'abc $a': No such file or directory
cat: ghi: No such file or directory
$cat "abc $a" ghi
cat: 'abc def': No such file or directory
cat: ghi: No such file or directory
$unset a
```

- Putem lansa comenzi shell externe a.î., deși procesele - copil lansate vor moșteni terminalul de control, vor avea descriptorii standard 0, 1, 2 redirectați spre alte fișiere - deci, în procesele - copil, funcțiile 'scanf()', 'printf()', ' perror()' vor lucra cu aceste fișiere, nu cu terminalul; mai exact, dacă 'comm' este o comandă oarecare (ce poate conține mai multe cuvinte) iar 'fis' este specificatorul unui fișier, atunci:

comm < fis

lansează 'comm' cu standard input către 'fis' în citire;

comm > fis

lansează 'comm' cu standard output către 'fis' în suprascriere (dacă fișierul nu există, este creat, iar dacă există, i se sterge vechiul conținut);

comm >> fis

lansează 'comm' cu standard output către 'fis' în apendare (dacă fișierul nu există, este creat, iar dacă există, nu i se sterge vechiul conținut ci se va scrie la sfârșit);

comm 2> fis

lansează 'comm' cu standard error către 'fis' în suprascriere;

comm 2>> fis

lansează 'comm' cu standard error către 'fis' în apendare.

Se pot cere mai multe redirectări simultan (nu contează ordinea), de exemplu:
comm < fis1 > fis2

Exemplu:

```
$echo abc > f
$echo def >> f
$echo dhi >> f
$cat f
abc
def
dhi
$cat < f
abc
def
dhi
$cat < f > g
$cat g
abc
def
dhi
```

```
$wc -l f
3 f
$wc -l < f
3
$wc -l < f > g
$cat g
3
$cat bazaconie 2> h
$cat h
cat: bazaconie: No such file or directory
```

Observații:

1. Ambele comenzi 'cat f' și 'cat < f' au citit din 'f' și au afișat pe terminal; prima a citit din 'f' printr-un descriptor $\neq 0, 1, 2$ (pe 0 l-a păstrat spre terminal), a doua a citit din 'f' prin descriptorul 0 (standard input, 'scanf()'); ambele comenzi au scris pe terminal prin descriptorul 1 (standard output, 'printf()').
2. Comanda 'cat < f > g' a copiat 'f' în 'g', asemenei unei comenzi 'cp f g', dar prin alti descriptori; prima comandă a citit din 'f' prin descriptorul 0 și a scris în 'g' prin descriptorul 1, în timp ce a doua comandă ar fi folosit descriptori $\neq 0, 1, 2$ (pe 0, 1, 2 i-ar fi păstrat pe terminal).
3. Cu comanda 'cat bazaconie 2> h' am încercat să afișăm un fișier inexistent ('bazaconie'); comanda a scris un mesaj de eroare prin descriptorul 2 (standard error, ' perror()') care a fost redirectat spre fișierul 'h', deci mesajul a fost scris în 'h', iar de acolo a fost afișat ulterior cu comanda 'cat h'.

- Putem lansa succesiv mai multe comenzi 'com1', 'com2'..., 'comn' (pot avea mai multe cuvinte), scriind:

com1 ; com2 ; ... ; comn

Exemplu:

```
$echo abc def > f ; cat f ; wc -w f  
abc def  
2 f
```

- Putem lansa în paralel în foreground mai multe comenzi 'com1', 'com2'..., 'comm' (pot avea mai multe cuvinte), a.î. standard output-ul comenzi i să fie conectat printr-un tub la standard input-ul comenzi i + 1 ($i = 1, \dots, n-1$), scriind:

```
com1 | com2 | ... | comm
```

Tuburile sunt fișiere de un tip special, care funcționează ca o coadă (vom vedea mai târziu).

Exemplu:

```
$echo abc > f ; echo ab >> f ; echo abc >> f ; echo ddd >> f ; echo aaa >> f  
$cat f  
abc  
ab  
abc  
ddd  
aaa  
$cat f | grep a | sort | uniq | wc -l  
3
```

Observații:

1. 'cat f' a trimis liniile din 'f' lui 'grep a', acesta a ales cele 4 linii care conțin 'a' și le-a trimis lui 'sort', acesta le-a sortat lexicografic crescător (cele două linii 'abc' au ajuns una lângă alta) și le-a trimis lui 'uniq', acesta a eliminat repetițiile successive (a dispărut o linie 'abc') și le-a trimis lui 'wc -l', acesta a numărat liniile și a scris numărul pe standard output care, nefiind redirectat, a fost pe terminal.
2. Comenzile de forma com1 | com2 | ... | comn s.n. filtre și prin ele putem cere prelucrări complexe prin combinarea de comenzi simple.
3. Comenzile utile în filtre sunt acelea care, măcar pentru o anumită combinație a opțiunilor și argumentelor, citesc de la standard input și scriu la standard output; acestea s.n. comenzi filtru; de exemplu, comenziile 'cat', 'grep', 'head', 'tail', 'sort', 'uniq' 'wc' sunt comenzi filtru.

Comanda 'cp f1 f2' nu este comandă filtru, deoarece întotdeauna citește și scrie (din/în cele două fișiere) prin descriptori $\neq 0,1,2$. Putem să o scriem într-un filtru dar informația transmisă prin filtru de la o comandă la alta nu va trece prin ea ('cp f1 f2' este lansat de shell cu descriptorul 0 redirectat la tubul din stânga și cu descriptorul 1 redirectat la tubul din dreapta dar nu folosește acești descriptori).

Dintronu motiv asemănător, comenziile 'ls' și 'echo' sunt utile doar la începutul filtrului (ele nu citesc de la descriptorul 0 dar scriu la descriptorul 1).

4. Se pot adăuga filtrului redirectări către fișiere dar operatorii '<', '>', '>>', etc. au prioritate față de '|'; pentru a schimba asocierea, se pot folosi paranteze '()'.

Exemplu:

```
$echo abc > f ; echo abc >> f  
$uniq | wc -l < f  
2  
123  
456  
789  
$(uniq | wc -l) < f  
1
```

Observații:

Filtrele au lansat procesele 'uniq' și 'wc -l' în paralel, conectate prin tub.

În cazul 'uniq | wc -l < f', redirectarea '< f' s-a asociat cu 'wc -l', deci comunicarea a fost (săgețile corespund tuburilor și am scris deasupra descriptorii redirectați la ele din procese):

```
0      1          0      1  
terminal ----> uniq ----> (nu sunt cititori)  f ----> wc -l ----> terminal
```

Astfel, 'wc' a citit și numărat 1 linie, apoi s-a terminat, iar 'uniq' a așteptat input de la tastatură; textul '123 ... 789' a fost introdus interactiv, urmat de Ctrl - D; atunci, 'uniq' a încercat să-l scrie în tubul fără cititor și a fost terminat (vom vedea mai târziu că un proces care încearcă să scrie într-un tub fără cititor primește semnalul SIGPIPE, al cărui handler implicit este terminarea).

În cazul '(uniq | wc -l) < f', redirectarea '< f' s-a asociat cu filtrul însuși, a cărui intrare este prin 'uniq', deci '< f' s-a asociat cu 'uniq', și atunci comunicarea a fost:

```
0      1  0      1  
f ----> uniq ----> wc -l ----> terminal
```

Deci, 'uniq' a citit cele 2 linii identice din 'f', a eliminat repetiția și a trimis linia rămasă lui 'wc -l', care a numărat și afișat 1 linie. Același efect l-ar fi avut și comanda 'uniq < f | wc -l'.

- Vom vedea mai târziu că atunci când un proces - copil se termină, furnizează procesului - părinte un cod de return întreg, de la 0 la 255. În programul copilului (dacă este în C), se poate specifica această valoarea ca argument al unei instrucțiuni 'return' din 'main()' ('main()' trebuie să returneze 'int') sau ca argument al unui apel 'exit()'; se poate returna orice valoare dar convenția uzuală este 0: succes, ≠0: eșec.

Procesele - copil lansate de shell în urma unor comenzi externe furnizează acestuia codul de return, iar shell-ul poate afișa codul de return al ultimei comenzi executate cu comanda:

```
echo $?
```

Exemplu:

```
$pwd  
/home/dragulici/Desktop/work  
$echo $?  
0  
$cat bazaconie  
cat: bazaconie: No such file or directory  
$echo $?  
1  
$echo $?  
0
```

Obs: Ultimul 'echo \$?' a afișat 0 deoarece ultima comandă dinaintea lui a fost precedentul 'echo \$?' iar acesta a avut succes.

- Am văzut că putem lansa un proces - copil al shell-ului în background, care să ruleze în paralel cu shell-ul (shell-ul să nu aștepte terminarea lui ci să afișeze imediat prompterul, putând prelua o nouă comandă), dacă adăugăm la sfârșitul liniei de comandă &.
- Am văzut, de asemenea, că putem semnala EOF unui proces care citește de la terminal, tastând Ctrl - D. Putem termina procesul care rulează în foreground la terminal, tastând Ctrl - C; vom vedea că atunci procesul va primi semnalul SIGINT, cu codul 2, iar handlerul implicit pentru el este terminarea. Unele proceze au asociat un handler de ignorare pentru SIGINT și nu pot fi terminate cu el (de exemplu, shell-ul însuși).
- Un semnal care termina procesul receptor și care nu poate fi blocat / ignorat este SIGKILL, cu codul 9. El poate fi trimis unui proces căruia îi cunoaștem identificatorul numeric 'pid' cu comanda 'kill -9 pid'. Procesul (shell) expeditor trebuie să aibă același proprietar ca procesul destinatar sau să aibă proprietarul privilegiat 'root'.

De exemplu, dacă un proces se blochează, de la un alt terminal/fereastră îi putem afla identificatorul 'pid' cu o comandă 'ps -u utilizator', apoi îl terminăm cu 'kill -9 pid'.

Exemplu (am lansat un shell copil în background, apoi l-am terminat):

```
$sh &
[1] 7242
$ps
    PID TTY          TIME CMD
  3448 pts/0        00:00:00 bash
  7242 pts/0        00:00:00 sh
  7243 pts/0        00:00:00 ps

[1]+  Stopped                  sh
$kill -9 7242
$ps
    PID TTY          TIME CMD
  3448 pts/0        00:00:00 bash
  7246 pts/0        00:00:00 ps

[1]+  Killed                  sh
$ps
    PID TTY          TIME CMD
  3448 pts/0        00:00:00 bash
  7247 pts/0        00:00:00 ps
```

Pentru dezvoltarea de programe C, distribuțiile de Linux au, de obicei, preinstalat compilatorul 'gcc' (GNU Compiler Collection), apelabil în linia de comandă shell.

Evident, se pot instala și alte compilatoare, inclusiv de tip IDE.

Exemplu de folosire gcc:

```
$ls  
prog.c  
$cat prog.c  
#include <stdio.h>  
int main() {  
    printf("Hello world !\n");  
    return 0;  
}
```

```
$gcc -Wall -o program prog.c  
$ls  
prog.c  program  
$program  
program: command not found  
$./program  
Hello world !
```

Observații:

1. Opțiunea '-Wall' activează afișarea tuturor avertismentelor (all warnings), chiar și a celor mai puțin importante; opțiunea '-o program' specifică faptul că fișierul executabil rezultat (output) va avea numele 'program' (în absența acestei opțiuni, fișierul rezultat va avea numele 'a.out', indiferent de numele fișierului sursă); 'prog.c' este fișierul sursă compilat (poate avea și cale). Opțiunile lui 'gcc' se pot pune în mai multe ordini dar între '-o' și 'program' nu trebuie intercalat altceva.
2. Specificatorii 'program' și 'prog.c' sunt primiți ca argument de 'gcc' și interpretați de apelurile sistem efectuate de acesta și pot fi scriși după toate regulile generale pe care le-am văzut mai devreme (inclusiv cele referitoare la '..' și '..').
3. Numele unei comenzi externe este un specificator interpretat de shell iar acesta are o regulă diferită pentru cazul când el este un nume fără cale: numele nu este căutat în directorul curent ci într-o listă de directoare reținută în variabila de environment 'PATH' a shell-ului (specificatori separați prin ':'). 'Command Prompt' din Windows ar fi căutat și în directorul curent (am văzut). De aceea, o comandă - fișier executabil din directorul curent va fi găsită fără cale d.d. directorul curent este în PATH. Altfel, poate fi găsită dacă specificăm calea directorului curent.

Mai sus, comanda 'program' nu a fost recunoscută, deoarece directorul curent '.' nu este în lista 'PATH', dar comanda './program' a fost recunoscută. Pentru a fi recunoscută și comanda 'program', putem adăuga directorul curent '.' la lista 'PATH'.

Exemplu (continuare):

```
$echo $PATH  
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:  
/sbin:/bin:/usr/games:/usr/local/games:/snap/bin  
$export PATH=$PATH:.  
$echo $PATH  
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:  
/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:.  
$program  
Hello world !
```

Obs: la terminarea procesului shell (de exemplu, cu comanda 'exit') se pierde noua setare a lui 'PATH' (valorile variabilelor sunt locale proceselor). Dacă vrem ca ea să fie permanentă, o putem adăuga în fișierul de configurare '~/.bashrc' (am văzut mai sus).

Compilatorul 'gcc' linkeditează automat (fără a fi nevoie de opțiuni suplimentare) cu biblioteca standard C dar există și biblioteci prezente în instalarea sa cu care nu linkeditează automat ci doar dacă o cerem prin opțiuni speciale.

De exemplu, pentru a folosi în program funcții matematice, trebuie să includem fișierul header `<math.h>` și să adăugăm în comanda de compilare opțiunea '`-lm`'. Similar, dacă vrem să folosim funcții de interfață avansată în mod text din biblioteca 'ncurses', trebuie să includem în programul sursă fișierul header `<curses.h>` și în linia de comandă opțiunea '`-lncurses`'.

Exemplu:

```
$cat prog.c
#include <stdio.h>
#include <math.h>
int main() {
    double x;
    printf("x = "); scanf("%lf", &x);
    printf("sqrt(%lf) = %lf\n", x, sqrt(x));
    return 0;
}

$gcc -Wall -o program  prog.c -lm
$./program
x = 4
sqrt(4.000000) = 2.000000
```

Cuprins

- 1 Sisteme de calcul, sisteme de operare
 - Organizarea unui sistem de calcul
 - Funcțiile sistemului de operare
- 2 Sisteme de operare uzuale
 - Incișiune în sistemul Windows
 - Incișiune în sistemul Linux
- 3 Interfața de programare C
 - Obiecte software și apeluri sistem
 - Utilizatori și grupuri
 - Fișiere și directoare
 - Procese
 - Semnale
 - Tuburi

- Biblioteca standard C (tipul 'FILE')
- 4 Interfață interactivă linie de comandă
 - Interfața Command Prompt (Windows)
 - Interfața Shell (Linux)
- 5 Interfață de scripting linie de comandă
 - Interfața Batch Scripting (Windows)
 - Interfața Shell Scripting (Linux)
- 6 Teme de laborator
 - Utilizatori și grupuri
 - Fișiere, directoare
 - Procese, semnale, tuburi

Cuprins

1 Sisteme de calcul, sisteme de operare

Organizarea unui sistem de calcul
Funcțiile sistemului de operare

2 Sisteme de operare uzuale

Incursiune în sistemul Windows
Incursiune în sistemul Linux

3 Interfața de programare C

Obiecte software și apeluri sistem
Utilizatori și grupuri
Fișiere și directoare
Procese
Semnale
Tuburi

Biblioteca standard C (tipul 'FILE')

4 Interfață interactivă linie de comandă

Interfața Command Prompt
(Windows)
Interfața Shell (Linux)

5 Interfață de scripting linie de comandă

Interfața Batch Scripting (Windows)
Interfața Shell Scripting (Linux)

6 Teme de laborator

Utilizatori și grupuri
Fișiere, directoare
Procese, semnale, tuburi

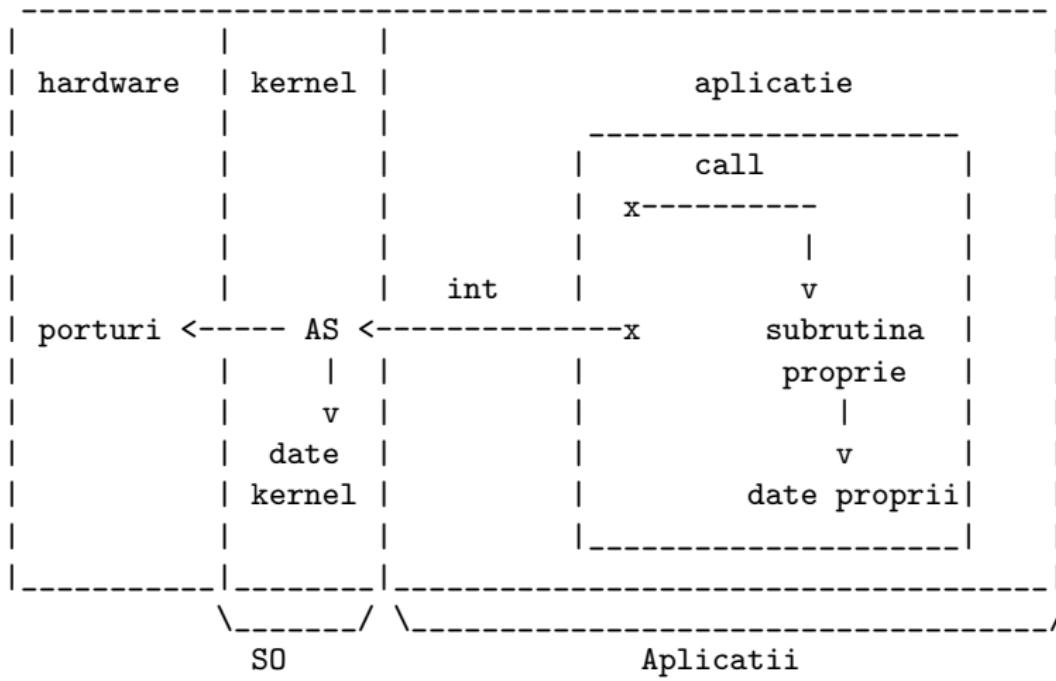
Am spus mai devreme că SO (Kernelul) gestionează resursele hardware și software ale sistemului de calcul și oferă o interfață prin care pot fi utilizate de aplicații, formată din obiecte software (unele corespund unor obiecte hardware) și apeluri sistem (AS). Din aplicații, pot fi cerute servicii SO apelând AS cu obiecte software indicate ca parametri.

Am spus, de asemenea, că SO nu permite aplicațiilor să acceseze anumite resurse hardware și software direct (de exemplu, executând instrucțiuni mașină de citire/scriere la porturile echipamentelor sau accesând memoria cu pointeri la adrese fizice) ci doar prin interfața sa (apelând AS) iar în acest scop sunt folosite facilități hardware, anume nivelurile de privilegiu ale procesorului și adresarea protejată a memoriei.

De aceea, aplicațiile nu pot apela AS cu instrucțiunile uzuale de apel subrutine ('call adresă' / 'ret' în arhitectura Intel), deoarece pe acestea le folosesc și pentru a apela subrutele proprii și, de aceea, aceste instrucțiuni nu schimbă nivelul de privilegiu al procesorului și nici spațiul de adrese accesibil.

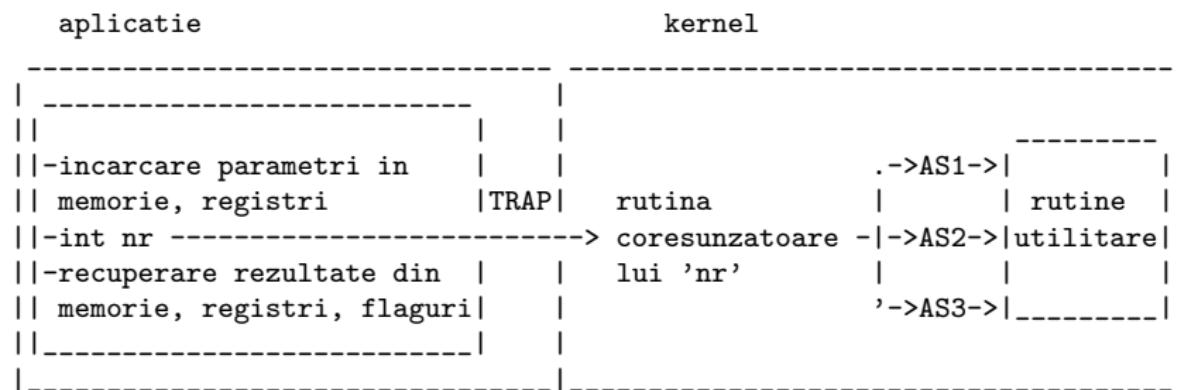
Aplicațiile pot apela însă AS cu instrucțiuni de TRAP, care generează întreruperi software ('int număr' / 'iret' în arhitectura Intel).

SC



Când se execută o instrucțiune 'int nr', numărul 'nr' (8 biți, valoare 0 ... 255) este folosit ca index într-o tabelă de descriptori (Interrupt Descriptor Table, IDT). Intrarea poate indica adresa unei subroutines care va gestiona cererea.

Cu o aceeași instrucțiune 'int nr', deci cu o aceeași subroutine lansată, se pot apela mai multe AS. Întrucât instrucțiunea 'int nr' nu are mai multe argumente, trebuie specificat pe altă cale ce AS trebuie lansat și cu ce parametri. Aceste informații vor fi încărcate în anumite locuri în memorie și în regiștrii procesorului. De asemenea, AS lansat produce rezultate accesibile la revenirea în programul apelant. Aceste informații vor fi furnizate în anumite locuri în memorie și în regiștrii și flagurile procesorului.



Cu 'int 21h' în Windows sau cu 'int 0x80' în Linux, se pot apela mai multe AS, de exemplu, operații cu fișiere: deschidere / închidere la operații, citire / scriere, etc. Exemplificăm modul de folosire al lui 'int 0x80' în Linux, într-un program care deschide un fișier în citire și scriere, scrie în el un mesaj de 2 linii, închide fișierul și se termină.

Programul este scris în limbajul de asamblare NASM (Netwide Assembler) pentru arhitectura x86, pentru care există compilatoare (asambloare) atât pentru Windows cât și pentru Linux.

NASM poate fi descărcat de la adresa: '<https://www.nasm.us/>'.

O varianta portabilă pentru Windows 64 biți poate fi descărcată și de la: '<https://ccm.net/download/download-1025-nasm>'.

În Linux, dacă stocăm programul sursă într-un fișier text numit 'prog.s', atunci, din linia de comanda shell, el poate fi compilat cu:

```
nasasm -f elf prog.s
```

linkeditat cu:

```
ld -m elf_i386 -s -o prog prog.o
```

și rulat cu:

```
./prog
```

Programul este următorul:

```
; in nasm, un comentariu tine de la simbolul ';' si pana la sfarsitul liniei
section .data
; urmeaza declaratii de variabilei; o declaratie contine:
; numele variabilei; este doar o eticheta, desemnand adresa sa de inceput;
; un tip: db = byte, dw = word, dd = double word;
; o lista de valori care vor fi stocate succesiv, incepand de la adresa
; desemnata de numele variabilei, fiecare avand un format si dimensiune
; conform tipului specificat
specificator_fisier db 'fis.txt', 0
; adresa de la care este stocat numele cu cale al fisierului ce trebuie
; deschis (exemplu: 'a/b/fisier.txt'); numele este sir de bytes, terminat
; cu byte-ul 0
descriptor dd 0
; adresa de la care este stocat 0, ca valoare de tip double word
mesaj db 'Salut !', 10, 'Ce mai faci ?', 10
; adresa de la care este stocat mesajul de scris in fisier;
; mesajul este un sir de bytes oarecare; el contine si doi bytes egali cu
; 10, care in Linux codifica sfarsitul de linie (caracterul line feed);
; astfel, la afisarea continutului fisierului pe ecran, 'Salut !' si
; 'Ce mai faci ?' vor fi linii separate
lungime_mesaj equ $ - mesaj
; simbolul $ inseamna "aici", adica adresa curenta, adica adresa de sfarsit
; a mesajului anterior; numele mesaj inseamna adresa de inceput a
; mesajului anterior; expresia $ - mesaj calculeaza diferența celor două
; adrese, care inseamna lungimea mesajului anterior, in numar de bytes;
; numele lungime_mesaj desemneaza acest numar
```

```
section .text
; urmeaza instructiuni
global _start
; in limbajul nasm, eticheta _start marcheaza in mod standard punctul de
; intrare in program; din S0, controlul executiei va trece la program,
; incepand de la eticheta _start;
; directiva global _start face eticheta _start vizibila in exterior;
; compilatorul va adauga simbolul _start in tabela de simboluri a
; fisierului obiect generat; linkeditorul va citi acest simbol in fisierul
; obiect si valoarea sa (o adresa) si va sti unde sa marcheze punctul de
; intrare in fisierul executabil generat

_start:
; de aici incepe executarea programului
```

```
; deschiderea fisierului:  
mov eax, 5  
; in registrul eax se incarca valoarea 5, care indica apelul sistem  
;   sys_open (deschide un fisier pentru operatii)  
mov ebx, specificator_fisier  
; in registrul ebx se incarca adresa de la care este stocat numele cu cale  
;   al fisierului  
mov ecx, 0102o  
; in registrul ecx se incarca valoarea octala 0102, insemnand modul de  
;   deschidere: 0002 inseamna deschidere pentru citire si scriere, 0100  
;   inseamna ca daca fisierul nu exista, se va crea  
mov edx, 0666o  
; in registrul edx se incarca valoarea octala 0666, specificand drepturile  
;   asociate fisierului, in caz ca se creaza un fisier nou; 0666 inseamna  
;   drept de citire si scriere pentru toti  
int 80h  
; instructiunea de TRAP; genereaza intreruperea software cu codul hexa 80;  
;   rutina de tratare a intreruperii inspecteaza argumentele din registri  
;   si memorie si apeleaza apelul sistem sys_open (deoarece in eax a fost  
;   incarcat 5), transmitandu-i ca argumente numele cu cale, modul de  
;   deschidere si drepturile;  
; la revenirea din apelul sistem si rutina de tratare a intreruperii,  
;   registrul eax va contine (ca valoare returnata) descriptorul fisierului  
;   deschis (intreg  $\geq$  0), in caz de succes, sau codul erorii (intreg  $<$  0),  
;   in caz de esec
```

```
mov [descriptor], eax
; in memorie, la adresa descriptor (deci in locul lui 0 de tip double word)
; se salveaza descriptorul fisierului (am presupus deschidere cu
; succes); in mod normal, trebuie consultata intai valoarea din eax, daca
; este sau nu >= 0;

; scrierea in fisier:
mov eax, 4
; in registrul eax se incarca valoarea 4, care indica apelul sistem
; sys_write (scrie intr-un fisier)
mov ebx, [descriptor]
; in registrul ebx, se incarca valoarea de tip double word de la adresa
; descriptor, adica descriptorul fisierului
mov ecx, mesaj
; in registrul ecx, se incarca adresa de la care este stocat mesajul
mov edx, lungime_mesaj
; in registrul edx se incarca lungimea mesajului
int 80h
; cu aceeasi intrerupere software 80h se apeleaza sys_write (eax are
; valoarea 4), transmitandu-i ca argumente descriptorul fisierului,
; mesajul de scris si lungimea acestuia;
; la revenire, registrul eax va contine (ca valoare returnata) numarul de
; bytes scrisi, in caz de succes, sau codul erorii, in caz de esec
```

```
; inchiderea fisierului:  
mov eax, 6  
; in registrul eax se incarca valoarea 6, care indica apelul sistem  
; sys_close (inchide un fisier)  
mov ebx, [descriptor]  
; in registrul ebx, se incarca valoarea de tip double word de la adresa  
; descriptor, adica descriptorul fisierului  
int 80h  
; cu aceeasi intrerupere software 80h se apeleaza sys_close (eax are  
; valoarea 6), transmitandu-i ca argument descriptorul fisierului;  
; la revenire, registrul eax va contine (ca valoare returnata) codul erorii,  
; in caz de exec  
  
mov eax, 1  
mov ebx, 0  
int 80h  
; cu aceeasi intrerupere software 80h se apeleaza sys_exit (eax are  
; valoarea 1), care incheie executia programului si reda controlul catre  
; S0, cu codul de retur 0 (ebx are valoarea 0), care inseamna succes
```

Testare:

```
$ls
prog.s
$nasm -f elf prog.s
$ld -m elf_i386 -s -o prog prog.o
$./prog
$ls
fis.txt  prog  prog.o  prog.s
$cat fis.txt
Salut !
Ce mai faci ?
```

Programul de mai înainte ilustrează cum pot fi apelate AS dintr-un program scris în limbaj de asamblare. Dacă dorim apelarea AS dintr-un limbaj de nivel mai înalt, de exemplu C, compilatorul trebuie să ne ofere, în bibliotecile sale, câte o funcție de interfațare (wrapper function) pentru fiecare AS din SO pentru care a fost dezvoltat - o asemenea funcție are ca scop principal apelarea AS respectiv, cu un efort de calcul minimal.

De exemplu, pentru a apela 'sys_open', 'sys_write', 'sys_close', compilatorul de C ofera în biblioteca sa funcțiile 'open()', 'write()', respectiv 'close()':

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *specifier_fisier,
         int mod_deschidere,
         mode_t drepturi);
```

deschide fișierul cu numele și calea date de 'specifier_fisier', în modul 'mod_deschidere', iar dacă cu această ocazie se crează un fișier nou, el se va crea cu drepturile 'drepturi'; returnează descriptorul asociat fișierului deschis, în caz de succes, sau -1, în caz de eșec.

```
#include <unistd.h>
ssize_t write(int descriptor, const void *buf, size_t nr);
```

scrie 'nr' bytes din zona de memorie pointată de 'buf' în fișierul cu descriptorul 'descriptor'; returnează numarul de bytes scriși, în caz de succes, sau -1, în caz de eșec.

```
#include <unistd.h>
int close(int descriptor);
```

închide descriptorul 'descriptor' (el devine liber, nu se mai referă la nici un fișier și poate fi refolosit); returnează 0, în caz de succes, sau -1, în caz de eșec.

În caz de eșec, AS apelate de funcțiile wrapper le furnizeaza acestora la revenire și un cod de eroare (de exemplu, 'sys_open' furnizează un cod de eroare < 0 în registrul EAX). Funcțiile wrapper vor furniza codul erorii către apelant folosind o variabilă globală predefinită:

```
int errno;
```

Variabila 'errno' adaugată automat de compilator se află în spațiul de memorie al programului utilizator, a.î. ea poate fi accesată în mod ușual de orice instrucțiune a programului, indiferent dacă este scrisă de programator sau adăugată de compilator. Dacă se dorește accesarea ei în mod direct (cu instrucțiuni în care să apară explicit cuvantul 'errno'), de exemplu:

```
errno = 2;  
printf("%d\n", errno);
```

atunci trebuie adăugată la program declarația:

```
extern int errno;
```

sau directiva (care conține, printre altele, declarația respectivă):

```
#include <errno.h>
```

altfel programul nu se poate compila.

Dacă variabila 'errno' urmează a fi accesată doar indirect (prin instrucțiuni adăugate automat de compilator, de exemplu, prin apelarea unor funcții predefinite, ca 'open()'), nu mai este necesară prezența declarației respective. Fișierul `<errno.h>` definește și constante simbolice pentru toate codurile de eroare posibile: ENOENT, EACCESS, etc.

scris de programator

adaugat de compilator

```
| int d;  
| main() {           descriptor, | corp open() | cod  
|   |           -1      |-----|  
|   d = open() <-----> |       int 0x80 -----> AS  
|   |           |       | | descriptor, |  
|   |           |       | | cod eroare v  
|   |           |       |-----<----- EAX  
|   |           |       | |  
|   printf("%d", errno); <-----|-----|  
| } |           |       | cod eroare  
|   |           |       |  
|-----|-----|-----|  
|   v |           |       | v |       | date  
|-----|-----|-----|  
|   d           |       | errno |  
-----|-----|-----|
```

Observații:

1. Există și alte funcții predefinite, în afară de wrapper-ele de AS, care setează 'errno'.
2. Valoarea lui 'errno' este relevantă doar atunci când valoarea returnată de funcție semnifică eroare (de exemplu -1 sau NULL); în caz de succes, funcția poate modifica 'errno' (dar nu este specificat cum).
3. Nici o funcție predefinită nu setează 'errno' cu 0. Astfel, o setare explicită 'errno = 0;' înaintea apelului și o consultare explicită a lui 'errno' după apel ne permite să detectăm modul cum acesta afectează 'errno', în toate cazurile.

Compilatorul pune la dispoziție funcții de bibliotecă cu care se pot face operații uzuale în legătură cu 'errno':

```
#include <stdio.h>
void perror(const char *s);
```

afișază pe standard error (descriptorul 2 al procesului, care de obicei este asociat unui terminal, dar poate fi asociat și unor fișiere) stringul 's', urmat de caracterul ':', urmat de un mesaj standard ce descrie valoarea curentă din 'errno'; de obicei, argumentul dat este numele obiectului (de exemplu, specificatorul unui fișier) asupra căruia s-a aplicat anterior un apel care a eșuat.

```
#include <string.h>
char *strerror(int errnum);
```

furnizează într-o zonă internă un mesaj care descrie codul de eroare 'errnum' și returnează adresa zonei; zona internă poate fi suprascrisă de apelurile ulterioare ale funcției, deci codul utilizator trebuie să utilizeze complet conținutul ei sau să-l salveze în altă parte înainte de un nou apel; de asemenea, codul utilizator nu trebuie să modifice conținutul zonei interne; dacă 'errnum' conține un număr nnn invalid ca și cod de eroare, mesajul furnizat este "Unknown error nnn".

Un exemplu uzual de folosire a funcțiilor de mai sus este:

```
int d;
if((d = open("/a/b/fis.txt",
              O_WRONLY|O_CREAT|O_TRUNC,
              S_IRUSR|S_IWUSR)) == -1){
    perror("/a/b/fis.txt");
    return 1;
}
write(d, "abc", 3 * sizeof(char));
close(d);
```

La rulare, dacă deschiderea a eşuat din imposibilitatea de a accesa fișierul ('open()') returneaza -1 iar 'errno' primește valoarea EACCES), se va afișa:

```
a/b/fis.txt: Permission denied
```

Între apelul unei funcții care setează 'errno' și căreia îi testam succesul și apelul ' perror()' corespunzător, nu trebuie să fie intercalate alte apeluri care setează 'errno', altfel mesajul afișat nu va fi cel corect.

Greșit:

```
int d1, d2;  
d1 = open("f.txt", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);  
d2 = open("g.txt", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);  
if(d1 == -1 || d2 == -1) { perror("Eroare"); return 1; }
```

Corect:

```
int d1, d2;  
d1 = open("f.txt", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);  
if(d1 == -1){ perror("f.txt"); return 1; }  
d2 = open("g.txt", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);  
if(d2 == -1){ perror("g.txt"); return 1; }
```

Exemplu: O variantă în limbajul C a programului nasm anterior, care în plus testează succesul/eșecul operațiilor și afisază mesaje adecvate:

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

char specifikator_fisier[] = "fis.txt";
int descriptor;
char mesaj[] = "Salut !\nCe mai faci ?\n";
int lungime_mesaj = sizeof(mesaj);

int main(){
    int mod_deschidere = O_CREAT | O_RDWR;
    mode_t drepturi =
        S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH;
```

```
if((descriptor = open(specificator_fisier,
                      mod_deschidere,
                      drepturi)) == -1){
    perror(specificator_fisier); return 1;
}
if(write(descriptor, mesaj, lungime_mesaj) == -1) {
    perror(specificator_fisier); return 2;
}
if(close(descriptor) == -1) {
    perror(specificator_fisier); return 3;
}
return 0;
}
```

Observație: Pentru instrucțiunile 'return' din 'main()', compilatorul adaugă cod care conduce la apelarea AS 'sys_exit', cu valoarea returnată transmisă ca argument în registrul EBX; aşadar, această valoare devine cod de return al procesului, către apelantul său.

Testare (presupunem că programul se află în fișierul 'prog.c'):

```
$gcc -Wall -o prog prog.c
$./prog
$cat fis.txt
Salut !
Ce mai faci ?
```

Funcțiile wrapper sunt în corespondență cu AS din SO pentru care a fost dezvoltat compilatorul iar când lucrăm în limbajul C vom asimila aceste funcții cu AS. Ele constituie nivelul inferior de la care putem accesa resursele SO dintr-un program C.

Accesarea resurselor SO la nivel inferior este rapidă (funcțiile wrapper conțin puține instrucțiuni în plus față de TRAP-ul care oricum trebuie efectuat) dar mai puțin portabilă, deoarece lista de AS poate dифeри de la un SO la altul. Pentru a mări portabilitatea, au fost elaborate standarde privind interfața SO (interfața API, interfața linie de comanda, etc.), cum ar fi POSIX (Portable Operating System Interface). O aplicație care cere doar AS standardizate POSIX va rula pe orice SO compatibil POSIX, via o compilare cu un compilator dezvoltat pentru sistemul respectiv.

Funcțiile wrapper pentru AS oferite de compilatorul C sunt rezultatul unui efort de compatibilizare între POSIX și standardului C - ele sunt incluse în biblioteca POSIX C (C POSIX library) și sunt utilizabile în SO compatibile POSIX (UNIX, Linux, etc.).

Exemplu: O variantă a programului C anterior, folosind o emulare a funcțiilor wrapper prin cod inline assembler:

TODO

Pentru un grad mai mare de portabilitate a aplicațiilor, compilatorul de C oferă biblioteca standard C (C standard library), care conține instrumente (macro-uri, tipuri, funcții) pentru manevrat stringuri, calcule matematice, procesarea intrărilor și ieșirilor, gestiunea memoriei și diverse servicii cerute SO, unele definite deasupra nivelului inferior dinainte. Acestea constituie nivelul superior de la care putem accesa resursele SO dintr-un program C.

Modul în care funcționează instrumentele din biblioteca standard C este descris de standardul limbajului C (ANSI C, ISO C, sau Standard C), independent de specificațiile unui anumit SO. Astfel, un program care utilizează numai funcții din biblioteca standard C va funcționa corect în orice SO, dacă este compilat cu un compilator dezvoltat pentru sistemul respectiv și care respectă standardul C.

Funcțiile din biblioteca standard C nu sunt nu corespondență cu AS (nu se limitează doar la interfațarea cu AS, unele pot să nu apeleze AS, deși pot seta 'errno'), deci nu le vom asimila cu AS. De asemenea, ele pot executa mai multe instrucțiuni în plus față de TRAP-uri (de exemplu, conversii de format în cazul funcțiilor 'fscanf()', 'fprintf()'), așa că față de funcțiile wrapper oferă mai multă portabilitate dar mai puțină viteză.

De exemplu, pentru gestiunea fișierelor, biblioteca standard C oferă tipul structură 'FILE' și funcții ca 'fopen()', 'fclose()', 'fwrite()', 'fprintf()', etc.

'FILE' este definit cu 'typedef' ca un tip structura opac (implementarea sa este ascunsă, poate fi specifică sistemului, el se manevrează ca un tot). El conține (printre altele):

- un descriptor de fișier (returnat de 'open()');
- un indicator de poziție curentă în fișier;
- un buffer pentru date citite/scrisă în fișier;
- flag-uri de stare (pot semnala o eroare);

Un fișier este gestionat cu ajutorul unui obiect 'FILE' alocat în spațiul de memorie al programului utilizator și manevrat cu ajutorul pointerilor.

```
#include <stdio.h>
FILE *fopen(const char *specifier_fisier,
            const char *mod_deschidere);
```

deschide fișierul cu numele și calea date de 'specifier_fisier', în modul 'mod_deschidere' (folosind 'open()'); poate crea un fișier nou; alocă o structură 'FILE' în spațiul de memorie al programului utilizator (poate fi accesată cu instrucțiuni obișnuite, ca și 'errno'), o initializează cu informații despre fișier (în particular, stochează aici descriptorul furnizat de 'open()') și returnează adresa structurii 'FILE' în caz de succes și NULL în caz de eșec (în acest caz, furnizează codul erorii în 'errno' și se poate folosi ' perror()').

Utilizatorul trebuie să-și salveze adresa structurii 'FILE' returnată de 'fopen()' într-o variabilă iar ulterior să o transmită ca argument funcțiilor care vor opera asupra fișierului respectiv.

```
#include <stdio.h>
size_t fwrite(const void *ptr,
              size_t size, size_t nmemb,
              FILE *stream);
```

scrie de la adresa de memorie 'ptr' în fișierul indicat de 'stream' 'nmemb' blocuri de date de dimensiune 'size'; returnează numărul de blocuri scrise; funcția scrie doar un număr întreg de blocuri, un bloc nu se poate scrie doar parțial.

```
#include <stdio.h>
int fclose(FILE *stream);
```

evacueză către fișier conținutul rămas în bufferul structurii 'FILE' indicate de 'stream', închide fișierul (descriptorul conținut în 'FILE') și dezalocă structura 'FILE'.

Observații:

1. Fiecare apel 'fopen()' alocă o nouă structură 'FILE', posibil dinamic, care se află în spațiul de memorie al programului utilizator și poate fi accesată cu instrucțiuni obișnuite (ca și variabilele definite de utilizator).

Este de preferat, însă, să nu fie accesată cu instrucțiuni directe (de exemplu, explicitând accesul la membri), deoarece organizarea structurii 'FILE' nu este specificată de standard iar considerarea unei anumite organizări poate conduce la un program neportabil.

De asemenea, adresa furnizată de 'fopen()' nu trebuie furnizată lui 'free()'. Funcțiile 'malloc()', 'calloc()', 'realloc()' rețin într-o listă internă adresele și dimensiunile zonelor alocate și returnează doar adresele; aceste adrese ajung transmise ca argument lui 'free()', care va căuta în lista respectivă dimensiunea zonei care trebuie dezalocată. Funcția 'fopen()', chiar dacă alocă dinamic structura 'FILE', este posibil să nu înregistreze în acea listă adresa și dimensiunea ei, a.î. dacă se transmite adresa lui 'free()', nu o va găsi în listă iar efectul va fi imprevizibil. Structurile 'FILE' alocate de 'fopen()' vor fidezalocate de 'fclose()' iar lui 'free()' i se vor transmite ca argument doar adrese furnizate de 'malloc()', 'calloc()', 'realloc()' (sau NULL).

Reamintim că 'FILE' este un tip opac - se manevrează ca un tot, cu funcții predefinite, standardul nu descrie structura lui și nu este nevoie să-i accesăm structura explicit.

2. Funcțiile de mai sus ilustrează modul cum nivelul superior de accesare a fișierelor (structuri 'FILE', funcții din biblioteca standard C) este construit deasupra nivelului inferior (descriptori, funcții wrapper de AS): structurile 'FILE' conțin un descriptor, 'fopen()' apelează 'open()':

scris de programator	adaugat de compilator
 FILE *f; main() { f = fopen("f.txt", "r"); -----> } adr FILE, NULL adr FILE ----- v ----- f	corp fopen() ----- corp open() ----- cod TRAP ----- open() -----> int 0x80 -----> AS <----- <----- <----- desc, desc, -1 err desc ----- ----- ----- ----- ----- ----- ----- ----- ----- date v v v ----- adr ----- ----- ----- ----- structura FILE ----- errno -----

Exemplu: O variantă a programului anterior, cu funcții de nivel superior:

```
#include <stdio.h>
char specifikator_fisier[] = "fis.txt";
FILE *f;
char mesaj[] = "Salut !\nCe mai faci ?\n";
int lungime_mesaj = sizeof(mesaj);
int main(){
    char *mod_deschidere = "w+";
    if((f = fopen(specifikator_fisier, mod_deschidere)) == NULL){
        perror(specifikator_fisier); return 1;
    }
    if(fwrite(mesaj, 1, lungime_mesaj, f) != lungime_mesaj) {
        perror(specifikator_fisier); return 2;
    }
    if(fclose(f) == -1) {
        perror(specifikator_fisier); return 3;
    }
    return 0;
}
```

Obs: "w+" corespunde combinației 'O_CREAT|O_TRUNC|O_RDWR' și drepturilor 'S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH' (pot fi modificate de masca de drepturi 'umask' a procesului apelant, vom vedea mai târziu).

Exemplu: O variantă a programului anterior, cu emularea funcțiilor de nivel superior prin funcții de nivel inferior:

TODO

Ca specificație, biblioteca POSIX C include biblioteca standard C, oferind facilități suplimentare legate, în general, de accesarea la nivel inferior a interfeței SO. Ca implementare, am văzut că unele instrumente de nivel superior sunt construite deasupra celor de nivel inferior.

Instrumentele din biblioteca POSIX funcționează în SO compatibile POSIX (UNIX, Linux, parțial Windows) iar cele din biblioteca standard funcționează în orice SO.

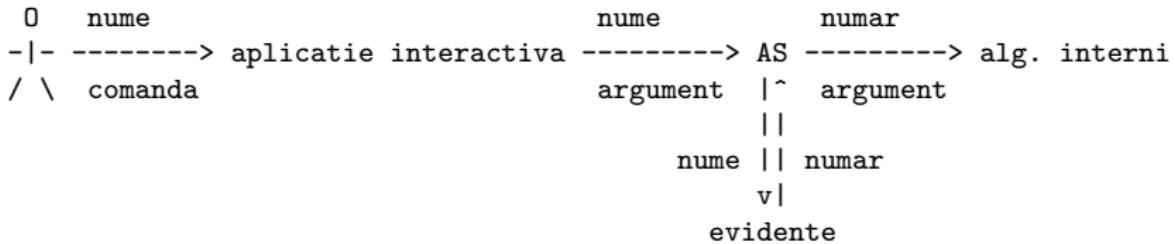
O documentație completă privind implementarea GNU a acestor biblioteci (compilatorul 'gcc') poate fi găsită la adresa:

<https://www.gnu.org/software/libc/manual/pdf/libc.pdf>

În secțiunile următoare, vom prezenta câteva concepte și instrumente de interfață SO cu aplicațiile (obiecte software și apeluri sistem) prezente în specificația POSIX și interfața C cu ele, anume funcții din biblioteca POSIX C și, în anumite cazuri, funcții de nivel superior din biblioteca standard C.

Obiectele software sunt implementate după un același tipar general:

- au asociate niște resurse;
- au asociate niște atribute;
- sunt gestionate cu o structură de date (conținând atribute);
- sunt desemnate intern printr-un număr întreg (algoritmii interni operează cu întregi);
- pot fi desemnate extern prin nume (în interfața SO cu aplicațiile, prin AS, și cu utilizatorii umani, prin aplicații interactive);
- legătura dintre nume și numere este păstrată în evidențe interne (evidența pentru utilizatori este păstrată în fișierele '/etc/passwd', '/etc/group' și '/etc/shadow', evidența pentru fișiere este păstrată în directoare, etc.);
- stabilirea legăturii dintre nume și numere este realizată tehnic de AS.



Cuprins

1 Sisteme de calcul, sisteme de operare

Organizarea unui sistem de calcul
Funcțiile sistemului de operare

2 Sisteme de operare uzuale

Incursiune în sistemul Windows
Incursiune în sistemul Linux

3 Interfața de programare C

Obiecte software și apeluri sistem

Utilizatori și grupuri

Fișiere și directoare

Procese

Semnale

Tuburi

Biblioteca standard C (tipul 'FILE')

4 Interfață interactivă linie de comandă

Interfața Command Prompt
(Windows)
Interfața Shell (Linux)

5 Interfață de scripting linie de comandă

Interfața Batch Scripting (Windows)
Interfața Shell Scripting (Linux)

6 Teme de laborator

Utilizatori și grupuri

Fișiere, directoare

Procese, semnale, tuburi

Utilizatorii (conturile) sunt mecanismul prin care SO multiuser fac distincție între persoanele fizice care îl folosesc: oricine dorește să utilizeze SO trebuie să specifice mai întâi un cont utilizator și o parolă (autentificare, logare) iar dacă sunt corecte sistemul îl acceptă și el va opera cu drepturile utilizatorului respectiv.

Utilizatorii sunt organizați în grupuri. Un utilizator poate face parte din mai multe grupuri dar pentru fiecare utilizator se specifică un grup preferențial (primary group).

D.p.v. tehnic, utilizatorii (user) și grupurile (group) sunt obiecte software definite în sistem, având anumite caracteristici: identificator numeric, nume, parolă, etc.. Ele pot fi create/distruse/modificate. Scopul lor principal este definirea posesiei diverselor resurse ale sistemului și controlul permiterii accesului la ele de către procese.

Diverse obiecte software ale sistemului (fișiere, procese, etc.) pot avea printre caracteristici utilizatori și grupuri proprietar (se reține identificatorul numeric) și drepturi de acces, iar un proces (interactiv sau nu) poate accesa un alt obiect software doar dacă proprietarii acestora și drepturile de acces verifică anumite condiții.

Fișierele sistem care conțin definițiile utilizatorilor și grupurilor sunt:

/etc/passwd
/etc/group
/etc/shadow
/etc/gshadow

Fișierul '/etc/passwd' conține definițiile utilizatorilor, are câte o linie pentru fiecare utilizator din sistem, conținând următoarele câmpuri:

- nume de login (login name, username);
- parola criptată; dacă sunt activate parolele shadow (cazul uzual), câmpul conține 'x' și este ignorat iar parola criptată este stocată în fișierul '/etc/shadow'; dacă câmpul este gol, logarea se poate face fără parolă;
- identificator numeric de utilizator (User ID, UID); pot exista mai multe înregistrări cu același UID, deci mai multe nume de login pentru același UID - astfel, mai mulți utilizatori pot accesa aceleasi resurse, folosind parole diferite (deoarece proprietarii obiectelor software sunt reținuti ca UID, nu ca nume); dacă $UID = 0$, utilizatorul este privilegiat; de obicei, există un singur utilizator privilegiat, cu numele 'root'; el este creat automat la instalarea sistemului, cerându-se specificarea unei parole pentru el;

- identificator de grup (group ID, GID) - este identificatorul numeric al grupului preferențial al utilizatorului; alte apartenențe ale acestuia la grupuri sunt definite în fișierul '/etc/group';
- comentariu: un text despre utilizator (de exemplu, real name); acest text este afișat de diverse utilitare, ca 'finger';
- directorul home (home directory): pentru fiecare utilizator se specifică unul dintre directoarele din arborescența (unică a) sistemului, ca fiind directorul său home; aici are toate drepturile și își poate construi propria arborescență de fișiere și directoare; de obicei, procesele obținute de utilizator la logare au directorul curent inițial și variabila de environment HOME setată la directorul său home; într-o comandă shell, '~' la începutul unei căi este expandat la directorul home al proprietarului shell-ului iar '~~nume' este expandat la directorul home al utilizatorului 'nume';
- login shell: este programul căruia îi este transferat controlul odată ce utilizatorul s-a logat (pe terminalele text, cu 'getty'); de obicei, este unul dintre shell-uri, de exemplu 'bash' (Bourne again shell, '/bin/bash'), dar poate fi orice program; dacă câmpul este gol, atunci login shell-ul este implicit '/bin/sh' (adică Bourne shell); acest câmp devine valoarea variabilei de environment SHELL a proceselor obținute la logare.

Exemplu:

```
$grep dragulici /etc/passwd  
dragulici:x:1000:1000:dragulici,,,:/home/dragulici:/bin/bash
```

Deoarece multe procese, ale diversilor utilizatori neprivilegiați, necesită citirea de informații din fișierul '/etc/passwd' (de exemplu, 'ls -l' găsește în directorul listat UID-ul proprietarilor fișierelor și consultă fișierul '/etc/passwd' pentru a le găsi și afișa numele), acest fișier este un fișier text, cu drept de citire pentru toți:

```
$ls -l /etc/passwd  
-rw-r--r-- 1 root root 2750 Nov  3 2020 /etc/passwd
```

Fișierul '/etc/passwd' poate fi citit la nivel de octet (cu 'open()', 'read()', etc.) și se poate implementa în program algoritmul de interpretare a informațiilor dar, pentru portabilitate, se recomandă utilizarea funcțiilor predefinite 'getpwuid()', 'getpwnam()', 'getpwent()', 'setpwent()', 'endpwent()'.

Fișierul '/etc/group' conține apartenențele la grupuri suplimentare pentru fiecare utilizator. Deci, mulțimea grupurilor din care face parte un utilizator este definită de combinația dintre campul GID din înregistrarea sa din fișierul '/etc/passwd' (i.e. grupul său preferențial) și grupurile sub care este listat utilizatorul în fișierul '/etc/group'.

Fișierul '/etc/group' are câte o linie pentru fiecare grup din sistem, conținând următoarele câmpuri:

- nume de grup (group name);
- parola criptată (optională); se folosește rar; o poate seta un utilizator privilegiat, cu comanda 'passwd'; cu comanda 'newgrp', se poate lansa un nou proces shell, a cărui apartenență la grupuri a proprietarului include și grupul respectiv, iar comanda cere introducerea acestei parole; dacă sunt activate parolele shadow, câmpul conține 'x' și este ignorat iar parola criptată este stocată în fișierul '/etc/gshadow';
- identificator numeric de grup (Group ID, GID); de obicei, există un singur grup cu GID = 0, numit 'root' (la fel ca înregistrarea cu UID = 0 din fișierul '/etc/passwd') și care este privilegiat;
- lista de utilizatori: lista numelor utilizatorilor, separate prin virgulă ',', care sunt membri ai acestui grup.

Exemplu:

```
$grep dragulici /etc/passwd
dragulici:x:1000:1000:dragulici,,,:/home/dragulici:/bin/bash
$grep dragulici /etc/group
adm:x:4:syslog,dragulici
cdrom:x:24:dragulici
sudo:x:27:dragulici
dip:x:30:dragulici
plugdev:x:46:dragulici
lpadmin:x:114:dragulici
dragulici:x:1000:
sambashare:x:134:dragulici
```

Așadar, utilizatorul 'dragulici' are grupul preferențial cu GID = 1000 (câmpul 4 din '/etc/passwd') și numele 'dragulici' (găsim în penultima linie afișată din '/etc/group') și mai este membru în grupurile 'adm', 'cdrom', 'sudo', 'dip', 'plugdev', 'lpadmin', 'sambashare'. Observăm că utilizatorul 'dragulici' nu este menționat în grupul său preferențial 'dragulici' în fișierul '/etc/group' ci doar în fișierul '/etc/passwd'.

Fișierul '/etc/group' este, de asemenea, un fișier text, cu drept de citire pentru toți:

```
$ls -l /etc/group  
-rw-r--r-- 1 root root 1104 Nov  3 2020 /etc/group
```

Observație: Deoarece obiectele software își specifică proprietarii și grupurile proprietar ca UID / GID, nu ca nume, și specifică drepturi în legătură cu aceste numere, calitatea unui utilizator sau grup de a fi privilegiat este dată de UID / GID = 0, nu de numele 'root'. Dealtfel, numele se poate schimba.

Tradițional, sistemele UNIX mențineau toata informația despre utilizatori, inclusiv parola criptată, în fișierul '/etc/passwd', care oferă drept de citire pentru toți. Aceasta prezinta o problema de securitate, deoarece putea fi atacat cu diverse programe de spart parole.

Pentru a preveni asemenea atacuri, parolele criptate ale utilizatorilor pot fi stocate în fișierul '/etc/shadow', care nu mai oferă drept de citire pentru toți și poate fi citit doar de procese privilegiate.

Similar, parolele criptate ale grupurilor pot fi stocate în fișierul '/etc/gshadow', care nu oferă drept de citire pentru toți și poate fi citit doar de procese privilegiate:

```
$ls -l /etc/*shadow
-rw-r----- 1 root shadow 926 Nov  3 2020 /etc/gshadow
-rw-r----- 1 root shadow 1408 Nov  3 2020 /etc/shadow
```

Funcții pentru aflarea informațiilor despre un utilizator:

```
#include <sys/types.h>
#include <pwd.h>
struct passwd *getpwnam(const char *name);
struct passwd *getpwuid(uid_t uid);
```

furnizează câmpurile înregistrării referitoare la utilizatorul cu numele 'name', respectiv cu UID-ul 'uid', din '/etc/passwd', plasate separat, în membrii unei structuri de tip 'struct passwd';

structura setată este una internă iar unii membrii ai structurii sunt pointeri la zone, de asemenea, interne; acestea pot fi suprascrise de apeluri ulterioare ale funcțiilor 'getpwnam()', 'getpwuid()', 'getpwent()'; de aceea, codul utilizator trebuie să utilizeze complet conținutul lor sau să-l salveze în altă parte înainte de un nou apel al acestor funcții; de asemenea, codul utilizator nu trebuie să modifice conținutul zonelor interne și nici să furnizeze adresele lor ca argument funcției 'free()';

funcțiile 'getpwnam()', 'getpwuid()' returnează adresa structurii interne în caz de succes sau NULL (și setează 'errno') în caz de eșec.

Tipul structură 'struct passwd' este:

```
#include <pwd.h>
struct passwd {
    char *pw_name;      /* username */
    char *pw_passwd;    /* user password (encrypted) */
    uid_t pw_uid;       /* user ID */
    gid_t pw_gid;       /* group ID */
    char *pw_gecos;    /* Comment (user information, ex. real name) */
    char *pw_dir;       /* home directory */
    char *pw_shell;     /* login shell */
};
```

Observație: Câmpul 'pw_passwd' conține o informație validă doar dacă nu sunt activate parolele shadow; în program, putem determina dacă sunt activate parolele shadow dacă urmăm un apel cu succes al funcției 'getpwnam()' cu un apel al funcției 'getspnam()' (care furnizează câmpurile înregistrării referitoare la utilizatorul respectiv din '/etc/shadow', plasate separat, în membrii unei structuri de tip 'struct spwd'), pentru a vedea dacă returnează o înregistrare shadow password pentru același username.

Observăm că structurile passwd nu conțin sirurile ci doar adresele lor (conțin pointeri, nu vectori). De aceea, dacă vrem să facem o copie completă a structurii inițializate de 'getpwnam()', 'getpwuid()' înainte de un nou apel al acestora, nu este suficientă o simplă atribuire de structuri (shallow copy):

```
struct passwd *s = getpwnam('ion');
struct passwd s1;
s1 = *s;
```

deoarece s-ar duplica doar adresele, nu și conținuturile sirurilor (de exemplu, dacă se modifică ulterior sirul indicat de 's -> pw_name', se va modifica și sirul indicat de 's1 . pw_name').

Pentru copierea tuturor informațiilor, structura trebuie parcursă în adâncime (deep copy). De exemplu, putem folosi funcțiile următoare:

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

struct passwd *newpwd(struct passwd *src) {
    struct passwd *dst;
    if((dst = malloc(sizeof(struct passwd))) == NULL) goto err1;
    if((dst -> pw_name = malloc((strlen(src->pw_name) + 1) * sizeof(char))) == NULL) goto err2;
    if((dst -> pw_passwd = malloc((strlen(src->pw_passwd) + 1) * sizeof(char))) == NULL) goto err3;
    if((dst -> pw_gecos = malloc((strlen(src->pw_gecos) + 1) * sizeof(char))) == NULL) goto err4;
    if((dst -> pw_dir = malloc((strlen(src->pw_dir) + 1) * sizeof(char))) == NULL) goto err5;
    if((dst -> pw_shell = malloc((strlen(src->pw_shell) + 1) * sizeof(char))) == NULL) goto err6;
    strcpy(dst -> pw_name, src -> pw_name);
    strcpy(dst -> pw_passwd, src -> pw_passwd);
    strcpy(dst -> pw_gecos, src -> pw_gecos);
    strcpy(dst -> pw_dir, src -> pw_dir);
    strcpy(dst -> pw_shell, src -> pw_shell);
    dst -> pw_uid = src -> pw_uid;
    dst -> pw_gid = src -> pw_gid;
    return dst;
/* error path*/
err6: free(dst -> pw_dir);
err5: free(dst -> pw_gecos);
err4: free(dst -> pw_passwd);
err3: free(dst -> pw_name);
err2: free(dst);
err1:
    return NULL;
}
```

```
#include <stdlib.h>

void freepw(struct passwd *s) {
    free(s -> pw_name);
    free(s -> pw_passwd);
    free(s -> pw_gecos);
    free(s -> pw_dir);
    free(s -> pw_shell);
    free(s);
}
```

Exemplu de utilizare:

```
struct passwd *s;
if((s = newpw(getpwnam("dragulici"))) != NULL) {
    printf("%d\n", (int) s -> pw_uid);
    freepw(s);
}
```

Funcții pentru aflarea informațiilor despre un grup:

```
#include <sys/types.h>
#include <grp.h>
struct group *getgrnam(const char *name);
struct group *getgrgid(gid_t gid);
```

furnizează câmpurile înregistrării referitoare la grupul cu numele 'name', respectiv cu GID-ul 'gid', din '/etc/group', plasate separat, în membrii unei structuri de tip 'struct group';

structura setată este una internă, unii membrii ai structurii sunt pointeri la zone, de asemenea, interne, acestea pot fi suprascrise de apeluri ulterioare ale funcțiilor 'getgrnam()', 'getgrgid()', 'getgrent()' iar restul comentariilor sunt asemanatoare ca în cazul funcțiilor 'getpwnam()', 'getpwuid()'.

funcțiile 'getgrnam()', 'getgrgid()' returnează adresa structurii interne în caz de succes sau NULL (și setează 'errno') în caz de eșec.

Tipul structura 'struct group' este:

```
#include <grp.h>
struct group {
    char    *gr_name;
    /* group name */
    char    *gr_passwd;
    /* group password (encrypted) (if not password shadowing);
       this field is not specified in SUSv3, but is available
       on most UNIX implementations */
    gid_t   gr_gid;
    /* group ID */
    char   **gr_mem;
    /* group members (NULL-terminated array of pointers to
       names of members listed in '/etc/group' */
};
```

Scanarea înregistrarilor din '/etc/passwd':

```
#include <sys/types.h>
#include <pwd.h>
struct passwd *getpwent(void);
void setpwent(void);
void endpwent(void);
```

- Funcția 'getpwent()' furnizează câmpurile unei înregistrări din '/etc/passwd', plasate separat, în membrii unei structuri interne de tip 'struct passwd', a cărei adresa o returnează;
la primul apel deschide fișierul și returnează prima înregistrare, la urmatoarele apeluri returnează înregistrările succesive următoare, iar dacă nu mai sunt înregistrari, returnează NULL;
în caz de eroare, returnează NULL și setează 'errno';
structura setată este una internă, unii membrii ai structurii sunt pointeri la zone, de asemenea, interne, acestea pot fi suprascrisse de apeluri ulterioare ale funcțiilor 'getpwnam()', 'getpwuid()', 'getpwent()', restul comentariilor sunt asemănătoare ca în cazul funcțiilor precedente.

- Funcția 'setpwent()' derulează înapoi de la începutul fișierului '/etc/passwd' parcurgerea efectuată de apelurile funcției 'getpwent()'.
- Funcția 'endpwent()' închide fișierului '/etc/passwd' după ce au fost efectuate toate procesările; un apel ulterior al lui 'getpwent()' va deschide fișierul și va scana de la început.

Exemplu: putem parurge '/etc/passwd' și afișa pentru fiecare înregistrare username și UID, astfel:

```
struct passwd *pwd;
while ((pwd = getpwent()) != NULL)
    printf("%-8s %5ld\n", pwd->pw_name, (long) pwd->pw_uid);
endpwent();
```

Scanarea înregistrarilor din '/etc/group':

```
#include <sys/types.h>
#include <grp.h>
struct group *getgrent(void);
void setgrent(void);
void endgrent(void);
```

(comentariile sunt similare).

Există funcții asemănătoare pentru consultarea fișierului '/etc/shadow'. Ele necesită însă privilegii speciale din partea procesului apelant (proprietar 'root', sau grup proprietar 'shadow'), a se vedea exemplul anterior cu comanda 'ls -l /etc/shadow'.

Câteva programe utilizare (utilizabile din linia de comandă shell) cu utilizatori și grupuri: 'useradd', 'usermod', 'userdel', 'groupadd', 'groupmod', 'groupdel', 'passwd', 'gpasswd', 'finger', 'id', 'groups', 'su', 'whoami', 'who', 'w' (vor fi prezentate în capitolul 'Interfața linie de comandă').

Exemplu: implementarea comenzi 'groups username', care afișază apartenențele la grupuri ale utilizatorului cu numele 'username':

```
#include <sys/types.h>
#include <pwd.h>
#include <grp.h>
#include <string.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
    struct passwd *pu; struct group *pg; char **p;
    if(argc != 2) {
        fprintf(stderr, "Utilizare: %s username\n", argv[0]); return 1;
    }
    if((pu = getpwnam(argv[1])) == NULL) {
        perror(argv[1]); return 1;
    }
    printf("%s : ", argv[1]);
    pg = getgrgid(pu -> pw_gid); printf(" %s", pg -> gr_name);
    while((pg = getgrent()) != NULL)
        for(p = pg -> gr_mem; *p; ++p)
            if(strcmp(argv[1], *p) == 0) {
                printf(" %s", pg -> gr_name);
                break;
            }
    printf("\n");
    return 0;
}
```

Testare (presupunem că programul sursă este 'mygroups.c'):

```
$gcc -Wall -o mygroups mygroups.c
$./mygroups dragulici
dragulici : dragulici adm cdrom sudo dip plugdev lpadmin sambashare
$groups dragulici
dragulici : dragulici adm cdrom sudo dip plugdev lpadmin sambashare
```

Cuprins

1 Sisteme de calcul, sisteme de operare

Organizarea unui sistem de calcul
Funcțiile sistemului de operare

2 Sisteme de operare uzuale

Incursiune în sistemul Windows
Incursiune în sistemul Linux

3 Interfața de programare C

Obiecte software și apeluri sistem
Utilizatori și grupuri

Fișiere și directoare

Procese
Semnale
Tuburi

Biblioteca standard C (tipul 'FILE')

4 Interfață interactivă linie de comandă

Interfața Command Prompt
(Windows)
Interfața Shell (Linux)

5 Interfață de scripting linie de comandă

Interfața Batch Scripting (Windows)
Interfața Shell Scripting (Linux)

6 Teme de laborator

Utilizatori și grupuri
Fișiere, directoare
Procese, semnale, tuburi

Fișierul (file) este o unitate abstractă de organizare a informației, prin care SO prezintă informația către softul de nivel superior. Fișierele desemnează de obicei colecții de date stocate pe un suport extern (disc, discheta, banda, CD-ROM, etc.), dar pot desemna și echipamente (terminale, imprimante, etc.) sau alte surse de informații.

În sistemele UNIX/Linux există un concept general de fișier, pentru care sunt implementate mecanisme generale, și care este încorporat în 7 - 8 tipuri de fișier, adăugându-se la mecanismele generale și mecanisme specifice.

La nivel general, un fișier:

- este gestionat de sistem cu ajutorul unei structuri de date, care este adresată intern (în algoritmii interni) printr-un număr și extern (via interfața AS) prin (0 sau mai multe) nume;
- poate fi accesat din procese prin descriptori numerici, alocați la nivel de proces (un același descriptor poate avea, pentru procese diferite, semnificații diferite);
- procesele pot citi scrie în fișier folosind apelurile 'open()', 'close()', 'read()', 'write()' (the universal I/O model).

Tipurile de fișier din sistemele UNIX/Linux sunt:

- Fișier obișnuit (regular file) - conține informații fără o semnificație sau organizare specială pentru sistem; sunt fișierele uzuale ale utilizatorilor.
- Fișier director (directory) - conține o listă de corespondențe nume - adresă pentru alte fișiere obișnuite/directoare/fișiere speciale;
- Fișier special (special file).

Ultimul tip are următoarele subtipuri:

- Fișier bloc (block file) - oferă acces (citire/scriere) aleator la informație; ele modelează echipamente de I/O de tip bloc, ex. partiții pe discuri fizice.
- Fișier special caracter (character device file) - oferă acces (citire/scriere) secvențial la informație (stream) (ex. terminale logice sau pseudoterminale logice); ele se pot asocia terminalelor fizice sau ferestrelor pe desktop în modul grafic.
- Tub (pipe) - fișier ce funcționează după o disciplină de coadă (FIFO): octetii pot fi citiți doar în ordinea în care au fost scriși, iar citirea lor îi elimină din tub; tuburile sunt un instrument de comunicare între procese aflate în aceeași instanță UNIX/Linux.
- Socket (socket) - este folosit în comunicarea între procese ce pot să nu fie în aceeași instanță UNIX/Linux.
- Legătură simbolică (symbolic link, soft link) - conține o referință către un alt fișier, sub forma reprezentării textuale a unei căi; de obicei apelurile sistem, când sunt aplicate unei legături simbolice, se transferă automat fișierului referit (deferențiază legătura simbolică); există și excepții.

Analogul legăturilor simbolice în sistemele Windows sunt shortcut-urile, implementate sub forma unor fișiere cu extensia '.lnk', ce conțin referințe către alte fișiere; când se execută dublu-click pe pictograma unui shortcut, rutina declanșată se aplică automat fișierului referit.

În sistemele UNIX (Sun Solaris) există și tipul de fișier special poartă (door file) folosit pentru comunicarea inter-proces dintre un client și un server.

Utilizatorul poate distinge mai multe tipuri de fișiere, în funcție de sensul pe care îl dă informațiilor conținute în ele, și poate implementa punctul său de vedere în aplicații. De exemplu, poate distinge între fișiere text și fișiere binare (programe executable). Această clasificare însă nu are legătură cu tipurile implementate în SO. De exemplu, pentru SO, fișierele text și cele binare sunt fișiere regular.

Fișierele sunt resurse ale SO. Partea din SO care se ocupă de gestiunea fișierelor s.n. sistem de fișiere (file system). Ea trebuie să asigure atât mecanisme de implementare a fișierelor, cât și mecanisme de interfațare între fișiere și softul de nivel utilizator (de regulă, sub forma unor apeluri sistem); în particular, trebuie să ofere un mecanism de protecție pentru a permite utilizatorilor să-și administreze accesul la fișiere.

Adesea, prin sistem de fișiere se înțelege doar ansamblul de informații sub forma căruia este stocată colectia de fișiere pe un suport de memorie (ex. disc), nu și mecanismele din SO folosite pentru a o manevra; în funcție de modul de organizare a acestor informații, există mai multe tipuri de sistem de fișiere: FAT32, NTFS, EXT3, etc. De multe ori, numele acestor tipuri de sistem de fișiere este asimilat/înlocuit cu numele primului SO care le-a folosit - de ex. sistemul de fișiere FAT se mai numește sistemul de fișiere (de la) MS-DOS.

Unele SO suportă (prin mecanismele încorporate) doar un singur tip de sistem de fișiere, altele mai multe - de ex. Linux, unele variante de UNIX, suportă: FAT (de la MS-DOS), FAT32 (de la Windows 95 versiunea 2 și Windows 98), NTFS (de la Windows NT), EXT3 (sistem nativ al Linux).

Acest lucru este posibil prin introducerea conceptului de sistem de fișiere virtual (virtual file system).

Gestionarul de fișiere este construit deasupra unui model de fișier abstract care este livrat (exportat) de componenta VFS (virtual file system).

VFS implementează operații independente de sistemul de fișiere; SO integrează extensii ale VFS ce adaugă operațiile dependente de anumite tipuri de sistem de fișiere; de ex. versiunea 2.x a VFS suportă formatele MS-DOS, MINIX, /proc, ext3, etc.

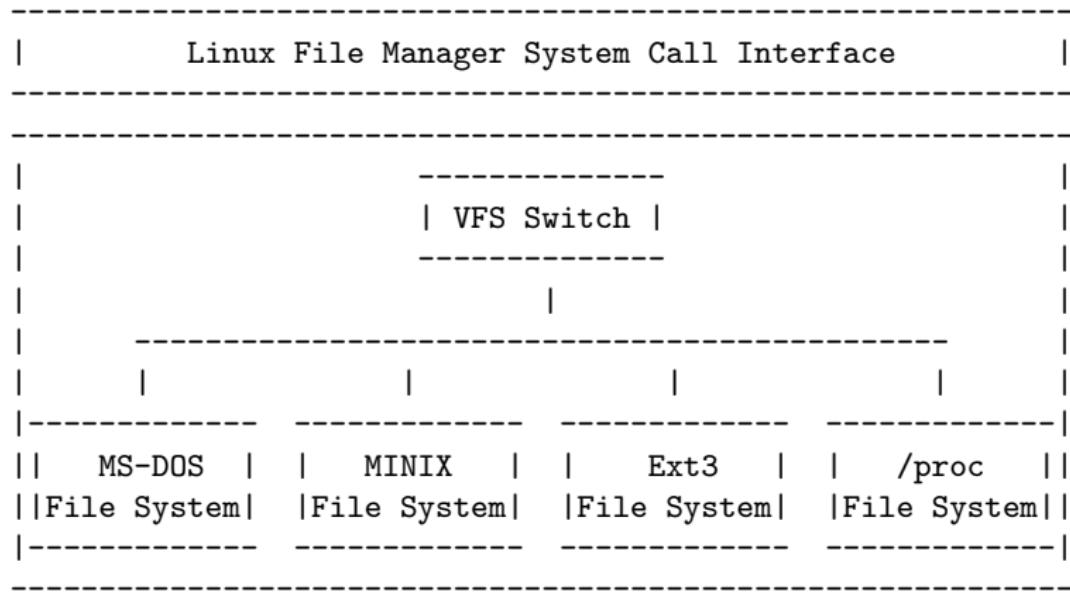
Esența VFS este comutatorul (switch); el oferă:

- o interfață cu programele utilizator (API), care livrează acestora modelul de fișier abstract și operațiile corespunzătoare;
- o interfață internă folosită de translatoare.

Un translator integrează suportul pentru un anumit tip de sistem de fișiere concret (FAT32, ext3, etc); el implementează operațiile dependente de sistemul de fișiere respectiv (strategia de organizare a informației pe disc, citirea/scriere proprietăților discului, a structurilor de informații asociate fișierelor (numite acum descriptori externi) (ex. i-nodurile), a blocurilor de date ale fișierelor, etc.).

Suportul pentru un nou tip de sistem de fișiere este adăugat sub forma unui nou translator.

Exemplu: comutatorul VFS din Linux:



Modelul abstract de fișier livrat de VFS este inspirat de UNIX; în particular, orice asemenea fișier abstract are asociată o structură cu informații (descriptor intern), asemănătoare i-nodurilor și care s.n. v-nod (v-node, virtual node); v-nodul are propriul lui format, adaptat abordării sistemului de fișiere multiplu.

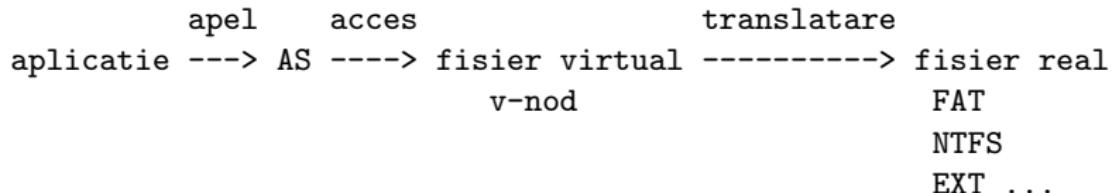
Când un fișier este deschis, translatorul folosit convertește conținutul descriptorului extern în formatul VFS V-NODE, și îl scrie în v-nodul asociat acestuia ca fișier abstract; când fișierul este închis, conținutul v-nodului este folosit pentru a actualiza descriptorul extern.

Un v-nod conține informații privind drepturile de acces, proprietarul, grupul acestuia, dimensiunea, momentul creării, momentul ultimei accesări, momentul ultimei modificări, etc. și un spațiu pentru mecanismul de pointeri pe care sistemul de fișiere concret îl folosește pentru a organiza blocurile pe disc, chiar dacă VFS nu știe cum vor fi organizați acești pointeri (acestă informație este încapsulată în translatorul folosit).

VFS suportă și directoare, presupunând că directoarele externe conțin în fiecare intrare cel puțin un nume și o adresă de descriptor extern (aproape toate tipurile de sistem de fișiere oferă acest minim de informație).

Înainte ca VFS să poată lucra cu un tip particular de sistem de fișiere, trebuie scris un translator pentru acel tip și apoi înregistrat în VFS.

Așadar, din aplicații, prin interfața AS, este vizibil un singur model de fișier, fișierul virtual, gestionat cu un v-nod; sistemul va pune în corespondență fișierul virtual cu un fișier real, aflat pe un suport:



Întrucât modelul fișierelor virtuale, gestionate cu v-noduri, seamănă cu cel al fișierelor UNIX/Linux, gestionate cu i-noduri, atunci când fișierul virtual corespunde unui fișier EXT, informația încărcată în v-nod este realistă - seamănă cu cea din i-nod (proprietar, drepturi, etc.); când fișierul virtual corespunde unui fișier dintr-un alt tip de file system (de exemplu FAT - aici nu există proprietar, drepturi, ci attribute Archive, Read-only, Hidden, System) informația încărcată în v-nod este una convențională - depinde de modul de montare logică a aceluiași file system.

Deși POSIX standardizează sisteme de tip UNIX/Linux, el nu dă o definiție a unui i-nod, ci doar folosește termenul 'file serial number' pentru a referi ceea ce este ușual cunoscut ca 'număr de i-nod', anume numărul unic al unei intrări fișier dintr-un file system - acesta trebuie să fie de tipul '`ino_t`', definit în `<sys/types.h>` ca un întreg fără semn.

Există instrumente (ex. funcția și tipul structură 'stat') cu ajutorul cărora se pot obține informațiile care descriu fișierul.

Așadar, AS operează cu fișiere virtuale, gestionate cu v-noduri.

Având în vedere asemănarea acestora cu fișierele UNIX/Linux, pentru a ilustra mecanismele interne ale SO legate de fișiere, vom considera un sistem Linux și fișiere EXT, gestionate cu i-noduri.

Zona de date a fișierului este, dpv. logic, o secvență finită, eventual vidă, de octeți, care începe de la o anumită adresă; în aceasta secvență, fiecare octet are un deplasament față de începutul secvenței, deplasamentele sunt numere naturale 0, 1, 2, ...; există posibilitatea de a lucra cu fișiere care au goluri logice (la anumite deplasamente nu sunt octeți), dar se poate scrie ulterior în ele.

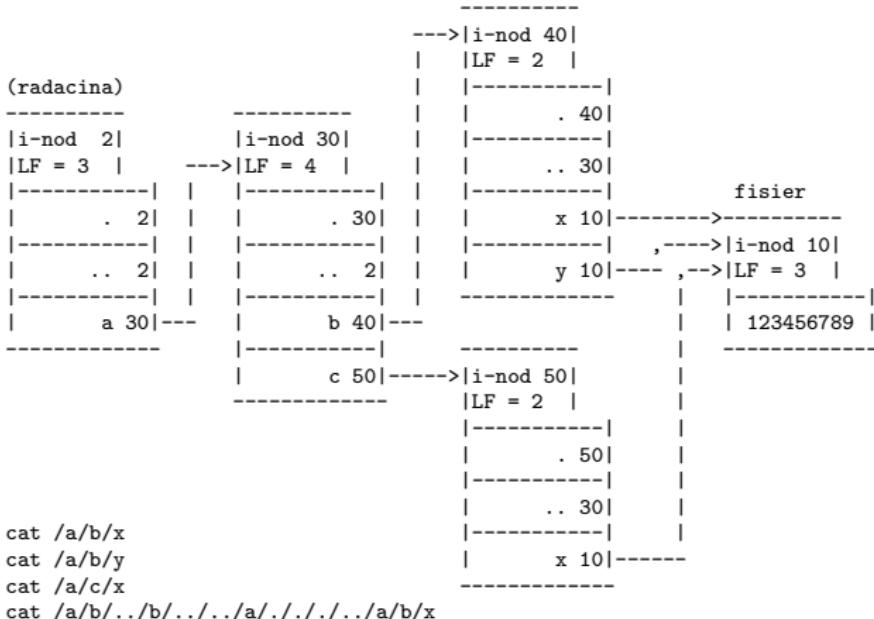
Fișierul este gestionat cu o structură de date cu informații despre el, numită i-nod (index node), desemnată intern (în structurile de date și algoritmii interni) printr-un număr de i-nod, iar extern (în interacțiunea cu utilizatorii, prin interfața AS) prin nume (legături fizice, hard link). Numele este un string unitar, nu se reține separat un nume și o extensie, dar numele poate conține '.', chiar de mai multe ori, ca un caracter obișnuit.

Evidența legăturii nume - numar de i-nod pentru diverse obiecte ale sistemului de fișiere este menținută în directoare; acestea sunt un tip de fișiere, a căror zonă de date este organizată ca o listă de intrări, o intrare reținând o corespondență nume - numar de i-nod (și alte proprietăți) pentru un fișier, alt director, etc. În i-nodul fiecărui fișier, există un contor cu numărul curent de legături fizice (nume) pe care le are.

După apartenența la directoare, obiectele sistemului de fișiere se organizează în arborescențe. Sistemul Linux expune în interfață să cu aplicațiile un arbore unic (toate căile sunt urmărite în același arbore), deși obiectele care figurează în acest arbore pot fi stocate fizic pe partiții diferite - sistemele de fișiere de pe diverse partiții pot fi montate logic ca subarbore în arborele unic (comenzile 'mount', 'umount').

Pentru a putea folosi același mecanism de urmărire și în cazul căilor care conțin '.' și '..', în fiecare director există două intrări care conțin numele '.', resp. '..', asociate numărului de i-nod propriu, resp. numărului de i-nod al directorului părinte.

Exemplu:



Comenzile de mai sus afișază același fișier cu i-nodul 10, care conține '123456789'.

La fiecare pas, algoritmul de parcurgere caută numele următor pe cale în directorul la care a ajuns, pentru a afla i-nodul directorului/fișierului următor. Putem considera că rădăcina are numele vid.

Notăm următoarele:

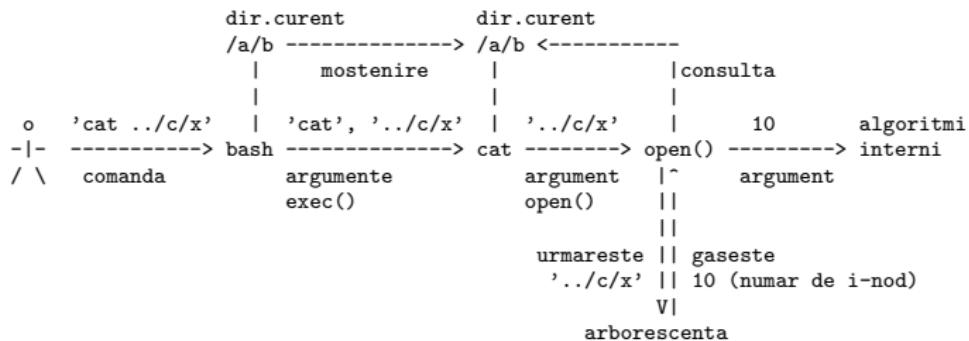
- Un obiect al sistemului de fișiere poate avea mai multe nume (legături fizice), în diverse directoare; de aceea, arborele unic expus în interfața sistemului nu este o descriere fidelă a situației de pe suportul de memorie.
- Dacă un obiect al sistemului de fișiere are numarul de i-nod n și numărul de legături fizice $LF = k$, înseamnă ca n apare în k intrări de director, având asociate diverse nume.
- Într-un director nu pot exista 2 intrări care să conțină același nume (deoarece regula de urmărire a căilor ar fi ambiguă).
- Un director are cel puțin 2 intrări, '.' și '..' și $LF \geq 2$.
- Un director gol este un director care conține doar intrările '.', '..', deci nu este un fișier vid (0 bytes); în general, directoarele au o dimensiune multiplu de bloc (de exemplu, multiplu de 4k).
- Dacă director are $LF = 2$, nu înseamnă că este gol (doar cu '.' și '..') ci doar că nu are subdirectoare (poate conține alte tipuri de fișiere).
- Dacă un director cu numele 'd' are $LF = k$, înseamnă că numărul lui de i-nod n apare o dată în directorul său părinte, cu numele 'd', o dată în el însuși, cu numele '.', și de $k - 2$ ori în subdirectoare ale sale, cu numele '..'.

Fiecare partitie EXT are propria tabela de i-noduri, în care numărătoarea i-nodurilor se reia de la 0. De aceea, un fișier este identificat complet de perechea număr de i-nod - număr de disc. Un obiect al sistemului de fișiere poate avea legături fizice doar în directoare aflate pe aceeași partitie cu el.

Orice proces are ca atribut un director curent; inițial, îl moștenește de la procesul părinte, ulterior îl poate modifica fără să afecteze directorul curent al altor procese. AS folosesc directorul curent al procesului apelant pentru a începe urmărirea căilor relative (care încep cu '.', '..' sau nume).

Dacă în exemplul anterior shell-ul are directorul curent '/a/b' (cu i-nodul 40), fișierul se poate afișa și cu comenziile:

```
cat x
cat y
cat ./x
cat ../c/x
cat ../../a/././..../a/b/x
```



Pentru accesarea fișierelor, la nivelul fiecarui proces există câte o tabela de descriptori (TD) iar la nivelul instanței SO o tabelă a deschiderilor de fișiere (TDF) și o tabelă a i-nodurilor accesate (TIA). Acestea sunt organizate ca array de structuri.

Dacă un proces a asociat unui fișier un descriptor *i* (număr natural), intrarea *i* din TD a să e completată cu informații specifice, printre care un pointer la o intrare din TDF; primele 3 intrări din TD, corespunzătoare descriptorilor 0, 1, resp. 2, semnifică intrarea standard (standard input), ieșirea standard (standard output), resp. ieșirea standard pentru erori (standard error) a procesului; într-un program C, cei 3 descriptori pot fi desemnați și prin constantele simbolice definite în <unistd.h>: STDIN_FILENO (=0), STDOUT_FILENO (=1), resp. STDERR_FILENO (=2).

O intrare din TDF corespunde unei 'deschideri' a unui fișier de către un proces și conține printre altele:

- numărul total de descriptori asociați acestei intrări de diverse procese;
- modul de deschidere (citire, scriere, citire și scriere);
- pozitia curentă în fișier;
- un pointer la o intrare din TIA.

O intrare din TIA corespunde i-nodului unui fișier curent accesat de procese și conține, printre altele, numărul intrărilor din TDF care pointează spre ea.

Mai mulți descriptori (din același proces sau din proceze diferite) pot pointa o aceeași intrare din TDF, mai multe intrări din TDF pot pointa o aceeași intrare din TIA, iar aceasta corespunde unui fișier; i-nodul unui fișier accesat se încarcă într-o singură copie în TIA și toate accesele (intrările TDF) o pointează pe aceasta.

Exemplu:

TD pt. Proces 1

```
---  
0 | |  
. | |  
. | |  
. ---  
i | a |  
PS1 ---  
j | b |  
. ---  
. | |  
. | |  
| | |  
---
```

TD pt. Proces 2

```
---  
0 | |  
. | |  
. | |  
. ---  
k | a |  
PS2 ---  
l | d |  
---  
m | d |  
---  
n | c |  
. ---  
| | |  
---
```

TDF

```
---  
0 | |  
. | |  
. | |  
. ---  
a | 2 | r | 100 | x  
---  
b | 1 | rw | 12 | x  
---  
. | |  
. | |  
. | |  
c | 1 | w | 55 | y  
---  
d | 2 | r | 0 | z  
---  
. | |  
. | |  
. | |  
---
```

TIA

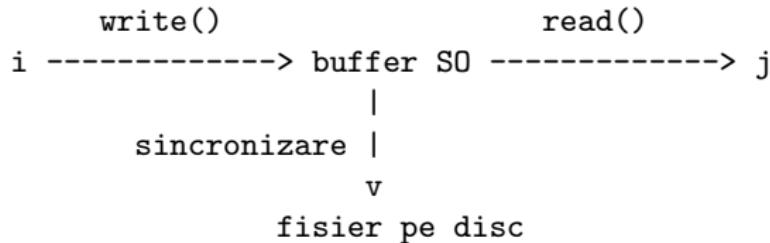
```
---  
0 | |  
. | |  
. | |  
. ---  
x | 2 | |  
---  
. | |  
. | |  
y | 1 | |  
---  
. | |  
. | |  
. | |  
z | 1 | |  
---
```

Observații:

- Dacă doi descriptori i și j (din același proces sau proceze diferite) pointează aceeași intrare din TDF (implicit, se referă la același fișier), operațiile făcute asupra fișierului prin intermediul lor vor fi de același fel (citire, scriere, citire și scriere) și vor folosi același indicator de pozitie curentă (modul de deschidere și indicatorul fiind conținute în intrarea din TDF) - deci o citire via i urmată de o citire via j vor furniza informații succesive din fișier.
- Dacă doi descriptori i și j (din același proces sau proceze diferite) pointează intrări diferite din TDF care se referă la același fișier, operațiile făcute asupra fișierului prin intermediul lor pot fi în moduri diferite (citire / scriere) și vor folosi indicatori de poziție curentă diferenți - deci o citire via i urmată de o citire via j ar putea furniza aceleași informații din fișier.

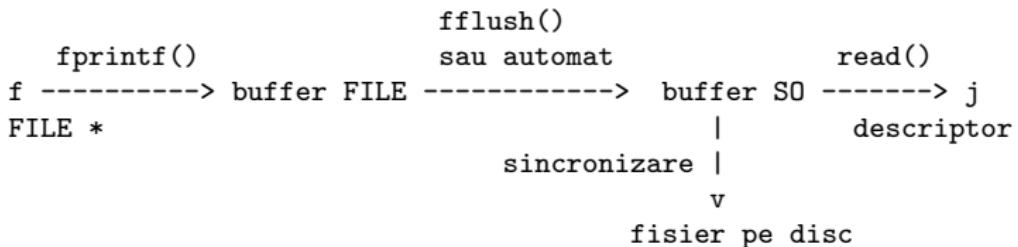
Citirile/scrierile din fișiere efectuate în mod bloc (mecanismul pe care îl vom presupune în cele ce urmează) sunt mediate de niște buffere ale sistemului; sistemul își scrie automat, periodic, informațiile din buffere pe disc (operația de sincronizare).

Bufferul pentru un fișier este asociat logic intrării din TIA, deci dacă prin doi descriptori i și j (din același proces sau procese diferite), care pointează intrări diferite din TDF cu moduri de deschidere diferite, se operează asupra unui același fișier, o scriere prin primul descriptor urmată de o citire prin al doilea descriptor vor furniza informații consistente, chiar dacă acestea nu au ajuns încă pe disc:



Aceasta mai înseamnă și că nu este necesară implementarea unei bufferizări în programul utilizator, în ideea că scrierea informației un caracter o dată - preupune mai multe accese la disc decât scrierea un grup de caractere o dată - 'write()' va scrie în bufferul SO, iar sincronizarea cu discul o face SO independent de programul utilizator. Este de preferat însă scrierea un grup de caractere o dată, deoarece se fac mai puține apele la 'write()' (care fac TRAP către SO, întrerupând procesul).

În cazul folosirii funcțiilor de nivel superior (care manipulează fișierele cu ajutorul unor structuri 'FILE'), operațiile sunt mediate de două niveluri de buffere: cel din programul utilizator, asociat structurilor 'FILE', apoi cel al SO. Atunci, pot apărea inconsistențe dacă bufferul FILE nu este evacuat la timp (automat sau cu 'fflush()') în bufferul SO:



(informația scrisă de 'fprintf()' poate rămâne în bufferul FILE din spațiul procesului utilizator și nu ajunge în bufferul SO de unde încearcă să citească 'read()').

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

determină ca toate modificările în curs privind fișierele și aflate în buffere să fie scrise în sistemul de fișiere subiacent; are întotdeauna succes.

Corespunzător acestei funcții este comanda shell 'sync'.

Modelul de I/O universal se bazează pe apelurile 'open()', 'close()', 'read()', 'write()'. Adițional, se pot folosi 'dup()', 'lseek()', etc.:

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
int open(const char *pathname, int flags [, mode_t mode]);
```

Efectuează o deschidere a unui fișier pentru operații;

pathname = specificatorul fișierului;

flags = tipul operațiilor pentru care se face deschiderea (citire, scriere, citire și scriere) și anumite acțiuni făcute cu ocazia deschiderii; se poate construi ca o disjuncție pe biți de următoarele constante simbolice (pentru ele putem include <fcntl.h>):

- exact una dintre:

- 0_RDONLY = deschidere doar pentru citire;

- 0_WRONLY = deschidere doar pentru scriere;

- 0_RDWR = deschidere pentru citire și scriere;

(informația specificată de ele va fi scrisă în intrarea creată în TDF);

- 0 sau mai multe dintre:

`O_TRUNC` = dacă fișierul există, este regular și am specificat `O_WRONLY` sau `O_RDWR`, vechiul conținut se șterge;

`O_CREAT` = dacă fișierul nu există, se va crea (ca fișier regular vid); în acest caz, trebuie să fie prezent și parametrul 'mode';

`O_EXCL` = dacă am folosit `O_CREAT` și fișierul există, funcția se termină cu eroare;

`O_APPEND` = înainte de fiecare scriere în fișier indicatorul poziției curente se va mută automat la sfârșit;

`O_NONBLOCK` = apelul lui `'open()'` este neblocant (în cazul când ar trebui să producă adormirea procesului (vezi mai jos), n-o face ci se termină cu eroare);

`O_SYNC` = orice scriere în fișier va bloca procesul până când informația ajunge efectiv din bufferul sistemului pe disc;

mode (parametru optional, dar obligatoriu dacă se crează un fișier nou, vezi `O_CREAT`) = precizează drepturile de acces asupra fișierului creat și se poate construi ca disjuncție pe biți ('|') de următoarele constante simbolice (definite în `<sys/stat.h>`):

`S_IRWXU` (= 700 octal), `S_IRUSR` (= 400 octal), `S_IWUSR` (= 200 octal), `S_IXUSR` (= 100 octal),
`S_IRWXG` (= 070 octal), `S_IRGRP` (= 040 octal), `S_IWGRP` (= 020 octal), `S_IXGRP` (= 010 octal),
`S_IRWXO` (= 007 octal), `S_IROTH` (= 004 octal), `S_IWOTH` (= 002 octal), `S_IXOTH` (= 001 octal);

drepturile cu care va fi creat în realitate fișierul vor fi '`mode & ~umask`' (conjuncție și negație pe biți), unde '`umask`' este masca de drepturi a procesului apelant (este un atribut al proceselor); pentru a seta exact drepturile specificate de '`mode`', se poate apela înainte '`umask(0)`' (masca procesului curent devine 0);

Efect: dacă fișierul nu există și cerem crearea lui (`O_CREAT`), se va crea (apare un i-nod nou); dacă i-nodul fișierului nu e încărcat în TIA, se încarcă; se crează o nouă intrare în TDF, ce pointează spre intrarea în TIA corespunzătoare i-nodului fișierului (chiar dacă mai există o intrarea în TDF ce pointa spre acea intrare în TIA); se găseste un descriptor liber în TD procesului apelant și se alocă intrarea respectivă - ea va pointa intrarea creată în TDF.

Returnează: descriptorul alocat în caz de succes sau -1 (și setează '`errno`') în caz de eșec.

Exemplu:

```
int d1, d2;  
d1=open("nae.txt",O_RDONLY); d2=open("nae.txt",O_RDWR);
```

TD a procesului	TDF	TIA
---	-----	-----
0	0	0
. ---	. -----	.
d1 --> a	a 1 r 0 x	.
. ---	. -----	. -----
.	.	x 2 "nae.txt"
---	. -----	. -----
d2 --> b	b 1 rw 0 x	.
---	-----	-----

(o citire via 'd1' urmată de o citire via 'd2' vor furniza însă aceeași informație, deoarece se folosesc indicatori de pozitie independenti).

Apelul 'open()' poate fi blocant dacă nu folosim O_NONBLOCK - de exemplu, se încearcă deschiderea unui tub cu nume pentru citire și nu există procese care scriu în el, sau pentru scriere și nu există procese care citesc din el (a se vedea mai încolo).

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat(const char *pathname, mode_t mode);
```

Deschide un fișier pentru suprascriere, creând fișierul dacă nu există; este echivalent cu apelul 'open()' cu 'flags' egal cu 'O_CREAT|O_WRONLY|O_TRUNC'.

```
#include<unistd.h>
int close(int desc);
```

Închide descriptorul 'desc' (intrarea sa din TD redevine liberă); ca efect, în intrarea pointată în TDF numărul descriptorilor asociați scade cu 1; dacă devine 0, ea este eliminată din TDF iar în intrarea din TIA pointată de ea numarul deschiderilor (numarul intrărilor din TDF care o pointează) scade cu 1; dacă devine 0, i-nodul respectiv este scos din TIA, iar dacă numărul legăturilor fizice la el este 0, el și fișierul respectiv sunt șterse de pe disc (unele fișiere, ca tuburile fără nume, nu au legături fizice și există în sistem doar cât timp există procese care le acceseză).

Returneaza: 0 în caz de succes sau -1 (și setează 'errno') în caz de eșec.

Exemplu:

TD a procesului

0 | |
d1 --> | a |
. ---
d2 --> | b |
. ---
d2 --> | c |
. ---

TDF

0 | |
a | 3 | | x |
. ---
b | 1 | | y |
. ---
c | 1 | | z |
. ---

TIA

0 | |
x | 2 | |
. ---
y | 2 | |
. ---
z | 1 | |
. ---

|| close(d1); close(d2); close(d3);
\\

0 | |
. ---
d1 --> | | |
. ---
d2 --> | | |
. ---
d2 --> | | |

0 | |
a | 2 | | x |
. ---

0 | |
x | 2 | |
. ---
y | 1 | |
. ---

```
#include<unistd.h>
int dup(int oldfd);
```

Alocă un nou descriptor în TD pentru aceeași intrare din TDF ca 'oldfd'; noul descriptor este cel mai mic descriptor liber; returnează noul descriptor în caz de succes sau -1 (și setează 'errno') în caz de eșec.

```
#include<unistd.h>
int dup2(int oldfd, int newfd);
```

Alocă forțat descriptorul 'newfd' pentru aceeași intrare în TDF ca 'oldfd'; dacă 'newfd' nu era liber, sistemul îl închide în prealabil în mod tacit; returnează noul descriptor (adică 'newfd') în caz de succes sau -1 (și setează 'errno') în caz de eșec.

Exemplu:

TD a procesului	TDF	TIA
---	-----	-----
0	0	0
.	.	.
d1 --> a	a 1 x	x
.	.	.
..	.	.
---	-----	-----
---	-----	-----
0	0	0
.	.	.
d1 --> a	a 2 x	x
.	.	.
d2 --> a	.	.
---	-----	-----

|| d2=dup(d1);
 \/\

```
#include<sys/types.h>
#include<unistd.h>
off_t lseek(int desc, off_t offset, int origine);
```

Poziționează indicatorul de poziție din intrarea în TDF asociată descriptorului 'desc' la distanța 'offset' de 'origine'; aceasta va afeca toți celilalți descriptori care sunt asociați cu intrarea TDF respectivă; 'origine' poate fi:

SEEK_SET (=0) = începutul fișierului

SEEK_CUR (=1) = poziția curentă

SEEK_END (=2) = sfârșitul fișierului

(constantele simbolice sunt definite în <unistd.h>); pentru tipul 'off_t' putem include <sys/types.h> și poate fi 'long';

Funcția returnează pozitia curentă față de începutul fișierului în caz de succes sau (off_t)-1 (și setează 'errno') în caz de eșec.

Exemple:

lseek(d, (off_t)0, SEEK_SET);

poziționare la începutul fișierului (returnează (off_t)0);

lseek(d, (off_t)0, SEEK_CUR);

poziția curentă nu se schimbă (și este returnată - astfel o putem determina);

lseek(d, (off_t)0, SEEK_END);

poziționare la sfârșitul fișierului (returnează dimensiunea fișierului);

```
#include<unistd.h>
ssize_t read(int desc, void *buf, size_t nr);
```

Citește 'nr' octeți din fișierul indicat de 'desc' în zona pointată de 'buf'; dacă de la poziția curentă până la sfârșitul fișierului sunt mai puțini octeți, se citesc câți sunt; în intrarea corespunzătoare din TDF trebuie să figureze permisiunea 'r' sau 'rw'; indicatorul de poziție din această intrare avansează până după ultimul octet citit; returnează numarul de octeți citiți în caz de succes sau -1 (și setează 'errno') în caz de eșec.

Pentru tipul 'ssize_t' putem include '<unistd.h>' și poate fi 'int' iar pentru tipul 'size_t' putem include tot '<unistd.h>' și poate fi 'unsigned'.

```
#include<unistd.h>
ssize_t write(int desc, void *buf, size_t nr);
```

Scrie 'nr' octeți din zona pointată de 'buf' în fișierul indicat de 'desc'; în intrarea corespunzătoare din TDF trebuie să figureze permisiunea 'w' sau 'rw'; indicatorul de poziție din această intrare avansează până după ultimul octet scris; returnează numarul de octeți scriși în caz de succes sau -1 (și setează 'errno') în caz de eșec.

Exemplu: simularea comenzi 'cp fișier1 fișier2' (dacă se crează fișierul destinație, se folosesc drepturile de citire și scriere pentru proprietar):

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
    int dsrc, ddst; char c;
    if(argc != 3) {
        fprintf(stderr, "Utilizare: %s fsrc fdst\n", argv[0]); return 1;
    }
    if((dsrc = open(argv[1], O_RDONLY)) == -1) {
        perror(argv[1]); return 1;
    }
    if((ddst = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR)) == -1) {
        perror(argv[2]); close(dsrc); return 1;
    }
    while(read(dsrc, &c, 1) == 1) write(ddst, &c, 1);
    close(dsrc); close(ddst);
    return 0;
}
```

Exemplu (accesarea unui fișier prin intrări TDF identice/diferite):

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
    int d1, d2, d3, d4; char c;
    d1 = open("f.txt", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);
    d2 = open("f.txt", O_RDONLY);
    d3 = open("f.txt", O_RDONLY);
    d4 = dup(d3);
    write(d1, "0123456789", sizeof("0123456789")); /* scrie: 0123456789 */
    read(d3, &c, sizeof(char)); printf("%c\n", c); /* afisaza: 0 */
    read(d4, &c, sizeof(char)); printf("%c\n", c); /* afisaza: 1 */
    lseek(d3, 2, SEEK_CUR);
    read(d4, &c, sizeof(char)); printf("%c\n", c); /* afisaza: 4 */
    read(d2, &c, sizeof(char)); printf("%c\n", c); /* afisaza: 0 */
    return 0;
}
```

Prezentăm câteva funcții care operează asupra directoarelor:

```
#include <sys/stat.h>
#include <sys/types.h>
int mkdir(const char *pathname, mode_t mode);
```

Crează directorul specificat de 'pathname' cu drepturile 'mode'.

Drepturile cu care va fi creat în realitate directorul vor fi

'mode & ~umask & 0777'

Returnează 0 în caz de succes sau -1 (și setează 'errno') în caz de eșec.

```
#include <unistd.h>
int rmdir(const char *pathname);
```

Șterge directorul 'pathname'; trebuie să fie gol.

Nu necesită vreun drept pe director, necesită drept de scriere pe directorul părinte, unde se află numele specificat - legătura fizică. Directoarele, însă, nu pot avea mai multe asemenea nume.

Returnează 0 în caz de succes sau -1 (și setează 'errno') în caz de eșec.

Pentru a consulta conținutul directoarelor se folosesc funcții de nivel înalt, care operează asupra directoarelor cu ajutorul unor (pointeri la) structuri de tip 'DIR' (asemănătoare cu 'FILE') sau 'struct dirent'.

Cu ajutorul tipului 'DIR' se gestionează un director.

'DIR' este un tip opac (interfața limbajului C nu expune organizarea sa, el se manevrează ca un tot, folosind funcții predefinite specializate) definit în <dirent.h>. În principiu, el este un tip structură, care conține (printre altele) un indicator de intrare curentă.

Cu ajutorul tipului 'struct dirent' se gestionează o intrare de director.

În Linux, el este definit în <dirent.h> astfel:

```
struct dirent {  
    ino_t          d_ino;        /* Inode number */  
    off_t          d_off;        /* Not an offset */  
    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 */  
};
```

POSIX impune doar prezența membrilor 'd_name', conținând numele (legătura fizică), și 'd_ino', conținând numărul de i-nod, ale fișierului copil respectiv (el trebuie să fie în același sistem de fișiere / partitie cu directorul părinte).

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
```

Deschide directorul 'name' pentru citire, alocă o structură 'DIR' în spațiul de memorie al programului utilizator, o initializează cu informații despre director, poziționează indicatorul pe prima intrare, și returnează adresa structurii; în caz de eroare, returnează NULL și setează 'errno'.

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dirp);
```

Închide directorul gestionat de 'dirp', (și descriptorul de fișier asociat) și dezalocă structura 'DIR' pointată de acesta.

Returnează 0 în caz de succes sau -1 (și setează 'errno') în caz de eșec.

Așadar, structurile 'DIR' nu trebuie alocate cu 'malloc()' sau eliberate cu 'free()', ele sunt alocate la 'opendir()' și eliberate la 'closedir()'.

```
#include <dirent.h>
struct dirent *readdir(DIR *dirp);
```

Furnizează intrarea curentă din directorul indicat de 'dirp' într-o structură internă, avansează indicatorul la următoarea intrare, și returnează adresa structurii.

Dacă indicatorul era la sfârșitul directorului, returnează NULL iar 'errno' nu este modificat.

Dacă a apărut o eroare, returnează NULL iar 'errno' este setat corespunzător. Pentru a distinge între sfârșitul directorului și eroare, se poate seta 'errno' cu 0 înainte de apel și verifica valoarea lui 'errno' dacă apelul a returnat NULL.

Structura internă a cărei adresă este furnizată poate fi suprascrisă de apeluri ulterioare ale funcției pentru aceeași structură 'DIR'; adresa nu trebuie transmisă ca argument lui 'free()'.

În practică, se poate observa că apeluri 'readdir()' asupra același structuri 'DIR' returnează aceeași adresă a unei structuri 'dirent' iar apeluri 'readdir()' asupra unor structuri 'DIR' diferite returnează adrese de structuri 'dirent' diferite. Aceasta sugerează că structura 'dirent' este implementată ca un membru al structurii 'DIR' asociate.

```
#include <dirent.h>
long telldir(DIR *dirp);
```

Returnează poziția curentă a indicatorului structurii 'DIR' indicate de 'dirp'. În caz de eroare, returnează -1 și setează 'errno'.

```
#include <dirent.h>
void seekdir(DIR *dirp, long loc);
```

Mută indicatorul structurii 'DIR' indicate de 'dirp' la poziția 'loc'. Următorul apel 'readdir()' va furniza intrarea de la această poziție. Argumentul 'loc' ar trebui să fie o valoare returnată de 'telldir()'.

```
#include <sys/types.h>
#include <dirent.h>
void rewinddir(DIR *dirp);
```

Mută (re)pozitionează indicatorul structurii 'DIR' indicate de 'dirp' la prima intrare. Astfel, directorul poate fi reparcurs cu 'readdir()' de la început.

Exemplu: Implementarea comenții 'ls -Uai1 director' (afișază toate intrările din director (a), indicând numărul de i-nod (i) și numele, câte una pe linie (1), în ordinea în care apar în director (U)):

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

int main(int argc, char *argv[]) {
    DIR *pd; struct dirent *pde;
    if(argc != 2) {
        fprintf(stderr, "Utilizare: %s director\n", argv[0]);
        return 1;
    }
    if((pd = opendir(argv[1])) == NULL) {
        perror(argv[1]);
        return 1;
    }
    while((pde = readdir(pd)) != NULL) {
        printf("%d %s\n", (int)pde -> d_ino, pde -> d_name);
    }
    closedir(pd);
    return 0;
}
```

```
#include <unistd.h>
int link(const char *oldpath, const char *newpath);
```

Crează o nouă legătură fizică (nume) (hard link) la un fișier existent (adaugă o intrare într-un director). Incrementează contorul cu numărul legăturilor fizice din i-nodul fișierului referit.

Dacă 'newpath' există, el nu este suprascris.

Noul specificator poate fi folosit în același fel ca și vechiul specificator pentru a referi același fișier (același i-nod, deci aceeași proprietate și drepturi) în orice operație, este imposibil de spus care specificator a fost cel inițial.

Funcția returnează 0 în caz de succes sau -1 (și setează 'errno') în caz de eșec.

De exemplu, după un apel 'link("/a/b/f.txt", "/a/x/g.doc")', comenziile 'cat /a/b/f.txt' și 'cat /a/x/g.doc' vor afișa același fișier.

```
#include <unistd.h>
int unlink(const char *pathname);
```

Șterge o legătură fizică (nume) din sistemul de fișiere (elimină o intrare într-un director); decrementează contorul cu numărul legăturilor fizice din i-nodul fișierului referit; dacă a fost ultimul nume al fișierului (contorul a ajuns la 0) și nici un proces nu are fișierul deschis (i-nodul său nu este în TIA), fișierul este șters (spațiul folosit de el este făcut disponibil pentru refolosire).

Dacă numele se referă la o legătură simbolică, este ștearsă aceasta.

Funcția returnează 0 în caz de succes sau -1 (și setează 'errno') în caz de eșec.

```
#include <stdio.h>
int rename(const char *oldpath, const char *newpath);
```

Redenumește un fișier, înlocuind o legătură fizică cu alta (elimină și adaugă intrări în directoare); 'oldpath' și 'newpath' trebuie să fie în același sistem de fișiere (într-un director pot figura cu nume doar fișiere din același sistem de fișiere / partiție cu directorul).

Dacă 'newpath' există, este înlocuită automat - intrarea respectivă de director va asocia același nume cu i-nodul fișierului referit de 'oldpath'; în i-nodul vechiului fișier cu care era asociat numele se decrementează contorul cu numărul legăturilor fizice, iar dacă a ajuns la 0 și nici un proces nu are fișierul deschis, acesta este șters.

Dacă 'oldpath' și 'newpath' referă același fișier (i-nod), funcția nu modifică nimic și returnează succes.

'oldpath' poate referi un director și atunci 'newpath' trebuie fie să nu existe fie să refere un director gol.

Dacă 'oldpath' referă o legătură simbolică, aceasta este redenumită.

Dacă 'newpath' referă o legătură simbolică, aceasta va fi suprascrisă.

Funcția returnează 0 în caz de succes sau -1 (și setează 'errno') în caz de eșec.

```
#include <unistd.h>
int symlink(const char *target, const char *linkpath);
```

Crează o legătură simbolică cu numele (legătura fizică) 'linkpath' care conține stringul 'target' (de obicei, stringul este specificatorul unui fișier).

Legăturile simbolice sunt interpretate la run time atunci când sunt întâlnite la parcurgerea unei căi, ca și când stringul conținut ar fi substituit în locul respectiv în calea urmărită.

Legăturile simbolice pot conține căi relative iar acestea se vor raporta la directorul în care figurează legătura (de exemplu, '..' la începutul căii va desemna directorul părinte al directorului în care figurează legătura).

De exemplu, dacă '/a/b/c/d' specifică o legătură simbolică ce conține stringul '/../c/../e', atunci 'cat /a/b/c/d/f' va căuta să afișeze fișierul '/a/b/c/../c/../e/f'.

O legătură simbolică (se mai numește și soft link) poate indica un fișier existent sau inexistent, caz în care ea se mai numește 'dangling link'.

Drepturile (permissions) unei legături simbolice sunt irelevante; proprietatea este ignorată la parcurgerea căilor, dar este verificată la stergerea sau redenumirea legăturii, dacă legătura se află într-un director având setat sticky bit (S_ISVTX) (vom vedea).

Dacă 'linkpath' există, nu va fi suprascris.

Funcția returnează 0 în caz de succes sau -1 (și setează 'errno') în caz de eșec.

Observație:

În comenziile uzuale, folosirea legăturilor simbolice crează aceeași percepție pentru utilizator ca și folosirea legăturilor fizice (numesc fișiere), doar că legăturile simbolice sunt fișiere de sine stătătoare, diferențiate automat și transparent.

Avantajul față de legăturile fizice este că pot referi fișiere din orice sistem de fișiere / partiție și astfel se poate crea iluzia că un fișier are un nume într-un director de pe altă partiție ca el (de fapt, în director este o legătură fizică către o legătură simbolică aflată pe aceeași partiție cu directorul și care conține o referință (specificare cale/nume) pentru fișierul în cauză).

```
#include <unistd.h>
ssize_t readlink(const char *pathname, char *buf, size_t bufsiz);
```

Furnizează conținutul legăturii simbolice în zona pointată de 'buf', fără a depăși dimensiunea 'bufsiz'; nu adaugă un octet nul la sfârșit.

Dacă zona este prea mică, trunchiază (tacit) conținutul furnizat la 'bufsiz' caractere.

În caz de succes, returnează numărul de octeți scriși în zona pointată de 'buf' (dacă valoarea returnată este egală cu 'bufsiz', este posibil să fi avut loc trunchierea); în caz de eroare, returnează -1 și setează 'errno'.

Putem afla valoarea 'bufsiz' necesară recuperării conținutului legăturii simbolice consultând dimensiunea acesteia cu 'lstat()' (a se vedea mai jos).

```
#include <stdlib.h>
int mkstemp(char *template);
```

Generează un nume de fișier temporar unic folosind şablonul 'template', crează fișierul cu drepturile 0600 (i.e. citire și scriere pentru proprietar), îl deschide pentru citire și scriere și returnează descriptorul deschis către el; în caz de eroare, returnează -1 și setează 'errno'.

Fișierul este deschis cu flagul `O_EXCL` al lui 'open()', ceea ce garantează că apelantul este procesul care crează fișierul.

Ultimle 6 caractere ale şablonului trebuie să fie "XXXXXX" iar acestea vor fi înlocuite cu un string care face numele fișierului unic. Deoarece va fi modificat, şablonul trebuie să nu fie un string constant, dar trebuie declarat ca array de char.

Şablonul poate începe cu o cale, dar toate directoarele menționate trebuie să existe.

```
#include <stdio.h>
FILE *tmpfile(void);
```

Deschide un fișier temporar unic în citire și scriere binară ('fopen()' cu 'w+b'). Fișierul va fi automat șters (deleted) atunci când este închis sau programul se termină.

Funcția returnează adresa structurii 'FILE' alocată (stream descriptor), în caz de succes, sau NULL (și setează 'errno') dacă nu poate fi generat un nume unic de fișier sau fișierul unic nu poate fi deschis.

Obs: POSIX.1-2001 menționează că ar putea fi scris pe stdout un mesaj de eroare, dacă stream-ul nu poate fi deschis. Standardul nu specifică directorul pe care îl va folosi 'tmpfile()'; glibc va încerca prefixul de cale 'P_tmpdir' definit în '<stdio.h>', iar dacă aceasta eșuează, directorul '/tmp'.

```
#include <unistd.h>
int isatty(int fd);
```

Testează dacă 'fd' este un descriptor de fișier deschis care referă un terminal.
În caz afirmativ (este terminal), returnează 1, altfel returnează 0 și setează 'errno'.

```
#include <unistd.h>
char *ttyname(int fd);
```

Furnizează specificatorul terminalului deschis la descriptorul de fișier 'fd' (string terminat cu caracterul nul '\0') într-un string intern, care poate fi suprascris de apelurile ulterioare.

În caz de succes, returnează adresa zonei interne, în caz de eșec (de exemplu, 'fd' nu este conectat la un terminal), returnează NULL și setează 'errno'.

Funcții pentru aflarea caracteristicilor (status) unui fișier:

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

int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

furnizează informații despre un fișier (în general, aflate în i-nodul fișierului); fișierul este indicat de un specificator, în cazul funcțiilor 'stat()', 'lstat()', și de un descriptor, în cazul funcției 'fstat()';

spre deosebire de 'stat()' și 'fstat()', funcția 'lstat()' nu deferențiază legăturile simbolice (dacă 'path' este o legătură simbolică, furnizează informații despre aceasta, nu despre fișierul la care se referă ea) (vom discuta despre legături simbolice mai târziu);

funcțiile nu necesită drepturi asupra fișierului însuși dar 'stat()' și 'lstat()' necesită dreptul de execuție (search) asupra tuturor directoarelor din calea 'path' care conduce la fișier;

funcțiile furnizează informații despre fișier în structura indicată de 'buf' și returnează 0 în caz de succes sau -1 (și setează 'errno') în caz de eșec.

Exemplu de utilizare:

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

#include<stdio.h>

struct stat s;
if(stat("/a/b/nae.txt", &s)==-1){
    perror("nae.txt");
    return; /* iesire din functia curenta */
}

/* Aflam informatii despre fisier
consultand campurile structurii 's' */
```

Tipul structură 'struct stat' este definit în fișierul <sys/stat.h> astfel:

```
struct stat {  
    dev_t      st_dev;      /* ID-ul dispozitivului ce contine fisierul */  
    ino_t      st_ino;      /* numarul de inod */  
    mode_t     st_mode;     /* protectia */  
    nlink_t    st_nlink;    /* numarul de legaturi fizice (hard links) */  
    uid_t      st_uid;      /* UID al utilizatorului proprietar */  
    gid_t      st_gid;      /* GID al grupului proprietar */  
    dev_t      st_rdev;     /* ID-ul despozitivului,  
                           daca este un fisier special,  
                           care reprezinta un dispozitiv */  
    off_t      st_size;     /* dimensiunea totala a fisierului, in bytes */  
    blksize_t  st_blksize;  /* dimensiunea "preferata" a blocului pentru  
                           I/O eficiente ale sistemului de fisiere  
                           (scrierea intr-un fisier in bucati mai  
                           mici poate cauza read-modify-rewrite  
                           ineficient */  
    blkcnt_t   st_blocks;   /* numarul de blocuri de 512B alocate  
                           fisierului (poate fi mai mic decat  
                           'st_size'/512 atunci cand fisierul  
                           are goluri) */  
    time_t     st_atime;    /* momentul (time) ultimului acces */  
    time_t     st_mtime;    /* momentul (time) ultimei modificari */  
    time_t     st_ctime;    /* momentul (time) ultimeei schimbari a  
                           caracteristicilor (last status change) */  
};
```

Observații:

1. ID-ul dispozitivului descris de câmpul 'st_dev' poate fi descompus ulterior cu macro-urile 'major()', și 'minor()'.
2. Dimensiunea unei legături simbolice (câmpul 'st_size') este lungimea specificatorului de fișier (pathname) pe care îl conține, fără un caracter nul la sfârșit.
3. Diverse sisteme de fișiere pot implementa altfel câmpurile 'time'.

Câmpul 'st_mode' este un întreg de 16 biți, care conțin:

- tipul fișierului (4 biți);
- SUID (set UID bit), SGID (set GID bit), sticky bit (câte 1 bit);
- drepturile de acces (permissions) în citire, scriere, execuție, pentru proprietar (utilizatorul proprietar al fișierului), grup (utilizatorii din grupul proprietar al fișierului), alții (alți utilizatori) (câte 1 bit):

```
--- \
|b15| \
--- |
|b14| |
--- | tipul fisierului
|b13| |
--- |
|b12| /
--- /
|b11|---- set UID
--- |
|b10|---- set GID
--- |
| b9|---- sticky bit
--- |
| b8|---- (R) citire pentru proprietar (USR)
--- |
```

```
--- |
| b7|---- (W) scriere pentru proprietar (USR)
--- |
| b6|---- (X) executie pentru proprietar (USR)
--- |
| b5|---- (R) citire pentru grup (GRP)
--- |
| b4|---- (W) scriere pentru grup (GRP)
--- |
| b3|---- (X) executie pentru grup (GRP)
--- |
| b2|---- (R) citire pentru altii (OTH)
--- |
| b1|---- (W) scriere pentru altii (OTH)
--- |
| b0|---- (X) executie pentru altii (OTH)
--- |
```

Pentru a selecta anumite componente, se pot folosi conjuncții pe biți '&' între 'st_mode' și următoarele flaguri (constante simbolice, definite în <sys/stat.h>):

S_IFMT	0170000	masca pentru campul de biti cu tipul fisierului
S_IFSOCK	0140000	socket
S_IFLNK	0120000	symbolic link
S_IFREG	0100000	fisier obisnuit (regular file)
S_IFBLK	0060000	dispozitiv block (block device)
S_IFDIR	0040000	director (directory)
S_IFCHR	0020000	dispozitiv caracter (character device)
S_IFIFO	0010000	fisier tub (FIFO)
S_ISUID	0004000	set UID bit
S_ISGID	0002000	set-group-ID bit
S_ISVTX	0001000	sticky bit
S_IRWXU	00700	masca pentru drepturile proprietarului
S_IRUSR	00400	dreptul de citire pentru proprietar (owner)
S_IWUSR	00200	dreptul de scriere pentru proprietar (owner)
S_IXUSR	00100	dreptul de executie pentru proprietar (owner)
S_IRWXG	00070	masca pentru drepturile grupului
S_IRGRP	00040	dreptul de citire pentru grup (group)
S_IWGRP	00020	dreptul de scriere pentru grup (group)
S_IXGRP	00010	dreptul de executie pentru grup (group)
S_IRWXO	00007	masca pentru drepturile altora
S_IROTH	00004	dreptul de citire pentru altii (others)
S_IWOTH	00002	dreptul de scriere pentru altii (others)
S_IXOTH	00001	dreptul de executie pentru altii (others)

Exemplu de utilizare:

```
struct stat s, s1;
stat("f.txt", &s); stat("g.pas", &s1);

if(s.st_mode & S_IRUSR !=0)
    printf("Proprietarul lui f.txt are drept de citire asupra lui\n");

if(s.st_mode & S_IRWXU == s1.st_mode & S_IRWXU)
    printf("f.txt si g.pas ofera aceleasi drepturi proprietarilor lor\n");

if(s.st_mode & S_IFMT == s1.st_mode & S_IFMT)
    printf("f.txt si g.pas sunt de acelasi tip\n");

if(s.st_mode & S_IFMT == S_IFDIR)
    printf("f.txt este director\n");
```

Pentru testarea tipului fișierului se pot aplica câmpului 'st_mode' următoarele macro-uri, definite în <sys/stat.h>:

```
S_ISREG(.) => este fisier obisnuit (regular file) ?
S_ISDIR(.) => este director (directory) ?
S_ISBLK(.) => este fisier special bloc (block device) ?
S_ISCHR(.) => este fisier special caracter (character device) ?
S_ISFIFO(.) => este tub (FIFO, named pipe) ?
S_ISSOCK(.) => este socket ?
S_ISLNK(.) => este legatura simbolica (symbolic link) ?
```

Exemplu de utilizare:

```
struct stat s;
stat("f.txt", &s);

if(S_ISDIR(s.st_mode))
    printf("f.txt este director\n");
```

Observație: POSIX nu descrie biții S_IFMT, S_IFSOCK, S_IFLNK, S_IFREG, S_IFBLK, S_IFDIR, S_IFCHR, S_IFIFO, S_ISVTX, dar cere folosirea macrourilor S_ISDIR(), etc.

Tipul unui fișier este stabilit implicit la crearea lui. De exemplu:

- apelul 'creat()' creaază fișiere regular;
- apelul 'mkdir()' creaază directoare;
- apelul 'mknod()' creaază (în funcție de un mod dat ca argument) diverse tipuri de fișier: regular, special caracter, special bloc, tub, socket;
- apelurile 'pipe()', 'mkfifo()' creaază tuburi fără, resp. cu, nume;
- apelul 'socket()' creaază socketuri;
- apelul 'symlink()' creaază legături simbolice.

Există comenzi shell cu care se pot crea diverse tipuri de fișier, folosind apelurile de mai sus: 'mkdir', 'mknod', 'mkfifo', etc.

Proprietarul și grupul proprietar ai unui fișier:

- se stabilesc implicit la crearea fișierului;
- pot fi modificate ulterior cu apelul 'chown()' (sau comanda shell 'chown').

În general, la crearea unui fișier (cu apelurile 'creat()', 'mkdir()', etc.), proprietarul și grupul proprietar ai fișierului sunt setate să fie proprietarul efectiv, resp. grupul efectiv, ai procesului creator. Există excepții legate de biții SUID, SGID ai directorului părinte al fișierului creat:

- dacă directorul părinte are SUID = 1, doar în anumite sisteme (BSD), proprietarul fișierului este setat să fie proprietarul directorului părinte; în alte sisteme, este ignorat;
- dacă directorul părinte are SGID = 1, grupul proprietar al fișierului este setat să fie grupul proprietar al directorului părinte; dacă fișierul creat este tot un director, va avea și el SGID = 1.

```
#include <unistd.h>
int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
```

Modifică proprietarul și grupul unui fișier; funcțiile diferă doar prin modul în care este specificat fișierul:

- la 'chown()', fișierul este specificat prin cale și nume, date în 'path', care este dereferențiat, dacă este o legătură simbolică;
- la 'fchown()', este un fișier deschis și referit prin descriptorul 'fd'.
- la 'lchown()' este ca la 'chown()', dar nu dereferențiază legăturile simbolice.

Pentru succes, este necesar să fie indeplinită măcar una din condițiile:

- proprietarul procesului coincide cu cel al fișierului; în acest caz, se poate schimba doar grupul și doar cu un alt grup din care face parte utilizatorul respectiv;
- procesul este unul privilegiat; în acest caz, se poate schimba atât proprietarul cât și grupul, în mod arbitrar (noul proprietar nu trebuie neapărat să facă parte din noul grup).

Observație: În anumite sisteme (4.4BSD version) apelul poate fi făcut doar de superuser (i.e., utilizatorii obișnuiți nu pot dărui fișiere).

Dacă 'owner' sau 'group' sunt specificate ca -1, atunci acel ID nu este modificat.

Când proprietarul sau grupul unui fișier executabil sunt modificate de un proces neprivilegiat, bitii set UID (S_ISUID) și set GID (S_ISGID) ai fișierului sunt setați la 0. POSIX nu specifică dacă aceasta trebuie să se întâpte și dacă proprietarul procesului este 'root', iar în Linux, comportamentul depinde de versiunea de kernel folosită.

În cazul unui fișier non-group-executable, i.e., nu are setat dreptul de execuție pentru grup (S_IXGRP), bitul S_ISGID indică încuierea obligatorie (mandatory locking), și nu este setat la 0 de un apel 'chown()'.

În caz de succes, funcțiile returnează 0. În caz de eșec, ele returnează -1 și setează 'errno'.

Exemplu (preluat din 'man 2 chown'): Programul următor modifică proprietarul fișierului dat ca al doilea argument în linia de comanda, la valoarea data în primul argument; noul proprietar poate fi specificat fie ca UID, fie ca username (care este convertit într-un UID folosind 'getpwnam()'):

```
#include <pwd.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
    uid_t uid; struct passwd *pwd; char *endptr;
    if (argc != 3 || argv[1][0] == '\0') {
        fprintf(stderr, "%s <owner> <file>\n", argv[0]); exit(EXIT_FAILURE);
    }
    uid = strtol(argv[1], &endptr, 10); /* Allow a numeric string */
    if (*endptr != '\0') { /* Was not pure numeric string */
        pwd = getpwnam(argv[1]); /* Try getting UID for username */
        if (pwd == NULL) { perror("getpwnam"); exit(EXIT_FAILURE); }
        uid = pwd->pw_uid;
    }
    if (chown(argv[2], uid, -1) == -1) {
        perror("chown"); exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

Drepturile (permission) asupra unui fișier:

- se stabilesc implicit la crearea fișierului; în general (există anumite excepții):
'open(pathname, flags, mode)', 'creat(pathname, mode)' crează fișierul
'pathname' cu drepturile 'mode & ~umask';
'mkdir(pathname, mode)' crează directorul 'pathname' cu drepturile
'mode & ~umask & 0777';
unde 'umask' este masca de drepturi a procesului apelant (file mode
creation mask, atribut al proceselor, se moșteneste la procesele copil, se
poate consulta/seta cu apelul 'umask()', a se vedea mai jos).
- pot fi modificate ulterior cu apelul 'chmod()' (sau comanda shell 'chmod').

```
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
int fchmod(int fd, mode_t mode);
```

Modifică drepturile (permissions) unui fișier; funcțiile diferă doar prin modul în care este specificat fișierul:

la 'chmod()', fișierul este specificat prin cale și nume (pathname), date în

'path', care este dereferențiat, dacă este o legătură simbolică;

la 'fchmod()', este un fișier deschis și referit prin descriptorul 'fd'.

Noile drepturi (file permissions) sunt specificate în 'mode', care este o mască pe biți creată prin disjuncția pe biți '|' a ≥ 0 dintre valorile următoare, folosite și pentru selectarea biților campului 'st_mode' al unei structuri 'stat' (a se vedea mai sus):

```
S_ISUID (04000), S_ISGID (02000), S_ISVTX (01000),
S_IRUSR (00400), S_IWUSR (00200), S_IXUSR (00100),
S_IRGRP (00040), S_IWGRP (00020), S_IXGRP (00010),
S_IROTH (00004), S_IWOTH (00002), S_IXOTH (00001).
```

Pentru succes, este necesar ca proprietarul efectiv (effective UID) al procesului apelant să coincidă cu proprietarul fișierului, sau procesul să fie unul privilegiat. Dacă procesul apelant nu este privilegiat și grupul fișierului nu coincide cu grupul efectiv al procesului sau cu unul din grupurile sale suplimentare, bitul S_ISGID va fi setat la 0, dar nu se va returna eroare.

Observații:

- Ca masură de securitate, în funcție de sistemul de fișiere, biții set UID și set GID pot fi setați la 0 dacă se scrie în fișier.
- În unele sisteme de fișiere, doar superuserul poate seta sticky bit.

În caz de succes, funcțiile returnează 0. În caz de esec, ele returnează -1 și setează 'errno'.

Drepturile de acces (citire, scriere, execuție) sunt folosite în felul următor:

- când un proces încearcă să acceseze fișierul (folosind un apel sistem), sistemul compară proprietarul efectiv și grupul efectiv ai procesului cu proprietarul și grupul fișierului, determinand categoria în care se situează procesul din punctul de vedere al fișierului - la proprietar, grup, alții;
- apoi, sistemul consultă bitul corespunzător operației încercate (citire, scriere, execuție) din categoria determinată (proprietar, grup, alții);
- dacă bitul are valoarea 1, accesul (apelul sistem) reușește; dacă are valoarea 0, eșuează.

Dreptul de citire (r) este necesar:

- în cazul fișierelor obișnuite (regular), pentru a fi deschise în citire (exemple: funcția 'open()', cu modurile de acces 'O_RDONLY' sau 'O_RDWR', comanda shell 'cat', editorul de text 'vi');
- în cazul directoarelor (directory), pentru a fi deschise în scopul consultării listei intrărilor (exemplu: funcția 'opendir()', comanda shell 'ls').

Dreptul de scriere (w) este necesar:

- în cazul fișierelor obișnuite (regular), pentru a fi deschise în scriere (exemplu: funcția 'open()', cu modurile de acces 'O_WRONLY' sau 'O_RDWR', editorul de text 'vi', dacă se dorește modificarea fișierului);
- în cazul directoarelor (directory), pentru a adăuga/modifica/elimina o intrare (exemplu: funcțiile 'link()', 'unlink()', 'rename()', 'mkdir()', 'rmdir()', comenzi shell 'cp', 'mv', 'ln', 'rm', 'mkdir', 'rmdir').

Dreptul de execuție (x) este necesar:

- în cazul fișierelor obișnuite (regular), pentru a fi lansate în execuție ca proces ce înlocuiește procesul curent (exemplu: funcțiile din familia 'exec()': 'execl()', 'execvp()', 'execle()', 'execv()', 'execvp()', shell-ul atunci când execută comenzi externe - le lansează ca procese copil cu 'fork()' și 'exec()');
- în cazul directoarelor (directory), pentru a accesa intrările sale; acesta include cazul când directorul este traversat, atunci când se parurge o cale, și cazul când directorul este setat ca director curent; în cazul directoarelor, dreptul de execuție (execute permission) se mai numește și drept de căutare (search permission).

Exemple:

```
open("/a/b/c/f.txt", O_RDONLY);
```

procesul curent deschide fișierul '/a/b/c/f.txt' în citire;
el necesită drept de executie (x) pe directoarele rădăcină, 'a', 'b', 'c'
și drept de citire (r) pe fișierul 'f.txt';

```
open("/a/b/c/f.txt", O_WRONLY);
```

procesul curent deschide fișierul '/a/b/c/f.txt' în scriere;
el necesită drept de execuție (x) pe directoarele rădăcină, 'a', 'b', 'c'
și drept de scriere (w) pe fișierul 'f.txt';

```
open("/a/b/c/f.txt", O_RDWR);
```

procesul curent deschide fișierul '/a/b/c/f.txt' în citire și scriere;
el necesită drept de execuție (x) pe directoarele rădăcină, 'a', 'b', 'c'
și drepturi de citire (r) și scriere (w) pe fișierul 'f.txt';

```
open("/a/b/c/f.txt", O_WRONLY | O_CREAT, S_IRUSR);  
procesul curent deschide fișierul '/a/b/c/f.txt' în scriere;  
dacă fișierul nu există, el este creat (O_CREAT), cu drept de citire  
pentru proprietar (S_IRUSR) (se ține cont și de masca de drepturi a  
procesului curent);  
dacă fișierul 'f.txt' există, procesul necesită drept de execuție (x) pe  
directoarele rădăcină, 'a', 'b', 'c' și drept de scriere (w) pe fișierul  
'f.txt';  
dacă fișierul 'f.txt' nu există, procesul necesită drept de executie (x)  
pe directoarele rădăcină, 'a', 'b', și drepturi de execuție (x) și  
scriere (w) pe directorul 'c' (dreptul de scriere este necesar, deoarece  
trebuie adăugată intrarea 'f.txt'); observăm că nu este necesară  
specificarea dreptului de scriere pe fișierul nou creat 'f.txt';
```

```
char *a[] = {"/a/b/c/f.exe", "xyz", "12", NULL};  
execv("/a/b/c/f.exe", a);  
se lansează în execuție fișierul 'f.exe' ca proces ce înlocuiește procesul  
curent; șirurile desemnate de 'a[0]', 'a[1]', ... vor deveni argumentele  
lui 'main()' 'argv[0]', 'argv[1]', ... pentru noul proces;  
procesul apelant necesită drept de execuție (x) pe directoarele rădăcină,  
'a', 'b', 'c' și pe fișierul 'f.exe';
```

```
opendir("/a/b/c");
procesul curent deschide directorul '/a/b/c', pentru a consulta lista
intrărilor sale;
el necesită drept de execuție (x) pe directoarele rădăcină, 'a', 'b' și
și drept de citire (r) pe directorul 'c';

link ("/a/b/c/f.txt", "/a/b/d/g.doc");
procesul curent crează o nouă legătură fizică '/a/b/d/g.doc' (practic, un
nou nume 'g.doc', ca intrare în directorul '/a/b/d') pentru fișierul cu
vechea legătura fizică '/a/b/c/f.txt'; în caz de succes, în continuare,
specificările "/a/b/c/f.txt" și "/a/b/d/g.doc" vor desemna același fișier;
procesul curent necesită drept de execuție (x) pe directoarele rădăcină,
'a', 'b', 'c', 'd' și, în plus, drept de scriere pe directorul 'd';

mkdir("/a/b/e", S_IRWXU);
procesul curent crează directorul 'e', ca subdirector (intrare) în
directorul '/a/b'; în caz de succes, directorul 'c' va avea drepturi
de citire, scriere, execuție pentru proprietar (S_IRWXU) (se ține cont și
de masca de drepturi a procesului curent);
procesul curent necesită drept de execuție (x) pe directoarele rădăcină,
'a', 'b' și, în plus, drept de scriere pe directorul 'b';
```

```
chdir("/a/b");
```

procesul curent își schimbă directorul curent în '/a/b';

el necesită drept de execuție (x) pe directoarele rădăcină, 'a', 'b'.

Drepturile necesitate de diversele comenzi shell derivă din drepturile necesitate de apelurile sistem pe care le fac. În funcție de implementarea lor, ele mai pot necesita și drepturi suplimentare.

Exemple:

`cat /a/b/c/f.txt`

se afișază pe standard output conținutul fișierului '/a/b/c/f.txt';
sunt necesare drepturile de execuție (x) pe directoarele rădăcină, 'a',
'b', 'c' și de citire (r) pe fișierul 'f.txt';

`vi /a/b/c/f.txt`

se deschide în editarea cu editorul de texte 'vi' fișierul '/a/b/c/f.txt';
sunt necesare drepturile de executie (x) pe directoarele rădăcină, 'a',
'b', 'c' și de citire (r) pe fișierul 'f.txt';
dacă se dorește posibilitatea de a salva modificările în fișier, este
necesar și dreptul de scriere (w) pe fișierul 'f.txt' (altfel, fișierul
este deschis read only);

`ls /a/b/c`

se afișază pe standard output lista intrărilor directorului '/a/b/c'; se
afișază doar numele intrărilor, fără alte detalii;
sunt necesare drepturile de execuție (x) pe directoarele rădăcină, 'a',
'b' și de citire (r) pe directorul 'c';

```
ls /a/b/c/f.txt
```

se afișază pe standard output intrarea 'f.txt' din directorul '/a/b/c',
dacă există și nu este un director (altfel, este ca la 'ls /a/b/c');

se afișază doar numele 'f.txt', fără alte detalii;

sunt necesare drepturile de executie (x) pe directoarele rădăcină, 'a',
'b', 'c' (nu mai este necesar dreptul de citire (r) pe directorul 'c');

```
ls -l /a/b/c
```

se afișază pe standard output lista intrărilor directorului '/a/b/c', cu
detalii (tipul fișierului, drepturile de acces, proprietarul, etc.);

sunt necesare drepturile de executie (x) pe directoarele rădăcină, 'a',
'b', 'c' și, în plus, dreptul de citire (r) pe directorul 'c';

deci, față de comanda 'ls /a/b/c', trebuie, în plus, drept de executie (x)
pe 'c', altfel el nu poate fi parcurs pentru a se ajunge la fișierele
care figurează în el ca intrări, pentru a se afla detalii despre ele
(ca în cazul funcției 'stat()');

```
ls -l /a/b/c/f.txt
```

se afișază pe standard output intrarea 't.txt' din directorul '/a/b/c', cu detalii (tipul fișierului, drepturile de acces, proprietarul, etc.), dacă există și nu este un director (altfel, este ca la 'ls -l /a/b/c'); sunt necesare drepturile de execuție (x) pe directoarele rădăcină, 'a', 'b', 'c'; deci, față de comanda 'ls -l /a/b/c', nu mai este necesar dreptul de citire (r) pe directorul 'c'

```
cp /a/b/c/f.txt /a/b/d/g.doc
```

se copiază fișierul '/a/b/c/f.txt' în fișierul '/a/b/d/g.doc'; sunt necesare drepturile de execuție (x) pe directoarele rădăcină, 'a', 'b', 'c', 'd'; fișierul 'f.txt' trebuie să existe și este necesar dreptul de citire (r) pe el; dacă fișierul 'g.doc' există, este necesar dreptul de scriere (w) pe el (conținutul lui va fi suprascris); dacă fișierul 'g.doc' nu există, este necesar, în plus, dreptul de scriere (w) pe directorul 'd' (deoarece se va adauga intrarea 'g.doc' la el);

```
mv /a/b/c/f.txt /a/b/d/g.doc
```

se înlocuiește legătura fizică '/a/b/c/f.txt' cu '/a/b/d/g.doc', pentru același fișier (presupunând că directorul destinație este pe același disc); practic, se elimină intrarea 'f.txt' din directorul '/a/b/c' și se adaugă intrarea 'g.doc' în directorul '/a/b/d', asociată la același fișier (i-nod); în continuare, fișierul nu va mai putea fi afișat cu 'cat /a/b/c/f.txt', dar va putea fi afișat cu 'cat /a/b/d/g.doc' (dacă există dreptul de citire (r) pe el);

sunt necesare drepturile de execuție (x) pe directoarele rădăcină, 'a', 'b', 'c', 'd' și, în plus, drepturile de scriere (w) pe directoarele 'c' și 'd' (deoarece din primul se elimină o intrare, iar în al doilea se adaugă o intrare);

observăm că nu sunt necesare drepturi pe fișierul 'f.txt';

```
ln /a/b/c/f.txt /a/b/d/g.doc
```

se crează o nouă legătură fizică '/a/b/d/g.doc' (practic, un nou nume 'g.doc', ca intrare în directorul '/a/b/d') pentru fișierul (i-nodul) cu vechea legătură fizică '/a/b/c/f.txt'; în continuare, fișierul va putea fi afișat atât cu 'cat /a/b/c/f.txt', cât și cu 'cat /a/b/d/g.doc' (dacă există dreptul de citire (r) pe el);
sunt necesare drepturile de execuție (x) pe directoarele rădăcină, 'a', 'b', 'c', 'd' și, în plus, drept de scriere (w) pe directorul 'd';
observăm că nu sunt necesare drepturi pe fișierul 'f.txt';

```
rm /a/b/c/f.txt
```

se elimină legătura fizică '/a/b/c/f.txt' (practic, se elimină intrarea 'f.txt' din directorul '/a/b/c'); dacă fișierul nu mai are alte legături fizice și nici nu este deschis de vreun proces, este eliminat și el (altfel, fișierul rămâne în sistem); în continuare, fișierul nu va mai putea fi afișat cu 'cat /a/b/c/f.txt';
sunt necesare drepturile de executie (x) pe directoarele rădăcină, 'a', 'b', 'c' și, în plus, drept de scriere (w) pe directorul 'c';

observăm că nu sunt necesare drepturi pe fișierul 'f.txt'; astfel, se poate întâmpla să nu putem citi sau scrie într-un fișier (pentru că nu avem drepturile (r) sau (w) pe el), dar să-l putem elimina din sistem (dacă avem drepturile (w) pe directorul său parinte și (x) pe calea până la el);

```
mkdir /a/b/e
```

```
rmdir /a/b/e
```

se crează/elimină directorul 'e', ca subdirector (intrare) în directorul '/a/b'; eliminarea reușește doar dacă subdirectorul 'e' este gol (i.e. are doar intrările '.' și '..');

sunt necesare drepturile de execuție (x) pe directoarele rădăcină, 'a', 'b' și, încă, drept de scriere pe directorul 'b';

```
cd /a/b
```

procesul shell curent își schimbă directorul curent în '/a/b'; el necesită drept de execuție (x) pe directoarele rădăcină, 'a', 'b'.

Notăm că, în general, funcțiile care deschid fișiere pentru operații ('open()', 'fopen()', 'opendir()', etc.) verifică drepturile, nu și funcțiile care efectuează operațiile ('read()', 'write()', 'fscanf()', 'fprintf()', 'readdir()', etc.).

Dacă în timp ce sunt efectuate operațiile de citire/scriere un proces paralel modifică drepturile ('chmod()') sau elimină legătura fizică ('unlink()') a fișierului, acesta încă există (deoarece este deschis de procesul curent) iar operațiile pentru a care a fost deschis încă funcționează.

De aceea, este recomandabil între o deschidere și o închidere a fișierului să se efectueze cât mai multe operații, nu să se deschidă / închidă deși probabilitatea ca toate operațiile să reușească este mai mare.

În aceeași idee, este preferabil să manevrăm fișierele pe cât posibil prin descriptori și (pointeri la) structuri 'FILE', 'DIR', nu prin specicatori cale/nume.

Pe de altă parte, nu este recomandabil să încercăm să deschidem prea multe fișiere simultan, deoarece tabela de descriptori a procesului este limitată și s-ar putea să nu avem suficienți descriptori liberi.

Bitul set UID (`S_ISUID`) are următoarea semnificație:

- În cazul unui fișier obișnuit (regular), bitul set UID influențează setarea proprietarului efectiv, atunci când fișierul este lansat în execuție (de exemplu, cu apelul '`execv()`').

Dacă bitul este 0, proprietarul efectiv al procesului rezultat va fi cel al procesului apelant, iar dacă este 1, va fi proprietarul fișierului executat.

Vom da mai multe detalii când vom discuta despre procese.

- În cazul unui director, bitul set UID este ignorat în majoritatea sistemelor; în anumite sisteme (BSD) un fișier creat într-un director cu `SUID = 1` va avea ca proprietar pe cel al directorului (nu proprietarul efectiv al procesului creator);

Bitul set GID (S_ISGID) are următoarea semnificație:

- În cazul unui fișier obișnuit (regular) care are bitul de execuție pentru grup (S_IXGRP) setat la 1, bitul set GID are efect asemănator cu bitul set UID, dar afectează grupul efectiv al procesului rezultat.

În cazul unui fișier obișnuit (regular) care are bitul de execuție pentru grup (S_IXGRP) setat la 0, bitul set GID indică încuierea obligatorie la înregistrare (mandatory file/record locking).

- În cazul unui director, bitul set GID influențeaza setarea grupurilor (GID) a noilor fișiere și subdirectoare create în subarborele cu originea în el; nu influențează setarea proprietarilor (UID); nu se modifică grupul fișierelor și subdirectoarelor deja existente (el se poate schimba manual, de exemplu, cu comanda 'chmod g+s').

Dacă bitul set GID este 0, noile fișiere și subdirectoare create în directorul respectiv vor avea ca grup grupul efectiv al procesului care le-a creat, iar subdirecoarele vor avea bitul set GID tot 0.

Dacă bitul set GID este 1, noile fișiere și subdirectoare create în directorul respectiv vor avea ca grup grupul acestui director, iar subdirecoarele vor avea bitul set GID tot 1.

Astfel, se poate institui un spațiu de lucru partajat pentru un grup, fără inconvenientul de a cere membrilor grupului să-și schimbe explicit grupul curent înainte de a crea noi fișiere și directoare în el.

Bitul sticky bit (`S_ISVTX`) are următoarea semnificație:

- În cazul unui fișier obișnuit (regular), sticky bit este ignorat (în Linux) sau poate avea diverse semnificații în diverse sisteme Unix sau Unix-like; de exemplu, în sistemul HP-UX, se păstrează imaginea codului programului în zona de swap atunci când ultimul proces care o utilizează se termină (procese diferite care execută același fișier folosesc în comun paginile de cod); astfel, când fișierul este lansat din nou mai tarziu, codul nu mai trebuie recitat din el ci poate fi direct swapped in, economisindu-se astfel timp; de aceea, sticky bit este setat la 1 pentru programele utilitare folosite frecvent (pentru a se începe execuția lor mai repede).
- În cazul unui director, sticky bit = 1 înseamnă eliminare restricționată (restricted deletion): un fișier ce figurează în el (intrarea respectivă) poate fi redenumit sau eliminat doar de proprietarul fișierului, proprietarul directorului, sau un proces privilegiat; altfel, redenumirea sau eliminarea poate fi făcută doar de cei care au drept de execuție (x) și scriere (w) pe director, a se vedea mai sus.

De exemplu, setarea sticky bit = 1 este prezentă asupra directorului '/tmp'; acest director este folosit pentru crearea de fișiere temporare de către procesele obținute de diversi utilizatori în urma lănsării unor programe instalate în sistem; de aceea, el trebuie să ofere drepturi de execuție (x) și scriere (w) pentru toți; se dorește însă ca aceste fișiere să poată fi eliminate în final doar de utilizatorul/procesul care le-a creat, nu de către oricine, în mod discreționar (de exemplu, cu comanda 'rm'); eliminarea fișierelor temporare create de procesul unui utilizator de către un alt utilizator ar putea împiedica funcționarea normală a procesului respectiv; de aceea, directorul '/tmp' are sticky bit = 1:

```
$ls -ld /tmp  
drwxrwxrwt 16 root root 4096 Nov 18 01:04 /tmp
```

(consemnarea 't' în poziția dreptului de executie 'x' pentru alții indică faptul că bitul set UID are valoarea 1).

```
#include<sys/types.h>
#include<sys/stat.h>
mode_t umask(mode_t mask);
```

Setează masca de drepturi a procesului apelant (file mode creation mask) la valoarea 'mask & 0777' (i.e. sunt afectați doar biții de drepturi) și se returnează vechea mască (apelul nu eșueaza niciodata); argumentul se poate construi ca disjuncție pe biți ('|') de aceleasi constante simbolice ca argumentul 'mode' al funcției 'open()'.

Această mască este un atribut al proceselor, se moșteneste la procesele copil, și afectează drepturile cu care procesele crează fișierele (a se vedea mai sus).

Pentru fișiere se rețin (și se pot consulta cu apelul 'stat()') momentele:

- ultimului acces (membrul 'st_atime' al structurii 'stat');
poate fi modificat de apelurile 'mknod()', 'utimes()', 'read()');
- ultimei modificări (membrul 'st_mtime' al structurii 'stat');
poate fi modificat de apelurile 'mknod()', 'utimes()', 'write()');
- ultimei schimbări a caracteristicilor (membrul 'st_ctime' al structurii 'stat');
poate fi modificat de apelurile 'chmod()', 'chown()', 'link()', 'mknod()',
'rename()', 'unlink()', 'utimes()', 'write()').

În SO mai vechi, acestea erau valori de tip 'time_t':

```
struct stat {  
    ...  
    time_t    st_atime;  
    time_t    st_mtime;  
    time_t    st_ctime;  
};
```

Tipul 'time_t' este un tip aritmetic (ISO C), de obicei întreg cu semn (POSIX), definit în '<time.h>' care conține un moment de timp împachetat, sub forma numărului de secunde care au trecut de la Unix epoch: 00:00, Jan 1 1970 UTC (Coordinated Universal Time), fără a lua în considerare ajustările ocazionale de 1 secundă (leap seconds).

În SO mai noi (Linux 2.6), sunt suportate momente cu precizie de nanosecunde. Structura 'stat' conține în acest scop membrii 'st_atim', 'st_mtim', 'st_ctim' dar, pentru compatibilitate backward, sunt definiți și 'st_atime', 'st_mtime', 'st_ctime':

```
struct stat {  
    ...  
    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  
};
```

Tipul 'struct timespec' este definit în '<time.h>' astfel:

```
struct timespec {  
    time_t      tv_sec;          /* seconds */  
    long        tv_nsec;         /* nanoseconds */  
};
```

Membrul 'tv_sec' reține numărul întreg de secunde care au trecut de la epoch (calendar time) sau alt moment de start (elapsed time) iar membrul 'tv_nsec' reține numărul de nanosecunde trecute (elapsed) de la momentul specificat de membrul 'tv_sec'. Valorile lui 'tv_nsec' cu care pot opera funcțiile din biblioteca C sunt de la 0 la 1,000,000,000.

Pentru conversii între diverse formate de timp, putem folosi funcțiile:

```
#include <time.h>
struct tm *gmtime(const time_t *timep);
time_t mktime(struct tm *tm);
```

Convertesc timpul între formatul împachetat, 'time_t', și cel despachetat, 'struct tm', care este un tip structură definit în '<time.h>' care are membri separați pentru secundă, minut, oră, etc:

```
struct tm {
    int tm_sec;      /* Seconds (0-60) */
    int tm_min;      /* Minutes (0-59) */
    int tm_hour;     /* Hours (0-23) */
    int tm_mday;     /* Day of the month (1-31) */
    int tm_mon;      /* Month (0-11) */
    int tm_year;     /* Year - 1900 */
    int tm_wday;     /* Day of the week (0-6, Sunday = 0) */
    int tm_yday;     /* Day in the year (0-365, 1 Jan = 0) */
    int tm_isdst;    /* Daylight saving time */
};
```

Funcția 'gmtime()' furnizează rezultatul într-o structură internă care poate fi suprascrisă de apelurile ulterioare ale diverselor funcții referitoare la timp - utilizatorul trebuie să folosească/salveze în altă parte valorile relevante înaintea unui nou apel și nu trebuie să transmită adresa lui 'free()'.

```
#include <time.h>
struct tm *localtime(const time_t *timep);
```

Convertește momentul dat ca argument în formatul despachetat, exprimat relativ la timezone specificat al utilizatorului. Rezultatul este furnizat într-o zonă internă, care poate fi suprascrisă de alte apeluri (ca mai devreme).

```
#include <time.h>
char *ctime(const time_t *timep);
char *asctime(const struct tm *tm);
```

Convertesc momentul dat ca argument într-o descriere string (terminat cu '\0') furnizat într-o zonă internă, care poate fi suprascrisă de alte apeluri (ca mai devreme).

Apelul 'ctime(t)' este echivalent cu 'asctime(localtime(t))'.

În caz de eroare, 'mktime()' returnează (time_t) -1, celelalte funcții returnează NULL, iar toate setează 'errno'.

Pentru a modifica momentele de timp ale unui fișier, putem folosi:

```
#include <fcntl.h>
#include <sys/stat.h>
int futimens(int fd, const struct timespec times[2]);
```

Funcția actualizează momentele unui fișier, cu precizie de nanosecundă.

'fd' este un descriptor deschis către fișier, 'times[0]' este noul 'last access time' (atime), 'times[1]' este noul 'last modification time' (mtime); valorile setate vor fi cele mai mari suportate de sistemul de fișiere, care sunt \leq valorile specificate.

Dacă vreunul dintre momente are câmpul 'tv_nsec' cu valoarea 'UTIME_NOW', momentul respectiv este setat la timpul curent din sistem, iar dacă are valoarea 'UTIME_OMIT', momentul respectiv este lăsat nemodificat; în ambele cazuri, valoarea câmpului corespunzător 'tv_sec' este ignorată.

Dacă 'times' este NULL, ambele momente sunt setate la timpul curent din sistem.

Pentru a seta ambele momente la timpul curent din sistem (i.e. 'times' este NULL sau ambele câmpuri 'tv_nsec' sunt 'UTIME_NOW') trebuie îndeplinită una din condițiile:

1. Procesul apelant trebuie să aibă drept de scriere (w) pe fișier;
2. Proprietarul efectiv al procesului apelant trebuie să coincidă cu proprietarul fișierului.
3. Procesul apelant trebuie să fie privilegiat.

Pentru a face orice altă setare a momentelor decât cea de mai sus, trebuie îndeplinită una din condițiile 2 sau 3 anterioare.

Dacă ambele câmpuri 'tv_nsec' sunt 'UTIME_OMIT' (deci momentele fișierului nu sunt modificate), nu sunt cerințe privind proprietarul fișierului sau drepturile.

Funcția returnează 0 în caz de succes sau -1 (și setează 'errno') în caz de eșec.

Este utilă și funcția:

```
#include <time.h>
time_t time(time_t *tloc);
```

Returnează timpul sistemului (momentul curent din sistem), ca număr de secunde de la Epoch, 1970-01-01 00:00:00 (UTC).

Dacă 'tloc' este non-NULL, valoarea returnată este furnizată și în zona de memorie pointată de 'tloc'.

În caz de eroare, returnează ((time_t) -1) și setează 'errno'.

Exemplu: Emularea comenzi 'ls -ailU [director]' (fără spațiere):

```
#include <sys/types.h>
#include <dirent.h>
#include <sys/stat.h>
#include <pwd.h>
#include <grp.h>
#include <unistd.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
    DIR *d; struct dirent *de; struct stat s;
    struct passwd *p; struct group *g;
    char *arg, *cale, *specifier, *aux;
    if(argc > 2) {fprintf(stderr, "Utilizare: %s [director]\n", argv[0]); return 1;}
    if(argc == 1) arg = ".";
    else arg = argv[1];
    if((d = opendir(arg)) == NULL) {perror(arg); return 1;}
    if((cale = malloc((strlen(arg) + 2) * sizeof(char))) == NULL)
        {perror("malloc"); closedir(d); return 1;}
    strcpy(cale, arg);
    if(strlen(cale) > 0 && cale[strlen(cale) - 1] != '/') strcat(cale, "/");
```

```

while((de = readdir(d)) != NULL) {
    if((specifier = malloc((strlen(cale) + strlen(de->d_name) + 1) * sizeof(char))) == NULL)
        {perror("malloc"); free(cale); closedir(d); return 1;}
    strcpy(specifier, cale); strcat(specifier, de->d_name);
    if(stat(specifier, &s) == -1)
        {perror(specifier); free(specifier); free(cale); closedir(d); return 1;}
    printf("%ld ", (long)s.st_ino);
    switch (s.st_mode & S_IFMT) {
        case S_IFREG: printf("-"); break; /* regular file */
        case S_IFDIR: printf("d"); break; /* directory */
        case S_IFBLK: printf("b"); break; /* block device */
        case S_IFCHR: printf("c"); break; /* character device */
        case S_IFIFO: printf("p"); break; /* FIFO/pipe */
        case S_IFSOCK: printf("s"); break; /* socket */
        case S_IFLNK: printf("l"); break; /* symlink */
    }
    if(s.st_mode & S_IRUSR) printf("r"); else printf("-");
    if(s.st_mode & S_IWUSR) printf("w"); else printf("-");
    if(s.st_mode & S_IXUSR)
        if(s.st_mode & S_ISUID) printf("s"); else printf("x");
    else
        if(s.st_mode & S_ISUID) printf("S"); else printf("-");
    if(s.st_mode & S_IRGRP) printf("r"); else printf("-");
    if(s.st_mode & S_IWGRP) printf("w"); else printf("-");
    if(s.st_mode & S_IXGRP)
        if(s.st_mode & S_ISGID) printf("s"); else printf("x");
    else
        if(s.st_mode & S_ISGID) printf("S"); else printf("-");
}

```

```
    if(s.st_mode & S_IROTH) printf("r"); else printf("-");
    if(s.st_mode & S_IWOTH) printf("w"); else printf("-");
    if(s.st_mode & S_IXOTH)
        if(s.st_mode & S_ISVTX) printf("t"); else printf("x");
    else
        if(s.st_mode & S_ISVTX) printf("T"); else printf("-");
    printf(" %ld ", (long)s.st_nlink);
    p = getpwuid(s.st_uid); g = getgrgid(s.st_gid);
    printf("%s %s %ld ", p->pw_name, g->gr_name, (long)s.st_size);
    for(aux = ctime(&s.st_mtime); *aux != '\n'; ++aux) printf("%c", *aux);
    printf(" %s\n", de->d_name);
    free(specifier);
}
free(cale);
closedir(d);
return 0;
}
```

```
#include <unistd.h>
#include <sys/types.h>
int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
```

Trunchiaza fișierul regular desemnat de specifiatourl 'path' sau descriptorul 'fd' la dimensiunea 'length' octeți.

Dacă fișierul era inițial mai mare, datele suplimentare sunt pierdute; dacă era mai mic, este extins cu octeți nuli ('\0').

Nu este modificată poziția curentă din fișier (file offset).

Dacă dimensiunea se schimbă, momentul ultimei schimbări a caracteristicilor (st_ctime) și momentul ultimei modificări (st_mtime) sunt actualizate iar biții set-user-ID și set-group-ID pot primi valoarea 0.

În cazul lui 'truncate()', fișierul trebuie să ofere drept de scriere; în cazul lui 'ftruncate()', fișierul trebuie să fie deschis pentru scriere.

Funcțiile returnează 0 în caz de succes sau -1 (și setează 'errno') în caz de eșec.

Exemplu care arată ce se întâmplă dacă un fișier este trunchiat la o dimensiune mai mică decât poziția curentă:

```
$cat prog.c
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
int main() {
    int d; char c;
    if((d = creat("test.txt", S_IRUSR | S_IWUSR)) == -1) {
        perror("t.txt"); return 1;
    }
    write(d, "abc", 3 * sizeof(char));
    ftruncate(d, 1);
    printf("%d\n", (int) lseek(d, 0, SEEK_CUR));
    write(d, "xyz", 3 * sizeof(char));
    close(d);
    return 0;
}
```

```
$gcc -o prog prog.c
$./prog
3
$ls -l test.txt
-rw----- 1 dragulici dragulici 6 Dec  2 23:16 test.txt
$cat test.txt
axyz$
```

```
$cat prog1.c
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
int main() {
    int d; char c;
    if((d = open("test.txt", O_RDONLY)) == -1) {
        perror("t.txt"); return 1;
    }
    while(read(d, &c, sizeof(char)) == sizeof(char))
        printf("%d\n", c);
    close(d);
    return 0;
}
```

```
$gcc -o prog1 prog1.c
$./prog1
97
0
0
120
121
122
$
```

Cuprins

1 Sisteme de calcul, sisteme de operare

Organizarea unui sistem de calcul
Funcțiile sistemului de operare

2 Sisteme de operare uzuale

Incursiune în sistemul Windows
Incursiune în sistemul Linux

3 Interfața de programare C

Obiecte software și apeluri sistem
Utilizatori și grupuri
Fișiere și directoare

Procese

Semnale
Tuburi

Biblioteca standard C (tipul 'FILE')

4 Interfață interactivă linie de comandă

Interfața Command Prompt
(Windows)
Interfața Shell (Linux)

5 Interfață de scripting linie de comandă

Interfața Batch Scripting (Windows)
Interfața Shell Scripting (Linux)

6 Teme de laborator

Utilizatori și grupuri
Fișiere, directoare
Procese, semnale, tuburi

Conceptual, procesul este o instanță de execuție a unui fișier executabil (program).

În sistemele multitasking, instanța SO poate avea mai multe procese aflate în execuție simultan, chiar mai multe procese ce execută același fișier - ele urmăresc același cod, dar au propriile lor locații și valori curente pentru variabilele din program, pe care le gestionează în mod independent.

Procesul este implementat ca un obiect software, pentru care sunt alocate resurse, atribuite și informații de control specifice. El este gestionat cu ajutorul unei structuri de date PCB (process control block, bloc control proces) și are la nivelul instanței SO un identificator numeric unic PID (process ID). SO are o tabelă de procese, menținută tot timpul în RAM, care conține PCB tuturor proceselor existente.

În Linux kernel, fiecare proces este reprezentat printr-o structură de tip 'struct task_struct' (PCB) într-o listă dublu înlăncuită (tabela de procese), a cărei cap este 'init_task' (procesul cu PID = 0). Tipul 'struct task_struct' este definit în '<include/linux/sched.h>'.

În user mode, tabela de procese este vizibilă proceselor utilizator sub '/proc' - un pseudo sistem de fișiere care oferă o interfață către structurile de date ale kernelului și conține informații despre procese (a se vedea 'man proc'); aici, un director poate corespunde unui proces, un fișier din acest director corespunde unui atribut al procesului, etc.; majoritatea fișierelor sunt readonly dar în unele se poate scrie, permitând modificarea unor variabile ale kernelului. De exemplu: /proc/[pid]

subdirectorul procesului cu PID-ul 'pid' (ex: '/proc/10811');

aici, de exemplu, găsim:

/proc/[pid]/cmdline

fișier care specifică linia de comandă a procesului (dacă nu e zombie);

/proc/[pid]/task/[tid]

subdirectorul threadului cu TID-ul 'tid' al procesului cu PID-ul 'pid'

Obs: specificatorul '/proc/self' corespunde unei legături simbolice prin care un proces își poate accesa propriul director '/proc/[pid]'.

Pseudo sistemul de fișiere montat în directorul '/proc' oferă doar aparență de fișier (i-noduri, legături fizice, etc.) dar datele aferente nu sunt persistente la reboot și toate operațiile au loc în memorie.

SO multitasking (Windows, Unix, Linux, etc.) permit executarea mai multor procese simultan. Planificarea la execuție a proceselor este una din sarcinile nucleului (kernel), și este realizată de o componentă a acestuia numită planificator (scheduler), pe baza unui algoritm de planificare.

Dacă sistemul dispune de suficient de multe procesoare, planificatorul poate repartiza, ca resursă, câte un procesor fiecărui proces, și atunci va avea loc un paralelism real. În acest caz, instrucțiunile de calcul din procese diferite se pot executa în paralel (pe procesoare diferite).

Dacă sistemul nu dispune de suficiente procesoare, de exemplu, este doar unul, atunci planificatorul poate simula paralelismul prin intercalare - comută din când în când de pe un proces pe altul, cu salvarea / restaurarea stării acestora (comutare de context, context switch). În acest caz, instrucțiunile de calcul din procese diferite se vor executa tot secvențial (pe singurul procesor disponibil), doar așteptările (de exemplu, așteptarea terminării activității unui dispozitiv de I/O) se pot desfășura în paralel. Dacă procesele execută preponderent calcule, ele se vor încetini reciproc (vor fi afectate de încărcarea sistemului), dacă ele execută preponderent așteptări (de exemplu, procese interactive, care în cea mai mare parte a timpului așteaptă o comandă), ele nu vor fi foarte afectate.

Ca terminologie (cf. Wikipedia):

- rularea fiecărui proces pe câte un procesor distinct s.n. simultaneitate;
- rularea mai multor procese pe același procesor ce comută rapid de la un proces la altul, facându-le să comute între a fi în execuție și a fi în așteptare, s.n. concurență sau multiprogramare.

Algoritmii de planificare la execuție concurrentă pot fi:

- non-preemptivi: procesul curent ales este lăsat să se execute până se termină, până cedează voluntar controlul, sau până se blochează (de exemplu, într-o I/O);
- preemptivi: pe lângă cazurile anterioare, procesul curent ales poate fi întrerupt și forțat (de exemplu, pentru că i s-a consumat o cantă de timp alocată).

Comutarea de context presupune ca execuția să treacă de la un proces1 la kernel și apoi de la kernel la un proces2. Pentru trecerea de la proces1 la kernel, registrul PC al procesorului (program counter) (în arhitectura Intel este perechea de regiștri CS:IP), care conține mereu adresa instruțiunii ce urmează a se executa și care pointa inițial în proces1, trebuie încărcat cu o adresă din kernel. Aceasta se poate face:

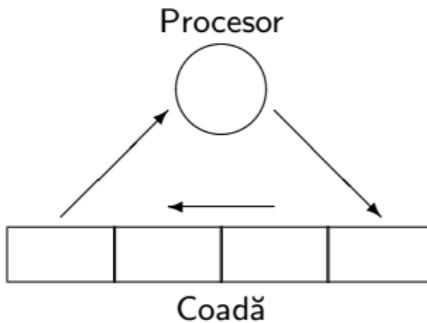
- pe cale hardware: o intrerupere hardware (semnal fizic) venită de la un echipament face ca procesorul (pe cale hardware) să încăreze executarea codului curent, să își salveze starea (de exemplu pe stiva curentă, unde pointează registrul SP (stack pointer), iar aceasta poate fi stiva utilizator a lui proces1), apoi să treacă la executarea unei anumite rutine din kernel (încărcând PC cu adresa rutinei respective); rutina mută informația salvată de procesor unde trebuie (nu are ce căuta pe stiva procesului utilizator), apoi lansează serviciul de tratare a intreruperii (care poate apela și planificatorul de procese pentru o comutare de context);
- pe cale software: se execută o instrucțiune de TRAP (care declanșază o intrerupere software), prezentă în proces1.

Calea software poate fi folosită doar la algoritmii de planificare non-preemptivi, deoarece presupune inserarea în programele utilizator de instrucțiuni de TRAP, care să cedeze (voluntar) controlul către kernel.

Implementarea unui algoritm preemptiv presupune existența unui echipament (hardware) care să emită din când în când câte un semnal de întrerupere hardware, iar rutina asociată să apeleze planificatorul. Un asemenea echipament poate fi un ceas (comutarea de context poate avea loc la fiecare întrerupere de ceas sau o dată la mai multe întreruperi de ceas).

Un algoritm preemptiv foarte folosit este Round-Robin:

Procesele sunt puse într-o coadă (sau o listă parcursă circular); primului proces i se alocă o cantă de timp în care i se permite să ruleze; dacă procesul este încă în execuție la sfârșitul cuantei (fapt verificat cu ajutorul unei întreruperi periodice de ceas), sau dacă se blochează din diferite motive, planificatorul îl trece la sfârșitul cozii și comută la procesul următor; procesele sunt manevrate fără priorități.



Algoritmul se poate îmbunătăți dacă proceselor le sunt asociate și priorități.

Linux folosește un algoritm preemptiv complex, unde sunt prezente și cuante de timp și priorități, numit 'Completely Fair Scheduler (CFS)'.

În timpul execuției, un proces se poate afla:

- în kernel mode: atunci poate executa instrucțiuni hardware privilegiate și are acces la datele nucleului; în acest mod se află când execută un apel sistem sau întrerupere;
- în user mode: atunci poate executa doar instrucțiuni hardware neprivilegiate și are acces la datele proprii; în acest mod se află când execută instrucțiuni scrise în programul executat.

Dacă un proces se află în kernel mode (de exemplu execută un apel sistem) și primește niște semnale, acestea nu sunt tratate ci ramân în aşteptare (pending) la proces.

Abia când procesul trece din kernel mode în user mode (revine din apelul sistem) tratează semnalele aflate în aşteptare, dacă nu sunt blocate prin program.

Tratarea unui semnal constă în întreruperea cursului normal al execuției și executarea handlerului asociat semnalului; pentru toate semnalele există handleuri implicite, pentru unele se poate instala un handler utilizator.

Pentru alte detalii, a se vedea secțiunea 'Semnale'.

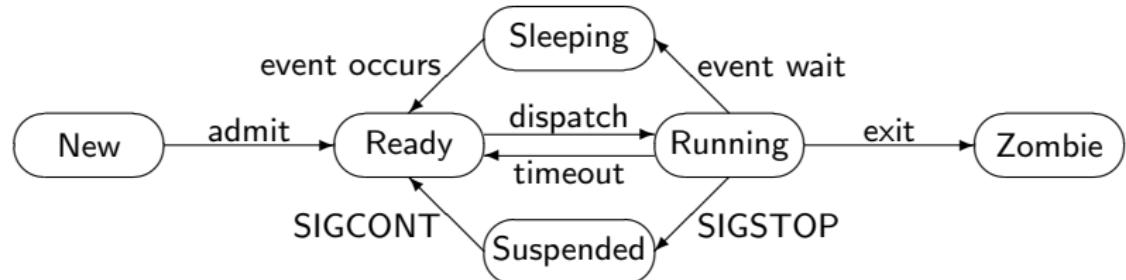
Apelurile sistem pot fi blocante (pot pune procesul în starea sleeping până când sunt întrunite anumite condiții, apoi fac return și lasă procesul să continue) sau nu.

De exemplu citirea cu 'read()' dintr-un fișier tub care este vid dar are scriitor (a se vedea secțiunea 'Tuburi') pune procesul apelant în sleeping până când cineva scrie date în tub (și atunci 'read()' efectuează citirea, face return, iar procesul continuă) sau tubul pierde scriitorii (și atunci 'read()' returnează 0 iar procesul continuă). Dacă însă tubul a fost deschis cu 'open()' și opțiunea "O_NONBLOCK", apelul 'read()' asupra tubului respectiv nu va bloca niciodată procesul apelant (va returna întotdeauna imediat, eventual un cod de eroare).

Într-un SO multitasking cu planificare preemptivă, comutarea proceselor (context switch) poate avea loc:

- când procesul curent se termină;
- când procesul curent se blochează (într-un apel sistem blocant, sau imperativ, printr-o comandă de suspendare);
- la o intrerupere de ceas.

În timpul executării (intercalate), un proces se poate afla în mai multe stări; acestea pot差别 de la un SO la altul, o schemă generală ar fi:



- new (proces nou creat): sistemul a creat inițializările necesare, dar încă nu a fost admis pentru execuție (de exemplu, încă nu există suficientă memorie);
- running (în execuție): folosește procesorul în acel moment;
aici există două substări:
 - * user mode - execută instrucțiuni din programul executat;
 - * kernel mode - execută un apel sistem sau întrerupere;
- ready (gata de execuție): nu folosește procesorul, dar poate fi executat;
- sleeping (adormit): blocat automat, în așteptarea anumitor condiții externe;
- suspended, (stopped, suspendat): blocat imperativ (la comandă);
- zombie: proces terminat; nu folosește procesorul dar PCB său încă se află în tabela de procese și conține informații minimale (codul de return); când părintele său îl colectează codul de return (cu 'wait()' sau 'waitpid()'), procesul zombie este eliminat - a se vedea mai departe.

Tranzițiile între aceste stări se efectuează astfel:

new → ready: când procesul este admis pentru execuție;

running ⇌ ready: prin decizii de planificare/pauzare (**dispatch/timeout**) ale planificatorului;

user mode → kernel mode: la apelarea unui apel sistem sau generarea unei întreruperi;

kernel mode → user mode: la revenirea dintr-un apel sistem sau handler al unei întreruperi;

running → sleeping: când procesul este în kernel mode și este penalizat, de exemplu pentru că încă nu sunt disponibile anumite resurse; de exemplu, a efectuat un apel sistem blocant care nu poate fi încă finalizat;

sleeping → ready: când penalizarea începează (de ex. resursa așteptată devine disponibilă); din acest moment, planificatorul îl poate alege oricând pentru running.

running → suspended: în Unix/Linux, la primirea semnalului SIGSTOP;

suspended → ready: în Unix/Linux, la primirea semnalului SIGCONT;

running → zombie: la terminare (normală sau anormală, de ex: return din 'main()', apel 'exit()', primirea unui semnal ucigaș precum SIGKILL).

Subliniem că:

- în starea sleeping se intră/iese automat, când sunt întrunite anumite condiții externe (ex: solicitarea unor resurse indisponibile/anumite resurse devin disponibile);
- în starea suspended se intră/iese imperativ (la comandă) (în Unix/Linux, prin trimitera semnalelor SIGSTOP/SIGCONT - aceste tranziții sunt exploataate în cadrul mecanismului de job control al anumitor shell-uri, pentru oprirea/(re)pornirea la comandă a anumitor procese lansate de utilizator).

În sistemul Linux, comanda 'ps' cu opțiunea '-o stat' afișază și starea proceselor, în felul următor, conform 'man ps':

Here are the different values that the s, stat and state output specifiers (header "STAT" or "S") will display to describe the state of a process:

- D uninterruptible sleep (usually IO)
- I Idle kernel thread
- R running or runnable (on run queue)
- S interruptible sleep (waiting for an event to complete)
- T stopped by job control signal
- t stopped by debugger during the tracing
- W paging (not valid since the 2.6.xx kernel)
- X dead (should never be seen)
- Z defunct ("zombie") process, terminated but not reaped by its parent

For BSD formats and when the stat keyword is used, additional characters may be displayed:

- < high-priority (not nice to other users)
- N low-priority (nice to other users)
- L has pages locked into memory (for real-time and custom IO)
- s is a session leader
- l is multi-threaded (using CLONE_THREAD, like NPTL pthreads do)
- + is in the foreground process group

O variabilă dintr-un program are la nivelul fișierului executabil o adresă (logică) iar la nivelul unui proces care execută fișierul adresă (fizică), locație de memorie, valoare curentă.

De exemplu:

```
-----  
|   int x;    ...  x = 10;  | fisier sursa (fis.c)  
-----
```

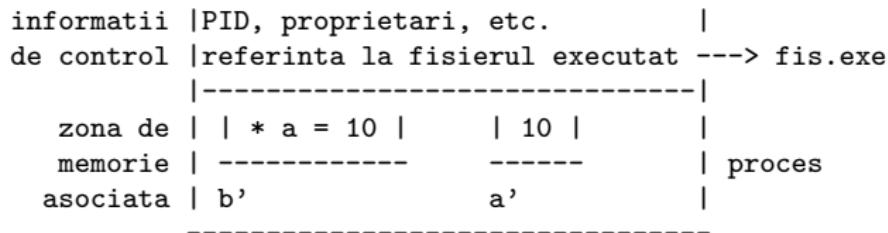
```
    |      |  
    v      v  
    a      b
```

la compilare, compilatorul stabilește adresele logice 'a' pentru variabila 'x' și 'b' pentru instrucțiunea 'x = 10;'

```
-----  
|       ...  /* a = 10; */ | fisier executabil (fis.exe)  
-----
```

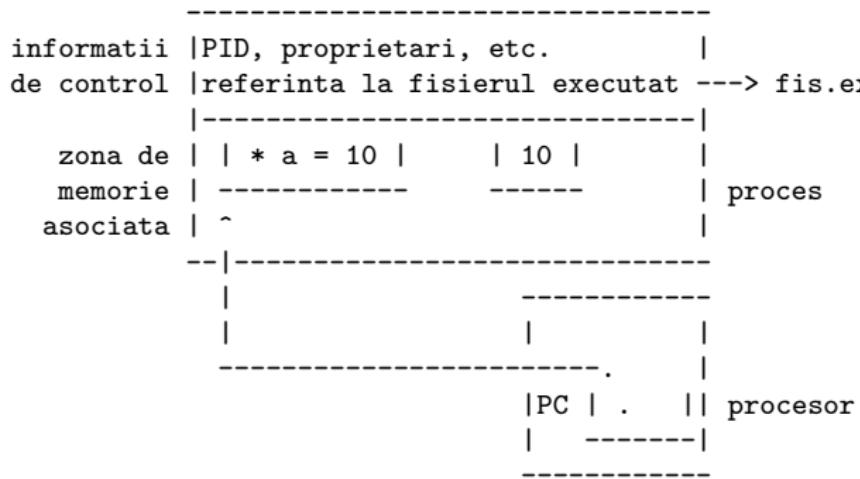
```
        b
```

În fișierul executabil, nu sunt stocate declarațiile ci doar instrucțiunile, în format mașina, iar numele variabilelor (early bound) este înlocuit cu adresa logică a lor, în toate instrucțiunile care se referă la ele; deci, în fișierul executabil ramâne doar instrucțiunea mașina '* a = 10' stocată de la adresa logică 'b'.



când fișierul executabil este executat de un proces, procesului î se asociază o zonă de memorie unde sunt încărcate instrucțiunile și unde sunt considerate locații pentru variabilele din program, care vor stoca valorile curente ale acestora; instrucțiunile și variabilele au, în această zonă de memorie, adrese fizice b', respectiv a'.

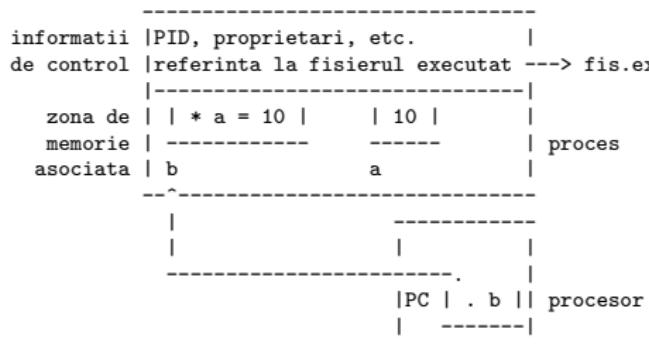
Procesorul are un registru PC (program counter, în arhitectura Intel este perechea de regiștri CS:IP) care reține adresa instrucțiunii care va fi executată și alți regiștri care rețin adresele locațiilor variabilelor accesate.



Procesoarele mai vechi foloseau modul real de adresare a memoriei (real mode): când o instrucțiune (ex. '`* a = 10`') era citită de procesor (de la adresa specificată în PC) și executată iar aceasta cerea acces la memoria la o adresă 'a', procesorul punea 'a' direct pe magistrala memoriei și accesa memoria fizică la adresa 'a'.

Astfel, adresele logice specificate în program erau considerate adrese fizice ($a' = a$, $b' = b$) iar din program se putea accesa direct toata memoria fizică.

Atunci, nu se puteau proteja datele unui proces de accesul altui proces și nu se putea face ca două procese diferite ce executa fișierul 'fis.exe' să aibă locații de memorie diferite (deci, cu adrese fizice diferite) pentru variabila 'x' din program - ambele, când executau instrucțiunea '`* a = 10`' accesau aceeași locatie fizică din memorie.



Procesoarele moderne au, pe lângă modul real, și modul protejat de adresare a memoriei (protected mode): când o instrucțiune (ex. '`* a = 10`') este citită de procesor (de la adresa specificată în PC) și executată iar aceasta cere acces la memorie la o adresa 'a', procesorul, pe cale hardware, o mapează într-o adresă 'a'', pe baza unei tabele de corespondență setabilă software, iar corespondentul 'a'' este pus pe magistrala memoriei și se accesează memoria fizică la adresa 'a''. SO crează câte o tabelă de corespondență pentru fiecare proces, a.î. fiecare tabelă să pună în corespondență spațiile de adrese logice specificabile din programe cu spații de adrese fizice disjuncte. Acum, adresele logice nu mai sunt totuna cu adresele fizice.

Dacă două procese execută în paralel fișierul 'fis.exe', care conține instrucțiunea '`* a = 10`', fiecare va avea propria tabelă setată de SO a.î. să pună în corespondență aceeași adresă logică 'a' cu adrese fizice diferite 'a1', 'a2'. Procesorul are un registru care pointează tabela curentă (de ex. CR3 în arhitectura Intel). La comutarea de context, este actualizat și acest registru a.î. să pointeze tabela procesului curent și astfel, aceeași instrucțiune '`* a = 10`', care specifică aceeași adresă logică 'a', executată în contextul celor două procese, va face ca 10 să ajungă în memorie la adrese fizice (locații) diferite.

fis.exe

```

|PID, proprietari, etc.      |      |
|referinta la fisierul executat -----
--> tabela - permite asocierea:
|      a => a'1               |
|      b => b'1               |
|
| * a = 10 |  10 |           |
|-----|-----| proces1
| b'1     | a'1 |           |
|
|-----.
|          |PC | . b || procesor
|          |-----|
|-----.
|          | R | . || 
|          |-----|
|
|-----.
| |PID, proprietari, etc.      |
| |referinta la fisierul executat -----
--> tabela - permite asocierea:
|      a => a'2               |
|      b => b'2               |
|
| * a = 10 |  10 |           |
|-----|-----| proces2
| b'2     | a'2 |           |
|
|-----.
|          |          |
|-----.

```

Astfel, nu numai că două procese diferite ce execută același fișier vor avea locații diferite (adrese fizice diferite) pentru o aceeași variabilă din program (aceeași adresă logică) dar un proces nu va putea accesa datele altui proces sau ale kernelului. Într-adevar, nu ar putea accesa memoria decât executând instrucțiuni din program care specifică adrese logice iar toate aceste adrese vor fi mapate doar în zona lui de memorie.

Dacă din program se pot specifica adrese logice pe 32 biți, spațiul de adrese logice va fi 0 ... 4G. Este dificil ca SO să asocieze proceselor curente zone de memorie fizică de 4G disjuncte - nu este suficient RAM.

Atunci, SO crează fiecare proces ca o imagine - colecție de informații care descriu starea sa curentă, incluzând valorile variabilelor, informațiile de control, etc. - și care poate fi parțial în memorie, parțial pe disc (în zona de swap) și este suficient să fie în memorie doar partea curent accesată de procesor (cod sau date). Părțile care nu sunt curent accesate pot fi evacuate pe disc (swap out) și încărcate în memorie (swap in) alte părți, eventual din alte procese.

Pentru o bună gestionare a memoriei, imaginile sunt împărțite în pagini (logice), memorie este împărțită în cadre (pagini fizice) iar paginile sunt încărcate în asemenea cadre. Paginile și cadrele au aceeași dimensiune, fixată hardware, de ex. 4k.

În Linux, se poate afla dimensiunea paginii cu apelul 'sysconf()', de ex. 'long pagesize = sysconf(_SC_PAGE_SIZE);'.

Funcția 'sysconf()' oferă la run time informații despre configurația sistemului. Ea necesită '<unistd.h>', primește ca argument un int ce specifică un parametru al sistemului și care poate fi specificat printr-o constantă simbolică și returnază un long ce reprezintă valoarea lui; în caz de eroare, returnează -1 și setează 'errno'.

Paginile conțin adrese logice și au un număr de ordine 0, 1, ... în spațiul adreselor logice. Cadrele conțin adrese fizice și au un număr de ordine 0, 1, ... în spațiul adreselor fizice.

Tabelele de corespondență ale proceselor sunt tabele de pagini și rețin corespondențe număr de pagină - număr de cadru. Mai exact, o tabelă are atâta intrări câte pagini poate avea un spațiu de adrese logice iar o intrare conține numărul cadrului în care este încărcată pagina, un bit de prezență care specifică dacă pagina este în memorie sau nu, alte informații.

Dacă în tabela de pagini a unui proces intrarea i conține valoarea j și bitul de prezență = 1, înseamnă că pagina cu numărul i a imaginii procesului se află în cadrul cu numărul j al memoriei.

Dacă, în contextul unui proces, se execută o instrucțiune ce specifică o adresă logică 'a', procesorul, pe cale hardware (folosind MMU - Memory Management Unit, Unitatea de gestiune a memoriei), calculează $i = a \text{ div } p$, $\text{ofs} = a \text{ mod } p$, unde 'p' este dimensiunea unei pagini/cadru, apoi citește din intrarea de indice i a tabelei de pagini a procesului valoarea j, apoi calculează adresa fizică $a' = j * p + \text{ofs}$ și o folosește pentru a accesa memoria.

În cazul SO care folosesc mecanisme complexe de segmentare, adresele abstracte din program (adrese logice) sunt mai întâi translatăte în alte adrese abstracte (adrese liniare sau adrese virtuale) și abia acestea sunt translatăte în adrese fizice, folosind tabelele de pagini. Astfel, procese diferite pot avea organizări diferite ale spațiului de adrese virtuale, cu alte zone de adrese valide.

În Linux, mecanismul de segmentare este simplu, a.î. adresele logice coincid practic (și le vom identifica cu) adresele virtuale. Există doar distincția între adrese virtuale și adrese fizice, puse în legătură prin tabelele de pagini.

O altă optimizare privește tabela de pagini: dacă spațiul adreselor virtuale este 4G iar o pagină are 4k, o tabelă de pagini ar trebui să aibă 1M intrări; dacă o intrare în tabela are 4 octeti, atunci tabela ar avea 4M, ceea ce este mult.

Pentru a evita tabelele foarte mari ținute tot timpul în memorie, unele calculatoare folosesc o tabelă de pagini pe mai multe niveluri: există o tabelă principală, ale carei intrări conțin fiecare adresa sau numărul de cadru al unei tabele secundare, ale cărei intrări conțin fiecare adresa sau numărul de cadru al unei tabele de nivel trei, etc., intrările tabelelor de pe ultimul nivel conțin fiecare numărul de cadru al unei pagini; o adresă virtuală este spartă în mai multe câmpuri, primul fiind un index în tabela principală, al doilea un index în tabela secundară gasită cu primul index, etc., ultimul câmp fiind o deplasare (offset) în cadrul paginii; dintre aceste tabele vor fi menținute în memorie doar cele care sunt necesare. Linux folosește, în funcție de arhitectura hardware, tabele pe 2 - 5 nivele.

Pentru simplitate, în cele ce urmează, ne vom referi generic la tabela de pagini a unui proces, fără a distinge nivelul.

Exemplu: dacă adresele virtuale sunt pe 16 biți, adresele fizice sunt pe 15 biți iar pagina are 4k, atunci:

- spațiul adreselor virtuale are 64k și este împărțit în 16 pagini, având numerele 0, 1, ..., 15;
- spațiul adreselor fizice are 32k și este împărțit în 8 cadre, având numerele 0, 1, ..., 7;
- tabela de pagini a unui proces are 16 intrări, având indicii 0, 1, ..., 15;
- dacă în contextul procesului respectiv se execută o instrucțiune ce specifică o adresă virtuală $a = 8196$, procesorul, pe cale hardware, calculează $i = 8196 \text{ div } 4k = 2$, $\text{ofs} = 8196 \text{ mod } 4k = 4$, apoi citește din intrarea de indice $i = 2$ a tabelei de pagini a procesului valoarea $j = 6$, și bitul de prezență = 1 (deci, pagina este în memorie), apoi calculeaza adresa fizică $a' = 6 * 4k + 4 = 24580$ și o folosește pentru a accesa memoria.
- practic, procesorul (MMU) a primit adresa virtuală în binar 0010000000000100 și a înlocuit în ea (pe baza tabelei de pagini) primii 4 biti 0010 cu 3 biti 110, obținând adresa fizica 110000000000100.

spatial					
de adrese					
virtuale pagini					

50K-64K	1 1111 = 15				

56K-60K	1 1110 = 14				

52K-56K	1 1101 = 13				

48K-52K	1 1100 = 12				

44K-48K	1 1011 = 11	bit			
-----		present/absent			
40K-44K	1 1010 = 10				

36K-40K	1 1001 = 9	de			
-----		pagina V		cadre de	adrese fizice
32K-36K	1 1000 = 8	-----		pagina	de memorie

28K-32K	1 0111 = 7	15 000 0	111 = 7		28K-32K

24K-28K	1 0110 = 6	14 000 0	110 = 6 xxx		24K-28K

20K-24K	1 0101 = 5	13 000 0	101 = 5		20K-24K

16K-20K	1 0100 = 4	12 000 0	100 = 4		16K-20K

12K-16K	1 0011 = 3	11 000 0	011 = 3		12K-16K

8K-12K	xxx 0010 = 2	10 000 1	010 = 2		8K- 12K

4K- 8K	1 0001 = 1	9 110 1	001 = 1		4K- 8K

0K- 4K	1 0000 = 0	8 010 1	000 = 0		0K- 4K

Din interiorul unui program sunt vizibile doar adresele logice, nu și cele fizice, nu se vede tabela de pagini și nici dacă valoarea curentă a unei variabile este în memorie sau pe disc.

Prin implementarea spațiului de adrese logice ca o imagine care poate fi stocată parțial în memorie și parțial pe disc, de unde este încărcată în memorie la nevoie (din program nu este vizibil unde se află o anumită informație, în memorie sau pe disc), este creată percepția că sistemul are o memorie mai mare - se mai spune că memoria fizică se extinde prin memoria virtuală.

Deși se folosește discul (zona de swap) alături de memorie pentru a gestiona procese (imagini) mai mari, spațiul disponibil poate fi prea mic pentru a implementa ca imagine tot spațiul virtual de adrese al proceselor - de exemplu, dacă adresele virtuale sunt pe 32 biti, spațiul virtual are 4G și ar trebui că fiecare proces să aibă o imagine de 4G.

Atunci, proceselor li se crează imagini mai mici, care implementează doar o parte din spațiul de adrese virtual iar în acest sens poate contribui compilatorul - acesta evaluează spațiul virtual utilizat de cod, date, etc. și consemnează informația respectivă în fișierul executabil (care nu este doar o listă de instrucțiuni ci are și un antet, etc. - executabilul are un format standardizat, de ex. formatul 'PE' în Windows, 'ELF' în Unix/Linux).

Când SO lansează fișierul executabil ca proces, implementează ca pagini de imagine doar zonele din spațiul virtual de adrese consemnate în fișierul executabil.

Dacă, la rulare, procesorul execută o instrucțiune din proces care solicită adresă virtuală 'a', el calculează indicele 'i' al paginii care conține adresa 'a' și consultă intrarea 'i' a tabelei de pagini a procesului; atunci, sunt posibile cazurile:

- intrarea 'i' conține bitul de prezenta = 1 și valoarea 'j'; atunci, pagina este în memorie în cadrul 'j', procesorul accesează memoria la adresa fizică calculată iar instrucțiunea se execută;
- intrarea 'i' conține bitul de prezență = 0; atunci se generează un TRAP către SO, numit defect de pagina (page fault); SO verifică dacă pagina există (a fost creată ca parte a imaginii) și este pe disc, blochează procesul și îl pune într-o coadă de așteptare la pagina respectivă, inițiază o rutină care începe încărcarea paginii într-un cadru liber (dacă nu există cadre libere, se evacuează o altă pagină pe disc) iar rutina se execută intercalat cu celelalte procese (între timp, cadrul este etichetat ca ocupat, prevenind utilizarea lui în alte scopuri); când rutina a terminat încărcarea, SO actualizează tabela de pagini a procesului și îl deblochează; la reluare, procesul își restartează instrucțiunea care a generat defectul de pagină iar aceasta va găsi pagina în memorie;
- dacă în urma defectului de pagină SO constată că pagina cerută nu există nici pe disc (nu a fost creată ca parte a imaginii procesului), atunci semnalează procesului o eroare de tip segmentation fault - în Unix/Linux, procesul primește semnalul SIGSECV, handlerul implicit produce terminarea procesului dar se poate instala un handler utilizator (atunci, doar instrucțiunea care a cauzat eroarea nu este executată).

O intrare într-o tabelă de pagini poate conține următoarele tipuri de informații (modul de implementare depinde de mașina):

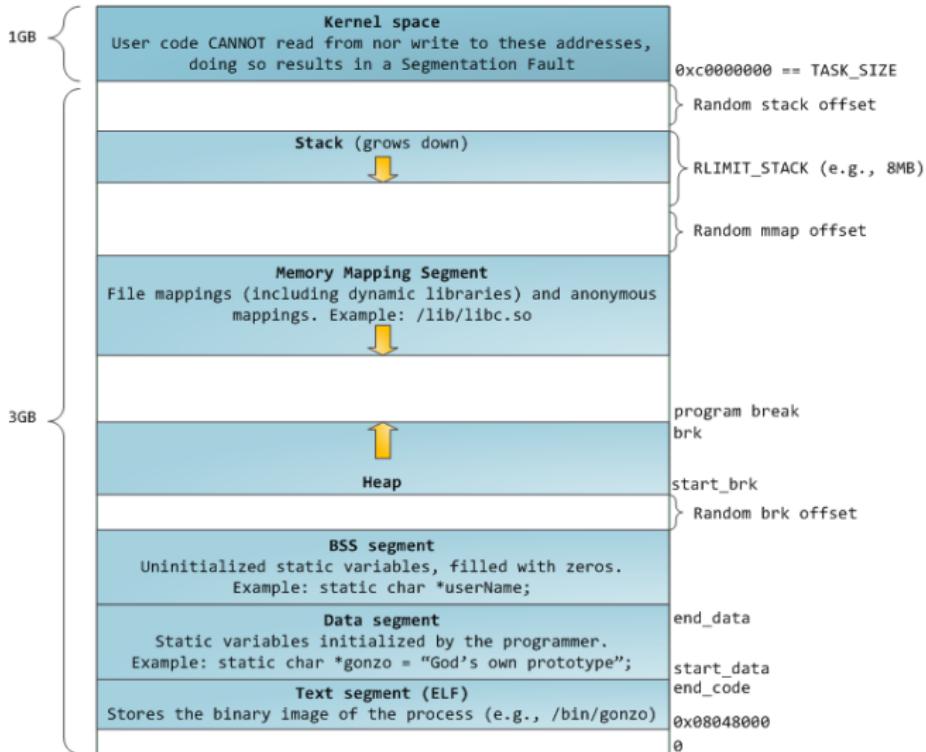
- Numărul cadrului de pagină: l-am discutat.
- Bitul present/absent: l-am discutat; dacă este 1, atunci pagina asociată este în memorie, în cadrul cu numărul menționat; dacă este 0, atunci pagina asociată nu este în memorie, iar accesarea acestei intrări va provoca un defect de pagină.
- Biții de protecție: arată tipul de acces permis asupra conținutului paginii: citire, scriere, execuție sau o combinație de acestea.
- Bitul user/supervisor: arată dacă pagina poate fi accesată doar de codul executat în kernel mode sau și de codul executat în user mode.
- Bitul modificat (sau bitul murdar (dirty bit)): când se scrie în pagina asociată din cadrul, acest bit devine 1 (i.e. pagina este marcată ca murdară (dirty)); dacă SO dorește refolosirea cadrului paginii și ea este murdară, mai întâi salvează pagina pe disc (altfel conținutul cadrului poate fi abandonat, deoarece copia de pe disc este încă validă);
- Bitul accesat: devine 1 oricând pagina asociată din cadrul este accesată, fie pentru citire fie pentru scriere; dacă se generează un defect de pagină iar SO dorește să elibereze un cadrul pentru a încărca pagina solicitată, va prefera un cadrul neaccesat (i.e. cu bitul accesat = 0).

- Bitul memorie intermediară dezactivată: permite dezactivarea caching-ului (i.e. menținerii în memoria tampon a unei copii) pentru pagina asociată; facilitatea este importantă pentru paginile corespunzatoare unor registri din diverse dispozitive, în cazul mașinilor cu Memory-Mapped I/O; dacă SO așteaptă într-o buclă ca un dispozitiv de I/O să raspundă la o comandă ce tocmai a fost dată, este esențial ca la fiecare iterație hardware-ul să livreze valoarea de la dispozitiv, nu o copie veche din memoria tampon (cache).

Notăm că pe parcursul rulării unui proces, o aceeași pagină a sa poate fi transferată de mai multe ori între RAM și disc și (re)încărcată în cadre diferite.

Adresa pe disc unde se găsește o pagină atunci când nu se află în memorie nu este reținută în tabela de pagini (tabela de pagini reține doar informația necesară hardware-ului pentru a transforma adresele virtuale în adrese fizice) ci în tabele interne ale SO (fiind folosite la tratarea defectelor de pagină).

Organizarea uzuială a spațiului de adrese virtuale al unui proces într-un sistem Linux pe 32 biți este următoarea:



sursa:

<https://manybutfinite.com/post/anatomy-of-a-program-in-memory/>

- Kernel space: adrese virtuale utilizate de nucleu; nu neapărat toate sunt mapate (în pagini), dar nucleul le poate mapea oricând are nevoie; kernel space este marcat în tabelele de pagini ca accesibil doar codului privilegiat; codul executat de proces în kernel mode (ex. un apel sistem) poate accesa acest spațiu, codul executat de proces în user mode (instrucțiuni scrise de utilizator) nu - dacă încearcă, se generează page fault, încheiat cu segmentation fault.

În Linux, kernel space mapează aceeași memorie fizică în toate procesele (paginile respective sunt comune tuturor imaginilor) și conțin cod și date nucleu și poate fi accesat de intreruperi și apeluri sistem.

- Stack: zona organizată ca o stivă, utilizată pentru a gestiona apelurile de subroutines; la apelare, se alocă o zonă asociată apelului (cadru de apel) în varful stivei, conținând datele automatice ale acestuia (variabile locale automate, parametri, etc.), iar la revenire (return), zona se eliberează; pentru a gestiona stiva unui proces, procesorul folosește registrul SP, care conține adresa vârfului stivei.

Stiva crește spre adrese mici (în jos); la intrarea în apel, valoarea lui SP scade, la revenirea din apel, valoarea lui SP crește.

Dacă sunt încărcate în stivă prea multe date, va fi epuizată zona din spațiul virtual pentru care sunt alocate pagini și atunci se generează page fault; SO tratează acest page fault alocând pagini suplimentare și mărind astfel domeniul de adrese virtuale utilizabile de către stivă.

Sistemul Linux specifică valori limită ale resurselor sale, iar dacă stivă crește mai mult de RLIMIT_STACK (de obicei 8MB), atunci nu se mai alocă pagini suplimentare iar page fault se va încheia cu segmentetion fault (stack overflow).

Dacă sunt descărcate date din stiva (stiva scade) nu sunt dezalocate paginile alocate la creștere - ele rămân alocate și pot fi folosite la o creștere ulterioară a stivei.

Limitele resurselor sistemului se pot afla și modifica (cu anumite restricții) folosind funcțiile 'getrlimit()', 'setrlimit()' - a se vedea mai jos.

Obs: creșterea stivei este singura situație când accesul la zone nealocate în pagini este valid - în urma accesării, page fault conduce la alocarea automată de pagini suplimentare; în cazul altor accese la zone nealocate în pagini generează page fault care conduce la segmentation fault.

- Memory mapping segment: zona folosită pentru maparea fișierelor în memorie. Utilizând apelul 'mmap()' (Linux), 'CreateFileMapping()' / 'MapViewOfFile()' (Windows) aplicația poate pune în corespondență un domeniu de drese virtuale cu conținutul unui fișier - practic, conținutul fișierului va fi considerat pagini alocate pentru zona respectivă.

Atunci, aplicația va putea accesa conținutul fișierului prin instrucțiuni de accesare a memoriei (pointeri, atribuirile de variabile, etc.), ceea ce este mai rapid decât cu apelurile 'read()' / 'write()', care generează TRAP-uri (TRAP va avea loc doar la page fault).

Mai multe procese pot mapa un același fișier și atunci paginile din fișier vor fi comune mai multor imagini - practic, un domeniu de adrese virtuale dintr-un proces va corespunde acelorași locații de memorie fizică cu un domeniu de adrese virtuale din alt iar procesele pot comunica prin zona de memorie comună.

Maparea fișierelor în memorie este un mod comod și performant de a efectua I/O în fișiere și astfel este folosit pentru încărcarea (și accesarea în comun de către mai multe procese) a bibliotecilor cu legare dinamică ('.so' în Linux, '.dll' în Windows).

Se pot crea mapări de memorie anonime, care nu corespund unui fișier existent iar paginile alocate se pot folosi pentru date ale programului. De exemplu, în Linux, dacă se cere alocarea unui bloc de memorie via funcția de biblioteca C 'malloc()' și care este mai mare decât limita MMAP_THRESHOLD, funcția va crea o mapare anonimă în loc să folosească zona heap.

- Heap: zona folosită la alocari dinamice ('malloc()', 'calloc()', 'realloc()').

Zona poate fi extinsă la cerere (se aloca pagini suplimentare), folosind apelurile 'brk()', 'sbrk()' - a se vedea mai jos.

Dacă funcțiile din biblioteca C 'malloc()', 'calloc()', 'realloc()' pot satisface cererile de memorie utilizând spațiul (paginile) deja alocat pentru heap, nu vor face apel la kernel; altfel, vor mări heap-ul folosind apelul 'sbrk()'.

Algoritmii folosiți de 'malloc()', 'calloc()', 'realloc()', 'free()' pentru gestionarea memoriei alocate sunt complecși și lenți, a.î. sistemele cu constrângeri de timp real (real-time systems) implementează alocatori dedicați.

- BSS segment: stochează conținutul variabilelor statice neinitializate (ex: 'static int x;'); maparea zonei este anonimă (paginile nu corespund unui fișier anume și initial sunt umplute cu octeți 0).
- Data segment: stochează conținutul variabilelor statice inițializate în codul sursă (ex: 'static int x = 10;'); maparea zonei nu este anonimă - paginile corespund zonei din fișierul binar executat care conține valorile de inițializare (în ex. anterior, 10); maparea este însă privată - modificările efectuate în memorie nu sunt salvate în fișier (ar însemna că modificarea valorilor variabilelor globale la executarea procesului să conducă la modificarea fișierului executabil de pe disc) și sunt vizibile doar procesului care le-a facut.

Maparea privată se realizează astfel: în intrările corespunzătoare din tabela de pagini se specifică accesul read only iar la prima încercare de scriere se generează un page fault la care SO răspunde înlocuind în imaginea procesului pagina din fișier (care este comună tuturor proceselor ce executa fișierul respectiv și accesibilă read only) cu o copie privată (prezentă doar în imaginea acestui proces) și modificând specificarea de acces din tabela de pagini în read / write; scrierea se va face în noua pagină. Mecanismul s.n. copy-on-write.

Astfel, dacă mai multe procese execută același fișier, inițial ele vor utiliza aceleași pagini cu variabile statice initializate (imaginile lor vor avea aceste pagini comune) dar la prima încercare a unui proces de a modifica o asemenea variabilă, el își va face o copie privată a paginii respective (în imaginea lui, noua pagină o va înlocui pe cea veche) și va gestiona o valoare curentă pentru variabila respectivă vizibilă doar lui.

- Text segment: stochează codul programului, alte obiecte adresabile, ca literalii string, și mapeaza fișierul binar executat; intrările din tabela de pagini specifică doar acces read și execute (nu și write) - încercarea da a scrie generează page fault finalizat cu segmentation fault.

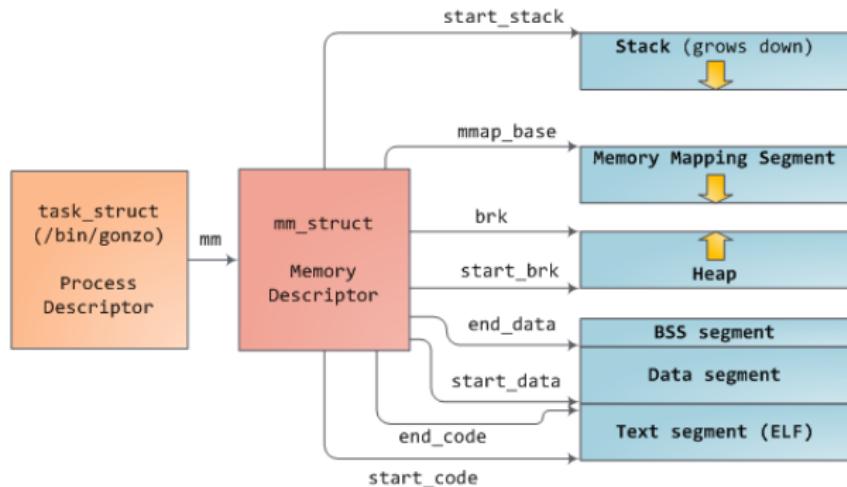
Așadar, dacă mai multe procese execută același fișier, imaginile lor vor conține aceleași pagini de cod (este un aspect de reentrantă).

- Pentru a proteja procesele împotriva exploatarii unor vulnerabilități de securitate de la distanță, Linux randomizează segmentele stack, memory mapping, heap, adăugând diverse deplasamente (offset) adreselor lor de început.

Observații:

- Uneori, este folosit termenul "segmentul de date" (data segment) pentru a desemna ansamblul Heap + Bss + Data.
- Tipul de acces permis asupra paginilor alocate pentru o regiune a spațiului virtual poate fi schimbat, folosind apelul 'mprotect()'.

În Linux, spațiul de adrese de memorie al procesului este descris cu ajutorul unei structuri de date de tip 'struct mm_struct', definită în <linux/sched.h>:



sursa:

<https://manybutfinite.com/post/how-the-kernel-manages-your-memory/>

Tabela de pagini a procesului este indicată de membrul 'pgd' al structurii de tip 'struct mm_struct'.

Zonele de memorie ale unui proces pot fi consultate în fișierul:

/proc/[pid]/maps

notăm că un segment poate conține mai multe zone - de exemplu, fiecare fișier mapat în memorie are în mod normal zona sa din Memory mapping segment iar bibliotecile cu legare dinamică au extra zone similare cu BSS și Data segment;

O linie din fișier descrie o regiune de memorie virtuală contiguă și are câmpurile:
address - adresele virtuale de început și sfârșit ale zonei;

permissions - tipul de acces permis asupra paginilor alocate pentru această regiune (citire, scriere, execuție, etc.);

offset - dacă regiunea a fost mapată dintr-un fișier (folosind 'mmap()'), acesta este deplasamentul (offset) în fișier de unde începe maparea; dacă memoria nu a fost mapată dintr-un fișier, este 0;

device - dacă regiunea a fost mapată dintr-un fișier, acesta este numărul major și cel minor al dispozitivului unde se află fișierului (în hexa);

inode - dacă regiunea a fost mapată dintr-un fișier, acesta este numărul fișierului (numărul de i-nod);

pathname - dacă regiunea a fost mapată dintr-un fișier, acesta este specifikatorul (nume/cale) fișierului; acest câmp este blank pentru regiuni mapate anonim;

există regiuni speciale cu nume, ca [heap], [stack], [vdso]; [vdso] înseamnă "virtual dynamic shared object" și este folosită de apelurile sistem pentru a comuta în kernel mode.

```
#include <sys/mman.h>
int mprotect(void *addr, size_t len, int prot);
```

Modifică tipul de acces permis asupra paginilor procesului apelant alocate pentru o regiune a spațiului virtual în intervalul de adrese [addr, addr+len-1]; 'addr' trebuie să fie aliniat la nivel de pagină.

Dacă procesul încearcă să acceseze memoria într-un mod care încalcă tipul de acces permis, produce segmentation fault - kernelul generează pentru el un semnal SIGSEGV.

Argumentul 'prot' poate fi un OR pe biți ('|') de valori care se pot specifica prin constante simbolice, de exemplu:

PROT_NONE : memoria nu poate fi accesată în nici un fel;

PROT_READ : memoria poate fi citită;

PROT_WRITE : memoria poate fi modificată;

PROT_EXEC : memoria poate fi executată.

Funcția 'mprotect()' returnează 0 în caz de succes sau -1 (și setează 'errno') în caz de eșec.

Obs: În Linux, se poate apela 'mprotect()' asupra oricărei adrese din spațiul virtual al procesului, cu excepția zonei 'kernel vsyscall area'; în particular, poate fi folosit pentru a permite scrierea în paginile de cod.

Exemplu: scrierea în Text segment - TODO

```
#include <stdlib.h>
void *valloc(size_t size);
#include <malloc.h>
void *memalign(size_t alignment, size_t size);
```

Alocă memorie aliniată.

Funcția 'memalign()' alocă o zonă de 'size' octeți și returnează adresa ei; adresa memoriei alocate va fi un multiplu de 'alignment', care trebuie să fie o putere a lui 2.

Funcția 'valloc()' alocă o zonă de 'size' octeți și returnează adresa ei; adresa memoriei alocate va fi un multiplu al dimensiunii paginii; apelul 'valloc(size)' este echivalent cu 'memalign(sysconf(_SC_PAGE-SIZE), size)'.

În ambele cazuri:

- memoria alocată nu este inițializată cu 0;
- implementările 'glibc' ale acestor funcții permit ca adresa alocată să poate fi furnizată lui 'free()' (POSIX nu cere asta);
- în caz de eroare, ambele funcții returnează NULL și setează 'errno'.

Observații:

- funcția 'malloc()' alocă zone de memorie aliniate la adrese multipli de 8 octeți;
- cu 'valloc()', putem aloca pagini pentru regiuni din zona Heap, a căror adresă de început ni se va face cunoscută; folosind această adresă, putem modifica tipul de acces permis asupra paginilor respective, folosind 'mprotect()'.

Exemplu: Alocăm 4 pagini pentru regiuni din zona Heap, modificăm tipul de acces permis asupra paginii a 3-a în 'PROT_READ', apoi încercăm scrieri în paginile respective, urmărind apariția segmentation fault cu ajutorul unui handler utilizator instalat pentru semnalul SIGSEGV:

```
$cat prog.c
#include <signal.h>
#include <sys/mman.h>
#include <unistd.h>
#include <malloc.h>
#include <stdio.h>
#include <setjmp.h>
jmp_buf jb;
void handler(int n) {longjmp(jb, 1);}
int main(int argc, char *argv[]) {
    long pagesize; char *buffer; char *p;
    if(signal(SIGSEGV, handler) == SIG_ERR) {
        perror("signal()");
        return 1;
    }
    if((pagesize = sysconf(_SC_PAGE_SIZE)) == -1) {
        perror("sysconf()");
        return 1;
    }
    if((buffer = memalign(pagesize, 4 * pagesize)) == NULL) {
        perror("memalign()");
        return 1;
    }
    /* Protectia initiala este PROT_READ | PROT_WRITE */
    if(mprotect(buffer + pagesize * 2, pagesize, PROT_READ) == -1) {
        perror("mprotect()");
        return 1;
    }
    printf("Dimensiunea paginii: %ld\nAdresa de start: %p\n",
    pagesize, buffer);
    if(! setjmp(jb)) for(p = buffer ; ; *p++ = 'x');
    printf("Primit SIGSEGV la adresa: %p\nDeplasamentul este: %ld\n",
    p, p - buffer);
    return 0;
}
```

```
$gcc -o prog prog.c
$./prog
Dimensiunea paginii: 4096
Adresa de start: 0x55db2fe82000
Primit SIGSEGV la adresa: 0x55db2fe84001
Deplasamentul este: 8193
```

Comentarii:

- cu 'signal(SIGSEGV, handler)' am instalat handlerul utilizator 'handler()' pentru semnalul SIGSEGV - la venirea semnalului, se va executa această funcție;
- cu 'setjmp(jb)' am memorat în 'jb' un checkpoint - o fază a execuției care poate fi restaurată ulterior; practic, în 'jb' a fost memorat contextul curent de execuție al stivei apelurilor de funcții; la intrarea în 'setjmp(jb)' prin apelare normală, acesta returnează 0 și se intră în 'for';
- după 2 * 4k accesări ale paginilor alocate, când se accesează pagina a 3-a, procesul primește SIGSEGV, se apelează funcția 'handler()' iar aceasta execută 'longjmp(jb, 1)', restaurându-se starea execuției de la momentul apelului 'setjmp(jb)'; din apelul 'setjmp(jb)' se mai revine o dată, acum returnându-se 1 (al doilea argument al lui 'longjmp()') și nu se mai intră în 'for';
- pentru alte detalii privind instalarea handlerelor pentru semnale și funcția 'longjmp()', a se vedea secțiunea semnale.

```
#include <sys/time.h>
#include <sys/resource.h>
int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, const struct rlimit *rlim);
```

Consultă / modifică limite de resurse.

Orice resursă are asociată o limită soft și un hard, aşa cum sunt definite în structura:

```
struct rlimit {
    rlim_t rlim_cur; /* Soft limit */
    rlim_t rlim_max; /* Hard limit (ceiling for rlim_cur) */
};
```

Limita soft este valoarea pe care kernelul o impune pentru resursa corespunzătoare.

Limita hard acționează ca o plafonare pentru limita soft: un proces neprivilegiat își poate seta limita soft doar la o valoare de la 0 la limita hard și își poate micșora (ireversibil) limita hard; un proces privilegiat poate modifica arbitrar oricare dintre limite.

Tipurile 'struct rlimit' și 'rlim_t' sunt definite în '<sys/resource.h>'; tipul 'rlim_t' este definit ca un tip întreg fără semn iar pentru el sunt definite câteva valori prin constante simbolice; de exemplu, valoarea RLIM_INFINITY a unei resurse înseamnă fără limită (nu sunt impuse limite asupra resursei respective) (de obicei, este -1).

Argumentul 'resource' poate fi specificat tot printr-o constantă simbolică; câteva exemple:

RLIMIT_AS : dimensiunea maximă a memoriei virtuale a procesului (spațiul de adresare), în octeți; este rotunjit în jos la dimensiunea paginii; expansiunea automată a stivei și apelurile 'brk()', 'mmap()', 'mremap()' eșuează dacă o depășesc);

RLIMIT_DATA : dimensiunea maximă a segmentului de date al procesului (date inițializate, date neinițializate și heap), în octeți; este rotunjit în jos la dimensiunea paginii; afectează apelurile 'brk()', 'sbrk()' și (începând cu Linux 4.7) 'mmap()';

RLIMIT_STACK

dimensiunea maximă a stivei procesului, în octeți; dacă stiva atinge această limită, procesul primește semnalul SIGSEGV; începând cu Linux 2.6.23, această limită determină și dimensiunea spațiului folosit pentru argumentele în linia de comandă și variabilele de environment ale procesului - a se vedea 'execve()', mai jos;

RLIMIT_FSIZE

dimensiunea maximă a în octeți pe care o poate avea un fișier creat de proces; afectează apelurile 'write()', 'truncate()';

RLIMIT_NOFILE

este $1 +$ valoarea maximă a unui descriptor de fișier deschis de proces; afectează apelurile 'open()', 'pipe()', 'dup()', etc.);

RLIMIT_NICE (începând cu Linux 2.6.12)

limită superioară până la care poate fi ridicată valoarea 'nice' a procesului (care definește prioritatea, a se vedea mai jos); afectează apelurile 'setpriority()', 'nice()'; limita superioară factuală pentru valoarea 'nice' este calculată ca $20 - \text{rlim_cur}$; intervalul util pentru această limită este de la 1 la 40 (corespunde intervalului de la 19 la -20 pentru valoarea 'nice');

RLIMIT_NPROC

numarul maxim de procese / threaduri cărora proprietarul real al procesului apelant le poate trimite semnale (extant process); dacă numărul curent de procese ale acestui utilizator este \geq această limită, procesul nu mai poate genera procese copil (cu 'fork()').

Funcțiile 'getrlimit()', 'setrlimit()' returneaza 0 în caz de succes sau -1 (și setează 'errno') în caz de eșec.

Observații:

- limitele resurselor sunt atribuite per-proces și sunt partajate (shared) de toate threadurile procesului;
- procesele copil, create cu 'fork()', sau cele care înlătăresc procesul, create cu 'execve()', moștenesc limitele resurselor;
- scăderea limitei soft a unei resurse sub nivelul curent la care procesul consumă resursa respectivă reușește dar împiedică procesul să își crească ulterior nivelul de consum al acelei resursei;
- se pot modifica limitele resurselor shell-ului folosind comanda internă 'ulimit'; limitele resurselor shell-ului sunt moștenite de procesele copil create de acestea pentru a executa comenzi externe;
- limitele resurselor unui proces se pot consulta (începând cu Linux 2.6.24) în fișierul:

/proc/[pid]/limits

Exemplu: setarea limitei pentru stiva:

```
$cat prog.c
#include <sys/resource.h>
#include <stdlib.h>
#include <stdio.h>
int n;
void f() {
    int buf[500000];
    printf("Apel %d\n", ++n);
    f();
}
int main(int argc, char *argv[]) {
    struct rlimit lim;
    if (getrlimit(RLIMIT_STACK, &lim) == -1) {
        perror("getrlimit"); return 1;
    }
    printf("Limitele curente hard: %ld, soft: %ld\n", (long)lim.rlim_max, (long)lim.rlim_cur);
    if(argc == 2) {
        lim.rlim_cur = atol(argv[1]);
        if (setrlimit(RLIMIT_STACK, &lim) == -1) {
            perror("setrlimit"); return 1;
        }
    }
    printf("Testare cu limita soft %ld:\n", (long)lim.rlim_cur);
    f();
    return 0;
}
```

```
$gcc -o prog prog.c
$./prog
Limitele curente hard: -1, soft: 8388608
Testare cu limita soft 8388608:
Apel 1
Apel 2
Apel 3
Apel 4
Segmentation fault (core dumped)
$./prog 6000000
Limitele curente hard: -1, soft: 8388608
Testare cu limita soft 6000000:
Apel 1
Apel 2
Segmentation fault (core dumped)
```

```
#include <unistd.h>
int brk(void *addr);
void *sbrk(intptr_t increment);
```

Modifică poziția 'program break', care definește sfârșitul segmentului de date al procesului.

'program break' este adresa (virtuală a) primei locații aflate după sfârșitul BSS segment (segmentul de date statice neinitializate); incrementarea 'program break' are ca efect alocarea de memorie pentru proces (se aloca pagini pentru noi adrese virtuale); decrementarea 'program break' dezalocă memorie (paginile respective sunt dezionate, adresele virtuale asociate rămân nemapate iar accesarea lor va produce segmentation fault).

Funcția 'brk()' setează sfârșitul segmentului de date la valoarea 'addr', dacă valoarea este rezonabilă, sistemul are suficientă memorie și nu este depășită limita maximă pentru segmentul de date al procesului (se poate afla cu 'setrlimit()' și 'RLIMIT_DATA').

Funcția 'sbrk()' incrementă segmentul de date al procesului cu 'increment' octeți. Apelul 'sbrk(0)' poate fi folosit pentru a afla poziția curentă (adresa) lui 'program break'.

Funcția 'brk()' returnează returnează 0 în caz de succes sau -1 (și setează 'errno') în caz de eșec.

Funcția 'sbrk()' returnează poziția (adresa) anterioară a lui 'program break' în caz de succes (dacă 'program break' a fost incrementat, aceasta este adresa de început a zonei de memorie nou alocate) sau (void *) -1 (și setează 'errno') în caz de eșec.

Funcția 'malloc()' aloca memorie folosind 'brk()', 'sbrk()', a.î. 'program break' desemnează de fapt sfârșitul zonei Heap. Se recomandă folosirea 'malloc()' pentru alocarea dinamică de memorie, în locul 'brk()', 'sbrk()', deoarece este mai confortabil și mai portabil.

Exemplu: Stiva de întregi alocată cu 'brk()', 'sbrk()':

```
#include <unistd.h>
#include <stdio.h>
int pushint(int n) {
    int *p;
    if((p = sbrk(sizeof(int))) == (void *) -1) return -1;
    *p = n;
    return 0;
}
int popint(int *n) {
    int *p;
    if(n == NULL) return -1;
    *n = ((int *) sbrk(0)) [-1];
    if((p = sbrk(- sizeof(int))) == (void *) -1) return -1;
    return 0;
}
int main() {
    int i, n;
    for(i = 0; i < 10000; ++i)
        if(pushint(i) == -1) {perror("pushint()"); return 0;}
    for(i = 9999; i >= 0; --i) {
        if(popint(&n) == -1) {perror("popint()"); return 0;}
        if(n != i) {printf("Eroare: i = %d, n = %d\n", i, n); return 0;}
    }
    return 0;
}
```

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
int munmap(void *addr, size_t length);
```

Asociaza pagini pentru zone din spațiul virtual de adrese al procesului apelant.

Paginile pot fi partajate cu alte procese (mai multe procese le pot include în spațiul lor de adrese virtuale, a.î. paginile respective vor putea fi folosite la comunicarea între procese pe calea memoriei) sau pot fi private (modificările făcute în ele vor fi vizibile doar procesului care le-a făcut - dacă și alte procese primesc mapările respective, le vor dobândi copy-on-write, i.e. la prima scriere într-o pagină se va face automat un duplicat iar procesul va păstra mapat duplicatul).

Paginile pot fi parte a unui fișier existent (file mapping) (paginile sunt swapate out în acest fișier) și atunci informația scrisă va persista în fișier și după de-mapare sau terminarea procesului, sau poate să nu fie parte a unui fișier (mapare anonimă, anonymous mapping) și atunci informația scrisă persistă doar atât timp cât există procese care au mapate paginile lor în spațiul propriu de adrese.

Parametri:

'addr' : adresa de început a zonei virtuale mapate; dacă este NULL, sistemul alege o adresă potrivită și care este aliniată la un multiplu al dimensiunii paginii (este metoda cea mai portabilă de a mapa); dacă nu este NULL și am specificat 'MAP_FIXED' (a se vedea mai jos), trebuie să fie o adresă aliniată la un multiplu al dimensiunii paginii (pot exista și alte restricții); dacă nu este NULL și nu am specificat 'MAP_FIXED', sistemul o consideră ca o "sugestie" și alege o adresă potrivită învecinate; funcția 'mmap()' va returna adresa de mapare;

'length' : dimensiunea zonei mapate; trebuie să fie ≥ 0 ;

'prot' : tipul de acces permis asupra paginilor mapate; în cazul file mapping, nu trebuie să fie în conflict cu modul de deschidere al fișierului indicat de 'fd'; poate fi un OR pe biți ('|') de următoarele valori:

PROT_EXEC : paginile pot fi executate

PROT_READ : paginile pot fi citite

PROT_WRITE : paginile pot fi modificate

PROT_NONE : paginile nu pot fi accesate

'flags' : determină vizibilitatea modificărilor făcute în paginile mapate;

în 'flags' trebuie specificată exact una dintre următoarele valori:

MAP_SHARED : mapare partajată; modificările în paginile mapate sunt vizibile și altor procese care mapează aceleași pagini iar, în cazul file mapping, modificările sunt transferate fișierului subiecent; pentru a controla precis momentul transferului în fișier, se folosește 'msync()' (a se vedea mai jos);

MAP_PRIVATE : mapare privată copy-on-write; modificările în paginile mapate nu sunt vizibile și altor procese care mapează aceleași pagini iar, în cazul file mapping, modificările nu sunt transferate fișierului subiecent; practic, ori se aloca pagini noi, ori se asociază pagini existente în modul copy-on-write - la prima scriere într-o pagină, se va face automat un duplicat iar procesul va păstra mapat duplicatul; este nespecificat dacă, în cazul file mapping, modificările făcute asupra fișierului ('write()') după mapare sunt vizibile în zona din spațiul virtual de adrese mapată;

în plus, în 'flags' se mai pot adăuga prin OR pe biți ('|') zero sau mai multe dintre următoarele valori:

MAP_ANONYMOUS sau MAP_ANON (învechit): anonymous mapping - paginile nu sunt swapate out în vreun fișier; conținutul paginilor este inițializat cu 0; argumentul 'fd' este ignorat (unele implementări cer să fie -1, a.î. aceasta este variabila portabilă); argumentul 'offset' trebuie să fie 0; în absența acestui flag, se face file mapping;

`MAP_FIXED` : nu este intrepretat 'addr' ca o "sugestie" - maparea este efectuată exact de la adresa respectivă; 'addr' trebuie să fie o adresă aliniată la un multiplu al dimensiunii paginii (pot exista și alte restricții); dacă zona virtuală specificată de 'addr' și 'offset' se suprapune cu regiuni din alte zone virtuale deja mapate, paginile asociate regiunilor respective din mapările vechi sunt înlăturate; dacă 'addr' specificat nu poate fi folosit, 'mmap()' eșuează;

`MAP_FIXED_NOREPLACE` (începând cu Linux 4.17) : similar cu `MAP_FIXED`, dar nu afectează zonele mapate existente - în caz de suprapunere, 'mmap()' eșuează; se poate folosi pentru a efectua încercările de mapare de către threaduri în mod atomic - dacă mai multe thread-uri din același proces încearcă simultan maparea unei zone comune, un thread va reuși, celelalte vor eșua;

'fd' : descriptorul fișierului mapat, în cazul file mapping (în cazul `MAP_ANONYMOUS`, argumentul 'fd' este ignorat);

'offset' : deplasamentul față de începutul fișierului indicat de 'fd' a zonei de conținut mapate (de lungime 'length' octeți); 'offset' trebuie să fie un multiplu al dimensiunii paginii.

În cazul file mapping, după revenirea din apelul 'mmap()' descriptorul 'fd' poate fi închis imediat, fără a invalida maparea. Un fișier poate fi mapat în multipli ai dimensiunii paginii; dacă fișierul nu are o asemenea dimensiune, se alocă pagini suplimentare umplute cu 0 iar modificările în zona nu sunt scrise în fișier. Efectul redimensionării fișierului asupra paginilor care corespund zonelor adăugate sau eliminate din fișier este nespecificat. La scrierea într-o pagină mapată 'MAP_SHARED' dintr-un fișier, se va modifica momentul (timestamps) ultimei modificări reținut pentru acesta.

Funcția 'munmap()' demapează paginile asociate domeniului specificat de adrese virtuale; 'addr' trebuie să fie un multiplu al dimensiunii paginii, dar 'length' nu este necesar să fie - toate paginile care conțin o parte din domeniul specificat vor fi demapate; accesarea ulterioră a memoriei virtuale în domeniul respectiv de adrese va cauza segmentation fault. Nu este o eroare dacă specificăm un domeniu care nu are asociate pagini.

Zonele mapate se demapează automat la terminarea procesului.

Funcția 'mmap()' returnează în caz de succes adresa zonei mapate iar în caz de eșec valoarea 'MAP_FAILED' (i.e. (void *) -1) și setează 'errno'.

Funcția 'munmap()' returnează în caz de succes 0 iar în caz de eșec -1 și setează 'errno'.

Observații:

- Procesele copil obținute cu 'fork()' moștenesc mapările efectuate cu 'mmap()' ale părintelui, cu aceleași atribut (a se vedea mai jos).
 - În cazul unei mapări anonime, nu avem un nume de fișier cu ajutorul căruia să comunicăm mai multor procese să asocieze în comun paginile respective, dar maparea se va moșteni proceselor copil obținute cu 'fork()'; în acest caz, dacă maparea este partajată, părintele și copilul vor avea asociate aceleași pagini și vor putea comunica prin ele pe calea memoriei, iar dacă maparea este privată, copilul va avea asociate inițial paginile părintelui dar la prima scriere pe care o va face în vreuna din ele, pagina se va duplica iar copilul va păstra copia.
 - Funcția de bibliotecă 'malloc()' alocă la cere din partea codului utilizator apelant memorie din spațiul deja rezervat pentru heap, ajustând la nevoie dimensiunea heap-ului cu 'sbrk()'; dacă sunt cerute spre alocare blocuri de memorie mai mari decât 'MMAP_THRESHOLD' octeți (valoare ajustabilă cu 'mallopt()', de obicei este 128k), implementarea glibc a funcției 'malloc()' alocă memoria ca o mapare anonimă privată folosind 'mmap()'.
- În Linux, se poate mări / mișora o zonă mapată, cu eventuală mutare în alt domeniu de adrese virtuale, folosind apelul 'mremap()' (specific Linux); pe baza acestui apel se poate implementa eficient funcția de bibliotecă 'realloc()'.

```
#include <sys/mman.h>
int msync(void *addr, size_t length, int flags);
```

Sincronizează un fișier cu conținutul său mapat în memorie cu 'mmap()' (evacuează (flush) modificările efectuate în zonele sale copiate în memorie înapoi în sistemul de fișiere). În absența acestui apel, nu se poate garanta că modificările au fost salvate înainte de demaparea cu 'munmap()'.

Este actualizată doar partea din fișier care corespunde zonei de memorie virtuală care începe de la adresa 'addr' și are lungime 'length'.

Argumentul 'flags' trebuie să specifice prin OR pe biți ('|') exact una dintre valorile 'MS_ASYNC' sau 'MS_SYNC' și poate specifica în plus și valoarea 'MS_INVALIDATE', având următoarele semnificații:

MS_ASYNC : este planificată o actualizare, dar apelul returnează imediat;

MS_SYNC : cere o actualizare și așteaptă finalizarea ei;

MS_INVALIDATE : cere invalidarea altor mapări ale aceluiași fișier (a.î. ele pot fi actualizate cu valorile noi care tocmai au fost scrise).

Funcția 'msync()' returnează 0 în caz de succes sau -1 (și setează 'errno') în caz de eșec.

Exemplu: comunicare între două procese la nivelul memoriei, printr-un fișier mapat în memorie partajat:

```
$cat expeditor.c
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    int d; char c; char *buf;
    if(argc != 2)
        {fprintf(stderr, "Utilizare: %s fisier\n", argv[1]); return 1;}
    if((d = open(argv[1], O_WRONLY | O_CREAT | O_TRUNC, S_IWUSR)) == -1)
        {perror(argv[1]); return 1;}
    write(d, "01", 2 * sizeof(char));
    close(d);
    if((d = open(argv[1], O_RDWR)) == -1)
        {perror(argv[1]); return 1;}
    if((buf = mmap(NULL, 2 * sizeof(char),
                  PROT_READ | PROT_WRITE, MAP_SHARED, d, 0)) == MAP_FAILED) {
        perror("mmap"); close(d); return 1;
    }
    while(read(0, &c, sizeof(char)) == sizeof(char)) {
        while(buf[1] != '1'); /* asteptare ocupata */
        buf[0] = c; buf[1] = '0'; /* conteaza ordinea atribuirilor */
    }
    while(buf[1] != '1');
    buf[1] = '2';
    return 0;
}
```

```
$cat destinatar.c
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    int d; char *buf;
    if(argc != 2)
        {fprintf(stderr, "Utilizare: %s fisier\n", argv[1]); return 1;}
    if((d = open(argv[1], O_RDWR)) == -1)
        {perror(argv[1]); return 1;}
    if((buf = mmap(NULL, 2 * sizeof(char),
                  PROT_READ | PROT_WRITE, MAP_SHARED, d, 0)) == MAP_FAILED) {
        perror("mmap"); close(d); return 1;
    }
    do{
        while(buf[1] != '0' && buf[1] != '2');
        if(buf[1] == '0') {write(1, buf, sizeof(char)); buf[1] = '1';}
        else break;
    }while(1);
    return 0;
}

$gcc -Wall -o expeditor expeditor.c
$gcc -Wall -o destinatar destinatar.c
$cat date.txt
abc
$(./expeditor fis.txt < date.txt &) ; (./destinatar fis.txt &)
$abc

$
```

Comentarii:

- Programele 'expeditor' și 'destinatar' comunică pe calea memoriei printr-un fișier mapat partajat de ambele; 'expeditor' citeste de la standard input și scrie în zona partajată 'buf[0]' iar 'destinatar' citeste din zona partajată 'buf[0]' și scrie la standard output.
- 'buf[1]' este folosit pentru sincronizare și evitarea condițiilor de cursă: fiecare program, după accesarea lui 'buf[0]' îi dă voie celuilalt să acceseze 'buf[0]' setând adekvat 'buf[1]'; 'expeditor' scrie în 'buf[0]' când găsește 'buf[1] == '1'' iar 'destinatar' citește din 'buf[0]' când găsește 'buf[1] == '0"'; 'expeditor' anunță pe 'destinatar' că a terminat de scris toate informațiile setând 'buf[1] == '2'' (ca 'destinatar' să nu mai aștepte degeaba).
- 'expeditor' crează fișierul partajat și scrie în el "01", ca să asigure ca va accesa zona partajată primul; funcționarea corectă tot nu este garantată, deoarece 'destinatar' ar putea mapa și accesa fișierul înainte ca 'expeditor' să scrie "01".
- comanda
`'./expeditor fis.txt < date.txt & ; ./destinatar fis.txt &'` a lansat 'expeditor' și 'destinatar' pe rând (operatorul ';') dar, pentru că aceștia au fost lansați cu '&', shell-ul nu așteptat terminarea primului ca să-l lanseze pe al doilea și astfel cele două procese au rulat în paralel.

TODO: segmente de memorie partajată (shm).

Un proces are asociate (printre altele):

- PID (process ID): număr întreg prin care procesul este identificat în mod unic la nivelul instanței SO din care face parte.
- PPID (parent process ID): PID-ul procesului părinte.

Un proces poate crea alte procese (cu apelurile 'fork()', 'exec()', vom vedea) iar între cele două procese se reține o legătură părinte - copil. Se formează astfel o ierarhie de procese, care în Unix/Linux este un arbore.

O instanță Unix/Linux are un arbore unic de procese, în care fiecare proces are un PID unic. În sistem, există un singur proces care este creat automat la boot-are și care are $\text{PID} = 0$, orice alt proces este creat de un proces existent (cu apelurile 'fork()', 'exec()') și are un părinte.

- proprietarul real (real user ID, RUID sau UID) și grupul real (real group owner, RGID sau GID): descriu posesiunea (utilizatorul / grupul utilizatorului care a lansat procesul).
- proprietarul efectiv (effective user ID, EUID) și grupul efectiv (effective group owner, EGID): definește drepturile de acces (access permissions) la resursele sistemului; de exemplu, un apel 'open()' va ține cont de drepturile proprietarului / grupului efectiv al procesului asupra fișierului vizat.

De obicei, proprietarul / grupul efectiv coincid cu cele reale (moștenite de la procesul părinte) dar uneori ele sunt setate (de 'exec()') la proprietarul / grupul fișierului executat (de obicei 'root') - aceasta permite unor utilizatori neprivilegiați să lanseze anumite comenzi ca proceze privilegiate (de exemplu, comanda 'passwd', cu care își poate schimba propria parolă și care scrie în fișierele protejate '/etc/passwd', '/etc/shadow').

- Saved UID, Saved GID: valori salvate pentru proprietar și grup.

Uneori, un proces cu privilegii (ex. 'root') are nevoie să efectueze niște acțiuni neprivilegiate și, ca atare, își schimbă proprietarul efectiv într-unul neprivilegat; vechiul EUID este salvat în saved UID iar procesul poate reveni ulterior la proprietarul privilegiat (funcțiile 'seteuid()', 'setegid()' - vom vedea).

- directorul curent: este o caracteristică per proces, folosită la parcurgerea căilor relative (am văzut în secțiunea 'Fișiere și directoare').
- terminalul de control (tty): în general, orice proces are un terminal de control; poate fi accesat prin descriptori, ca orice fișier; pot fi create și procese fără terminale de control (ex. procesele daemon), dar care ulterior să achiziționeze unul.

Procesele shell interactive au, de obicei, descriptorii standard 0, 1, 2 pe terminalul de control.

Din program, terminalul de control poate fi specificat generic prin '/dev/tty'.

- environmentul: listă de variabile asignate la valori string; se moștenește la 'fork()', uneori la 'exec()', și este un instrument de comunicare între procese la run time, pe relația părinte - copil.

Variabilele de environment nu sunt cunoscute la momentul scrierii programului sursă ci sunt stabilite la rularea ca proces, prin moștenire de la părinte; dacă un același program (fișier) este lansat ca proces copil de mai multe ori, de către procese diferite, poate moșteni environment-uri diferite.

Prin environment, părintele poate transmite copilului informații despre setările sistemului, a.î. acesta să nu mai fie nevoie să cerceteze aceste setări.

- tabela de descriptori de fișiere: cu aceasta, procesul gestionează fișierele asupra cărora operează (am văzut în secțiunea 'Fișiere și directoare'); în această tabelă, intrările (descriptorii) 0, 1, 2 sunt standard input, standard output, standard error.
- informații despre zona de memorie accesată; în Linux, spațiul de adrese de memorie al procesului este descris cu ajutorul unei structuri de date de tip 'struct mm_struct', definită în <linux/sched.h>.
- tabela de pagini: folosită de sistem în mecanismul adresării protejate paginate a memoriei; acesta permite ca, dacă un același fișier (program) este executat de mai multe proceze simultan, o aceeași variabilă din program să aibă locații diferite și valori curente independente în proceze diferite.

Când procesorul execută o instrucțiune în contextul unui proces iar aceasta specifică o adresă (adresă virtuală), procesorul o mapează pe cale hardware într-o altă adresă (adresă fizică) pe baza tabelei de pagini a procesului respectiv (procesorul are un registru cu care pointează tabela curentă, registrul 'CR3' în arhitectura Intel) și folosește noua adresă pentru a accesa memoria.

SO gestionează tabelele de pagini ale proceselor și le setează a.î. să pună în corespondență spațiile de adrese virtuale ale programelor executate de proceze diferite cu spații de adrese fizice disjuncte. Se pot implementa și excepții, de exemplu folosind segmente de memorie partajate.

În Linux, tabela de pagini a procesului este indicată de membrul 'pgd' al structurii de tip 'struct mm_struct'.

- tabela de gestiune a semnalelor primite și tabela de handle-uri asociate semnalelor (vom vedea în secțiunea "Semnale").
- umask: masca de drepturi folosită la crearea fișierelor (file mode creation mask) (am văzut în secțiunea 'Fișiere și directoare').
- prioritatea (Nice number): număr întreg care descrie prioritatea față de alte procese (prioritatea este calculată din numerele 'nice' și utilizarea recentă a CPU).
- starea procesului: running, ready, etc.
- timpul procesor consumat.
- regiștrii (dacă nu este running)
- informații despre fișierul executat de proces, linia de comandă executată.

- PCB (process control block, bloc control proces): structură de date utilizată de SO pentru a gestiona procesul - ea conține informații despre proces.

SO are o tabelă de procese (process table), menținută permanent în RAM, care ține evidența proceselor curente; PCB unui proces este intrarea sa în tabela de procese.

Întrucât tabela de procese este menținută tot timpul în RAM, care este limitat, în intrările sale sunt stocate doar informațiile de care este nevoie tot timpul, chiar și când procesele respective nu rulează (sunt ready, sleeping, etc.), de exemplu, prioritatea - pentru ca SO să știe în permanență pe care proces poate comuta (context switch).

Informațiile legate de un proces de care este nevoie doar când procesul rulează (running), pot fi stocate în altă parte în RAM, în pagini de imagine care, la nevoie, pot fi swap-ate in / out pe disc (de exemplu, tabela descriptorilor de fișiere).

Putem consulta / modifica câteva caracteristici ale procesului curent folosind următoarele funcții:

```
#include <unistd.h>
#include <sys/types.h>
uid_t getuid(void);
uid_t geteuid(void);
```

Funcția 'getuid()' returnează proprietarul real (real user ID, RUID, UID) iar 'geteuid()' returnează proprietarul efectiv (effective user ID, EUID), ai procesului apelant; ele au mereu succes.

```
#include <unistd.h>
#include <sys/types.h>
gid_t getgid(void);
gid_t getegid(void);
```

Funcția 'getgid()' returnează grupul real (real group ID, RGID, GID) iar 'getegid()' returnează grupul efectiv (effective group ID, EGID), ale procesului apelant; ele au mereu succes.

```
#include <sys/types.h>
#include <unistd.h>
int setuid(uid_t uid);
int setgid(gid_t gid);
```

Funcția 'setuid()' setează proprietarul efectiv (EUID) pentru procesul apelant; dacă procesul este privilegiat, setează și UID, sau saved set-user-ID.

Un proces neprivilegiat își poate schimba EUID doar în UID sau saved set-user-ID.

Un proces privilegiat poate folosi orice 'uid' și își va schimba în 'uid' toți proprietarii: UID, EUID, sau saved set-user-ID.

Funcția 'setgid()' operează similar, cu grupurile.

Funcțiile returnează 0 în caz de succes sau -1 (și setează 'errno') în caz de eșec.

```
#include <sys/types.h>
#include <unistd.h>
int seteuid(uid_t euid);
int setegid(gid_t egid);
```

Funcția 'seteuid()' setează proprietarul efectiv (EUID) pentru procesul apelant. Un proces neprivilegiat își poate schimba EUID doar în UID, EUID sau saved set-user-ID.

Un proces privilegiat își poate schimba EUID în orice user; în versiunile glibc recente (2.1) nu se va modifica saved set-user-ID, ca în cazul funcției 'setuid()'.

Funcția 'setegid()' operează similar, cu grupurile.

Funcțiile returnează 0 în caz de succes sau -1 (și setează 'errno') în caz de eșec.

Observație:

Funcțiile 'setuid()', 'seteuid()' permit unui proces să renunțe temporar la privilegiile deținute (schimbând EUID la UID), să efectueze niște activități neprivilegiate, apoi să revină la privilegiile anterioare (schimbând EUID la saved set-user-ID).

Dacă însă 'root' folosește 'setuid()', își va schimba toți proprietarii în cel indicat de argumentul 'uid' și nu va putea reveni la 'root'; el ar trebui să folosească 'seteuid()' (va putea păstra 'root' în saved set-user-ID).

```
#include <unistd.h>
int nice(int inc);
```

Modifică prioritatea procesului curent, adunând incrementul 'inc' la valoarea atributului 'nice'; o valoare 'nice' ridicată înseamnă o prioritate mare.

Intervalul valorilor 'nice' este de la +19 (low priority) la -20 (high priority); încercările de a seta valori în afara intervalului sunt ajustate la interval.

Tradițional, doar procesele privilegiate puteau coborî valoarea 'nice' (i.e., setând o prioritate mai mare). De la Linux 2.6.12, un proces neprivilegiat poate coborî valoarea 'nice' până la o anumită limită (limita soft 'RLIMIT_NICE', se poate consulta cu 'getrlimit()').

Funcția returnează noua valoare 'nice' în caz de succes sau -1 (și setează 'errno') în caz de eșec.

Întrucât un apel cu succes poate returna -1, pentru a detecta o eroare se poate seta 'errno' cu 0 înainte de apel și verifica dacă este nonzero după ce apelul returnează -1.

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
char *get_current_dir_name(void);
```

Funcțiile furnizează un string terminat cu caracterul nul ('\'0') care conține specificarea cu cale absolută a directorului curent al procesului apelant.

Funcția 'getcwd()' copiază specificatorul directorului curent în zona pointată de 'buf', fără a depăși dimensiunea 'size'.

Dacă lungimea specificatorului, inclusiv terminatorul nul, excede 'size' octeți, funcția 'getcwd()' returnează NULL și setează 'errno'; aplicația trebuie să verifice această eroare și să aloce, eventual, o zonă mai mare.

Funcția 'getcwd()' returnează adresa 'buf' a zonei indicate de utilizator (și care va conține specificatorul directorului curent) în caz de succes sau NULL (și setează 'errno') în caz de eșec; în caz de eșec, conținutul zonei indicate de 'buf' este nedefinit.

Funcția 'get_current_dir_name()' aloca cu 'malloc()' o zonă suficient de mare și copiază în ea specificatorul directorului curent; dacă variabila de environment 'PWD' este setată la o valoare corectă, este copiată această valoare. Zona alocată trebuie dezalocată ulterior cu 'free()'.

Funcția 'get_current_dir_name()' returnează adresa zonei alocate dinamic (și care va conține specificatorul directorului curent) în caz de succes sau NULL (și setează 'errno') în caz de eșec.

Obs: Constanta 'PATH_MAX', care poate fi găsită în <limits.h> sau <linux/limits.h> oferă o limită maximă pentru lungimea unui specificator cale/nume, dar valoarea respectivă nu e sigură.

```
#include <unistd.h>
int chdir(const char *path);
int fchdir(int fd);
```

Schimbă directorul curent al procesului apelant în cel indicat de 'path' (specificator cale/nume), respectiv de 'fd' (descriptor de fișiere deschis).

Funcțiile returnează 0 în caz de succes sau -1 (și setează 'errno') în caz de eșec.

```
#include <stdlib.h>
char *getenv(const char *name);
```

Returnează adresa de început a strigului - valoare a variabilei de environment cu numele 'name'; dacă numele nu corespunde nici unei variabile de environment, returnează NULL.

```
#include <stdlib.h>
int putenv(char *string);
```

Stringul 'string' trebuie să fie de forma 'name=value'.

Funcția schimbă valoarea variabilei de environment 'name' la 'value', dacă variabila 'name' există, sau adaugă această variabilă cu această valoare, dacă nu există.

Stringul pointat de 'string' devine parte a environment-ului, deci alterarea sa ulterioară modifică environmentul.

Funcția 'putenv()' returnează 0 în caz de succes sau nonzero (și setează 'errno') în caz de eșec.

```
#include <stdlib.h>
int setenv(const char *name, const char *value, int overwrite);
int unsetenv(const char *name);
```

Funcția 'setenv()' adaugă sau modifică o variabilă de environment:

- dacă nu există variabila cu numele 'name', adaugă această variabilă, cu numele 'name';
- dacă există variabila cu numele 'name' și 'overwrite' este nonzero, schimbă valoarea variabilei în 'value';
- dacă există variabila cu numele 'name' și 'overwrite' este 0, nu schimbă valoarea variabilei și returnează succes;

Funcția 'setenv()' face copii ale stringurilor indicate de 'name' și 'value', în contrast cu 'putenv()'.

Funcția 'unsetenv()' elimină (delete) variabila 'name' din environment.

Dacă 'name' nu există în environment, 'unsetenv()' returnează succes și environmentul rămâne neschimbat.

Ambele funcții 'setenv()' și 'unsetenv()' returnează 0 în caz de succes sau -1 (și setează 'errno') în caz de eșec.

```
#include <stdlib.h>
int clearenv(void);
```

Șterge (clear) environmentul procesului apelant și setează valoarea variabilei 'environ' (a se vedea mai jos) la NULL.

Ulterior, se pot adăuga noi variabile cu 'putenv()' și 'setenv()'.

Funcția 'clearenv()' returnează 0 în caz de succes sau nonzero în caz de eșec.

Environmentul procesului curent poate fi consultat și folosind variabila globală predefinită 'environ'; pentru aceasta, trebuie să includem declarația:

```
extern char **environ;
```

Variabila pointează un vector (array) conținând adresele de început ale sirurilor de environment, de forma 'name=value', și o componentă NULL la sfârșit.

Deci, codul următor afișază environmentul procesului curent:

```
extern char **environ;
...
char **p;
for(p = environ; *p; ++p) printf("%s\n", *p);
```

Obs: putem să parcurgem environmentul chiar cu 'environ' (nu este 'const') dar vom pierde începutul listei și nu vom putea să o parcurgem din nou, dacă dorim.

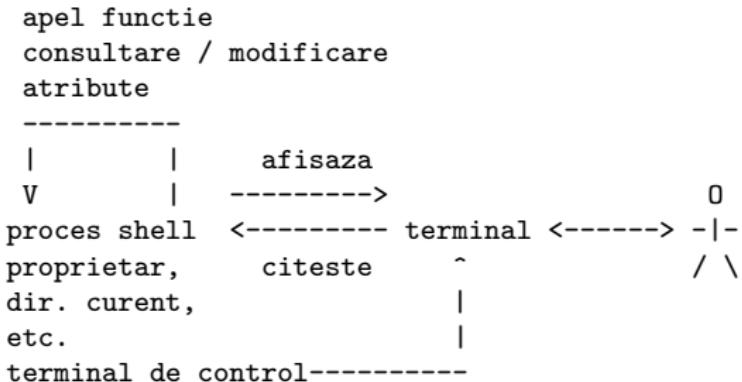
Un proces își poate consulta environmentul și folosind un al treilea parametru al funcției 'main()', dacă este definit astfel:

```
int main(int argc, char *argv[], char *envp[])
```

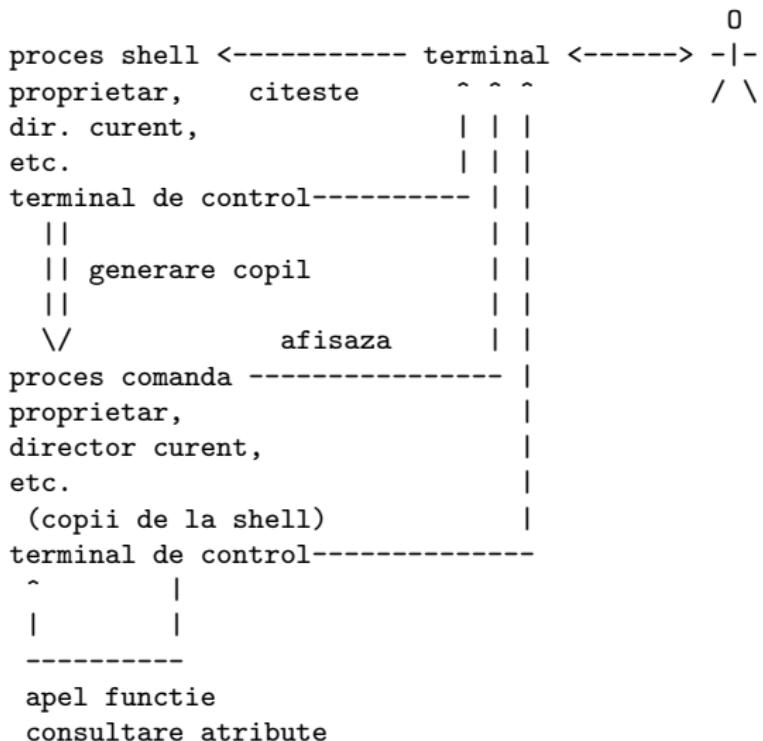
Parametrul 'envp' va indica un vector (array) conținând adresele de început ale șirurilor de environment, de forma 'name=value', și o componentă NULL la sfârșit.

Folosirea celui de-al treilea parametru al lui 'main()' nu este însă specificată în POSIX.1; conform POSIX.1, environmentul ar trebui accesat prin variabila 'environ'.

Procesul shell își poate consulta și afișa atributele sale ca proces (executând apeluri ca mai sus), ca urmare a unor comenzi date de utilizator.



Comenzile care doar consultă și afișază pot fi implementate atât ca interne cât și ca externe. Într-adevăr, procesul copil va moșteni de la shell terminalul de control și copii ale atributelor de la shell-ul părinte și, când își vor afișa atributele lor pe terminalul lor, vor fi afișate atributele shell-ului pe terminalul shell-ului - utilizatorul nu poate distinge de efectul unei comenzi interne.



Comenzile care modifică trebuie să fie neapărat interne - procesele copil, chiar dacă moștenesc copii ale atributelor shell-ului părinte, rulează independent de acesta și când își vor modifica attributele lor, nu se vor modifica attributele shell-ului.

Câteva comenzi shell cu care acesta își poate consulta și afișa / modifica attributele (lansate doar cu argumentele indicate):

id - afișază UID, GID, grupurile lui UID

whoami - afișază numele proprietarului efectiv (EUID)

nice - afișază prioritatea (valoarea 'nice')

pwd - afișază directorul curent

cd director - schimbă directorul curent în 'director'

tty - afișază terminalul la care este conectat standard input (de obicei, este terminalul de control)

env - afișază environmentul

Comanda 'cd director' este internă, restul sunt externe (fișierele executabile se află în '/bin').

Exemplu: implementarea comenzii 'whoami':

```
$cat mywhoami.c
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
#include <stdlib.h>
#include <stdio.h>

int main(){
    uid_t euid; struct passwd *pw;
    euid = geteuid();
    if((pw = getpwuid(euid)) == NULL){
        perror("getpwuid");
        exit(1);
    };
    printf("%s\n", pw->pw_name);
    return 0;
}

$gcc -o mywhoami mywhoami.c
$./mywhoami
dragulici
$whoami
dragulici
```

Exemplu: implementarea comenzii 'pwd':

```
$cat mypwd.c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <linux/limits.h>

int main(){
    char *buf, *buf_nou; size_t size, size_old;
    size = 0; buf = NULL;
    do{
        size_old = size;
        size += PATH_MAX;
        if(size <= size_old) {
            fprintf(stderr, "Cale prea lunga\n");
            free(buf);
            exit(1);
        }
        if((buf_nou = realloc(buf, size)) == NULL) {
            perror("realloc");
            free(buf);
            exit(1);
        }
        buf = buf_nou;
    }while(getcwd(buf, size) == NULL);
    printf("%s\n", buf);
    free(buf);
    return 0;
}
```

```
$gcc -o mypwd mypwd.c  
$./mypwd  
/home/dragulici/Desktop/work  
$pwd  
/home/dragulici/Desktop/work
```

Comentarii:

- Am realocat buffer-ul până a încăput specificatorul directorului curent.
- La fiecare iterație, dimensiunea este incrementată cu PATH_MAX; astfel, crește şansa de a reuşi de la prima iterație.
- Creșterea valorii 'size' poate conduce la depășirea limitelor tipului 'size_t' și alterarea valorii; eroarea este detectată constatănd că noua valoare este \leq valoarea veche.
- Noua adresă rezultată după reallocare nu este recuperată direct în 'buf', deoarece în caz de eșec 'realloc()' nu dezalocă zona veche și returnează NULL, a.î. zona veche ar rămâne alocată și inaccesibilă pentru a o dezaloca ulterior cu 'free()' (memory leak).

Exemplu: implementarea comenzii 'env':

```
$cat myenv.c
#include<stdio.h>
extern char **environ;
int main(){
    char **p;
    for(p = environ; *p; ++p)
        printf("%s\n", *p);
    return 0;
}
```

```
$gcc -o myenv myenv.c
$./myenv > rez1.txt
$env > rez2.txt
$diff rez1.txt rez2.txt
60c60
< _=./myenv
---
> _=/usr/bin/env
```

Comentarii:

Întrucât environmentul shell-ului este mare, pentru testare am preferat să nu-l afișăm pe terminal ci să-l scriem într-un fișier, iar fișierele rezultate cu '`./myenv`' și '`env`' să fie comparate cu '`diff`'.

Diferența afișată de '`diff`' se explică prin faptul că shell-ul are o variabilă de environment '`$_`' pe care o initializează automat cu specificatorul ultimului program executat - în primul caz, a fost '`./myenv`', în al doilea caz a fost '`/bin/env`'.

Pentru a crea noi procese, se folosesc apelurile 'fork()' și 'exec()', iar pentru a aștepta schimbarea stării (de exemplu, terminarea) unui proces copil, se folosesc apelurile 'wait()' și 'waitpid()':

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

Procesul se duplică - apare un nou proces, cu alt PCB și alt PID (unice în instanța SO); noul proces este copil al primului; cele două procese vor rula înapoiîn paralel, independent unul de altul.

Copilul execută același fișier (program) ca părintele, punctul de plecare fiind ieșirea din apelul 'fork()'; copilul moșteneste o copie a datelor părintelui, cu valorile din momentul duplicării, și copii ale informațiilor de control: proprietarul real și cel efectiv, grupul real și cel efectiv, prioritatea, directorul curent, terminalul de control, environmentul, tabela de descriptori de fișiere, sesiunea, grupul.

Chiar dacă execută același program ca părintele, copilul și părintele au spații separate de memorie și propriile lor locații și valori curente pentru variabilele din program; în momentul duplicării, variabilele aveau aceleași valori în cele două procese, dar ulterior aceste valori vor evolua independent.

În Linux, 'fork()' este implementat folosind pagini copy-on-write, a.î. doar se duplică tabela de pagini a părintelui și se alocă un task structure unic pentru copil. Copilul va duplica efectiv doar paginile părintelui în care scrie, atunci când va scrie.

Scrierile în memorie și mapările / demapările de fișiere ('mmap()', 'munmap()') efectuate ulterior apelului 'fork()' de către unul dintre cele două procese nu îl vor afecta pe celălalt.

Copilul nu moștenește mulțimea semnalelor în pending (la copil este inițial vidă) ('sigpending()') și alarma ('alarm()') dar moștenește mulțimea semnalelor blocate ('sigprocmask()') (a se vedea în secțiunea 'Semnale').

Funcția 'fork()' returnează în caz de succes:

în părinte: PID-ul fiului copilului (> 0)

în copil: 0

În caz de eșec, procesul inițial nu se duplică iar 'fork()' returnează (în el) -1 și setează 'errno' (o cauză: prea multe procese în sistem pentru utilizatorul respectiv).

Așadar, dacă în programul pe care îl rulează cele două procese 'fork()' apare la condiția unui 'if', 'while', 'switch', etc., părintele va continua pe o ramură iar copilul pe alta.

```
if(fork()) {  
    /* parinte */  
} else {  
    /* copil */  
}
```

sau:

```
pid_t p;  
switch(p = fork()) {  
    case -1: perror("fork()"); break;  
    case 0: /* copil */ break;  
    default: /* parinte, are PID-ul copilului in p */  
}
```

Deoarece copilul moștenește o copie a tabelei de descriptori de fișiere a părintelui, un același descriptor deschis în părinte înainte de 'fork()' va fi deschis și în copil, indicând către aceeași intrare TDF și utilizând același mod de deschidere și același indicator de poziție curentă.

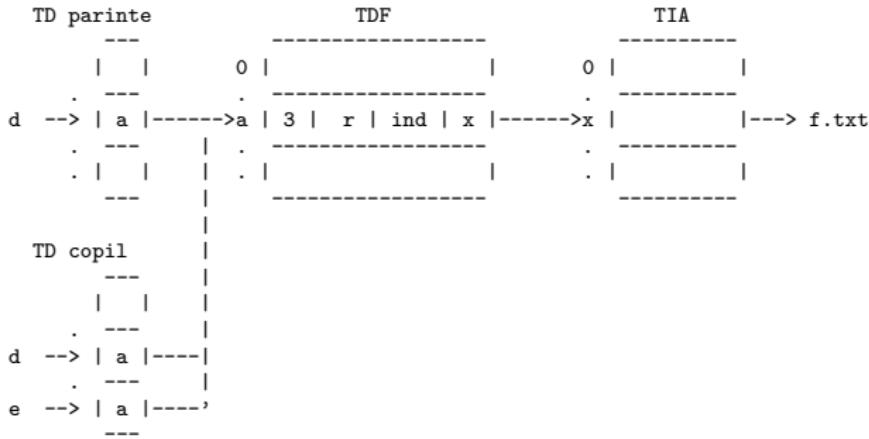
De asemenea, copilul moștenește aceeași mulțime de directoare deschise, gestionate cu copii ale structurilor 'DIR' din părinte, și este posibil să nu utilizeze același indicator de intrare curentă (POSIX afirmă ca pot partaja aceeași poziție curentă, Linux/glibc afirmă că nu). Într-adevăr, structurile 'DIR' au propriul lor membru care indică intrarea curentă iar părintele și copilul au structuri 'DIR' diferite pentru același director (chiar dacă imediat după 'fork()' structurile 'DIR' aveau conținuturi identice, ulterior ele se pot modifica independent).

Exemplu:

```
$cat prog.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main() {
    int d, e; char buf[256];
    d = open("f.txt", O_RDONLY);
    if(fork()) {
        read(d, buf, 3 * sizeof(char)); buf[3] = 0; printf("P: %s\n", buf);
        sleep(2);
        read(d, buf, 3 * sizeof(char)); buf[3] = 0; printf("P: %s\n", buf);
    } else {
        sleep(1);
        read(d, buf, 3 * sizeof(char)); buf[3] = 0; printf("C: %s\n", buf);
        e = dup(d);
        lseek(e, 3, SEEK_CUR);
    }
    return 0;
}
```

```
$gcc -o prog prog.c
$cat f.txt
abcdefghijklmnopqrstuvwxyz
$./prog
P: abc
C: def
P: jkl
```

În exemplul de mai sus, cele două procese au în total 3 descriptori pe aceeași intrare TDF, utilizând același mod de deschidere (citire) și același indicator de poziție curentă pentru fișierul 'f.txt':



Apelul 'sleep(k)' pune procesul apelant în aşteptare 'k' secunde (vom vedea mai târziu) iar cu asemenea apeluri am mărit şansa ca cele 3 apeluri 'read()' să fie executate în ordinea: părinte, copil, părinte (deși din program nu putem garanta timpii de execuție și ordinea de rulare intercalată a proceselor).

La terminarea proceselor, cei 3 descriptori către intrarea TDF au dispărut iar intrarea TDF a fost eliminată, chiar dacă n-am apelat explicit 'close()'.

Reamintim că la terminarea unui proces copil, procesul părinte primește semnalul SIGCHLD iar copilul intră în starea zombie și pune la dispoziția părintelui codul său de return - părintele îl poate prelua cu apelurile 'wait()', 'waitpid()', care vor și elibera copilul zombie din sistem. Dacă părintele se va termina fără să aplique copilului 'wait()' sau 'waitpid()', orfanul este preluat de procesul 'init', care îl va elibera atunci când se termină.

În general, este greu să "prindem" un proces zombie și să-l vizualizăm - dacă este orfan, este eliberat repede de 'init'. Șansa ar fi dacă părintele său încă rulează și nu a apelat încă 'wait()' sau 'waitpid()'.

Exemplu: producem și vizualizăm un proces zombie:

```
$cat prog.c
#include <sys/types.h>
#include <unistd.h>
int main(){
    if(!fork()) if(fork()) while(1);
    return 0;
}

$gcc -o prog prog.c
$./prog
$ps -o ppid,pid,cmd
  PPID      PID CMD
  4303      4341 bash
  958      10662 ./prog
10662      10663 [prog] <defunct>
  4341      10664 ps -o ppid,pid,cmd
$kill -9 10662
$ps -o ppid,pid,cmd
  PPID      PID CMD
  4303      4341 bash
  4341      10687 ps -o ppid,pid,cmd
```

Atenție că dintr-o eroare de programare se poate intra în 'fork()' infinit, de exemplu:

```
while(fork()) { ... } exit(0);
```

(generează la infinit procese ce se termină și devin imediat zombie, dar nu sunt înfiatați de 'init' ca să-i elimine, deoarece părintele lor original nu se termină).

Într-o asemenea situație, se umple tabela de procese a sistemului și nu se mai pot genera noi procese, inclusiv procese de executare a unei comenzi 'kill' de la un alt terminal, pentru a omorâ din procesele respective - practic, trebuie resetat (reboot) sistemul.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *wstatus, int options);
pid_t wait(int *wstatus);
```

Procesul așteaptă schimbarea stării unui proces copil al său, obținând informații despre copilul a cărui stare s-a schimbat.

Se consideră schimbare de stare: copil terminat (devenit zombie), copil stopat de un semnal, copil trezit de un semnal. În cazul unui copil terminat, se cere sistemului să elibereze resursele asociate copilului și să eliminate copilul din sistem.

Dacă copilul deja și-a schimbat starea, aceste apeluri returnează imediat; altfel, ele blochează procesul apelant până copilul își schimbă starea sau până sunt întrerupte de un semnal (al cărui handler nu a fost instalat cu flagul SA_RESTART al lui 'sigaction()', vom vedea în secțiunea 'Semnale').

Reamintim că dacă părintele unui copil se termină fără să-i fi aplicat copilului 'wait()' sau 'waitpid()', orfanul este adoptat de procesul 'init', iar când se termină și devine zombie (sau era deja zombie când a fost adoptat), este eliminat cu 'wait()' / 'waitpid()' de 'init'.

În continuare, un copil a cărui stare s-a schimbat și care încă nu a fost așteptat cu 'wait()' sau 'waitpid()' este denumit așteptabil (waitable).

Apelul 'waitpid()' blochează procesul apelant până când un copil specificat de argument și-a schimbat starea. Implicit, 'waitpid()' așteaptă doar copii terminați, dar acest comportament este modificabil via argumentul 'options'.

Valoarea lui 'pid' poate fi:

- < -1 înseamnă așteptarea oricărui copil cu GID = abs(pid).
- 1 înseamnă așteptarea oricărui copil
- 0 înseamnă așteptarea oricărui copil cu GID = GID-ul apelantului (părintelui) la momentul apelului
- > 0 înseamnă așteptarea copilului cu PID = pid

Valoarea lui 'options' este o disjuncție pe biți ('|') de zero sau mai multe dintre următoarele constante:

WNOHANG returnează imediat, chiar dacă copilul nu și-a schimbat starea
(nu așteaptă)

WUNTRACED returnează și dacă copilul a fost stopat (nu doar terminat)

WCONTINUED (începând cu Linux 2.6.10) returnează și dacă copilul era
stopat și a fost trezit cu un semnal SIGCONT

Dacă 'wstatus' nu este NULL, funcția stochează în zona de tip 'int' pointată informații despre starea copilului respectiv. Acest întreg poate fi consultat cu următoarele macro-uri (care primesc întregul respectiv ca argument, adică '*wstatus'):

WIFEXITED() returnează true dacă copilul s-a terminat normal (i.e. 'exit()', '_exit()', sau return din 'main()')

WEXITSTATUS() returnează codul de return (exit status) al copilului; acesta constă din cei mai puțin semnificativi 8 biți ai valorii date ca argument lui 'exit()', '_exit()', sau returnată din 'main()'; acest macro este util doar dacă 'WIFEXITED()' a returnat true

WIFSIGNALED() returnează true dacă copilul a fost terminat de un semnal

WTERMSIG() returnează numărul semnalului care a terminat copilul; acest macro este util doar dacă 'WIFSIGNALED()' a returnat true

`WCOREDUMP()` returnează true dacă copilul a produs un core dump (un fișier pe disc conținând imaginea memoriei procesului copil la momentul terminării); de exemplu, dacă copilul a primit un semnal SIGQUIT și l-a tratat cu handler-ul implicit, el s-a terminat și a produs un core dump; acest macro este util doar dacă '`WIFSIGNALED()`' a returnat true;
obs: acest macro nu este specificat în POSIX, nu este disponibil în unele implementări Unix, și de aceea utilizarea lui trebuie inclusă între

```
#ifdef WCOREDUMP ... #endif
```

`WIFSTOPPED()` returnează true dacă copilul a fost stopat de un semnal; este util doar dacă s-a folosit WUNTRACED

`WSTOPSIG()` returnează numărul semnalului care a stopat copilul; acest macro este util doar dacă '`WIFSTOPPED()`' a returnat true

`WIFCONTINUED(wstatus)` (începând cu Linux 2.6.10) returnează true dacă copilul a fost trezit de semnalul SIGCONT

Valoarea returnată de '`waitpid()`' este:

- în caz de succes, returnează PID-ul copilului care și-a schimbat starea;
- dacă a fost specificat 'WNOHANG' și unul sau mai mulți copii specificați de argumentul 'pid' există dar încă nu și-au schimbat starea, returnează 0;
- în caz de eroare, returnează -1 și setează 'errno'.

Apelul 'wait()' blochează procesul apelant până când unul dintre copii săi se termină. Apelul 'wait(&wstatus)' este echivalent cu:

```
waitpid(-1, &wstatus, 0)
```

Valoarea returnată de 'waitpid()' este:

- în caz de succes, returnează PID-ul copilului terminat;
- în caz de eroare, returnează -1 și setează 'errno'.

Observație:

Întrucât codul de return (exit status, valoarea dată ca argument lui 'exit()', '_exit()', sau returnată din 'main()') este preluată de părinte ca valoare pe 8 biți (unsigned char), aceasta trebuie să fie în intervalul 0, ..., 255, altfel va fi preluată alterată.

Se poate furniza ca cod de return orice valoare, dar se obișnuiește să se furnizeze 0 în caz de succes și ≠ 0 în caz de eșec.

Mai portabil, se pot furniza valorile indicate de constantele EXIT_SUCCESS, EXIT_FAILURE, definite în <stdlib.h>.

Apelurile 'wait()' și 'waitpid()' sunt utile când un proces se duplică cu 'fork()' și dorim ca părintele să aștepte terminarea copilului înainte de a continua, nu să ruleze împreună cu el:

```
if(fork()) {
    wait(NULL);
} else {
    /* diverse */
    exit(0);
}
/* aici continua parintele*/
```

Exemplu:

```
$cat prog.c
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main(){
    pid_t p; int s;
    if(!fork()) exit(1);
    if(!fork()) exit(2);
    if(!fork()) exit(3);
    while((p = wait(&s)) != -1)
        if(WIFEXITED(s))
            printf("Copil %d, cod retur %d\n", (int) p, WEXITSTATUS(s));
    return 0;
}

$gcc -o prog prog.c
./prog
Copil 18538, cod retur 1
Copil 18539, cod retur 2
Copil 18540, cod retur 3
```

În urma unui apel 'fork()', copilul moștenește de la părinte terminalul de control și este creat în aceeași sesiune și grup ca părintele.

În particular, dacă părintele se află în (grupul din) foreground la terminalul respectiv, și copilul va fi în (grupul din) foreground la acel terminal - astfel, poate citi și scrie la el și va primi semnalele generate de la terminal (Ctrl-c, Ctrl-\, Ctrl-z).

Dacă părintele se termină fără să fi aplicat copilului 'wait()' sau 'waitpid()' iar acesta încă rulează, se poate muta în background la terminalul respectiv și, de exemplu, nu va putea să scrie pe el dacă terminalul este setat 'stty tostop'.

De aceea, este util ca părintele să aștepte terminarea tuturor copiilor înainte de a se termina și el.

Exemple:

```
$cat prog1.c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(){
    if(fork()) {printf("Parinte\n");}
    else {sleep(1); printf("Copil\n");}
    return 0;
}

$gcc -o prog prog1.c
$stty tostop
$./prog
Parinte
$
```

```
$cat prog2.c
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
int main(){
    if(fork()) {printf("Parinte\n");}
    else {sleep(1); printf("Copil\n");}
    wait(NULL);
    return 0;
}

$gcc -o prog prog2.c
$stty tostop
$./prog
Parinte
Copil
$
```

```
#include <unistd.h>
int execve(const char *pathname, char *const argv[],
            char *const envp[]);
int execl(const char *pathname, const char *arg, ...
          /* (char *) NULL */);
int execlp(const char *file, const char *arg, ...
           /* (char *) NULL */);
int execle(const char *pathname, const char *arg, ...
           /*, (char *) NULL, char *const envp[] */);
int execv(const char *pathname, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
            char *const envp[]);
```

Funcțiile din familia 'exec()' (termen generic pentru a denumi una dintre funcțiile de mai sus) înlocuiesc procesul apelant cu un altul, care îi ia locul în sistem, preluând aceleași PCB, același PID și același părinte.

Noul proces va executa un fișier (program) dat ca argument, de la început și cu date inițiale (nu ca în cazul 'fork()', de la ieșirea din apel și cu datele de acolo). Poate fi și același program, dar va fi executat de la început.

Noul proces preia același PCB, PID și poziție în arborele de procese (același părinte), preluând și majoritatea informațiilor de control: proprietarul real și uneori cel efectiv, grupul real și uneori cel efectiv, prioritatea, directorul curent, terminalul de control, uneori environmentul, parțial tabela de descriptori de fișiere, sesiunea, grupul.

Se pot modifica:

- proprietarul efectiv: dacă bitul SUID al fișierului executat este 1, proprietarul efectiv al noului proces va fi proprietarul fișierului executat, altfel se va păstra proprietarul efectiv al procesului apelant;
- grupul efectiv: dacă bitul SGID al fișierului executat este 1, grupul efectiv al noului proces va fi grupul fișierului executat, altfel se va păstra grupul efectiv al procesului apelant;
- environmentul: dacă este prezent argumentul 'envp', acesta va inițializa environment-ul pentru noul proces, altfel se va păstra environment-ul de la procesul apelant;

- tabela de descriptori de fișiere: se va păstra cea de la procesul apelant, dar anumiți descriptori pot fi setați să se închidă la 'exec()' ('fcntl()' cu 'FD_CLOEXEC');
- nu se păstrează deschiderile de directoare ale procesului apelant ('opendir()');
- nu se păstrează handlerele de ieșire ('atexit()', 'on_exit()');
- nu se păstrează handlerele asociate semnalelor (se revine la handlerele implicite), dar se păstrează multimea semnalelor în pending ('sigpending()'), multimea semnalelor blocate ('sigprocmask()') și alarma ('alarm()') - a se vedea secțiunea 'Semnale'.

Obs: dacă se schimbă EUID, resp. EGID, pe baza biților SUID, resp. SGID, vechile valori EUID, resp. EGID, sunt reținute pentru noul proces în saved set-user-ID, resp. saved set-group-ID (pot fi folosite ulterior de 'setuid()', 'seteuid()', 'setgid()', 'setegid()').

Dintr-un apel reușit al funcțiilor 'exec()' nu există return (procesul apelant nu mai există). În caz de eșec, returnează -1 și setează 'errno'.

Observăm că nu este nevoie să testăm valoarea returnată:

```
if(exec(...) == -1) ...
```

deoarece codul scris după 'exec()' se execută doar dacă 'exec()' a eșuat.

Funcția 'execve()' este apel sistem, celelalte sunt scrise deasupra ei (front-end). Argumentul 'pathname' specifică fișierul (programul) executat de noul proces; poate fi și un script, a se vedea 'man execve'.

Argumentele 'arg...' sau 'argv' vor inițializa parametrii 'argc' și 'argv' ai funcției 'main()' pentru noul proces; de aceea, 'arg...' trebuie să fie o listă scrisă explicit de elemente separate prin virgulă, care sunt pointeri la char (stringuri), și terminată cu o componentă NULL, iar 'argv' trebuie să fie adresa de început a unui array de pointeri la char (stringuri) terminat cu o componentă NULL. Uzual, prima componentă din 'arg...' și 'argv' specifică același string ca 'pathname' (și va inițializa 'argv[0]' pentru funcția 'main()' a noului proces).

Argumentul 'envp' va inițializa environmentul noului proces - deci și parametrul 'envp' al funcției 'main()' și variabila 'environ' pentru acesta; de aceea, 'envp' trebuie să fie adresa de început a unui array de pointeri la char (stringuri) terminat cu o componentă NULL. Uzual, stringurile din array sunt de forma "nume=valoare".

Dacă argumentul 'envp' lipsește, se va transmite environmentul procesului apelant (ca și când am fi transmis ca argument 'environ').

În general, există o limită maximă a dimensiunii memoriei folosite de argumente și environment; ea se poate afla cu 'sysconf(_SC_ARG_MAX)'.

Funcțiile din familia 'exec()' pot fi clasificate după literele de la sfârșitul numelui:

I - execl(), execlp(), execle()

funcții variadice în care stringurile argument sunt date explicit;

castul '(char *) NULL' în expresia ultimului argument este necesar pentru conversia corectă a valorii la intrarea în apel, deoarece funcția, fiind variadică, nu specifică un tip al parametrului formal corespunzător;

v - execv(), execvp(), execvpe()

stringurile argument sunt date printr-un vector (array);

e - execle(), execvpe()

este specificat un environment (prin lipsă, se transferă environmentalul procesului apelant - ca și când argumentul ar fi 'environ');

p - execlp(), execvp(), execvpe()

folosesc variabila de environment 'PATH' a procesului apelant pentru a căuta fișierul executat, dacă specificatorul 'pathname' este un nume (nu conține '/'); practic, se reproduce acțiunea shell-ului; dacă 'pathname' conține '/', variabila 'PATH' este ignorată.

Cele de mai sus pot fi sintetizate spunând că 'fork()' duplică imaginea procesului apelant și asociază alte structuri de control iar 'exec()' înlocuiește imaginea procesului apelant, păstrând structurile de control.

Astfel, din combinații 'fork()' și 'exec()', arborele de procese al instanței SO se extinde, cu procese diverse. Schema uzuală de lucru este:

```
if(fork()) {  
    wait(NULL);  
} else {  
    execve(program, argumente, environment);  
    perror("exec()");  
    exit(EXIT_FAILURE);  
}
```

Exemplu: program care primește ca argumente în linia de comandă specificatorul unui program (fișier executabil), o listă de argumente în linia de comandă pentru el, precedată de opțiunea '-a', și o listă de stringuri de environment pentru el, precedată de opțiunea '-e' (dacă lipsesc, se transmite environmentul apelantului):

```
$cat prog.c
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
extern char **environ;
int main(int argc, char* argv[]) {
    int i; char **starta, **starte;
    starta = argv; starte = environ;
    for(i = 1; i < argc; ++i)
        if(!strcmp(argv[i], "-a"))
            {starta = argv + (i + 1); argv[i] = NULL;}
        else if(!strcmp(argv[i], "-e"))
            {starte = argv + (i + 1); argv[i] = NULL;}
    execve(argv[1], starta, starte);
    perror("execve()");
    return EXIT_FAILURE;
}
```

```
$gcc -o prog prog.c
$./prog /bin/env -e abc=xy d=123
abc=xy
d=123
```

```
$cat progaux.c
#include <stdio.h>
extern char **environ;
int main(int argc, char *argv[]) {
    char **p;
    printf("Program:\n %s\n", argv[0]);
    printf("Argumente:\n");
    for(p = argv + 1; *p; ++p) printf(" %s\n", *p);
    printf("Environment:\n");
    for(p = environ; *p; ++p) printf(" %s\n", *p);
    return 0;
}
```

```
$gcc -o progaux progaux.c
```

```
$. ./prog ./progaux -a Ana are mere -e abc=xy d=e=f ggg
```

Program:

Ana

Argumente:

are

mere

Environment:

abc=xy

d=e=f

ggg

Observație: Ultimul test ne arată că nu este necesar ca primul argument să fie numele fișierului executat (nu a fost necesar './progaux', a funcționat și cu 'Ana') și nu este necesar ca șirurile de environment să fie de forma 'nume=valoare' (a funcționat și cu 'd=e=f' și cu 'ggg'). Acestea sunt doar convenții uzuale.

O implementare tipică a conceptelor și mecanismelor referitoare la procese este programul shell.

În mod uzual, el are un terminal de control pe care îl gestionează în mod text, are descriptorii standard 0, 1, 2 pe acest terminal și, într-un ciclu infinit, la fiecare iterare, afișază un prompter, citește o linie de comandă și o execută. Prompterul este afișat pe standard output ('puts()', 'printf()', etc.), linia de comandă este citită de la standard input ('gets()', 'scanf()', etc.) iar mesajele de eroare sunt afișate la standard error (' perror()') - toate acestea fiind pe terminal.

Unele comenzi shell sunt interne - le execută shell-ul prin propriile sale instrucțiuni, altele sunt externe - sunt specificatorii unor fișiere executabile pe care shell-ul le lansează ca procese copil ('fork()' și 'exec()'), iar acestea execută sarcina.

Procesele copil care execută comenzi externe moștenesc de la shell terminalul de control și, în absența unor specificări speciale în linia de comandă (redirectări '<', '>', '>>', '>2>', '>2>>', filtre '|', etc.), moștenesc descriptorii 0, 1, 2 pe acest terminal, a.î. vor executa 'scanf()', 'printf()', 'perror()' pe terminalul shell-ului, efectul percepții de utilizator fiind similar comenziilor interne.

De asemenea, în absența unor specificări speciale în linia de comandă ('&'), procesele copil care execută comenzi externe sunt rulate în foreground la terminal (pot citi, scrie sau intercepta semnale generate de la tastatură), în timp ce shell-ul părinte așteaptă terminarea lor. După terminarea copilului curent, shell-ul părinte se trezește și reia ciclul, pentru o nouă comandă.

Schema uzuală de lucru este:

```
do {
    puts(prompter);
    gets(linie_de_comanda);
    parsare(linie_de_comanda);
    /* interpretare constructii speciale, impartire in cuvinte, etc.*/
    if(comanda_interna(linie_de_comanda)) {
        /* executata shell-ul ceva */
    } else if(fork()) { /* parinte */
        wait(NULL);
    } else {           /* copil */
        prelucrare(linie_de_comanda);
        /* de exemplu, redirectarea descriptorilor 0, 1, 2, daca e cazul */
        exec(linie_de_comanda);
        perror(linie_de_comanda);
        return 1;
    }
} while (1);
```

Detaliile de implementare pot diferi.

Exemplu: shell care suportă comanda internă 'exit' și comenzi externe cu mai multe argumente și mai multe redirectări '<', '>', '>>', '2>', '2>>':

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

char ldc[256], *a[256], *p;
int nc, i, inw;

int main(){
    while(1){
        printf(">>");
        gets(ldc);
        inw = 0; nc = 0;
        for(p = ldc; *p; ++p)
            if(!isspace(*p) && !inw) {inw = 1; a[nc++] = p;}
            else if(isspace(*p) && inw) {inw = 0; *p = '\0';}
        a[nc] = NULL;
        if(nc == 0) continue;
```

```

if(!strcmp(a[0],"exit")){
    exit(0);
}else if(fork()){
    wait(NULL);
}else{
    int d, s, i, j;
    for(i = 1; i < nc - 1; ++i) {
        if(!strcmp(a[i], ">")) {
            d = open(a[i+1], O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR); s = 1;
        } else if(!strcmp(a[i], ">>")) {
            d = open(a[i+1], O_WRONLY | O_CREAT | O_APPEND, S_IRUSR | S_IWUSR); s = 1;
        } else if(!strcmp(a[i], "2>")) {
            d = open(a[i+1], O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR); s = 2;
        } else if(!strcmp(a[i], "2>>")) {
            d = open(a[i+1], O_WRONLY | O_CREAT | O_APPEND, S_IRUSR | S_IWUSR); s = 2;
        } else if(!strcmp(a[i], "<")) {
            d = open(a[i+1], O_RDONLY); s = 0;
        } else continue;
        if(d == -1) {perror(a[i]); exit(1);}
        dup2(d, s); close(d);
        for(j = i; j < nc - 1; ++j) a[j] = a[j+2];
        --i; nc -= 2;
    }
    execv(a[0],a);
    perror(a[0]);
    return 1;
}
return 0;
}

```

Testare (executabilul s.n. 'mysh'):

```
$ ./mysh
>>/bin/echo abc > test
>>/bin/cat test
abc
>>/bin/cat test > test1
>>/bin/cat < test1 > test2
>>/bin/cat test2
abc
>>/bin/diff test test2
>>exit
```

Comentarii:

- Secvența de cod:

```
inw = 0; nc = 0;
for(p = ldc; *p; ++p)
    if(!isspace(*p) && !inw) {inw = 1; a[nc++] = p;}
    else if(isspace(*p) && inw) {inw = 0; *p = '\0';}
a[nc] = NULL;
```

identifică cuvintele liniei de comandă, le delimitizează (scriind un '\0' la sfârșit) și le pointează cu 'a' chiar în linia de comandă 'ldc' (nu le copiază în altă parte); în ultima componentă a lui 'a' scrie NULL, a.î. 'a' să se poată folosi ca argument 'argv' la 'execv()' iar mai apoi 'nc' și 'a' să devină 'argc' și 'argv' pentru programul lansat.

Identificarea cuvintelor s-a făcut după o schemă de decizie cu stările specificate de 'inw': 0 = în afara unui cuvânt, 1 = într-un cuvânt; deciziile se referă la pointarea începutului, delimitarea sfârșitului, incrementarea numărului de cuvinte 'nc' și schimbarea stării 'inw'.

- La partea de 'prelucrare(linie_de_comanda)' au fost identificate specificările de redirectare '< fisier', etc., acestea au fost scoase din linia de comandă curentă (translatarea cuvintelor:

```
for(j = i; j < nc - 1; ++j) a[j] = a[j+2];  
) și s-au făcut redirectările descriptorilor standard corespunzători ('open()',  
'dup2()', 'close()').
```

Programul lansat va primi argumentele rămase (el nu va cunoaște linia de comandă originală) și va moșteni descriptorii standard redirectați.

- La testare, utilitarele 'echo', 'cat', 'diff' au trebuit apelate cu cale, deoarece am folosit apelul 'execv()', care nu folosește variabila de environment 'PATH' moștenită de la shell-ul sistem; dacă foloseam 'execvp()', puteam omite căile.

- Inspectând codul, se poate constata că comenziile externe se pot lansa și intercalând argumentele acestora cu redirectările sau scriind mai multe redirectări de același tip; experimental, se poate constata că și shell-ul din sistem procedează aşa:

```
$ls -l  
total 0  
$echo abc > test.txt  
$cat test.txt  
abc  
$cat > test1.txt > test2.txt test.txt  
$ls -l  
total 8  
-rw-rw-r-- 1 dragulici dragulici 0 Dec 1 18:03 test1.txt  
-rw-rw-r-- 1 dragulici dragulici 4 Dec 1 18:03 test2.txt  
-rw-rw-r-- 1 dragulici dragulici 4 Dec 1 18:03 test.txt  
$cat test2.txt  
abc  
$diff test.txt test2.txt  
$
```

(așadar, am putut insera redirectările între numele comenzi 'cat' și argumentul ei 'test.txt' și am putut scrie două redirectări de același tip '>' - shell-ul a deschis în suprascriere ambele fișiere 'test1.txt', 'test2.txt' (în particular, le-a creat sau le-a șters vechiul conținut) dar a păstrat doar redirectarea standard output doar către ultimul.

Dintr-un program, putem lansa foarte simplu o comandă shell folosind:

```
#include <stdlib.h>
int system(const char *command);
```

Lansează un proces shell copil al procesului curent, care execută comanda shell 'command' și se termină.

Practic, funcția 'system()' crează cu 'fork()' un proces copil care folosește 'execl()' pentru a lansa Bourne Shell ('/bin/sh') pentru a executa doar comanda shell 'command' și apoi se termină:

```
execl("/bin/sh", "sh", "-c", command, (char *) NULL);
```

În timpul executării lui 'command':

- în procesul care a apelat 'system()', semnalul SIGCHLD va fi blocat iar semnalele SIGINT și SIGQUIT vor fi ignorate;
- în procesul copil care execută comanda, semnalele SIGCHLD, SIGINT, SIGQUIT vor fi gestionate cu handler-ul implicit (a se vedea secțiunea 'Semnale').

Apelul 'system()' returnează după ce executarea comenzi s-a încheiat; valoarea returnată este următoarea:

- dacă 'command' este NULL, returnează 0 dacă shell-ul este disponibil și o valoare \neq 0 dacă shell-ul nu este disponibil;
- dacă procesul copil nu a putut fi creat sau status-ul său nu a putut fi recuperat, returnează -1 și setează 'errno';
- dacă un shell nu a putut fi executat în procesul copil, returnează o valoare ca și cum shell-ul copil s-a terminat prin apelul '_exit(127)'.
- dacă toate apelurile sistem reușesc, returnează codul de return (termination status) al copilului shell utilizat să execute comanda; codul de return al unui shell este codul de return al ultimei comenzi executate de acesta.

În ultimele două cazuri, valoarea returnată este un "wait status" care poate fi examinat cu macro-urile descrise pentru 'waitpid()' ('WIFEXITED()', 'WEXITSTATUS()', etc.).

Apelul 'system()' nu afectează wait status -ul nici unui alt copil.

Exemplu: Program care primește ca argumente cuvintele unei linii de comandă shell, execută comanda cu 'system()' și afișază valoarea returnată de acesta:

```
$cat prog.c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
extern char **environ;
int main(int argc, char* argv[]) {
    int i, dim, s; char *buf;
    dim = 0;
    for(i = 1; i < argc; ++i) dim += (strlen(argv[i]) + 1);
    if ((buf = malloc(dim * sizeof(char))) == NULL) {
        perror("malloc");
        return 1;
    }
    strcpy(buf, "");
    for(i = 1; i < argc - 1; ++i) {strcat(buf, argv[i]); strcat(buf, " ");}
    if(i < argc) strcat(buf, argv[i]);
    s = system(buf);
    printf("Codul returnat de 'sistem()' este: %d\n", s);
    free(buf);
    return 0;
}
```

```
$gcc -o prog prog.c
$./prog ls -l
total 24
-rwxrwxr-x 1 dragulici dragulici 16960 Dec  1 18:54 prog
-rw-rw-r-- 1 dragulici dragulici    535 Dec  1 18:53 prog.c
Codul returnat de 'sistem()' este: 0
$./prog ls -l bazaconie
ls: cannot access 'bazaconie': No such file or directory
Codul returnat de 'sistem()' este: 512
```

Am văzut ca setarea bițiilor SUID, SGID la un fișier executabil permite lansarea lui ca proces având ca proprietar sau grup efectiv pe cele ale fișierului lansat (de obicei un utilizator / grup privilegiat, 'root') nu pe cele ale procesului apelant (de obicei un utilizator / grup neprivilegiat), a.î. un utilizator neprivilegiat să poată opera asupra sistemului cu proceze privilegiate, având posibilitatea să acționeze (în mod limitat) asupra anumitor resurse restricționate.

De exemplu, editorul de texte 'vi' ('/bin/vi'), care permite modificarea discreționară a unui fișier oarecare, și programul 'passwd' ('/bin/passwd'), care permite doar modificarea fișierului '/etc/passwd' și doar pentru a schimba parola utilizatorului curent (în cazurile uzuale), sunt foarte asemănătoare ca restricții de funcționare și, în absența bitului SUID, nu s-ar putea implementa distincția ca un utilizator obișnuit să nu poată modifica fișierul '/etc/passwd' cu 'vi' dar să-l poată modifica cu 'passwd'.

Astfel:

- fișierul text '/etc/passwd' și fișierele binare '/bin/vi', '/bin/passwd' au proprietar 'root', pentru că sunt fișiere ale SO iar un utilizator obișnuit nu trebuie, de exemplu, să le poată modifica atributele;
- fișierele binare '/bin/vi', '/bin/passwd' au drept de execuție pentru toată lumea a.î. orice utilizator să le poată lansa (din shell, care apelează 'exec()') pentru ca să-și poată modifica discreționar fișierele proprii, resp. să-și poată schimba parola proprie;
- fișierul text '/etc/passwd' are drept de citire (r) pentru toată lumea a.î., de exemplu, orice utilizator să-l poată consulta cu 'ls -l' (care, pentru intrările de director afișate, află codul UID al proprietarului cu 'stat()' și numele acestuia cu 'getpwuid()', care citește '/etc/passwd');
- fișierul text '/etc/passwd' nu are drept de scriere (w) pentru toată lumea, pentru ca nu orice utilizator să-l poată modifica discreționar cu un editor de texte (care își deschide fișierul întă în citire și scriere);
- ambele programe 'vi' și 'passwd' își deschid fișierul întă în citire și scriere;

- dar comanda 'vi /etc/passwd' va trebui să refuze modificarea și salvarea (în fapt, va deschide fișierul readonly) iar comanda 'passwd' va trebui să reușească modificarea lui '/etc/passwd';

distincția nu poate fi făcută pe baza informațiilor privind proprietarii și drepturile și atunci intervine SUID: pentru '/bin/vi' este 0, pentru '/bin/passwd' este 1; astfel, orice utilizator 'ion' poate lansa atât 'vi /etc/passwd' cât și 'passwd' (are drept de execuție pe cele două fișiere executabile), va obține procese care au proprietarul real 'ion' (moștenit de la shell) dar procesul 'vi' va avea proprietarul efectiv 'ion' (și nu va putea scrie în '/etc/passwd') iar procesul 'passwd' va avea proprietarul efectiv 'root' (și va putea scrie în '/etc/passwd');

cei doi proprietari efectivi sunt setați de 'exec()' apelat din shell (care are proprietarii real și efectiv 'ion'), în funcție de biții SUID ai fișierelor binare '/bin/vi', '/bin/passwd'.

Bitul SGID are efect asemănător cu SUID, dar referitor la grupul efectiv.

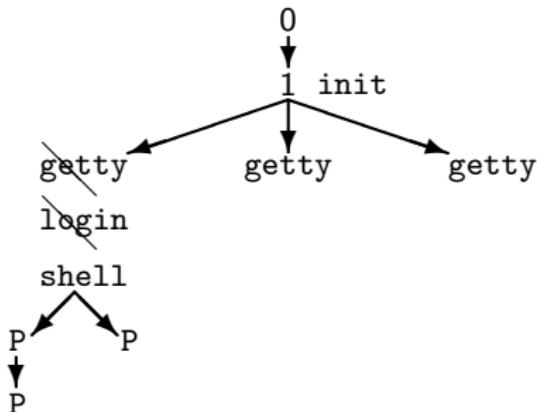
```

cazul
shell   cazul  cazul
-----  generare copil -----  copil   vi     passwd
|copil: ======>|
|pr.real:  ion          |pr.real:  ion  ion  ion
|pr.efectiv:ion          |pr.efectiv:ion  ion  root
|imagine:----->|imagine:----->
|program:---           |program:--- |
----- |           v      fork() |----- | -----
|       imagine proces bash ----->| | |
|           |           ^           |           v           v
|           |           |           |       imagine  inlocuire  imagine
|           |           |           |       proces ======> proces
|           |           |           |       bash           vi,
|           v           |           |           |           passwd
|       afisaza  citeste comenzi: |       | exec()           |
|       prompter  vi /etc/passwd |       |           | open(0_RDWR)
|       $         passwd          |       v           v
|           |           |           |       /bin/vi           /etc/passwd
|           |           |           |       <->| /bin/passwd      pr.fisier: root
|           |           |           |       pr.fisier: root  drepturi: rw-r--r--
|           v           |           |           |       drepturi: rwxr-xr-x
|       /bin/bash          |           |           |       SUID: 0 (vi), 1 (passwd)
pr.fisier: root

```

O instanță Unix/Linux are un arbore unic de procese, în care toate procesele, cu excepția celui inițial (PID = 0) sunt create de procese existente cu apeluri 'fork()', 'exec()'.

Unele dintre procese sunt standard, altele apar în funcție de activitatea curentă. Ilustrăm mai jos o parte din arborescența standard, în cazul unui sistem Unix tradițional (în sistemele Linux moderne, mai ales dacă au instalată și interfața X-Windows, mai pot exista și alte procese):



- Procesul cu PID=0, apare la boot-are; este un proces atipic, de ex. nu îl se pot transmite semnale; pe unele sisteme Unix (System V) el se ocupă cu planificarea la execuție. El generează procesul cu PID = 1 ('init').
- Procesul 'init' (PID = 1) are mai multe sarcini, printre care:
 - Pune în funcțiune restul sistemului, urmând anumite scripturi de startup, de exemplu, scriptul '/etc/inittab', care specifică ce programe trebuie rulate și în ce condiții (runlevel); de asemenea, gestionează oprirea sistemului (shut down).
 - Lansează procesele daemon (uzual, au nume terminate în 'd') - ele sunt procese de fundal, nu au terminal de control și supervizează sistemul sau oferă funcționalități altor procese.
De exemplu, 'lpd' gestionează sarcinile de printare trimise de alte procese.
Procesele daemon sunt analoage serviciilor Windows.
 - Pentru fiecare terminal disponibil în sistem (tty), procesul 'init' generează un copil 'getty' ('fork()' și 'exec()') care va gestiona logarea la terminalul respectiv.
 - Adoptă proceele orfane și le elimină din sistem când devin zombie (a se vedea mai jos).

- Procesul 'getty' lansat de 'init' la un terminal stabilește parametrii comunicării prin terminalul respectiv și, dacă detectează o activitate la terminal, afișază invitația de login ('login:'); în continuare, procesul 'getty' citește numele de login al utilizatorului și se înlocuiește ('exec()') cu un proces 'login' căruia îi transmite numele de login ca argument în linia de comandă; procesul 'login' devine copilul lui 'init', în locul lui 'getty'.
- Procesul 'login' invită utilizatorul (al cărui nume a fost primit ca argument) să introducă parola, care este citită fără afișare în ecou.
Dacă parola este greșită, repetă de câteva ori operația, apoi se termină iar procesul 'init' detectează terminarea copilului 'login' ('wait()') și lansează un alt proces 'getty' la terminalul respectiv.
Dacă parola este corectă, își schimbă proprietarul ('setuid()') în utilizatorul specificat (proprietarul inițial este 'root' și poate să facă asta), își schimbă directorul curent ('chdir()') în directorul home al utilizatorului specificat, aflat din '/etc/passwd' ('getpwnam()'), la fel și alte atribute, apoi se înlocuiește ('exec()') cu un proces shell, care execută login shell-ul utilizatorului respectiv (aflat tot din '/etc/passwd'); procesul 'getty' devine copilul lui 'init', în locul lui 'login'.

- Procesul shell moștenește terminalul de control și descriptorii standard pe el, de aceea afișaza prompterul și primește comenzi de la același terminal de unde s-a făcut logarea; moștenește proprietarul, dar acesta este utilizatorul logat; moștenește directorul curent, iar acesta este directorul home al utilizatorului logat - primul director curent primit de utilizator la logare este directorul său home; și rulează programul specificat ca login shell pentru utilizatorul respectiv. Când procesul shell se termină (de exemplu, utilizatorul dă comanda 'exit'), procesul 'init' detectează terminarea copilului shell ('wait()') și lansează un alt proces 'getty' la terminalul respectiv.
- Procesului shell îl se pot da comenzi interactiv; unele sunt executate chiar de shell (comenzi interne), pentru altele sunt lansate alte programe ca procese copil (cu 'fork()' și 'exec()') (comenzi externe). Procesele copil lansate în urma comenziilor externe pot fi generate succesiv sau în paralel, ele pot genera alte procese copil, etc. - astfel, arborele de procese se mărește.

Observație:

Folosirea a două procese separate pentru stabilirea comunicării la terminal ('getty') și autentificarea unui utilizator ('login') oferă mai multă flexibilitate.

De exemplu, 'getty' nu este necesar să execute login - el poate executa un program care transmite faxuri, un daemon PPP care emulează o conexiune rețea pe o linie serială, sau un program pentru a gestiona un modem cu "voicemail".

De asemenea, 'login' poate fi lansat și printr-o comandă shell, pentru a schimba sesiunea de lucru curentă cu cea a unui alt utilizator (shell-ul poate să nu o lanseze ca un subproces ci să facă direct 'exec()').

La terminare, un proces furnizează procesului părinte:

- semnalul SIGCHLD (17); handlerul implicit este de ignorare, dar poate fi instalat un handler utilizator; părintele tratează semnalul atunci când vine - execută handlerul; alte detalii, în secțiunea 'Semnale';
- codul de return; este valoarea returnată de 'main()' sau furnizată ca argument funcției 'exit()'; părintele preia codul de return atunci când dorește - când execută 'wait()' sau 'waitpid()'; până atunci, copilul rămâne în sistem, în starea zombie; se poate returna orice cod, convenția uzuală este de a returna 0: success și $\neq 0$: eșec; încrucișat 'wait()' și 'waitpid()' furnizează codul de return pe un octet, acesta trebuie să fie în limitele tipului 'unsigned char' (0, ..., 255).

Când un proces se termină, el intră în starea zombie: nu folosește procesorul (este considerat terminat) dar este încă menținut în sistem, PCB al său este menținut în tabela de procese și conține informații minimale (codul de return); în momentul când părintele său îl colectează codul de return (cu 'wait()' sau 'waitpid()'), procesul zombie este eliminat complet.

Dacă părintele unui proces se termină fără să îl fi așteptat cu 'wait()' sau 'waitpid()', procesul respectiv, devenit orfan, fie el zombie sau nu, devine copil al procesului 'init'. Acesta îl va aștepta cu 'wait()' sau 'waitpid()' și când va deveni (sau dacă era deja) zombie îl va elimina.

Cel mai "curat" mod de a termina voluntar un proces este apelul:

```
#include <stdlib.h>
void exit(int status);
```

Determină terminarea normală a procesului apelant, iar byte-ul low din 'status' (adică 'status & 0xFF') este furnizat ca cod de return procesului părinte.

Sunt apelate funcțiile înregistrate cu 'atexit()' și 'on_exit()' în ordinea inversă înregistrării, sunt evacuate bufferele stream-urilor stdio deschise (de ex. 'FILE') apoi acestea se închid, sunt șterse fișierele create cu 'tmpfile()'.

Funcția 'exit()' nu returnează (deoarece procesul apelant nu mai există).

Pentru a specifica codul de return conform convenției uzuale (0: success și ≠0: eșec), standardul C menționează două constante, EXIT_SUCCESS și EXIT_FAILURE (semnificând succes, respectiv eșec), care pot fi transmise ca parametru lui 'exit()'.

Obs: Este posibil ca una din funcțiile înregistrate cu 'atexit()' sau 'on_exit()' să apeleze 'atexit()' sau 'on_exit()' pentru a înregistra alte funcții; noile înregistrări vor fi adăugate la începutul listei funcțiilor care au mai rămas de apelat. Dacă una din aceste funcții nu face return (de exemplu, apelează '_exit()' sau se sinucide cu un semnal), atunci nici una dintre funcțiile rămase de apelat nu va mai fi apelată iar restul acțiunii 'exit()' (de exemplu, evacuarea bufferelor 'stdio') este abandonată. Dacă una din aceste funcții apelează 'exit()', comportamentul este nedefinit.

```
#include <stdlib.h>
int atexit(void (*function)(void));
```

Înregistrează funcția 'function' pentru a fi apelată la terminarea normală a procesului, via apelul 'exit()' sau return din 'main()'.

Funcțiile înregistrate astfel sunt apelate în ordinea inversă înregistrării și nu le sunt transmise argumente.

Dacă o funcție este înregistrată de mai multe ori, este apelată de atâtea ori cât este înregistrată.

Cf. POSIX.1, pot fi înregistrate cel puțin 'ATEXIT_MAX' funcții; limita reală poate fi obținută cu 'sysconf()' (ex: 'long a = sysconf(_SC_ATEXIT_MAX);').

Înregistrările de funcții se moștenesc la 'fork()', nu se moștenesc la 'exec()'.

Funcția 'atexit()' returnează 0 la succes și nonzero la eșec.

```
#include <stdlib.h>
int on_exit(void (*function)(int , void *), void *arg);
```

Înregistrează funcția 'function' pentru a fi apelată la terminarea normală a procesului, via apelul 'exit()' sau return din 'main()'. Funcției 'function' îi sunt transmise argumentul 'status' transmis ultimului apel al lui 'exit()' și argumentul 'arg' al lui 'on_exit()'.

Dacă o funcție este înregistrată de mai multe ori, este apelată de atâtea ori cât este înregistrată.

Înregistrările de funcții se moștenesc la 'fork()', nu se moștenesc la 'exec()'.

Funcția 'on_exit()' returnează 0 la succes și nonzero la eșec.

Obs: La momentul executării funcției 'function', variabilele alocate automatic (pe stiva apelurilor de funcții, auto variable) pot avea deja durata de existență depășită (out of scope). De aceea, 'arg' nu trebuie să pointeze o variabilă automatică; poate pointa, însă, o variabilă alocată dinamic (heap variable) sau o variabilă globală.

Obs: Funțiile 'atexit()' și 'on_exit()' înregistrează funcții în aceeași listă; la terminarea normală a procesului, funcțiile înregistrate sunt apelate în ordinea inversă înregistrării.

Exemplu:

```
$cat prog.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void f1(void) {printf("Bye !\n");}
void f2(void) {printf("Pa !\n");}
void f3(int n, void *s) {printf("%d %s\n", n, (char *) s);}

int main() {
    long limita;
    limita = sysconf(_SC_ATEXIT_MAX);
    printf("ATEXIT_MAX = %ld\n", limita);
    if(atexit(f1)) {fprintf(stderr, "Err\n"); exit(EXIT_FAILURE);}
    if(on_exit(f3, "salutari !")){fprintf(stderr,"Err\n");exit(EXIT_FAILURE);}
    if(atexit(f2)) {fprintf(stderr, "Err\n"); exit(EXIT_FAILURE);}
    if(atexit(f1)) {fprintf(stderr, "Err\n"); exit(EXIT_FAILURE);}
    if(on_exit(f3, "felicitari !")){fprintf(stderr,"Err\n");exit(EXIT_FAILURE);}
    exit(EXIT_SUCCESS);
    return 0;
}
```

```
$gcc -Wall -o prog prog.c
$./prog
ATEXIT_MAX = 2147483647
0 felicitari !
Bye !
Pa !
0 salutari !
Bye !
```

TODO: sesiuni și grupuri, foreground și background.

Prezentăm alte câteva funcții utile:

```
#include <stdlib.h>
int rand(void);
void srand(unsigned int seed);
```

Funcția 'rand()' returnează un întreg pseudo-aleator în intervalul [0, RAND_MAX]; secvența de numere generate de apeluri succesive este doar aparent aleatoare, numerele sunt puternic dispersate dar secvența este unic determinată de o anumită valoare germen (seed) - dacă repetăm generarea numerelor pornind de la un aceeași germen, obținem aceeași secvență.

Funcția 'srand()' setează argumentului 'seed' ca germen pentru o nouă secvență de numere generate de 'rand()'.

Dacă se repetă apelarea lui 'srand()' cu aceeași valoare 'seed', se repetă și secvența care poate fi generată ulterior cu 'rand()'. De aceea, se pot folosi artificii gen: 'srand(getpid())' sau 'srand(time(NULL))'.

Dacă nu este furnizat un germen (nu se apelează 'srand()' înainte de utiliza 'rand()'), secvența generată de 'rand()' folosește germeneul 1.

Funcția 'rand()' folosește o stare ascunsă care se modifică la fiecare apel, de aceea nu este reentrantă; o variantă reentrantă (utilă într-o aplicație multithred) este funcția 'rand_r()' (a se vedea 'man rand_r').

Obs: un cod s.n. reentrant dacă poate fi întrerupt în mijlocul executiei sale și apoi poate fi apelat din nou în siguranță ("re-intrat") înainte ca invocările sale anterioare să finalizeze execuția.

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

Procesul apelant este adormit până trec 'seconds' secunde de timp real sau se primește un semnal neignorat (a se vedea secțiunea 'Semnale').

Funcția 'sleep()' returnează 0, dacă a trecut timpul cerut, sau numărul de secunde rămase, dacă apelul a fost întrerupt de handlerul unui semnal.

```
#include <unistd.h>
int usleep(useconds_t usec);
```

Procesul apelant este suspendat pentru cel puțin 'usec' microsecunde; durata poate fi mărită puțin de activitatea din sistem, de timpul consumat cu procesarea apelului sau de granularitatea timerelor sistemului. Așteptarea poate fi întreruptă de primirea unui semnal.

Funcția 'usleep()' returnează 0 în caz de succes sau -1 (și setează 'errno') în caz de eșec.

Obs: 4.3BSD, POSIX.1-2001. POSIX.1-2001 declară această funcție depreciată (obsolete); se recomandă folosirea în schimb a lui 'nanosleep()'; POSIX.1-2008 a eliminat specificația lui 'usleep()'.

```
#include <time.h>
int nanosleep(const struct timespec *req, struct timespec *rem);
```

Procesul apelant este suspendat până trece (cel puțin) timpul specificat în '*req' (timpul poate avea precizie de nanosecunde, a se vedea secțiunea 'Fișiere și directoare') sau se primește un semnal care lansează un handler în procesul apelant sau care termină procesul (a se vedea secțiunea 'Semnale').

Dacă apelul este întrerupt de un handler de semnal, funcția 'nanosleep()' returnează -1, setează 'errno' la valoarea 'EINTR', și scrie timpul rămas în structura '*rem', cu excepția cazului când 'rem' este NULL; valoarea lui '*rem' poate fi folosită pentru a apela din nou 'nanosleep()' și a completa perioada de așteptare.

Dacă a dormit cu succes perioada de timp cerută, funcția 'nanosleep()' returnează 0; dacă apelul a fost întrerupt de handlerul unui semnal sau a apărut o eroare, funcția returnează -1 și setează 'errno'.

Cuprins

1 Sisteme de calcul, sisteme de operare

Organizarea unui sistem de calcul
Funcțiile sistemului de operare

2 Sisteme de operare uzuale

Incursiune în sistemul Windows
Incursiune în sistemul Linux

3 Interfața de programare C

Obiecte software și apeluri sistem
Utilizatori și grupuri

Fișiere și directoare

Procese

Semnale

Tuburi

Biblioteca standard C (tipul 'FILE')

4 Interfață interactivă linie de comandă

Interfața Command Prompt
(Windows)
Interfața Shell (Linux)

5 Interfață de scripting linie de comandă

Interfața Batch Scripting (Windows)
Interfața Shell Scripting (Linux)

6 Teme de laborator

Utilizatori și grupuri
Fișiere, directoare
Procese, semnale, tuburi

Semnalele (signal) sunt entități software ce au ca principală informație asociată un cod numeric întreg (tipul semnalului).

Codurile semnalelor sunt de la 1 la NSIG-1 și se pot desemna prin mnemonice scrise cu majuscule și care încep cu "SIG" (SIGINT, SIGQUIT, etc.); atât NSIG cât și menmonicile sunt constante simbolice definite în '<signal.h>'.

Semnalele pot fi primite de procese în mod asincron, fiind trimise de alte procese sau generate de anumite evenimente; când la un proces ajunge un semnal, de obicei (dacă semnalul nu este tratat cu un handler instalat cu 'sigaction()' și flagul 'SA_SIGINFO') acesta nu cunoaște expeditorul sau evenimentul generator ci doar tipul semnalului (codul numeric al semnalului).

Un semnal poate fi trimis de un proces altor procese folosind:

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

⇒ dacă 'pid' este > 0 , se trimit semnalul 'sig' procesului cu PID-ul 'pid';
dacă 'pid' este $= 0$, se trimit 'sig' tuturor proceselor din același grup cu procesul expeditor;

dacă 'pid' este $= -1$, se trimit 'sig' tuturor proceselor pentru care procesul apelant are dreptul să trimită semnale, mai puțin procesului 'init' care are PID-ul 1 (cu unele excepții);

dacă 'pid' este < -1 , se trimit 'sig' tuturor proceselor din grupul al cărui lider are PID-ul -PID;

dacă 'sig' este 0, nu se transmite nici un semnal, dar se face verificarea erorilor (astfel, putem verifica dacă un proces există trimițându-i 0 și verificând codul de eroare furnizat de 'kill()' în 'errno');

procesul expeditor poate trimite un semnal doar dacă este proces privilegiat sau dacă UID-ul sau EUID-ul lui coincide cu UID-ul sau 'saved set-user-ID'-ul procesului destinatar; în cazul semnalului SIGCONT este suficient ca expeditorul și destinatarul să fie în aceeași sesiune;

la succes returnează 0, la eșec returnează -1 și setează 'errno'.

```
#include <signal.h>
int raise(int sig);
    => se trimită semnalul 'sig' însuși procesului curent;
este echivalent cu 'kill(getpid(), sig);'
la succes returnează 0, la eșec returnează ≠ 0.
```

Din linia de comandă shell putem transmite semnale către procese cu comanda:

```
kill -semnal proces
```

unde 'semnal' este codul numeric sau sfârșitul mnemonicului unui semnal (partea fără 'SIG'), iar 'proces' este PID-ul procesului destinatar; în general, comanda reușește dacă proprietarul său real coincide cu cel al procesului destinatar sau este root; mai sunt și alte detalii.

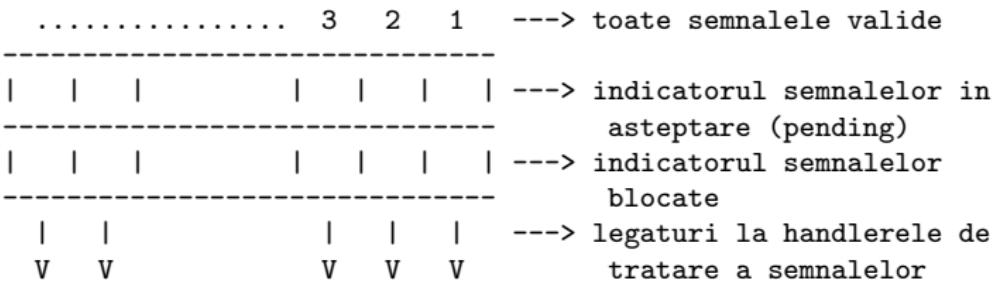
Cu comanda:

```
kill -l
```

se afișază o listă cu toate semnalele implementate în sistemul curent.

Câteva evenimente care generează semnale: tastarea Ctrl-c, resp. Ctrl-\ (procesele aflate în foreground la terminalul respectiv primesc 'SIGINT', resp. 'SIGQUIT'), accesarea ilegală a memoriei (procesul primește SIGSEGV), scrierea într-un tub fără cititori (procesul primește 'SIGPIPE', vom vedea în secțiunea 'Tuburi'), un proces copil se termină (părintele primește 'SIGCHLD').

Pentru a gestiona semnalele primite, un proces folosește o structură de date cu următoarea schemă:



(pentru un semnal, indicatorul de pending si cel de blocare sunt biti, iar handlerul este o functie)

Detaliile de implementare pot dифri вn func\$ie de SO. În Linux, structura de tip 'struct task_struct' (blocul de control) a unui proces con\$ine referiri la structurile de date folosite pentru gestionarea semnalelor:

```
struct task_struct {  
    ...  
    /* signal handlers */  
    struct signal_struct *signal;  
    /* descriptorul semnalelor, membrul signal->shared_pending  
       specifica semnale in pending asociate unui grup de threaduri */  
    struct sighand_struct *sighand;  
    /* handlerele semnalelor asociate procesului */  
    sigset_t blocked, real_blocked;  
    /* semnale blocate asociate procesului */  
    sigset_t saved_sigmask; /* restaurata la folosirea set_restore_sigmask() */  
    struct sigpending pending;  
    /* semnale in pending asociate procesului */  
    unsigned long sas_ss_sp;  
    size_t sas_ss_size;  
    unsigned sas_ss_flags;  
    /* descriu o stiva dedicata executarii handlerelor,  
       alocata cu 'sigaltstack()' */  
    ...  
};
```

Când la un proces ajunge un semnal n:

- bitul de aşteptare (pending) al lui n devine 1; dacă pe perioada cât acest bit este 1 mai vine un semnal n, acesta se va pierde (din fiecare semnal poate fi la un moment dat în pending doar un singur exemplar);
- dacă bitul de blocaj al lui n este 1, semnalul n nu va fi tratat (el rămâne în pending); dacă acest bit e 0, semnalul n va fi tratat atunci când va fi posibil (a se vedea mai jos);
- în general, un proces tratează (delivery) semnalele aflate în pending doar atunci când este în user mode, deci atunci când execută instrucțiuni ale utilizatorului; astfel, când procesul execută un apel sistem, el nu tratează semnalele (toate sunt blocate temporar);
- tratarea unui semnal n aflat în pending constă în apelarea handlerului (disposition) asociat; chiar de la începutul executării handlerului, bitul de pending al lui n se reposiționează pe 0 (deci procesul poate primi imediat un alt n, care va intra în pending); pe parcursul executării handlerului, n este în principiu blocat suplimentar, astfel că dacă între timp vine un nou n, el va fi blocat în pending până la sfârșitul executării handlerului, când va putea fi tratat lansând din nou handlerul; un handler poate fi însă instalat și a.î. acest blocaj suplimentar să nu mai apară (a se vedea mai jos).

Pentru fiecare semnal există handlere implicate (ale sistemului), dar pentru majoritatea putem instala handlere utilizator (excepții: SIGKILL, SIGSTOP, uneori SIGCONT). În general, handlerele implicate sunt fie de ignorare fie de terminare a procesului.

Pentru un semnal putem reinstala handlerul implicit, desemnat prin SIG_DFL (care însă pentru fiecare semnal poate înseamna altceva), sau un handler de ignorare al sistemului, desemnat prin SIG_IGN (pentru cele două constante simbolice putem include '<signal.h>').

Observații:

- dacă un proces primește un semnal neblocat n și este în user mode, el va fi întrerupt și se va executa handlerul h asociat lui n; apoi, dacă h nu a cerut terminarea programului, acesta se reia de unde s-a întrerupt;
- pe parcursul executării lui h mai poate veni un semnal neblocat p și atunci sunt posibile cazurile:
 - dacă h este un handler al sistemului (SIG_DFL sau SIG_IGN): atunci p rămâne în pending până la sfârșitul lui h, deoarece h se execută în kernel mode și toate semnalele sunt blocate temporar; după terminarea lui h, blocajul suplimentar dispare și p va fi tratat - se va lansa handlerul f al lui p;
 - dacă h este un handler al utilizatorului (chiar și funcția cu corp vid), atunci sunt posibile subcazurile:
 - dacă p = n:
atunci noul n nu se pierde (pentru că bitul de pending al lui n s-a reposiționat pe 0 chiar de la începutul lui h), dar rămâne în pending pe toată perioada lui h (pentru că n este blocat temporar); după terminarea lui h, blocajul suplimentar asupra lui n dispare și noul n va fi tratat (se va lansa iar h); am presupus însă că h nu a fost instalat cu eliminarea blocajului suplimentar, cum am spus mai sus că se poate;
 - dacă p ≠ n:
atunci h se întrerupe temporar (el se execută în user mode), se execută f, apoi se continuă h de unde a rămas.

În cele de mai sus am presupus că h și f nu produc terminarea programului.

Astfel, presupunând că n și p nu sunt blocate iar h și f sunt handle ale utilizatorului care nu termină programul și blochează semnalul pentru care au fost lansate pe perioada execuției lor, dacă procesul primește foarte repede secvența n, n, p, va executa: un început de h, apoi un f, apoi sfârșitul primului h, apoi un alt h.

Observație: Semnalele discutate în această secțiune sunt "semnale standard" (standard signal) - ele au coduri din intervalul 1 - 31; începând cu versiunea 2.2, Linux suportă și "semnale în timp real" (real-time signals, descrise în POSIX.1-2001) - ele au coduri din intervalul 'SIGRTMIN' - 'SIGRTMAX' (în Linux 32 - 64), nu au un sens predefinit, sunt la dispoziția utilizatorului, handlerul implicit este terminarea, se pot afla în pending simultan mai multe semnale de același tip, sunt tratate în ordine crescătoare a tipurilor (codurilor) și în ordinea venirii pentru semnale de același tip, pot fi trimise însotite de o valoare numerică auxiliară (folosind 'sigqueue()') iar această valoare poate fi obținută folosind un handler instalat cu 'sigaction()' și flagul 'SA_SIGINFO'. Pentru alte detalii, a se vedea 'man 7 signal'.

Pentru a manipula semnale din program, avem următoarele instrumente (furnizate de '<signal.h>'):

```
#include <signal.h>
sigset_t
    ⇒ tipul "mulțime de semnale";
int sigemptyset(sigset_t *set);
    ⇒ setează *set := {};
        returnează: 0=succes, -1=eșec;
int sigfillset(sigset_t *set);
    ⇒ setează *set := {1, ..., NSIG-1};
        returnează: 0=succes, -1=eșec;
int sigaddset(sigset_t *set, int signum);
    ⇒ adaugă *set := *set ∪ {signum};
        returnează: 0=succes, -1=eșec;
int sigdelset(sigset_t *set, int signum);
    ⇒ elimină *set := *set \ {signum};
        returnează: 0=succes, -1=eșec;
int sigismember(const sigset_t *set, int signum);
    ⇒ testeaza dacă signum aparține *set;
        returneaza: 0=nu aparține, 1=aparține, -1=eșec;
```

În caz de eșec, funcțiile setează 'errno' cu valoarea 'EINVAL'.

Putem bloca/debloca semnale (modificând linia a 2-a, a indicatorilor semnalelor blocate, din structura de gestiune a semnalelor primite de procesul curent descrisă schematic mai devreme) folosind:

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

⇒ setează/consultă masca procesului de blocare a semnalelor;

*set=noua mască (dacă set=NULL, nu se schimbă masca);

*oldset=aici se recuperează vechea mască (dacă oldset=NULL, nu se mai recuperează);

how=poate fi indicat prin următoarele constante simbolice (furnizate de <signal.h>):

SIG_SETMASK ⇒ noua mască devine *set

SIG_BLOCK ⇒ noua mască devine *oldset ∪ *set

SIG_UNBLOCK ⇒ noua mască devine *oldset \ *set

returnează: 0=succes, -1=eșec și setează 'errno';

nu pot fi blocate SIGKILL și SIGSTOP (încercarea eșuează);

masca de semnale blocate se moștenește la 'fork()' și la 'execve()';

în procesele multithread, fiecare thread are propria mască de semnale blocate.

Putem afla semnalele în pending la procesul curent (consultând prima linie, a indicatorilor semnalelor în pending, din structura de gestiune a semnalelor primite descrisă schematic mai devreme) folosind:

```
#include <signal.h>
int sigpending(sigset_t *set);
```

⇒ pune în *set mulțimea semnalelor aflate în pending în acel moment; dacă un semnal este și blocat ('sigprocmask()') și cu handlerul 'SIG_IGN', atunci la primirea lui nu este păstrat în pending; procesele copil generate cu 'fork()' au inițial mulțimea semnalelor în pending vidă; mulțimea semnalelor în pending se transmite la 'execve()'; în procesele multithread, se rețin semnale în pending și pentru fiecare thread în parte; funcția 'sigpending()' returnează: 0=succes, -1=esec și setează 'errno'.

Exemplu:

```
$cat prog.c
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>

int main() {
    sigset(SIGCHLD, handler);
    if(! fork()) {
        pid_t pp = getppid();
        kill(pp, SIGSEGV); kill(pp, SIGSEGV);
        return 0;
    }
    while(wait(NULL) != -1);
    sigpending();
    for(i = 1; i < NSIG; ++i) if(sigismember(SIGCHLD, i)) printf("%d ", i);
    printf("\n");
    sigempty(SIGCHLD);
    return 0;
}
```

```
$gcc -o prog prog.c
$./prog
11 17
Segmentation fault (core dumped)
```

Comentarii:

- Procesul copil trimite părintelui două semnale 'SIGSEGV' (la cerere explicită) și un semnal 'SIGCHLD' (la terminare, în mod automat).
- Părintele are cele două semnale blocate (blocajul este moștenit și de copil) iar 'SIGCHLD', chiar dacă are handlerul implicit de ignorare, nu are instalat handlerul de ignorare 'SIG_IGN' (deci, se poate păstra în pending); atunci, un 'SIGSEGV' și 'SIGCHLD' rămân în pending, al doilea 'SIGSEGV' se pierde (în pending nu pot fi păstrate simultan două semnale de același tip); după terminarea copilului, părintele consultă și afișază semnalele pe care le are în pending - un 'SIGSEGV' (11) și un 'SIGCHLD' (17).
- În final, părintele își deblochează semnalele iar cele două semnale din pending sunt tratate cu handlerul implicit - 'SIGCHLD' este ignorat, 'SIGSEGV' termină procesul, iar shell-ul părinte detectând că comanda './prog' s-a terminat ca urmare a primirii unui semnal ('wait()' sau 'waitpid()', 'WIFSIGNALED()' și 'WTERMSIG()'), afișază 'Segmentation fault (core dumped)'.

Pentru majoritatea semnalelor se pot instala handlere utilizator.

În general, pentru a putea fi folosită ca handler, o funcție trebuie să aibe un singur parametru de tip 'int' și să returneze 'void', de exemplu:

```
void h(int n){...}
```

Când procesul va primi semnalul asociat, el va fi transmis ca argument acestei funcții. Astfel, dacă aceeași funcție este instalată ca handler pentru mai multe semnale, ea va ști la fiecare apel semnalul pentru care a fost apelată (și poate fi scrisă să facă ceva diferit în fiecare caz).

Putem instala un handler utilizator pentru un semnal cu apelurile 'signal()' (mai simplu) și 'sigaction()' (mai elaborat).

Obs: Cu 'sigaction()' și flagul 'SA_SIGINFO' se pot instala handlere utilizator cu mai mulți parametri, prin care handlerul utilizator poate primi mai multe informații despre semnalul care a venit, inclusiv procesul expeditor (PID).

Apelul 'signal()' este mai simplu:

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

⇒ se instalează funcția specificată de 'handler' pentru semnalul 'signum';
'handler' poate specifica o funcție utilizator (cu semnatura menționată),
poate fi SIG_IGN (handler-ul de ignorare) sau SIG_DFL (handler-ul implicit al
semnalului 'signum'); 'signum' nu poate fi 'SIGKILL' sau 'SIGSTOP';
'signal()' returnează adresa vechiului handler folosit pentru semnalul
'signum', sau SIG_ERR în caz de eroare (iar atunci setează 'errno');

În unele implementări UNIX/Linux, 'signal()' instalează handler-ele utilizator
a.î. blochează temporar pe perioada apelului semnalul pentru care au fost
instalate sau reinstalează pentru semnalul respectiv, chiar de la începutul
apelului, handlerul SIG_DFL; în acest caz, dacă dorim să instalăm handler-ul
permanent, punem la începutul său un apel de reinstalare a sa pentru același
semnal:

```
void h(int n){signal(n,h); ...}
```

(dacă în plus pe perioada apelului 'n' este blocat, nu există riscul ca un nou 'n'
să întrerupă 'h' între '{' și 'signal(n,h);' iar pentru el să se execute
SIG_DFL); dacă nu vrem să ne bazăm pe aceste comportamente implicate,
folosim 'sigaction()'.

```
$cat prog.c
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
void h(int n) {
    signal(n, h);
    printf("Incepe h(%d)...\\n", n);
    sleep(5);
    printf("... se termina h(%d).\\n", n);
}
int main() {
    int sem[] = {2, 2, 2, 3}, n = sizeof(sem)/sizeof(int), i;
    signal(2, h); signal(3, h);
    if(! fork()) {
        pid_t p = getppid();
        for(i = 0; i < n; ++i) {kill(p, sem[i]); sleep(1);}
        return 0;
    }
    wait(NULL);
    return 0;
}
```

```
$gcc -o prog prog.c
$./prog
Incepe h(2)...
Incepe h(3)...
... se termina h(3).
... se termina h(2).
Incepe h(2)...
... se termina h(2).
```

Comentarii:

- Procesul copil trimite părintelui semnalele 2, 2, 2, 3 la interval de o secundă iar părintele le va trata cu un handler care afișază momentul de început, momentul de sfârșit iar între ele trec 5 secunde; apelurile 'sleep()' cresc probabilitatea ca semnalele să fie primite la anumite momente anticipate la scrierea codului, dar nu există garanții în acest sens - sistemul poate întrerupe procesele de mai multe ori pentru diverse alte activități, într-un mod care nu se poate anticipa la scrierea codului, a.î. nu putem face din interiorul programului predicții precise asupra duratelor.

- La rulare, au avut loc, în ordine următoarele evenimente (momentul este aproximativ):
 - secunda 0: copilul a trimis primul 2, care a intrat în pending la părinte; părintele încă nu ajunsese la 'wait()' era în user mode, a.î. a început apelul 'h(2)'; de la începutul executării lui 'h(2)', bitul de pending al lui 2 a redevenit 0 iar bitul de blocaj al lui 2 a devenit 1 (la sfârșitul apelului 'h(2)' blocajele suplimentare dispar iar bitul de blocaj al lui 2 va reveni la valoarea anterioară apelului); apelul 'h(2)' a afișat 'Incepe h(2) ...', apoi a intrat în'sleep(5)';
 - secunda 1: copilul a trimis al doilea 2, care a intrat în pending la părinte (deoarece apelul 'h(2)' început a pus deja bitul de pending al lui 2 pe 0), dar nu a întrerupt apelul 'h(2)' (deoarece apelul 'h(2)' început a pus bitul de blocaj al lui 2 pe 1); în părinte, apelul 'h(2)' continuă 'sleep(5)' (au mai rămas 4 secunde);
 - secunda 2: copilul a trimis al treilea 2; întruțăt în părinte era deja un 2 în pending, acest 2 se pierde; în părinte, apelul 'h(2)' continuă 'sleep(5)' (au mai rămas 3 secunde);
 - secunda 3: copilul a trimis 3, care a intrat în pending la părinte; întruțăt părintele era în user mode (executa 'h(2)') și nu avea pe 3 blocat (ci doar pe 2), apelul 'h(2)' s-a întrerupt în timpul lui 'sleep(5)' ca să se execute în întregime 'h(3)' (care a durat 5 secunde);

- secunda 8: părintele a revenit să continue apelul 'h(2)' - aici, nu a mai reluat 'sleep(5)' (din el a consumat doar 3 secunde), ci doar s-a finalizat cu '... se termina h(2)', apoi a revenit în 'main()';
aici, deoarece era în user mode și avea un 2 în pending, înainte de a ajunge la 'wait()' a executat complet un al doilea 'h(2)', care a duret 5 secunde;
- secunda 13: părintele a revenit din al doilea 'h(2)' în 'main()' și cu 'wait(NULL)' a așteptat terminarea copilului și a eliminat procesul zombie (dacă copilul era terminat, nu așteaptă ci doar elimină procesul zombie).

Apelul 'sigaction()' permite specificarea unor setări mai fine, cu ajutorul unei structuri de tip 'struct sigaction':

```
#include <signal.h>
int sigaction(int signum,
              const struct sigaction *act,
              struct sigaction *oldact);
```

⇒ dacă 'act' este ≠NULL, se instalează acțiunea descrisă de structura '*act' pentru semnalul "signum";

dacă 'oldact' este ≠NULL, este recuperată vechea acțiune în '*oldact';
'signum' nu poate fi SIGKILL sau SIGSTOP;

```
#include <signal.h>
struct sigaction{
    void (*sa_handler)(int);
    ...
    sigset_t sa_mask;
    int sa_flags;
    ...
};
```

acțiunea descrisă de o structură 'sigaction' conține următoarele informații:
'sa_handler' = pointer la funcția handler (poate fi o funcție utilizator sau
SIG_IGN sau SIG_DFL);

'sa_mask' = o mască de semnale adăugate la masca de semnale blocate a
procesului / threadului, pe perioada executării handlerului; în plus, va fi
blocat și semnalul care a declanșat handlerul, cu excepția cazului când a
fost folosit flagul SA_NODEFER;

'sa_flags' = 0 sau disjuncție pe biți de mai multe opțiuni;

câteva dintre opțiunile utilizabile la 'sa_flags':

SA_NOCLDSTOP = Dacă se instalează un handler utilizator, și 'signum' este SIGCHLD, nu se primește notificare când procesul copil este stopat sau reluat (i.e. primește SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU, respectiv SIGCONT);

SA_ONSTACK = Dacă se instalează un handler utilizator, el se va executa pe o stivă alternativă oferită de 'sigaltstack()'; dacă nu este disponibilă o stivă alternativă, se va folosi stivă implicită.

SA_RESETHAND, SA_ONESHOT (mai vechi) = Dacă se instalează un handler utilizator, atunci după o singură execuție a handlerului se va reinstala automat handlerul implicit; reinstalarea va avea loc chiar de la începutul executării noului handler (a se vedea funcția 'signal()');

SA_NODEFER, SA_NOMASK (mai vechi) = semnalul căruia i s-a asociat handlerul nu va mai fi blocat pe perioada executării sale (deci handlerul se va putea întrerupe de către el însuși, la venirea unui nou semnal de același tip);

SA_RESTART = Dacă se instalează un handler utilizator, atunci anumite apeluri sistem vor fi restartabile după primirea semnalului; de exemplu, dacă procesul doarme într-un 'wait()', la primirea semnalului va executa handler-ul apoi va continua să doarmă în acel 'wait()' (altfel, după executarea handler-ului nu se reia 'wait()' ci se trece mai departe).

Funcția 'sigaction()' returnează 0 în caz de succes sau -1 (și setează 'errno') în caz de eșec.

Observații:

- Handlerul unui semnal este un atribut per proces - este același pentru toate threadurile procesului.
- Procesele copil generate cu 'fork()' moștenesc handlerele asociate semnalelor. La 'execve()', tratarea semnalelor neignorate este restaurată la handlerele implicate iar a celor ignorate (SIG_IGN) este lăsată neschimbată.
- Conform POSIX, comportamentul unui proces este nedefinit după ce ignoră un semnal SIGFPE, SIGILL, sau SIGSEGV care nu a fost generat de 'kill()' sau 'raise()'; împărțirea întreagă la 0 are rezultat nedefinit; pe anumite arhitecturi, va genera un semnal SIGFPE; de asemenea, împărțirea celui mai negativ întreg la -1 poate genera SIGFPE. Ignorarea acestui semnal poate duce la ciclu infinit.

Exemplu: reluăm exemplul anterior, dar handlerul este instalat cu 'sigaction()' fără flagul SA_RESETHAND (deci nu mai este nevoie să-l reinstalăm la fiecare execuție) dar cu flagul SA_NODEFER (deci semnalul căruia î s-a asociat handlerul nu va mai fi blocat pe perioada executării sale):

```
$cat prog.c
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
void h(int n) {
    printf("Incepe h(%d)... \n", n);
    sleep(5);
    printf("... se termina h(%d). \n", n);
}
int main() {
    int sem[] = {2, 2, 2, 3}, n = sizeof(sem)/sizeof(int), i;
    struct sigaction s;
    s.sa_handler = h; sigemptyset(&s.sa_mask); s.sa_flags = SA_NODEFER;
    sigaction(2, &s, NULL); sigaction(3, &s, NULL);
    if(! fork()) {
        pid_t p = getppid();
        for(i = 0; i < n; ++i) {kill(p, sem[i]); sleep(1);}
        return 0;
    }
    wait(NULL); return 0;
}
```

```
$gcc -o prog prog.c
$./prog
Incepe h(2)...
Incepe h(2)...
Incepe h(2)...
Incepe h(3)...
... se termina h(3).
... se termina h(2).
... se termina h(2).
... se termina h(2).
```

Comentarii:

- Întrucât de la începutul executării handlerului bitul de pending al semnalului pentru care a fost lansat se reposiționează pe 0 iar bitul de blocaj nu devine 1, la fiecare nou semnal venit s-a întrerupt apelul curent al handlerului pentru a se executa un apel îmbricat;
- Apelurile încuibate au început la interval de 1 secundă, fiecare a întrerupt 'sleep(5)' din apelul exterior; apelul 'h(3)' a durat 5 secunde; la revenirea dintr-un apel încuibat, nu s-a continuat 'sleep(5)' din apelul exterior, a.î. mesajele de terminare au apărut rapid unul după altul.

Dacă am asociat unui semnal un handler utilizator ce modifică niște variabile globale care influențează cursul ulterior al execuției programului, este foarte important să știm de câte ori și în ce momente ale execuției va veni semnalul respectiv (de asta depinde traseul execuției).

Totuși, la momentul scrierii programului, nu putem anticipa ce semnale va primi el pe parcursul execuției și în ce momente - asta se decide abia la run-time.

De aceea, în general asemenea programe sunt greu de scris, deoarece trebuie să avem în vedere toate cazurile posibile.

Variabilele globale modificate de handlerele utilizator asociate semnalelor e bine să fie declarate cu calificatorul 'volatile' - aceasta este o indicație către compilator că valoarea variabilei poate fi modificată oricând prin cod în afara domeniului de aplicare (scope) a codului curent.

Compilatorul interpretează această indicație prin aceea că nu optimizează accesarea variabilei respective în memorie; mai exact orice accesare / modificare a variabilei va fi făcută cu citirea și scrierea valorii ei curente în memorie (valoarea curentă nu va fi menținută mai mult timp în registri).

Exemplu fără 'volatile':

```
$cat prog.c
int x,y,z,t;
int a, b;
int main(){
    x=a; y=a; z=a;
    b=t; b=b+10; b=b+20;
    return 0;
}
```

```
$gcc -O3 -S -masm=intel prog.c
$cat prog.s
```

```
...
mov eax, DWORD PTR a[rip]
mov DWORD PTR x[rip], eax
mov DWORD PTR y[rip], eax
mov DWORD PTR z[rip], eax
mov eax, DWORD PTR t[rip]
add eax, 30
mov DWORD PTR b[rip], eax
xor eax, eax
ret
```

```
...
```

Comentarii:

- Cu opțiunea '-O3', 'gcc' va căuta să optimizeze puternic codul generat, a.î. vom putea face ușor diferența între modul în care este compilată accesarea variabilelor ne-vloatile (optimizat) și volatile (ne-optimizat); cu '-S', 'gcc' va produce un fișier de ieșire assembler, având extensia '.s'; cu '-masm=intel', 'gcc' va genera cod assembler cu sintaxa Intel (asemănătoare sintaxei NASM folosită într-un exemplu anterior).
- Compilatorul a constatat că de-a lungul secvenței 'x=a; y=a; z=a;' valoarea lui 'a' nu se schimbă și atunci a tradus optimizat secvența încărcând 'a' în registrul 'eax' și apoi salvând din 'eax' în 'x', 'y', 'z', fără a reîncărca de fiecare dată din 'a'; astfel, dacă vine un semnal în timpul secvenței iar handlerul modifică pe 'a', el va opera pe 'a' din memorie, nu din 'eax' (compilatorul nu poate anticipa momentul venirii semnalului și nu poate optimiza handlerul a.î. să opereze pe 'eax'), a.î. în 'x', 'y', 'z' va ajunge valoarea veche.
- Asemănător, compilatorul a constatat că de-a lungul secvenței 'b=t; b=b+10; b=b+20;' nu este nevoie de valoarea lui 'b' în altă parte, a.î. prin optimizare valoarea inițială a lui 'b' s-a încărcat în registrul 'eax', acolo s-a calculat valoarea finală iar aceasta a fost salvată în 'b' la sfârșit; astfel, dacă vine un semnal în timpul secvenței iar handlerul consultă pe 'b', el va opera pe 'b' din memorie, deci nu va consulta valoarea lui 'b' cea mai recentă, care este în 'eax'.

Exemplu cu 'volatile':

```
$cat prog.c
int x,y,z,t;
volatile int a, b;
int main(){
    x=a; y=a; z=a;
    b=t; b=b+10; b=b+20;
    return 0;
}
```

```
$gcc -O3 -S -masm=intel prog.c
$cat prog.s
```

```
...
mov eax, DWORD PTR a[rip]
mov DWORD PTR x[rip], eax
mov eax, DWORD PTR a[rip]
mov DWORD PTR y[rip], eax
mov eax, DWORD PTR a[rip]
mov DWORD PTR z[rip], eax
```

```
mov eax, DWORD PTR t[rip]
mov DWORD PTR b[rip], eax
mov eax, DWORD PTR b[rip]
add eax, 10
mov DWORD PTR b[rip], eax
mov eax, DWORD PTR b[rip]
add eax, 20
mov DWORD PTR b[rip], eax
xor eax, eax
ret
...
...
```

Comentariu: deși am folosit '-O3', 'gcc' a compilat accesarea variabilelor 'a', 'b' neoptimizat, a.î. fiecare consultare să se facă cu (re)încărcarea valorii curente din memorie iar fiecare modificare să se facă cu salvarea valorii curente în memorie; astfel, dacă vine un semnal între timp, are şanse mai mari să prindă valoarea curentă a lui 'a' sau 'b' în memorie (nu în regisztr).

Combinarea calificării 'volatile' cu 'const' are sens și este permisă de compilator - din cauza lui 'const', compilatorul nu va accepta instrucțiunile care modifică variabila, iar din cauza lui 'volatile', toate instrucțiunile care consultă variabila se vor compila ne-optimizat (cu (re)încărcarea valorii curente din memorie).

În schimb, calificarea 'volatile' este în contradicție cu 'register' și nu este permisă de copilator - într-adevăr, 'volatile' cere ca valoarea curentă a variabilei să fie păstrată cât mai mult în memorie iar 'register' cere ca valoarea curentă a variabilei să fie păstrată într-un registru.

Semnale

Putem adormi procesul în aşteptarea unui semnal cu apelurile 'pause()' sau 'sigsuspend()':

```
#include <unistd.h>
int pause(void);
```

⇒ procesul/threadul adoarme până primește un semnal neignorat (deci cu alt handler decat SIG_IGN) și neblocat ('sigprocmask()'); excepție: dacă semnalul este SIGCONT cu handlerul SIG_DFL, procesul rămâne adormit;

returnează după ce handler-ul semnalului a facut return - anume 'pause()' returnează -1 și setează errno = EINTR;

```
#include <signal.h>
int sigsuspend(const sigset_t *mask);
```

⇒ instalează temporar (până la ieșirea din apel) '*mask' ca mască de blocare a semnalelor (în locul celei curent folosite) și adoarme threadul până la primirea unui semnal neignorat și neblocat de '*mask'; acțiunea este ATOMICĂ (cele două efecte sunt realizate printr-o singură operație, deci nu există riscul să se trateze un semnal neignorat și neblocat de '*mask' între momentul instalării acestei măști și adormirea threadului);

la primirea unui semnal neignorat și neblocat de '*mask' comentariile și return-ul sunt ca la 'pause()'.

Exemplu: aşteptarea (INCORECTĂ a) unui semnal într-un anumit loc din program, folosind instrucțiuni separate de deblocare ('sigprocmask()') și intrare în aşteptare ('pause()'):

```
$cat prog.c
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
void h(int n) {signal(n, h);}
int main() {
    sigset_t ms; pid_t pid[10]; int i;
    signal(SIGTERM, h);
    sigemptyset(&ms); sigaddset(&ms, SIGTERM);
    sigprocmask(SIG_SETMASK, &ms, NULL);
    sigemptyset(&ms);
    for(i = 0; i < 10; ++i)
        if(pid[i] = fork()) {usleep(10); kill(pid[i], SIGTERM);}
        else {sigprocmask(SIG_SETMASK, &ms, NULL); pause();
              printf("Terminat %d\n", i);
              return 0;}
    sleep(1);
    for(i = 0; i < 10; ++i) kill(pid[i], SIGKILL);
    return 0;
}
```

```
$gcc -o prog prog.c
$./prog
Terminat 0
$./prog
Terminat 1
Terminat 9
```

Comentarii:

- De 10 ori, lansăm câte un proces copil căruia îi trimitem SIGTERM; vrem ca procesul copil să primească semnalul într-un anumit loc din program, cu 'pause()', și ca atare blocăm semnalul în afara locului de aşteptare (cu 'sigprocmask()' în părinte, blocajul se moștenește și în copil) și îl deblocăm imediat înaintea locului de aşteptare (cu 'sigprocmask()' în copil); din program, nu se poate garanta că cele două instrucțiuni se execută atomic, a.î. poate veni semnalul între ele - atunci, semnalul este tratat înainte de 'pause()' iar la 'pause()' se aşteaptă la infinit.
- Rulări diferite arată că unii dintre copii (nu întotdeauna aceiași) rămân blocați în 'pause()'.
- După lansarea copiilor, părintele aşteaptă o durată "rezonabilă" (deși nu sigură) de 1 secundă ca toți copii fie să se termine fie să se blocheze, iar în final trimit SIGKILL (semnal mortal ce nu poate fi blocat sau ignorat) tuturor pid-urilor de copii pe care le detine - semnalul va fi primit doar de copii rămași (blocați în 'pause()'), a.î. se vor termina ei. Procesele zombie rezultate vor fi adoptate și eliminate de 'init'.

Exemplu: aşteptarea (CORECTĂ a) unui semnal într-un anumit loc din program, folosind o singură instrucţiune ('sigsuspend()') pentru deblocare şi intrare în aşteptare, în mod atomic:

```
$cat prog.c
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
void h(int n) {signal(n, h);}
int main() {
    sigset(SIGTERM, h);
    sigemptyset(SIGTERM);
    if(fork() == 0) {
        sleep(1);
        for(i = 0; i < 10; ++i)
            if(pid[i] = fork()) {usleep(10); kill(pid[i], SIGTERM);}
            else {sigsuspend(SIGTERM);
                  printf("Terminat %d\n", i);
                  return 0;}
    }
    sleep(1);
    for(i = 0; i < 10; ++i) kill(pid[i], SIGKILL);
    return 0;
}
```

```
$gcc -o prog prog.c
$./prog
Terminat 0
Terminat 6
Terminat 5
Terminat 3
Terminat 9
Terminat 4
Terminat 1
Terminat 8
Terminat 7
Terminat 2
$./prog | wc -l
10
$./prog | wc -l
10
```

Comentariu: acum, toți cei 10 copii au primit semnalul în locul unde l-au așteptat (cu 'sigsuspend()') - la fiecare rulare, sunt afișate toate cele 10 mesaje (ordinea nu este fixată).

Dacă dorim ca două procese să comunice corect prin semnale, trebuie să asigurăm două condiții:

- Semnalele nu trebuie tratate în afara locului unde sunt așteptate - aceasta poate conduce la blocaje, ca în exemplele anterioare; în acest scop: în afara locului unde sunt așteptate, blocăm semnalele cu 'sigprocmask()' iar în locul unde sunt așteptate le deblocăm și intrăm în așteptare în mod atomic ('sigsuspend()').
- Emițătorul nu trebuie să trimită semnale cu o rată mai mare decât le tratează receptorul - altfel, poate veni un nou semnal în timp ce altul este încă în pending și atunci noul semnal se va pierde; în acest scop: implementăm un protocol între emițător și receptor, a.î. emițătorul să nu trimită un nou semnal până nu primește un semnal de confirmare de la receptor (că a terminat de tratat semnalul precedent).

Exemplu: Un proces generează un copil și îl trimite 2000 de semnale SIGUSR1, apoi un SIGUSR2, apoi așteaptă terminarea copilului și se termină; copilul, cu ajutorul unor handle, numără câte SIGUSR1 a primit iar la SIGUSR2 afișază numărul și se termină; pentru confirmare, după fiecare SIGUSR1, copilul trimite părintelui tot un SIGUSR1:

```
$cat prog.c
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
volatile int nr;
void f(int n){signal(n,f);}
void g(int n){signal(n,g); ++nr;}
void h(int n){printf("Primit: %d\n", nr); exit(0);}
```

```
int main(){
    pid_t p; sigset_t ms;
    sigemptyset(&ms); sigaddset(&ms,SIGUSR1); sigaddset(&ms,SIGUSR2);
    sigprocmask(SIG_SETMASK,&ms,NULL); sigemptyset(&ms);
    if(p=fork()){
        signal(SIGUSR1,f);
        for(nr = 0; nr < 2000; ++nr) {kill(p,SIGUSR1); sigsuspend(&ms);}
        kill(p, SIGUSR2);
        wait(NULL);
    }else{
        p=getppid(); nr = 0; signal(SIGUSR1, g); signal(SIGUSR2, h);
        while(1){sigsuspend(&ms); kill(p,SIGUSR1);}
    }
    return 0;
}
```

```
$gcc -o prog prog.c
$./prog
Primit: 2000
```

Funcții pentru gestionat timere software alocate la nivel de proces:

```
#include <sys/time.h>
int getitimer(int which, struct itimerval *curr_value);
int setitimer(int which, const struct itimerval *new_value,
              struct itimerval *old_value);
```

Consultă / setează un timer de interval la nivel de proces.

Un asemenea timer expiră inițial la un anumit moment în viitor și (optional) la intervale regulate după aceea. De fiecare dată când un timer expiră, procesul apelant primește un semnal iar timerul este resetat pentru a expira din nou la intervalul specificat (dacă intervalul este non-zero).

Există 3 tipuri de timere, fiecare urmărind un alt fel de timp și generând la expirare un al tip de semnal. Orice proces are câte un timer din fiecare tip.

Argumentul 'which' specifică timerul:

ITIMER_REAL : Măsoară timpul real; la fiecare expirare, generează semnalul **SIGALRM**.

ITIMER_VIRTUAL : Măsoară timpul CPU (timp consumat cu calcule, se omit timpii de așteptare) consumat în user mode de proces; la fiecare expirare, generează semnalul **SIGVTALRM**.

ITIMER_PROF = Măsoară timpul CPU total (i.e. consumat în user mode și kernel mode) de proces; la fiecare expirare, generează semnalul **SIGPROF**.

Valorile timerelor sunt definite prin următoarele structuri:

```
struct itimerval {  
    struct timeval it_interval; /* Interval for periodic timer */  
    struct timeval it_value;    /* Time until next expiration */  
};  
  
struct timeval {  
    time_t tv_sec;           /* seconds */  
    suseconds_t tv_usec;     /* microseconds */  
};
```

('time_t' și 'suseconds_t' sunt tipuri întregi definite în '<sys/types.h>', ultimul trebuie să suporte valori cel puțin în domeniul [-1, 1000000]).

Funcția 'getitimer()' furnizează în '*curr_value' valoarea curentă a timerului specificat de 'which':

- substructura 'it_value' va conține timpul rămas până la următoarea expirare a timerului; această valoare este cea care descrește odată cu trecerea timpului și se resetează la 'it_interval' atunci când timerul expiră. Dacă ambii membri ai lui 'it_value' sunt zero, atunci timerul este curent dezarmat (inactiv).
- substructura 'it_interval' va conține intervalul timerului. Dacă ambii membri ai lui 'it_interval' sunt zero, atunci timerul este single-shot (i.e. expiră doar o dată).

Funcția 'setitimer()' armează sau dezarmează timerul specificat de 'which'.

Timerul este setat la valoarea specificată de '*new_value'; dacă 'old_value' este non-NULL, în '*old_value' este furnizată valoarea anterioară a timerului (i.e. valoarea furnizată de 'getitimer()'); POSIX nu precizează ce se întâmplă dacă 'new_value' este NULL (în Linux, dacă 'new_value' este NULL, este ca și când câmpurile sale ar fi zero, iar aceasta înseamnă că timerul este dezarmat).

Dacă cel puțin un subcâmp din 'new_value -> it_value' este nonzero, atunci timerul este armat să expire inițial la timpul specificat; dacă ambele subcâmpuri din 'new_value -> it_value' sunt zero, timerul este dezarmat;

Câmpul 'new_value -> it_interval' specifică noul interval pentru timer; dacă ambele sale subcâmpuri sunt zero, timerul este single-shot.

Funcțiile 'getitimer()' și 'setitimer()' returnează zero în caz de succes, sau -1 (și setează 'errno') în caz de eșec.

Timerele interval nu se moștenesc la 'fork()' dar se transmit la 'execve()'.

Observații:

- După setarea unui timer, procesul nu așteaptă - el rulează în continuare iar timerul cronometrează în paralel; la expirarea inițială și la fiecare expirare la interval, procesul va primi câte un semnal; handlerul implicit al celor 3 semnale este terminarea, dar semnalele pot fi blocate și li se poate instala un alt handler.
- Momentele de expirare pot fi întârziate din cauza rezoluției timerelor și încărcării sistemului.
- Funcțiile 'alarm()' (a se vedea în continuare), 'sleep()', 'usleep()' sunt interfețe pentru 'setitimer()'.
- Dacă sistemul este foarte încărcat, un timer ITIMER_REAL poate expira înainte ca semnalul de la expirarea precedentă să fie tratat (delivered); atunci, al doilea semnal se pierde (din fiecare tip de semnal poate fi în pending doar câte un singur exemplar).

Exemplu: Pornim o numărătoare inversă care expiră inițial la 0.5 secunde și apoi la interval de 1 secundă; dacă până ajunge la 0 tastăm 'stop', ea se oprește cu mesajul 'Numaratoarea anulata.'; dacă nu tastăm 'stop' în timp util și ajunge la 0, se va afișa 'Gata !':

```
$cat prog.c
#include <sys/time.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void h(int n) {
    static int t = 5;
    signal(n, h);
    if(t) {printf("%d\n", t--); return;}
    printf("Gata !\n"); exit(0);
}
```

```
int main() {
    struct itimerval s1 = {{1, 0}, {0, 500000}}, s2 = {{0, 0}, {0, 0}};
    char buf[256];
    signal(SIGALRM, h);
    printf("Incepe numaratoarea inversa:\n");
    setitimer(ITIMER_REAL, &s1, NULL);
    do fgets(buf, 256, stdin); while(strcmp(buf, "stop\n") != 0);
    setitimer(ITIMER_REAL, &s2, NULL);
    printf("Numaratoarea anulata.\n");
    return 0;
}
```

```
$gcc -o prog prog.c
$./prog
Incepe numaratoarea inversa:
5
4
3
stop
Numaratoarea anulata.
$./prog
Incepe numaratoarea inversa:
5
4
3
2
1
Gata !
$
```

Putem programa mai ușor primirea de către procesul curent a unui semnal SIGALRM după un anumit interval de timp real, cu 'alarm()' (care este o interfață către 'setitimer()'):

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

Programează trimiterea unui semnal SIGALRM către procesul apelant după trecerea a 'seconds' secunde (apelul nu blochează procesul în aşteptarea semnalului, ci procesul își continuă execuția iar după trecerea intervalului de timp primește semnalul):

dacă anterior a mai fost efectuat un apel 'alarm()', el este anulat (numărătoarea descrescătoare a secundelor se reia de la 'seconds');

dacă apelam 'alarm()' cu parametrul 'seconds'=0 doar se anulează orice apel anterior;

returnează numărul de secunde rămase de numărat în urma apelului anterior, sau 0 dacă nu a existat un apel anterior.

Observații:

- Alarmele create cu 'alarm()' se transmit la 'execve()' dar nu sunt moștenite la 'fork()'.
- Apelul 'sleep()' poate fi implementat folosind SIGALRM; de aceea mixarea apelurilor 'alarm()' și 'sleep()' este contraindicată.
- Cu 'alarm()' se poate seta venirea alarmei după un număr întreg de secunde; cu 'setitimer()' putem seta durele cu număr fraționar de secunde.

Exemplu: folosire 'alarm()':

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

void h(int n){printf("Timpul a expirat\n"); exit(1);}

int main(){char c;
    signal(SIGALRM,h);
    printf("Asa e ? (d/n): ");
    alarm(10);
    scanf("%c",&c);
    alarm(0);
    printf("Multumesc pentru raspuns !\n");
    return 0;
}
```

Comentariu: după afişarea întrebării "Asa e ? (d/n): ", dacă utilizatorul răspunde în < 10 secunde, se afișază "Multumesc pentru raspuns !\n", alarma se anulează iar procesul se termină; dacă nu raspunde în timp util, procesul primește alarma, handler-ul afișază "Timpul a expirat\n" iar procesul se termină (nu mai este afișat "Multumesc...").

Câteva semnale importante și semnificațiile lor:

SIGHUP = 1

Dacă un terminal îintrerupe conexiunea (hangup) procesul lider al sesiunii asociate terminalului primește SIGHUP; când liderul de sesiune se termină, toate procesele aflate în grupul din foreground al sesiunii primesc SIGHUP.

Handler implicit: terminarea.

Astfel, când ne delogam (deci login shell-ul, care era lider de sesiune se termină), toate procesele lansate prin comenzi shell se termină automat.

Putem însă lansa un program 'prog' cu comanda 'nohup prog' și atunci el va fi 'vaccinat' la semnalul SIGHUP iar cand îl va primi nu se va termina - astfel putem lansa programe care să ruleze și după ce ne-am delogat (de exemplu de pe-o zi pe alta). Dacă intrarea standard a procesului este un terminal, ea este redirectată automat către un fișier ne-citibil; dacă ieșirea standard este un terminal, ea este redirectată automat către un fișier 'nohup.out' aflat în directorul curent (sau în directorul home, dacă cel curent nu oferă drept de scriere), cu excepția cazului când s-a indicat explicit o altă redirectare (de exemplu dăm: 'nohup prog > f'), caz în care se respectă această redirectare; dacă ieșirea standard pentru erori este un terminal, ea este redirectată automat către ieșirea standard.

SIGINT = 2, SIGQUIT = 3

Când tastăm de la un terminal Ctrl-c, respectiv Ctrl-\ (de fapt caracterele logice 'intr' și 'quit'), primul, respectiv al doilea semnal este trimis către toate procesele aflate în foreground la terminalul respectiv.

Handler implicit: terminarea (în cazul lui SIGQUIT se face și un fișier 'core' cu imaginea memoriei).

SIGKILL = 9

Este neblocabil și nu îi putem asocia alt handler decât cel implicit.

Handler implicit: terminarea.

Acest semnal este folosit de regulă pentru a termina explicit un proces dat.

SIGTERM = 15

Handler implicit: terminarea.

Este o alternativă la SIGKILL care poate fi blocat și i se poate schimba handlerul; cu SIGKILL forțăm terminarea unui proces, cu SIGTERM doar îi semnalăm că ar trebui să se termine. Este trimis implicit de comanda shell 'kill'.

SIGUSR1 = 10, SIGUSR2 =12

Handler implicit: terminarea.

Nu sunt folosite de sistem, sunt lăsate la dispoziția programatorilor.

SIGILL = 4 (instrucțiune ilegală)

SIGBUS = 7 (acces incorrect la memorie, de ex. acces la adrese nealiniate)

SIGFPE = 8 (floating point exception)

SIGSEGV = 11 (acces la memorie în afara spațiului de adrese alocate sau care încalcă permisiuni)

Handler implicit: terminarea și fișier 'core'.

SIGPIPE = 13

Scriere într-un fișier tub fără cititori (a se vedea secțiunea 'Tuburi').

Handler implicit: terminarea.

SIGALRM = 14

SIGVTALRM = 26

SIGPROF = 27

Handler implicit: terminarea.

Sunt folosite de program pentru a-și planifica (folosind 'setitimer()', 'alarm()') un comportament anume după un anumit interval de timp, indiferent de momentul unde se află cu execuția (cu ajutorul unui handler utilizator asociat semnalului).

SIGCHLD = 17

Handler implicit: ignorare

Când un proces se termină, părintele său primește un semnal SIGCHLD.

SIGSTOP = 19, SIGCONT = 18

Semnale folosite la suspendarea / reluarea explicită a unor procese (mecanisme de job control).

Handler implicit: suspendare, respectiv reluare.

SIGSTOP nu poate fi blocat și nu i se poate asocia alt handler decât cel implicit.

SIGTSTP = 20

Handler implicit: suspendare.

Este asemănător cu SIGSTOP, dar poate fi blocat și i se poate schimba handlerul.

Când tastăm de la un terminal Ctrl-z (de fapt caracterul logic 'susp'), semnalul este trimis către toate procesele aflate în foreground la terminalul respectiv ("terminal stop").

SIGTTIN = 21, SIGTTOU = 22

Este primit de procesele aflate în background la un terminal, atunci când încearcă să citească, respectiv. scrie, pe el.

Handler implicit: suspendare.

Toate semnalele de mai sus, în afară de SIGKILL și SIGSTOP, pot fi blocate și îi se poate schimba handlerul. Dacă lui SIGCONT îi instalăm un handler utilizator, efectul de trezire din suspendare se va păstra dar, în plus, după trezire, se va executa handlerul.

Cu următoarele funcții putem efectua un transfer nonlocal al controlului, în afara domeniului curent ("nonlocal goto"):

```
#include <setjmp.h>
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

'setjmp()' salvează contextul de execuție și stiva (în principiu registrul contor de program și regiștrii de stivă) în 'env'; tipul 'jmp_buf' este definit ca un tip vector (array), deci 'env' este de fapt un pointer către zona în care se vor salva aceste informații (parametru de ieșire); de asemenea, rezultă că entitățile declarate de tip 'jmp_buf' sunt nume de vectori, deci nu sunt lvalues (nu pot apărea în stânga atribuirii);

'longjmp()' restaurează contextul de execuție și stivă memorat în 'env'; aceasta face ca execuția să continue de la ieșirea din apelul 'setjmp()' care a salvat aceste informații; notăm că o entitate 'jmp_buf' memorează doar poziția pe stivă, nu și conținutul ei, deci dacă între timp s-au apelat și alte funcții cu aceeași zonă curentă a stivei, ele poate au suprascris conținutul din locul respectiv, a.î. la revenirea la ieșirea din 'setjmp()' execuția să nu se mai facă corect (inclusiv pot apărea erori de execuție fatale);

constatăm că într-un apel 'setjmp()' se poate intra/reveni (return) de mai multe ori: o dată prin apelarea normală și mai multe ori în urma unui 'longjmp()'; când se revine în urma apelului normal, 'setjmp()' returnează 0; când se revine în urma saltului cu un 'longjmp()', 'setjmp()' returnează valoarea parametrului 'val' al acelui 'longjmp()'; 'val' nu poate fi 0; dacă îl dăm 0, se va considera 1;

'longjmp()' nu returnează niciodată (dintr-un apel 'longjmp()' se sare direct într-un apel anterior al lui 'setjmp()' prin restaurarea contextului).

```
#include <setjmp.h>
int sigsetjmp(sigjmp_buf env, int savesigs);
void siglongjmp(sigjmp_buf env, int val);
```

Sunt similare lui 'setjmp()'/'longjmp()', cu următoarele diferențe:

- o entitate de tip 'sigjmp_buf' poate stoca nu numai un context de execuție și stivă, ci și o mulțime de semnale;
- dacă 'savesigs' este $\neq 0$, 'sigsetjmp()' salvează în 'env' și masca de semnale blocate a procesului;
- 'siglongjmp()' restaurează ce a fost stocat în 'env'; dacă apelul 'sigsetjmp()' a stocat acolo și masca de semnale blocate, o va restaura și pe aceasta, altfel va restaura doar contextul de execuție și stivă.

Existența acestor apeluri este datorată faptului că standardul POSIX nu specifică dacă 'setjmp()'/'longjmp()' salvează/restaurează și masca de semnale blocate (comportamentul poate diferi de la o implementare de UNIX/Linux la alta); dacă vrem să fim siguri că se salvează/restaurează și masca de semnale blocate, vom folosi 'sigsetjmp()'/'siglongjmp()'.

Observații:

- Funcțiile 'setjmp()' / 'longjmp()' pot fi utile pentru a trata erorile care apar în interiorul apelurilor de funcții îmbricate profund sau pentru a permite unui handler de semnal să transfere controlul unui anumit punct al programului, mai degrabă decât să revină la punctul în care handlerul a întrerupt programul principal.

În al doilea caz, am putea dori să restaurăm și masca de semnale blocate a programului principal, pentru a putea intercepta corect același semnal, dacă reappears (dacă handlerul din care am efectuat 'longjmp()' a blocat semnalul pe perioada executării lui, întrucât nu s-a ieșit din el prin 'return', semnalul a rămas blocat) - atunci, folosim 'sigsetjmp()' / 'siglongjmp()'.

- Compilatorul poate optimiza alocarea variabilelor în registri, iar 'longjmp()' poate restaura și valorile altor registri, în plus față de conținutul de program și registrii de stivă; atunci, valorile variabilelor automatice care îndeplinesc simultan criteriile următoare sunt nespecificate după un apel 'longjmp()':

- sunt locale pentru funcția care a efectuat apelul corespunzător 'setjmp()';
- valorile lor sunt modificate între apelurile 'setjmp()' și 'longjmp()';
- nu sunt declarate ca 'volatile'.

Comentarii similare sunt valabile și pentru 'siglongjmp()'.

- Dacă funcția care a apelat 'setjmp()' returnează înainte de a fi apelat 'longjmp()', comportamentul este nedefinit; de exemplu, dacă compilatorul implementează returnul din funcții prin stivă iar valoarea returnată are dimensiunea 'sizeof' mare, cadrul de apel al funcției poate fi translatat iar în locul respectiv în stivă să fie construită valoarea returnată, și atunci la revenirea cu 'longjmp()' contextul stivă este găsit alterat.
- Dacă, într-un program multithreading, un apel 'longjmp()' vizează un buffer 'env' care a fost inițializat printr-un apel 'setjmp()' efectuat într-un thread diferit, comportamentul este nedefinit.

Exemplu: gestionarea erorilor fatale (involuntare):

```
#include<setjmp.h>
#include<signal.h>
#include<stdio.h>
sigjmp_buf stare;
void h(int n){siglongjmp(stare,1);}
int main(){
    int n,*p=NULL;
    signal(SIGSEGV,h);
    if(!sigsetjmp(stare, 1))
        n=*p;
    else
        fprintf(stderr,"Eroare.\n");
    return 0;
}
```

Comentariu: la prima întâlnire a apelului 'sigsetjmp(stare, 1)' acesta salvează în 'stare' contextul de dinaintea operației riscante și returnează 0; atunci se intră pe ramura operației riscante 'n=*p;', aceasta generează o eroare fatală - accesarea ilegală a memoriei - procesul primește semnalul SIGSEGV, iar handler-ul asociat restaurează contextul salvat în 'stare'; atunci se întâlnește pentru a doua oară apelul 'sigsetjmp(stare, 1)', acum acesta returnează 1 (al doilea parametru al lui 'siglongjmp()') și astfel se intră pe ramura 'else'.

O problemă legată de handlerele utilizator asociate semnalelor este siguranța în raport cu semnalele (signal-safety).

O funcție este sigură în raport cu semnalele asincrone (async-signal-safe function) dacă poate fi apelată în siguranță dintr-un handler de semnal.

Un semnal poate întrerupe o funcție 'f()' iar handlerul să execute o funcție 'g()', a.î. la rulare se încuibă un apel 'g()' în apelul 'f()' într-un mod neanticipat la momentul scrierii programului și pot apărea probleme dacă 'g()' este apelat în anumite locuri ale execuției lui 'f()' - de exemplu, dacă ambele operează pe niște date comune, pot apărea inconsistențe, condiții de cursă, etc.

Funcțiile non-reentrantne nu sunt, evident, async-signal-safe.

Funcțiile descrise în '<stdio.h>' nu sunt async-signal-safe; într-adevăr, funcțiile de aici operează asupra fișierelor folosind buffere alocate static în programul utilizator și care au asociate contoare și indicatori de poziție curentă; astfel, dacă un apel 'printf()' este întrerupt de un handler care execută tot 'printf()', al doilea apel va opera pe date inconsistente în bufferul comun iar rezultatele sunt nonpredictibile.

Pentru a evita problemele cauzate de funcții nesigure, există două abordări:

- Să ne asigurăm că (a) handlerul de semnal apelează doar funcții `async-signal-safe` și (b) handlerul propriu-zis este reentrant în raport cu variabilele globale ale programului.
- Să blocăm în program semnalele atunci când apelam funcții care nu sunt sigure sau operăm pe date globale care sunt accesate și de handlerele de semnal.

În general, a doua abordare este mai dificilă și se preferă prima.

Funcțiile predefinite `async-signal-safe` sunt așa fie pentru că sunt reentrant, fie pentru că sunt atomice în raport cu semnalele (i.e. execuția lor nu poate fi întreruptă de un handler de semnal). Lista funcțiilor predefinite care sunt `async-signal-safe` se poate afla cu '`man 7 signal-safety`'.

Dacă un handler de semnal încearcă să blocheze o funcție nesigură, apoi handlerul se termină prin '`longjmp()`' sau '`siglongjmp()`', apoi programul apelează o funcție nesigură, comportamentul programului este nedefinit.

Apelurile 'setjmp()'/'longjmp()' (și generalizările lor 'sigsetjmp()' / 'siglongjmp()') se pot folosi pentru a implementa o formă de **corutine** - componente de program ce generalizează subrutele (proceduri, funcții) în sensul că permit existența mai multor puncte de intrare și ieșire dintr-un același apel, pentru întreruperea și reluarea execuției apelului în anumite locuri (iar la fiecare revenire se regăsesc datele locale apelului, cu aceleași valori); într-un apel de subrutină se intră/iese doar o dată, prin mecanismul de apel/revenire.

Practic, inserând în diverse locuri apeluri 'setjmp()'/'longjmp()', putem scrie funcții a.î.din interiorul apelului uneia să se sară în interiorul apelului celeilalte și invers, la fiecare revenire regăsindu-se variabilele locale automatice ale apelului cu aceleași valori. Funcțiile respective trebuie să fie însă apelate normal în prealabil, pentru a-și crea cadrul de apel pe stivă, și trebuie să avem grijă ca aceste cadre să nu fie suprascrise accidental - de ex. să începem salturile înainte ca din funcțiile respective să facem return, sau să avem grijă să nu fie apelate între timp alte funcții cu zonele respective din stivă ca zone curente (ca să-și creeze acolo cadrul de apel).

Prin asemenea mecanisme, putem emula multithreadingul și tratarea exceptiilor (din C++) acolo unde compilatorul și sistemul de operare nu ni le oferă în mod nativ. Exemple - TODO.

Cuprins

1 Sisteme de calcul, sisteme de operare

Organizarea unui sistem de calcul
Funcțiile sistemului de operare

2 Sisteme de operare uzuale

Incursiune în sistemul Windows
Incursiune în sistemul Linux

3 Interfața de programare C

Obiecte software și apeluri sistem
Utilizatori și grupuri

Fișiere și directoare

Procese

Semnale

Tuburi

Biblioteca standard C (tipul 'FILE')

4 Interfață interactivă linie de comandă

Interfața Command Prompt
(Windows)
Interfața Shell (Linux)

5 Interfață de scripting linie de comandă

Interfața Batch Scripting (Windows)
Interfața Shell Scripting (Linux)

6 Teme de laborator

Utilizatori și grupuri
Fișiere, directoare
Procese, semnale, tuburi

Tuburile sunt un tip de fișiere speciale, folosit la comunicarea între procese aflate în aceeași instanță SO.

Tuburile au caracteristicile generale ale unui fișier: i-nod cu informații despre el (proprietar, drepturi, etc.), număr de i-nod, poate fi accesat de procese prin descriptori, se poate opera asupra lui cu 'open()', 'close()', 'read()', 'write()', etc., dar au și caracteristici specifice:

- Tuburile oferă acces secvențial la informație, nu există un mecanism de adresare a informației, de tip file offset; nu se poate folosi 'lseek()'.
- Tuburile au o organizare de coadă (FIFO) - informația poate fi citită doar în ordinea în care a fost scrisă; citirea dintr-un tub este distructivă - odata citită, o informație dispare din tub.
- Comunicarea prin tub este de tip byte stream: mai multe procese pot scrie concurențial într-un tub, în tub informațiile vor apărea intercalate într-o secvență continuă de octeți, fără a se reține limitele pachetului scris de fiecare proces; o citire va furniza un prefix al acestei secvențe, fără se putea determina ce proces a scris fiecare octet.

- O cerere de scriere ('write()') în tub a \leq PIPE_BUF octeți va fi executată atomic - octeții vor apărea adiacent în tub (secvența contiguă); scrierea unui număr mai mare de octeți poate conduce la intercalarea lor cu cei scriși concurrent de alte procese. Constanta 'PIPE_BUF' este definită în '<limits.h>', POSIX.1 cere să fie \geq 512 octeți, în Linux este 4K.
- Un tub are capacitate limitată. De aceea, tubul poate ajunge să fie vid sau plin.

Capacitatea unui tub nu este standardizată, în Linux < 2.6.11 era de o pagină (ex. 4k), ulterior a devenit 16 pagini (ex. 64K, dacă paginile sunt 4K), iar de la Linux 2.6.11 a devenit consultabilă / setabilă: capacitatea unui tub indicat prin descriptorul 'fd', deschis în citire sau scriere, poate fi aflată ca valoare returnată de apelul 'fcntl(fd, F_GETPIPE_SZ)' și poate fi setată (între anumite limite) la 'arg' octeți cu apelul 'fcntl(fd, F_SETPIPE_SZ, arg)'; pentru funcția 'fcntl()' trebuie incluse '<unistd.h>' și '<fcntl.h>'.

Putem folosi apelul 'ioctl(fd, FIONREAD, &nbytes)' pentru a obține în variabila de tip int 'nbytes' numărul de octeți curent aflați în tubul indicat prin descriptorul întreg 'fd', deschis în citire sau scriere; pentru funcția 'ioctl()' trebuie inclus '<sys/ioctl.h>'.

- Ca și în cazul altor tipuri de fișiere, mai multe procese pot opera concurrent asupra unui tub, pentru citire, scriere, un proces poate avea mai mulți descriptori către un tub, unii în citire, alții în scriere, descriptorii diverselor procese către un tub pot folosi aceeași intrare TDF sau intrări diferite. Un proces care are tubul deschis în citire s.n. cititor din tub, un proces care are tubul deschis în scriere s.n. scriitor în tub; nu este nevoie ca procesele să citească sau să scrie efectiv, contează doar că au tubul deschis. În ce privește numărul curent de cititori și scriitori în tub, ceea ce contează sunt intrările TDF.

Citirea și scrierea într-un tub au anumite particularități față de cazul fișierelor obișnuite:

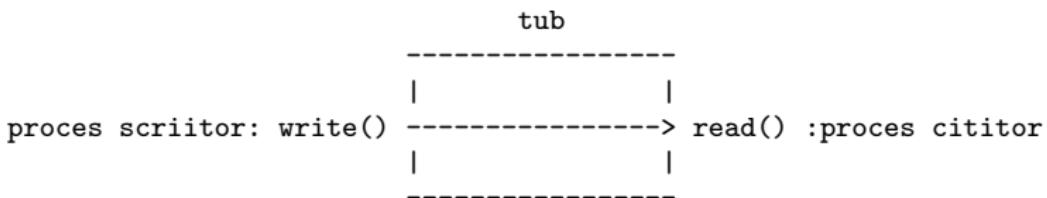
- citirea dintr-un tub nevid cu suficientă informație se finalizează imediat, cu citire date;
scrierea într-un tub ne-plin cu suficient loc și cu cititori se finalizează imediat, cu scriere date;
- citirea dintr-un tub vid (sau cu insuficientă informație):
 - dacă nu există scriitorii, se finalizează imediat, cu date citite parțial sau EOF;
 - dacă există scriitorii, apelantul se blocheaza până:
 - cineva scrie suficintă informație în tub și atunci operația se finalizează cu citire date;
 - tubul pierde scriitorii și atunci operația se finalizează cu date citite parțial sau EOF;

- scrierea într-un tub plin (sau cu insuficient loc):

- dacă nu există cititori, apelantul primește semnalul 'SIGPIPE'; handlerul implicit este terminarea procesului dar se poate instala un handler utilizator - atunci, doar cererea de scriere va eșua dar procesul va continua; notăm că orice încercare de a scrie într-un tub fără cititori se soldează cu 'SIGPIPE', nu este necesar ca tubul să fie plin;
- dacă există cititori, apelantul se blochează până:
 - cineva citește informație, creând suficient loc, și atunci operația se finalizează cu scriere date;
 - tubul pierde cititorii și atunci apelantul primește 'SIGPIPE'.

Aceste proprietăți permit ca două procese să comunice printr-un tub a.î. cel mai rapid să-l aștepte din când în când pe cel mai lent și fiecare proces să fie informat când partenerul lui a disparut - tubul trimite două feluri de feedback:

- dacă partenerul este prezent și prea lent iar tubul a ajuns într-un caz limită (citire din tub vid, scriere în tub plin), procesul va aștepta;
- dacă partenerul a disparut (a închis tubul), procesul își finalizează cererea imediat (cu citire parțială, EOF sau 'SIGPIPE') și poate lua o alta decizie (nu mai așteaptă).



Se poate seta a.î. citirea sau scrierea într-un tub să fie nonblocantă. Atunci, dacă în tub nu sunt date suficiente, citirea este parțială, iar dacă nu este loc suficient, tubul are cititori și se cere scrierea a $>$ 'PIPE_BUF' octeți, scrierea poate fi parțială; scrierea neblocantă a \leq 'PIPE_BUF' octeți reușește sau eșuează integral (nu există scriere parțială).

Notăm, de asemenea, că așteptarea în 'read()' / 'write()' poate fi întreruptă de un semnal.

Tuburile pot fi cu nume (legături fizice) și atunci persistă în sistem chiar dacă nu există procese care le țin deschise, sau pot fi anonime și atunci persistă în sistem doar cât timp există procese care le țin deschise (în momentul când au pierdut toți cititorii și scriitorii, sunt eliminate automat).

Tuburile cu nume sunt afișate de comanda 'ls -l' cu tipul 'p'.

De obicei, se folosesc termenii "FIFO" sau "named pipe" pentru a desemna tuburile cu nume și "pipe" pentru a desemna tuburile anonime.

Din program, putem crea un tub cu nume folosind funcția:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

Crează un FIFO cu specificatorul 'pathname' și cu drepturile 'permissions' - acestea se pot specifica folosind aceleași constante simbolice ca la 'open()' și sunt afectate de masca de drepturi 'umask' a procesului apelent în modul ușual: drepturile setate la fișier vor fi '(mode & ~umask)'.

Tubul creat poate fi operat în același fel ca fișierele obișnuite: descriptori, 'open()', 'close()', 'read()', 'write()', etc.; deschiderea poate fi pentru citire ('O_RDONLY') sau scriere ('O_WRONLY') (dacă există drepturile necesare).

Există următoarea particularitate:

- dacă un proces încearcă deschiderea unui tub cu nume care este închis, va fi blocat până când un proces va încerca deschiderea pentru operația duală; dacă, între timp, alte proceze încearcă deschiderea pentru același tip de operații, vor fi blocați și ele; când un proces va încerca deschiderea pentru operația duală, atât el cât și procezele care erau blocați își finalizează deschiderea și își continuă rularea în paralel; astfel, deschiderea unui tub închis poate fi un instrument de sincronizare între două proceze; aşteptarea în 'open()' poate fi întreruptă de un semnal;
- dacă un proces încearcă să deschidă un tub deja deschis, nu se blochează.

Un tub cu nume poate fi deschis cu `'open(pathname, O_RDONLY | O_NONBLOCK)'` sau `'open(pathname, O_WRONLY | O_NONBLOCK)'` și atunci nici deschiderea, nici citirea, resp. scrierea, prin descriptorul obținut nu vor fi blocante. De asemenea, citirea /scrierea se pot seta să fie neblocante ulterior deschiderii, folosind `'fcntl(fd, F_SETFL, O_NONBLOCK)'` (unde `'fd'` este descriptorul către tub).

Tubul cu nume peristă în sistemul de fișiere chiar și dacă nu este deschis de procese, deoarece are legături fizice, dar nu are salvat conținutul pe disc - când procesele comunică printr-un tub, toate datele sunt transmise prin memorie. În particular, dacă la închiderea tubului de către toate procesele mai erau octeți necititi în tub, aceștia se pierd (tubul rămâne cu dimensiune 0).

Funcția `'mkfifo()'` returnează 0 în caz de succes sau -1 (și setează `'errno'`) în caz de eșec.

Exemplu: presupunem că două procese "PS1" și "PS2" încearcă deschiderea tubului cu nume "tub" astfel:

PS1:

```
int dr,dw;  
...  
(1) dw=open("tub", O_WRONLY);  
...  
(2) dr=open("tub", O_RDONLY);
```

PS2:

```
int dr,dw;  
...  
(1') dr=open("tub", O_RDONLY);  
...  
(2') dw=open("tub", O_WRONLY);
```

Atunci, primul dintre procesele "PS1" și "PS2" care ajunge la (1), respectiv (1'), îl așteaptă pe celălalt; când ambele ajung acolo, realizează apelul 'open()' simultan apoi merg mai departe independent; în punctul (2), respectiv (2'), ele nu se mai așteaptă, deoarece în acel moment tubul are deja o deschidere pentru operația duală.

Exemplu: talk intre mai multi utilizatori, folosind tuburi cu nume:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>
#include <limits.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <setjmp.h>

sigjmp_buf jb;
void h(int n) {signal(n, h); siglongjmp(jb, 1);}
int main(){
    char tubp[11], tuba[11], buf[PATH_MAX], *mes, chr;
    uint8_t pachet[PIPE_BUF];
    int dp, da[10], na, len, i;
    pid_t pid;

    printf("Dati tubul propriu (max.10 car.): ");
    fgets(buf, PATH_MAX, stdin); sscanf(buf, "%10s", tubp); fflush(stdin);
    if(mkfifo(tubp, S_IRUSR | S_IWUSR | S_IWGRP | S_IWOTH) == -1)
        {perror(tubp); return -1;}

    strcpy((char *)pachet, tubp); strcat((char *)pachet, ": ");
    len = (PIPE_BUF - strlen((char *) pachet)) * sizeof(char) / sizeof(char);
    mes = (char *) (pachet + strlen((char *) pachet) * sizeof(char));
```

```

if(fork()) dp = open(tubp, O_RDONLY); else {dp = open(tubp, O_WRONLY); return 1;}
open(tubp, O_WRONLY);
printf("Tubului propriu este: %s\n", realpath(tubp, buf));

printf("Dati tuburile adverse (max.10), unul pe linie, o linie goala la sfarsit:\n");
for(na = 0; na < 10; ++na) {
    fgets(buf, PATH_MAX, stdin); if(strcmp(buf, "\n") == 0) break;
    sscanf(buf, "%10s", tuba); fflush(stdin);
    if((da[na] = open(tuba, O_WRONLY)) == -1) {perror(tuba); --na;}
}
printf("Incepe conversatia:\n");

if(pid = fork()){
    signal(SIGPIPE, h);
    while(na && fgets(mes, len, stdin) != NULL) {
        fflush(stdin);
        if(mes[strlen(mes) - 1] != '\n') mes[strlen(mes) - 1] = '\n';
        for(i = 0; i < na; ++i)
            if(sigsetjmp(jb, 1)) {close(da[i]); --na; if (i < na) {da[i] = da[na]; --i;}}
            else write(da[i], pachet, strlen((char *) pachet) * sizeof(char));
    }
    kill(pid, SIGKILL); while(wait(NULL) != -1); unlink(tubp);
} else {
    while(1) {read(dp, &chr, sizeof(char)); write(1, &chr, sizeof(char));}
}
return 0;
}

```

Comentarii:

- Se lansează de la mai multe terminale; se introduce numele tubului propriu, apoi numele tuburilor adverse, ale celor care și-au introdus deja numele tuburile proprii (pentru ca tuburile să fie deja create); este bine să folosim nume sugestive (de exemplu, nume de persoane), deoarece vor fi afișate la începutul mesajelor; numele pot avea și cale.

Apoi are loc comunicarea. Când cineva dorește să încheie, tastează Ctrl-d.

- Din tubul propriu se citește, în tuburile adverse se scrie. Tubul propriu este menținut deschis și în citire și în scriere, pentru a preveni confuzia între situația când toți au încheiat comunicarea sau nimeni încă nu a început. Pentru a împiedica autoblocajul, prima deschidere a tubului propriu este efectuată cu două procese paralele.

- Fiecare linie citită este scrisă în 'pachet', împreună cu numele expeditorului și trimisă în toate tuburile adverse; pentru ca liniile să nu fie intercalate, 'pachet' are max. 'PIPE_BUF' octeți.

- Procesul se bifurcă părinte - copil, părintele citește linii de la standard input și le scrie în tuburile adverse rămase deschise în citire, copilul afișază la standard output conținutul primit prin tubul propriu.

- Părintele detectează când unul din tuburile adverse a fost închis la citire folosind un handler instalat pentru 'SIGPIPE'; handlerul restaurează contextul de dinaintea scrierii eşuate, folosind 'siglongjmp()'; nu ar fi fost suficient 'longjmp()', deoarece 'signal()' instalează handlerul a.î. pe perioada apelului 'SIGPIPE' este blocat, iar un 'longjmp' efectuat din apel nu ar fi restaurat masca de semnale blocate anteroară - 'SIGPIPE' ar fi rămas blocat și nu s-ar mai fi detectat alte închideri ale tuburilor adverse.

Dacă se tastează Ctrl-d (EOF) sau nu mai sunt tuburi adverse, părintele termină copilul ('kill()') și elimină tubul propriu ('unlink()').

- Copilul afișază în contiuu conținutul venit prin tubul propriu, până este terminat de părinte; citirea din tubul propriu nu se va finaliza niciodată cu EOF, deoarece tubul are cel puțin un scriitor - procesul curent; întrucât pachetele scrise în tuburi se termină cu '\n', afișarea se va face pe linii, fără cod suplimentar.

Din linia de comandă shell, putem crea un tub cu nume folosind comanda `'mkfifo [--mode=drepturi] specificator'` ('specificator' este numele cu cale, 'drepturi' se poate construi simbolic ca la comanda 'chmod', prin lipsă se vor seta drepturile 'a=rw & ~umask', unde 'umask' este masca de drepturi a shell-ului).

Exemplu: comunicarea între două terminale, folosind comenzi 'cat' și un tub cu nume:

Lansăm două procese shell la terminale (ferestre) diferite, având același director curent; vom numi terminalele '/dev/pts/0', '/dev/pts/1'; dăm în ele următoarele comenzi:

Terminalul '/dev/pts/0':

```
| $tty
| /dev/pts/0
| $cd /home/dragulici/Desktop/work
| $mkfifo tub
| $cat > tub
| abcd
| ad
| a123g
^d->$rm tub
| $
```

Terminalul '/dev/pts/1':

```
| $tty
| /dev/pts/1
| $cd /home/dragulici/Desktop/work
| $cat < tub
| abcd
| ad
| a123g
| $
|
```

Comentarii:

- Comenzile 'tty' și 'cd /home/dragulici/Desktop/work' pot fi date de la cele două terminale în orice ordine și prin ele aflăm numele terminalelor și setăm același director curent.
- Apoi, de la 'dev/pts/0', dăm comenziile 'mkfifo tub' și 'cat > tub'.
- Apoi, de la 'dev/pts/1', dăm comanda 'cat < tub'.
- Apoi, de la 'dev/pts/0', tastăm diverse linii de text, de ex. 'abcd', 'ad', 'a123g' și observăm că ele sunt afișate la 'dev/pts/1' pe masură ce sunt introduse (cele două procese 'cat' comunică prin tubul 'tub').
- Apoi, de la 'dev/pts/0', tastăm Ctrl-d, ceea ce va semnala procesului 'cat' EOF; acesta, ieșind din ciclul citire - scriere, se va termina iar tubul 'tub' va pierde singurul scriitor. Atunci, după ce procesul 'cat' de la 'dev/pts/1' va golii tubul, se va afla în citire din tub vid fără scriitor; ca atare, citirea îi va semnala EOF și, ieșind din ciclul citire - scriere, se va termina și el.
- În final, de la unul dintre terminale, ștergem tubul cu 'rm tub'.

Am dat mai înainte un exemplu de implementare a unui shell care suportă comanda internă 'exit' și comenzi externe cu mai multe argumente și mai multe redirectări '<', '>', '>>', '>2', '>2>'.

Am văzut acolo că, în cazul unei comenzi cu redirectări, shell-ul copil deschide fișierele spre care se face redirectarea, apoi lansează comanda cu 'exec()' iar aceasta moștenește redirectările respective.

Dacă o redirectare se face într-un tub, shell-ul va fi blocat în deschidere până când un alt proces, de exemplu tot un shell, va încerca să deschidă tubul pentru operația duală, apoi ambele procese vor continua în paralel; în particular, comanda externă nu va fi lansată cu 'exec()' până când shell-ul copil nu va finaliza deschiderea.

Așa s-a întâmplat și în exemplul de mai sus - un shell a executat 'cat > tub', alt shell a executat 'cat < tub', ambele au generat câte un shell copil care a încercat să deschidă tubul 'tub', unul în citire, altul în scriere, și nici unul dintre shell-urile copil nu s-a substituit cu 'cat' până când celălalt nu a finalizat deschiderea.

Aceasta se poate observa dacă modificăm exemplul de mai sus în felul următor:

Exemplu:

Terminalul '/dev/pts/0':

```
| $tty  
| /dev/pts/0  
| $cd /home/dragulici/Desktop/work  
| $mkfifo tub  
| $cat > tub  
| abcd  
| ad  
| a123g  
^d->$rm tub  
| $
```

Terminalul '/dev/pts/1':

```
| $tty  
| /dev/pts/1  
| $cd /home/dragulici/Desktop/work  
| $cat < tub  
| abcd  
| ad  
| a123g  
| $
```

Terminalul '/dev/pts/2':

```
| $ps -t /dev/pts/0 -o ppid,pid,cmd |  
|     PPID      PID CMD  
|     3980      3987 bash  
|     3987      12194 bash  
| $ps -t /dev/pts/0 -o ppid,pid,cmd |  
|     PPID      PID CMD  
|     3980      3987 bash  
|     3987      12194 cat  
| $
```

Comentarii:

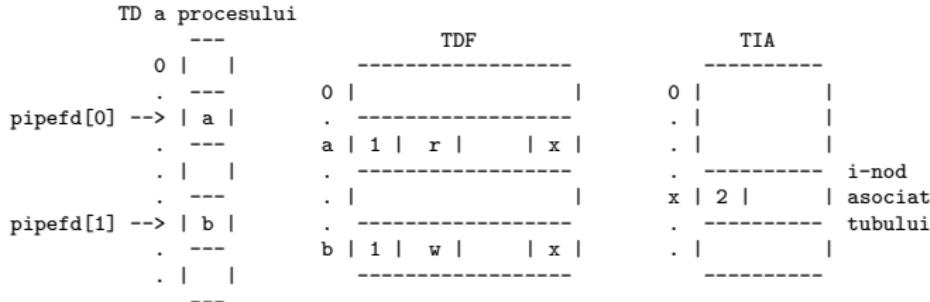
- Lansăm trei procese shell la terminale (ferestre) diferite și vom numi terminalele '/dev/pts/0', '/dev/pts/1', '/dev/pts/2'.
- În '/dev/pts/0' dăm comenziile 'tty', 'cd /home/dragulici/Desktop/work', 'mkfifo tub', 'cat > tub' (în continuare, procesul aşteaptă).
- Apoi, de la '/dev/pts/1' dăm comenziile 'tty', 'cd /home/dragulici/Desktop/work' (încă nu dăm 'cat < tub').
- Apoi, de la '/dev/pts/2' dăm comanda 'ps -t /dev/pts/0 -o ppid,pid,cmd' și observăm că la terminalul '/dev/pts/0' sunt două procese shell, aflate în relația părinte - copil; deci, acolo încă nu a început 'cat' (există încă shell-ul copil, care n-a efectuat 'exec()').
- Apoi, de la '/dev/pts/1', dăm comanda 'cat < tub'.
- Apoi, de la '/dev/pts/2' dăm din nou comanda 'ps -t /dev/pts/0 -o ppid,pid,cmd' și observăm că la terminalul '/dev/pts/0' shell-ul copil s-a înlocuit cu 'cat' (deci, a trecut de 'open()' și a efectuat 'exec()').
- În continuare, de la '/dev/pts/0' se introduc liniile de text 'abcd', 'ad', 'a123g', se tastează Ctrl-d și se dă comanda 'rm tub', ca în exemplul anterior.

Din program, putem crea un tub anonim cu functiile:

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

Crează un tub anonim și îl deschide de două ori (allocă două intrări în TDF): o dată pentru citire și o dată pentru scriere; în procesul apelant, i se alocă doi descriptori: unul asociat deschiderii pentru citire și care va fi stocat în 'pipefd[0]', altul asociat deschiderii pentru scriere și care va fi stocat în 'pipefd[1]'; parametrul 'pipefd' este implementat de fapt sub forma 'int *pipefd'.

Funcția returnează 0 în caz de succes sau -1 (și setează 'errno') în caz de eșec.
Începând cu POSIX.1-2016 și în Linux, se menționează că, în caz de eșec, funcția 'pipe()' nu va modifica zona pointată de 'pipefd'.



Deci, un tub deschis ca mai sus are deodată un cititor și un scriitor (ambii fiind procesul însuși) iar apelul 'pipe()' nu blochează; ulterior, numărul proceselor cititor sau scriitor în tubul respectiv se poate modifica (în plus sau minus), cu 'dup()', 'fork()', 'close()', etc.

Tubul, fiind anonim (fără legături fizice), va exista doar atât timp cât va avea cititori și / sau scriitori; când tubul va pierde toți cititorii și scriitorii, va fi eliminat automat.

Datele scrise în tub sunt menținute în memorie până sunt citite de un proces (nu sunt salvate pe disc). Dacă tubul mai are date în el atunci când a pierdut toți cititorii și scriitorii, el va fi eliminat iar datele respective se vor pierde.

Observatii:

- pe anumite arhitecturi (Alpha, IA-64, MIPS, SuperH, SPARC/SPARC64), funcția 'pipe()' este implementată sub forma 'struct fd_pair pipe();', și returnează descriptorii sub forma unei structuri de tip 'struct fd_pair {long fd[2];};'
- pe langă funcția 'pipe()', care este standardizată POSIX, există și funcția specifică Linux 'pipe2()', care permite crearea unui tub anonim specificând un anumit mod în care se face citirea și scrierea - de exemplu, apelul 'pipe2(pipefd, O_NONBLOCK)' crează tubul cu operațiile neblocante; 'pipe2(pipefd, 0)' este echivalent cu 'pipe(pipefd)'; în absența funcției 'pipe2()', pentru operații neblocante se poate folosi 'fcntl()'.

Când lucrăm cu tuburi (cu nume sau anonime) există riscul de a apărea autoblocaj al unui proces sau interblocaj între mai multe procese:

Exemplu (autoblocaj):

```
int p[2]; char buf[10];
pipe(p);
read(p[0], buf, 1);
/* ... */
write(p[1], buf, 1);
```

Aici are loc un autoblocaj: când se ajunge la 'read()', tubul e vid și nu are alt scriitor decât procesul însuși iar acesta doarme în 'read()', a.î. nu mai ajunge la 'write()' unde ar fi scris ceva în tub.

Exemplu (autoblocaj):

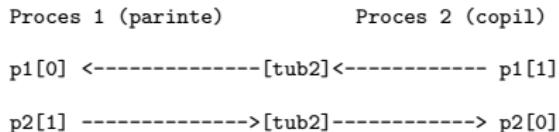
```
int p[2]; char buf[10];
pipe(p);
while(read(0, buf, 1)) write(p[1], buf, 1);
/* ... */
while(read(p[0], buf, 1)) write(1, buf, 1);
```

Dacă de la standard input provin mai mulți octeți decât capacitatea tubului, la un moment dat procesul se blochează la scrierea în tubul plin dar cu cititor - procesul însuși este și cititor în tub, prin 'p[0]'.

Exemplu (interblocaj):

```
int p1[2], p2[2]; char buf[10];
pipe(p1); pipe(p2);
if(fork()) {read(p1[0],buf,1); /* ... */ write(p2[1],buf,1);}
else {read(p2[0],buf,1); /* ... */ write(p1[1],buf,1);}
```

Aici, două procese încearcă să comunice prin două tuburi, fiecare tub corespunzând unui sens de circulație a informației:



Procesul inițial crează și deschide ambele tuburi, alocând patru descriptori; după 'fork()', procesul copil moștenește cei patru descriptori, aşa că fiecare tub are acum doi cititori și doi scriitori (deși fiecare proces folosește efectiv doar doi descriptori, prezentați în desenul de mai sus); în continuare, ambele procese ajung la 'read()' înainte de a apuca să scrie ceva în tubul "celalalt" și adorm pe o perioadă infinită - fiecare așteaptă ca celălalt să scrie în tubul din care el trebuie să citească (tuburile sunt în acest moment vide, dar au scriitori, fiecare proces are descriptori în scriere pe ambele tuburi).

O recomandare pentru a evita asemenea blocaje/interblocaje este ca tot timpul să fie păstrați deschisi pe tub doar descriptorii folosiți (pe ceilalți să-i închidem) - atunci, unele adormiri în 'read()' la tubul vid se pot sfârși (cu EOF) prin faptul tubul nu mai are scriitori, iar unele adormiri în 'write()' la tubul plin se pot sfârși (cu 'SIGPIPE') prin faptul că tubul nu mai are cititori.

Tuburile anonime sunt folosite de shell pentru implementarea comenziilor filtru 'comanda_1 | ... | comanda_n':

- shell-ul crează n - 1 tuburi anonime, apoi generează n copii shell (cu 'fork()'), apoi își închide descriptorii pe tuburi (nu îi folosește) și, în absența lui '&', așteaptă terminarea copiilor;
- fiecare copil shell moștenește descriptori pe toate tuburile dar își redirectează standard input la tubul "din stanga", își redirectează standard output la tubul "din dreapta", își închide ceilalți descriptori pe tuburi și se substituie (cu 'exec()') cu un proces ce execută comanda 'comanda_i' corespunzătoare;
- în final, cele n comenzi sunt copii ai shell-ului părinte, rulează în paralel și comunică prin tuburi, iar fiecare tub are un singur cititor (comanda "din dreapta") și un singur scriitor (comanda "din stanga");
- când ambele procese - comenzi care sunt la capetele unui tub se termină, tubul pierde singurul cititor și singurul scriitor și atunci (neavând nume) este eliminat automat (nu trebuie eliminat explicit).

Exemplu: shell care suportă comanda internă 'exit' și comenzi externe de forma 'comanda', 'comanda > fisier', 'comanda1 | comanda2', unde 'comanda', 'comanda1', 'comanda2' sunt specificatorii unor programe (fișiere executabile), fără argumente în linia de comandă:

```
$cat mysh.c
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<stdio.h>
#include<string.h>

char ldc[256], c0[256], c1[256], c2[256], *a[256];

int main(){
    while(1){
        printf(">>");
        fgets(ldc, 256 * sizeof(char), stdin);
        c0[0]=c1[0]=c2[0]=0;
        sscanf(ldc,"%s%s%s",c0,c1,c2);
        if(!strcmp(c0,"exit")){
            return 0;
        }
    }
}
```

```
else if(!strcmp(c1,"|")){
    int d[2];
    pipe(d);
    if(!fork()){
        close(1); dup(d[1]); close(d[0]); close(d[1]);
        a[0]=c0; a[1]=NULL;
        execv(c0,a);
        perror(c0);
        return 1;
    }else if(!fork()){
        close(0); dup(d[0]); close(d[0]); close(d[1]);
        a[0]=c2; a[1]=NULL;
        execv(c2,a);
        perror(c2);
        return 1;
    }else{
        close(d[0]); close(d[1]); while(wait(NULL)!=-1);
    }
}else if(fork()){
    wait(NULL);
}else{
    if(!strcmp(c1,>"")){
        int d;
        if((d=open(c2,O_WRONLY|O_CREAT|O_TRUNC,S_IRUSR|S_IWUSR))==-1)
            {perror(c2); return 1;}
        close(1); dup(d); close(d);
    }
    a[0]=c0; a[1]=NULL;
    execv(c0,a);
    perror(c0);
    return 1;
}
return 0;
}
```

```
$gcc -o mysh mysh.c
$./mysh
>>/bin/pwd
/home/dragulici/Desktop/work
>>/bin/pwd | /bin/wc
      1      1     29
>>/bin/pwd > out.txt
>>exit
$cat out.txt
/home/dragulici/Desktop/work
```

Comentarii:

- Cu 'sscanf()' se citesc din linia de comandă 'ldc' maxim 3 cuvinte, în 'c0', 'c1', 'c2' ('sscanf()' recunoaște cuvintele delimitate de spații); dacă 'ldc' conține mai puțin de 3 cuvinte, ultimile dintre 'c0', 'c1', 'c2' rămân cu valoarea anterioară, care este stringul vid (setat cu 'c0[0]=c1[0]=c2[0]=0;').

În cazul unei comenzi de forma 'comanda1 | comanda2':

- Shell-ul inițial crează un tub anonim și lansează doi copii shell.

Procesele copil moștenesc descriptorii părintelui pe tub, a.î. tubul ajunge să aibă 3 cititori și 3 scriitori; ulterior, primul copil își redirektează standard output la tub, al doilea copil își redirektează standard input la tub și toate procesele își închid descriptorii nefolosiți pe tuburi, a.î. în final tubul are un singur cititor - al doilea copil shell, prin standard input, și un singur scriitor - primul copil shell, prin standard output; în continuare, copiii shell se înlocuiesc cu respectiv comenziile 'c0', 'c2', a.î. acestea devin copiii shell-ului părinte și singurul scriitor, respectiv cititor, în tub.

Când 'c0' și 'c2' se termină, tubul pierde singurul cititor și singurul scriitor și, neavând nume, este eliminat automat (nu este nevoie să fie eliminat explicit)

- Shell-ul părinte așteaptă terminarea copiilor 'c0' și 'c2' cu 'while(wait(NULL)!=-1);' (în care face doar 3 iterații: la primele două adoarme până se mai termină un copil, la a treia 'wait()' returneaza -1).

- Comportamentul tubului în cazurile limită - citirea din tub vid și scrierea în tub plin - de a adormi un proces dacă există deschidere la operația duală și de a semnala imediat, în mod diferit, dacă nu mai există deschidere la operația duală, permite ca procesele 'c0' și 'c2' să se aștepte unul pe altul, dacă nu rulează cu aceeași viteză, și să afle când celalalt a disparut, pentru a lua o altă decizie.
- Dacă nu închideam descriptorii nefolosiți pe tuburi, puteau apărea interblocaje.

De exemplu, dacă părintele nu efectua 'close(d[0]);' iar 'c2' se termina primul, atunci 'c0' putea ajunge să umple tubul și să se afle în scriere în tub plin dar cu cititor - părintele; scriitorul 'c0' ar fi așteptat în 'write()' ca părintele să citească, părintele ar fi așteptat în 'wait()' ca copilul 'c0' să se termine și ar fi apărut un interblocaj între părinte și copilul 'c0'.

Similar, dacă părintele nu efectua 'close(d[1]);' iar 'c0' se termina primul, atunci 'c2' putea ajunge să golească tubul și să se afle în citire din tub vid dar cu scriitor - părintele; cititorul 'c2' ar fi așteptat în 'read()' ca părintele să scrie, părintele ar fi așteptat în 'wait()' ca copilul 'c2' să se termine și ar fi apărut un interblocaj între părinte și copilul 'c2'.

De asemenea, dacă primul copil nu ar fi închis 'd[0]' sau al doilea copil nu ar fi închis 'd[1]', atunci 'c0', respectiv 'c2', ar fi putut ajunge să se autoblocheze, așteptându-se pe sine în 'write()', respectiv 'read()', să facă operația duală.

- Deoarece am folosit o forma de 'exec()' fără 'p' (deci, care nu folosește 'PATH'), utilitarele 'pwd' și 'wc' au trebuit apelate cu cale.

Comanda 'cat out.txt' cu care am afișat fișierul rezultat în urma comenzi '"/bin/pwd > out.txt' nu a putut fi dată din shell-ul nostru, deoarece acesta nu suportă comenzi externe cu argumente.

Exemplu: program care primește ca argument în linia de comandă rădăcina unei arborescențe a sistemului de fișiere și afișază numărul obiectelor din arborescența respectivă al căror proprietar este proprietarul efectiv al procesului:

```
$cat prog.c
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <dirent.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
```

```

int count(uid_t u, char const *nd) {
    struct stat s; DIR *pd; struct dirent *pde; int d[2];
    int n, e = errno, x;
    if(stat(nd, &s) == -1) {n = -1; e = errno; goto err1;}
    n = s.st_uid == u;
    if(! S_ISDIR(s.st_mode)) goto err1;
    if((pd = opendir(nd)) == NULL) {n = -1; e = errno; goto err1;}
    if(pipe(d) == -1) {n = -1; e = errno; goto err2;}
    while((pde = readdir(pd)) != NULL)
        if(strcmp(pde->d_name, ".") && strcmp(pde->d_name, ".."))
            switch(fork()) {
                case -1: close(d[1]); n = -1; e = errno; goto err3;
                case 0: if((n = chdir(nd)) == -1) perror(nd);
                          else if((n = count(u, pde->d_name)) == -1) {
                              e = errno;
                              fprintf(stderr, "%s/%s: %s\n", nd, pde->d_name, strerror(e));
                          }
                          write(d[1], &n, sizeof(int)); exit(2);
            }
    close(d[1]);
    while(read(d[0], &x, sizeof(int)) == sizeof(int))
        if(x == -1) {n = -1; e = ESPIPE; goto err3;}
        else n += x;
err3: close(d[0]); while(wait(NULL) != -1);
err2: closedir(pd);
err1: errno = e;
        return n;
}

```

```
int main(int argc, char *argv[]) {
    int n;
    if(argc != 2) {
        fprintf(stderr, "Utilizare: %s director\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if((n = count(geteuid(), argv[1])) == -1) {
        perror(argv[1]); exit(EXIT_FAILURE);
    }
    printf("Ai %d fisiere\n", n);
    return EXIT_SUCCESS;
}
```

```
$gcc -o prog prog.c
$ls -allR a
a:
total 24
drwxrwxr-x 3 dragulici dragulici 4096 Dec 30 01:48 .
drwxrwxr-x 6 dragulici dragulici 4096 Jan  6 00:14 ..
drwxrwxr-x 2 dragulici dragulici 4096 Dec 10 02:41 b
-rw----- 1 dragulici dragulici 148 Dec 10 02:32 ff
-rw-rw-r-- 1 dragulici dragulici 148 Dec 10 02:32 fg
-rw-r--r-- 1 root      root     2750 Nov  3  2020 p
```

```
a/b:
total 8
drwxrwxr-x 2 dragulici dragulici 4096 Dec 10 02:41 .
drwxrwxr-x 3 dragulici dragulici 4096 Dec 30 01:48 ..
-rw-rw-r-- 1 dragulici dragulici     0 Dec   4 17:50 g
$./prog a
Ai 5 fisiere
```

Comentarii:

- Funcția 'count()' calculează progresiv valoarea de returnat 'n' (-1 sau numărul de obiecte) și o valoarea furnizată în 'errno', a.î., dacă funcția 'count()' eșuează din cauza unei funcții predefinite care setează 'errno', ea să indice prin 'errno' motivul eșecului acestei funcții - astfel, apelantul va putea afișa cu ' perror()' motivul corect al eșecului lui 'count()'.
- Dacă argumentul funcției 'count()' este un director, îl parcurge și, pentru fiecare intrare diferită de '.' și '..', lansează un proces copil care numără obiectele cu proprietarul dorit din subarborescența care începe cu acea intrare și comunică părintelui numărul respectiv sau -1, printr-un tub; procesele copil rulează în paralel; procesul părinte adună valorile citite din tub la 'n' iar dacă citește -1, va returna -1.
- Dacă un proces copil detectează o eroare (apelul 'count()' efectuat returnează -1), afișază el eroarea cu ' perror()' - nu raportează părintelui valoarea 'errno' ci doar pe -1 (altfel, părintele ar putea primi prin tub mai multe valori 'errno' diferite și afișarea ar fi confuză); dacă însă părintele citește din tub -1, returnează mai departe -1 și furnizează o valoare 'errno' convențională (am ales 'ESPIPE').

- Dacă părintele nu efectuează 'close(d[1])' înainte de citirea din tub, s-ar putea autobloca (copii s-ar termina și nu ar mai scrie în tub, părintele ar goli tubul, apoi s-ar afla în citirea din tub vid dar cu scriitor, el însuși).
- Parintele efectuează 'while(wait(NULL) != -1);' pentru a nu lăsa în sistem copii terminați, ca zombie; închiderea 'close(d[0]);' pe care o face înainte este importantă dacă a ieșit din 'while(read...)' înainte de terminarea copiilor (pentru că a citit -1) - fără aceasta, copii rămași ar fi putut umple tubul din care părintele nu mai citea (dar îl păstra deschis în citire) și ar fi adormit în scrierea în tub plin cu cititor; atunci, părintele nu ar fi putut trece de 'while(wait(NULL) != -1);' (interblocaj). După 'close(d[0]);' efectuat de părinte, dacă mai sunt copii neterminați, ei vor încerca să scrie în tubul rămas fără cititor și se vor termina cu 'SIGPIPE'.

Cuprins

- 1 Sisteme de calcul, sisteme de operare
 - Organizarea unui sistem de calcul
 - Funcțiile sistemului de operare
- 2 Sisteme de operare uzuale
 - Incișiune în sistemul Windows
 - Incișiune în sistemul Linux
- 3 Interfața de programare C
 - Obiecte software și apeluri sistem
 - Utilizatori și grupuri
 - Fișiere și directoare
 - Procese
 - Semnale
 - Tuburi

Biblioteca standard C (tipul 'FILE')

- 4 Interfață interactivă linie de comandă
 - Interfața Command Prompt (Windows)
 - Interfața Shell (Linux)
- 5 Interfață de scripting linie de comandă
 - Interfața Batch Scripting (Windows)
 - Interfața Shell Scripting (Linux)
- 6 Teme de laborator
 - Utilizatori și grupuri
 - Fișiere, directoare
 - Procese, semnale, tuburi

TODO

Cuprins

- 1 Sisteme de calcul, sisteme de operare
 - Organizarea unui sistem de calcul
 - Funcțiile sistemului de operare
- 2 Sisteme de operare uzuale
 - Incișiune în sistemul Windows
 - Incișiune în sistemul Linux
- 3 Interfața de programare C
 - Obiecte software și apeluri sistem
 - Utilizatori și grupuri
 - Fișiere și directoare
 - Procese
 - Semnale
 - Tuburi

- Biblioteca standard C (tipul 'FILE')
- 4 Interfață interactivă linie de comandă
 - Interfața Command Prompt (Windows)
 - Interfața Shell (Linux)
- 5 Interfață de scripting linie de comandă
 - Interfața Batch Scripting (Windows)
 - Interfața Shell Scripting (Linux)
- 6 Teme de laborator
 - Utilizatori și grupuri
 - Fișiere, directoare
 - Procese, semnale, tuburi

Prezentăm câteva programe utilitare utilizabile din linia de comandă și câteva opțiuni uzuale pentru acestea. Unele au mai fost prezentate mai înainte.

Vom folosi următoarele convenții de notare:

Specificarea '...' (de exemplu 'opțiuni...') arată că pot fi mai multe.

Parantezele '()' arată că este optional.

În general, liniile de comandă pot conține mai mult decât invocarea unor comenzi interne sau externe cu argumente - pot conține operatori, definiții de variabile, etc. Acest lucru este valabil și când ele sunt introduse interactiv și când ele sunt citite dintr-un fișier script. Mai multe detalii vor fi prezentate în secțiunea 'Interfața de scripting linie de comandă'.

Cuprins

- 1 Sisteme de calcul, sisteme de operare
 - Organizarea unui sistem de calcul
 - Funcțiile sistemului de operare
- 2 Sisteme de operare uzuale
 - Incișiune în sistemul Windows
 - Incișiune în sistemul Linux
- 3 Interfața de programare C
 - Obiecte software și apeluri sistem
 - Utilizatori și grupuri
 - Fișiere și directoare
 - Procese
 - Semnale
 - Tuburi

- Biblioteca standard C (tipul 'FILE')
- 4 Interfață interactivă linie de comandă
 - Interfața Command Prompt (Windows)
 - Interfața Shell (Linux)
- 5 Interfață de scripting linie de comandă
 - Interfața Batch Scripting (Windows)
 - Interfața Shell Scripting (Linux)
- 6 Teme de laborator
 - Utilizatori și grupuri
 - Fișiere, directoare
 - Procese, semnale, tuburi

TODO: de completat ...

Cuprins

- 1 Sisteme de calcul, sisteme de operare
 - Organizarea unui sistem de calcul
 - Funcțiile sistemului de operare
- 2 Sisteme de operare uzuale
 - Incișiune în sistemul Windows
 - Incișiune în sistemul Linux
- 3 Interfața de programare C
 - Obiecte software și apeluri sistem
 - Utilizatori și grupuri
 - Fișiere și directoare
 - Procese
 - Semnale
 - Tuburi

- Biblioteca standard C (tipul 'FILE')
- 4 Interfață interactivă linie de comandă
 - Interfața Command Prompt (Windows)
 - Interfața Shell (Linux)
- 5 Interfață de scripting linie de comandă
 - Interfața Batch Scripting (Windows)
 - Interfața Shell Scripting (Linux)
- 6 Teme de laborator
 - Utilizatori și grupuri
 - Fișiere, directoare
 - Procese, semnale, tuburi

Utilizatori și grupuri

În general, crearea unui utilizator nou (comanda 'useradd') presupune:

- specificarea unui username, parola, grup preferential;
- specificarea unui login shell;
- specificarea și crearea unui director home (poate fi și un director existent);
- copierea în directorul home a fișierelor și directoarelor din directorul skel (schelet), care implicit este '/etc/skel'; acestea formează structura de bază a directoarelor home ale utilizatorilor și sunt, în general, fișiere de configurare ale utilizatorului pentru diverse programe instalate, de exemplu: '.bash_profile', '.bashrc', etc.

continutul lui '/etc/skel' poate fi modificate doar de administrator;
copiile lor din directoarele home ale utilizatorilor aparțin acestora și pot fi
modificate de ei;

când un utilizator este eliminat din sistem de către administrator cu comanda 'userdel', directorul home al utilizatorului, inclusiv fișierele și directoarele copiate acolo din '/etc/skel', ramân intacte;

- crearea unui fișier de poștă (mail spool), de ex. '/var/spool/mail/username'.

Utilizatori și grupuri

```
useradd [options] username  
useradd -D
```

Crează un utilizator nou, cu numele 'username', care are caracteristicile implicite specificate în fișierul sistem '/etc/default/useradd' și pe cele specificate prin opțiuni (opțiunile prevalează asupra celor implicite); cu '-D' doar afișază valorile implicite; câteva de opțiuni:

-c comment

comentariul despre utilizator; textul este afișat de 'finger';

-d homedir

directorul home;

-g group

numele (group name) sau identificatorul numeric (GID) al grupului principal (grupul preferențial) (trebuie să existe);

-G group,...

lista grupurilor suplimentare din care face parte utilizatorul este 'group,...' (listă de nume de grupuri, separate prin virgulă ',' și fără spații);

-s shell

login shell-ul (cale/nume de fișier executabil).

Utilizatori și grupuri

În general, UID-ul va fi ales automat, a.î. să fie primul ID liber de după cel mai mare folosit, dintr-un anumit domeniu.

Comanda poate necesita privilegii de superuser (ex. 'root').

Pe unele sisteme, este implementată și comanda mai prietenoasă 'adduser'.

```
usermod [options] username
```

Modifică un utilizator folosind valorile specificate ca argumente.

Opțiunile seamănă cu cele de la 'useradd'.

Comanda 'usermod' poate necesita privilegii de superuser (ex. 'root').

```
userdel [options] username
```

Elimină (delete) utilizatorul cu numele 'username' și elibera (remove) toate intrările care se referă la el din grupuri.

Cu opțiunea '-r' se elibera întregul director home și fișierul de poștă (mail spool) al utilizatorului; fișierele din alte directoare trebuie eliminate (delete) manual.

Pe unele sisteme, este implementată și comanda mai prietenoasă 'deluser'.

Comanda 'userdel' poate necesita privilegii de superuser (ex. 'root').

Utilizatori și grupuri

`groupadd [options] group`

Crează un grup nou, cu numele 'group', folosind valorile implicite sau cele specificate în linia de comandă.

În general, GID-ul va fi ales automat, a.î. să fie primul ID liber de după cel mai mare folosit, dintr-un anumit domeniu.

Comanda 'groupadd' poate necesita privilegii de superuser (ex. 'root').

`groupmod [options] group`

Modifică grupul 'group' folosind valorile din linia de comandă.

Comanda 'groupmod' poate necesita privilegii de superuser (ex. 'root').

`groupdel [options] group`

Elimină grupul existent 'group'; grupul primar (grupul preferențial) al oricărui utilizator existent nu poate fi eliminat (întâi trebuie eliminat utilizatorul); toate sistemele de fișiere (filesystems) trebuie controlate manual pentru a asigura că nu a ramas nici un fișier cu grupul numit.

Utilitarul 'groupdel' poate necesita privilegii de superuser (ex. 'root').

Utilizatori și grupuri

```
passwd [options] [username]
```

Schimbă parole pentru utilizatori; un administrator poate schimba parola oricărui utilizator, unui utilizator normal își permite să schimbe doar parola propriului cont.

Fără 'username', un utilizator își schimbă parola proprie.

```
gpasswd [option] group
```

Schimbă parola pentru grupuri; doar un administrator poate schimba parola oricărui grup.

```
finger [options] [user ...]
```

```
finger [options] [user@host ...]
```

Afișază informații despre utilizatori locali și distanți (remote).

Cu opțiunile '-l', '-s', se folosește formatul lung (informații detaliate), resp. scurt. Pe o mașină distantă se poate folosi doar '-l'.

Utilizatori și grupuri

```
id [option]... [username]
```

Afișază UID, EUID, GID, EGID pentru utilizatorul specificat, resp. cel curent (dacă 'username' este omis). Câteva opțiuni:

-g

afișază doar EGID

-G

afișază toate GID-urile grupurilor utilizatorului

-n

afișază nume în loc de număr, pentru -ugG

-r

afișaza ID real în loc de ID efectiv

```
groups [option]... [username]...
```

Afișază apartenența la grupuri pentru fiecare 'username' sau, dacă nu a fost specificat nici un 'username', pentru cel curent.

Utilizatori și grupuri

```
su [options] [-] [user [argument...]]
```

Lansează un proces shell copil, având EUID și EGID cele ale lui 'user', care este interactiv sau execută o singură comandă și se termină. Poate cere parola pentru 'user'. Câteva opțiuni:

-, -l, --login

Ianseză un login shell (în acest scop, modifică niște variabile de environment și schimbă directorul curent în directorul home al lui 'user'); un simplu '-' implică '-l'; dacă 'user' nu este dat, se presupune 'root'.

-c, --command=COMMAND

Ianseză un shell cu opțiunea '-c', ce execută o singură comandă, 'COMMAND'.

-s, --shell=SHELL

Ianseză 'SHELL', în locul shell-ului din intrarea passwd a lui 'user'.

```
whoami [option]...
```

Afișază username-ul asociat cu EUID curent (al procesului shell de la care s-a dat comanda); efectul este similar cu cel al comenzi 'id -un'.

Utilizatori și grupuri

```
who [option]... [ file | arg1 arg2 ]
```

Afișază informații despre utilizatorii curent logați în sistem; informațiile sunt căutate în fișierul 'file' specificat, care implicit este '/var/run/utmp' (dar se poate folosi și '/var/log/wtmp'); argumentele 'arg1 arg2' uzuale sunt 'am i' sau 'mom likes'.

```
w [options] user [...]
```

Afișază informații despre utilizatorii aflați curent pe mașină și procesele lor.

Fișiere și directoare

stat [OPTION]... FILE...

Afișază caracteristicile (status) unor fișiere sau sisteme de fișiere. Opțiuni:

-L, --dereference

deferențiază legăturile simbolice;

-f, --file-system

afișază caracteristicile (status) unui sistem de fișiere, în loc de cele ale unui fișier (dacă se dă ca argument un fișier, se afișază caracteristicile sistemului de fișiere din care face parte);

-c --format=FORMAT

foloseste formatul de afișare 'FORMAT', în locul formatului implicit; afișază un newline după fiecare folosire a lui 'FORMAT';

--printf=FORMAT

ca și '--format', dar interpretează secvențele escape cu backslash și nu afișază un newline la urmă obligatoriu; dacă dorim un newline, includem '\n' în 'FORMAT';

-t, --terse

afișază informația în formă concisă.

Fisiere și directoare

Secvențele de format valide pentru fișiere (fără '--file-system'):

%a	Drepturile de acces in octal
%A	Drepturile de acces in human readable form
%b	Numarul de blocuri alocate (a se vedea '%B')
%B	Dimensiunea in bytes a fiecarui bloc raportat de '%b'
%C	SELinux security context string
%d	Numarul dispozitivului (device number) in zecimal
%D	Numarul dispozitivului (device number) in hex
%f	Raw mode in hex
%F	Tipul fisierului
%g	GID al proprietarului
%G	Group name al proprietarului
%h	Numarul legaturilor fizice (hard links)
%i	Numarul de inod
%n	Numele fisierului
%N	Numele citat (quoted file name) cu deferentiere, daca este legatura simbolica (symbolic link)
%o	Dimensiunea blocului I/O
%s	Dimensiunea totala, in bytes
%t	Tipul major al dispozitivului (major device type) in hex
%T	Tipul minor al dispozitivului (minor device type) in hex
%u	UID al proprietarului
%U	User name al proprietarului
%x	Momentul ultimului acces (time of last access)
%X	Momentul ultimului acces (time of last access), in secunde de la Epoch (momentul de referinta 1 ianuarie 1970)
%y	Momentul ultimei modificari (time of last modification)
%Y	Momentul ultimei modificari (time of last modification), in secunde de la Epoch (momentul de referinta 1 ianuarie 1970)
%z	Momentul ultimei schimbari (time of last change)
%Z	Momentul ultimei schimbari (time of last change) in secunde de la Epoch (momentul de referinta 1 ianuarie 1970)

Fișiere și directoare

Secvențele de format valide pentru sisteme de fișiere (file systems):

%a	Numarul de blocuri libere disponibile pentru non-superuser
%b	Numarul total de blocuri de date din sistemul de fisiere
%c	Numarul total de noduri de fisiere din sistemul de fisiere
%d	Numarul de noduri libere de fisiere din sistemul de fisiere
%f	Numarul de blocuri libere din sistemul de fisiere
%i	Identifierul sistemului de fisiere (File System ID) in hex
%l	Lungimea maxima a numelor de fisiere
%n	Numele fisierului
%s	Dimensiunea blocului (block size) (pentru transferuri mai rapide)
%S	Dimensiunea fundamentală a blocului (fundamental block size) (pentru numarari de blocuri)
%t	Tipul in hex
%T	Tipul in human readable form

Obs: Shell-ul poate avea propria versiune a comenzi 'stat', care de obicei elimină (supersedes) versiunea descrisă mai sus.

De exemplu, în GNU/Linux, dacă avem GNU 'stat' din 'coreutils' 8.6 sau mai recent, comanda:

```
stat -c %m sursa
```

afișază punctul de montare logică (mount point) al sistemului de fișiere / partitiei care conține fișierul sau directorul 'sursa' în arborele de directoare și fișiere al instanței SO.

Fișiere și directoare

Exemple:

```
$stat /etc/passwd
  File: /etc/passwd
    Size: 2750      Blocks: 8          IO Block: 4096   regular file
Device: 806h/2054d Inode: 655588      Links: 1
Access: (0644/-rw-r--r--) Uid: (    0/    root)  Gid: (    0/    root)
Access: 2021-11-16 21:17:01.266829826 +0200
Modify: 2020-11-03 15:35:29.865357242 +0200
Change: 2020-11-03 15:35:29.893357381 +0200
 Birth: -


$stat -f /etc/passwd
  File: "/etc/passwd"
    ID: ff54a3e1215e8153 Namelen: 255      Type: ext2/ext3
Block size: 4096      Fundamental block size: 4096
Blocks: Total: 5127135      Free: 1211890      Available: 945685
Inodes: Total: 1310720      Free: 676025


$stat --printf='%n\n%u    %U\nDimensiunea = %s\n' /etc/passwd
/etc/passwd
0    root
Dimensiunea = 2750
```

Fișiere și directoare

```
chown [OPTION]... [OWNER][:[GROUP]] FILE...
chown [OPTION]... --reference=RFILE FILE...
```

Comanda 'chown [OPTION]... [OWNER][:[GROUP]] FILE...' modifică proprietarul și/sau grupul fiecărui fișier specificat de 'FILE...', conform cu specificarea '[OWNER][:[GROUP]]'. Mai exact:

- dacă este specificat doar 'OWNER', utilizatorul respectiv devine proprietarul tuturor fișierelor specificate, iar grupul acestora rămâne nemodificat;
- dacă este specificat 'OWNER:GROUP' (fără spații între ele), este modificat atât proprietarul cât și grupul fișierelor;
- dacă este specificat 'OWNER:' (fără 'GROUP' după ':'), atunci proprietarul fișierelor devine 'OWNER', iar grupul fișierelor devine grupul preferențial (login group) al lui 'OWNER';
- dacă este specificat ':GROUP', este modificat doar grupul fișierelor; în acest caz, comanda 'chown' efectuează aceeași funcție ca și comanda 'chgrp';
- dacă este specificat ':', sau întregul operand '[OWNER][:[GROUP]]' este vid, nu este modificat nici proprietarul, nici grupul fișierelor.

'OWNER' și 'GROUP' pot fi numeric ID sau nume simbolice.

Fișiere și directoare

Comanda 'chown [OPTION]... --reference=RFILE FILE...' modifică proprietarul și grupul fiecărui fișier specificat de 'FILE...', să fie cele ale fișierului specificat de 'RFILE'.

Comanda 'chown' este supusă unor restricții, asemănătoare celor din cazul funcției 'chown()' (a se vedea mai sus).

Câteva opțiuni:

--dereference

afectează referentul fiecărei legături simbolice (este implicit);

-h, --no-dereference

afectează legăturile simbolice, în locul fișierelor referite;

--from=CURRENT_OWNER:CURRENT_GROUP

modifică proprietarul și/sau grupul fiecărui fișier doar dacă proprietarul și/sau grupul său curent se potrivesc cu cele specificate aici; oricare din atrbute poate fi omis și atunci nu este necesară o potrivire pentru el;

--reference=RFILE

ia în considerație proprietarul și grupul fișierului specificat de 'RFILE', în locul specificării valorilor 'OWNER:GROUP'.

Fișiere și directoare

-R, --recursive

operează asupra fișierelor și directoarelor recursiv; în acest caz, se mai pot adăuga următoarele opțiuni (dacă apar mai multe, are efect doar ultima):

-H

dacă un argument al liniei de comandă este o legătură simbolică la un director, este traversată;

-L

traversează toate legăturile simbolice la un director întâlnite.

-P

nu traversează nici o legătură simbolică (este implicit).

Fișiere și directoare

Exemple:

```
chown root /u
```

Modifică proprietarul lui '/u' în 'root'.

```
chown root:staff /u
```

Asemănător, dar modifică și grupul său în 'staff'.

```
chown -hR root /u
```

Modifică proprietarul lui '/u' și al subfișierelor sale în 'root'.

Fișiere și directoare

```
chgrp [OPTION]... GROUP FILE...
chgrp [OPTION]... \verb|--|reference=RFILE FILE...
```

Modifică grupul fiecărui fișier specificat de 'FILE...', în 'GROUP'.

Cu '--reference', modifică grupul fiecărui fișier specificat de 'FILE...' în cel al fișierului specificat de 'RFILE'.

Comanda 'chgrp' este supusa unor restrictii, asemanatoare celor din cazul functiei 'chown()' (a se vedea mai sus).

Multe opțiuni seamănă cu cele de la comanda 'chown': '--dereference', '-h', '--no-dereference', '-R', '--recursive' iar în acest caz: '-H', '-L', '-P'.

Opțiunea '--reference=RFILE' ia în considerație grupul fișierului specificat de 'RFILE', în locul specificării unei valori 'GROUP'.

Fișiere și directoare

Exemple:

```
chgrp staff /u
```

Modifică grupul lui '/u' în 'staff'.

```
chgrp -hR staff /u
```

Modifică grupul lui '/u' și al subfișierelor sale în 'staff'.

Fișiere și directoare

```
chmod [OPTION]... MODE[,MODE]... FILE...
chmod [OPTION]... OCTAL-MODE FILE...
chmod [OPTION]... --reference=FILE FILE...
```

Modifică biții de mod ai fiecărui fișier dat, conform modului indicat, care poate fi o reprezentare simbolică a modificărilor ce trebuie făcute, sau un număr octal ce reprezintă tiparul de biți pentru noii biți de mod.

Formatul unui mod simbolic este:

```
[ugoa...] [[+-=] [perms...]...]
```

unde 'perms' poate fi ≥ 0 litere din 'rwxXst', sau o literă din 'ugo'.

Pot fi date mai multe moduri simbolice, separate prin virgule ','. În toată transcrierea lui 'MODE[,MODE]' nu trebuie să apară nici un spațiu (să fie un singur argument în linia de comandă pentru 'chmod').

O combinație de literele 'ugoa' specifică utilizatorii a căror drepturi vor fi modificate (prin lipsă înseamnă 'a', dar nu sunt afectați biții setați în 'umask'):

'u' proprietarul fișierului;

'g' alți utilizatori din grupul fișierului;

'o' alți utilizatori ce nu sunt în grupul fișierului;

'a' toți utilizatorii.

Fișiere și directoare

Operatorii '+-=' au următoarea semnificație:

'+' biți specificați sunt adăugați la cei existenți ai fiecărui fișier;

'-' biți specificați sunt eliminați;

'=' biți specificați sunt adăugați iar cei nespecificați sunt eliminați, cu excepția faptului că biți set UID și set GID nemenționați ai unui director nu sunt afectați.

Intuitiv, '+' înseamnă reuniunea de mulțimi (se poate implementa prin disjuncția pe biți '|' cu masca bițiilor specificați), '-' înseamnă diferența de mulțimi (se poate implementa prin conjuncția pe biți '&' cu negația pe biți '~~' a maștii bițiilor specificați), iar '=' înseamnă atribuirea de mulțimi (se poate implementa prin atribuirea de întregi).

Fișiere și directoare

Literele 'rwxXst' specifică biții de mod modificați:

'r' read;

'w' write;

'x' execute (search pentru directoare);

'X' execute/search doar dacă fișierul este un director sau are deja dreptul execute pentru vreun utilizator;

's' set UID sau set GID la execuție;

't' flagul de eliminare restricționată (restricted deletion flag) sau sticky bit.

În loc de una sau mai multe dintre aceste litere, se poate specifica exact una din literele 'ugo' (semnificând proprietarul fișierului, alții utilizatori din grupul fișierului, utilizatorii care nu sunt în categoriile anterioare), și atunci va fi vorba de drepturile prezente la categoria de utilizatori respectivă.

Fișiere și directoare

Un mod numeric este de 1 - 4 cifre octale (0-7); fiecare cifră octală este derivată prin reunirea bitilor corespunzatori valorilor 4 (100 octal), 2 (010 octal) și 1 (001 octal); cifrele octale omise se presupun a fi zerouri aflate la început (leading zeros); de exemplu, modul 67 înseamnă modul 0067.

Prima cifră octală specifică atributelor set UID (4), set GID (2), restricted deletion/sticky (1); diversele lor combinații dau toate valorile octale 0-7.

A doua cifră octală specifică drepturile proprietarului fișierului: read (4), write (2), execute (1).

A treia cifră octală specifică drepturile altor utilizatori din grupul fișierului, cu aceleași valori 4, 2, 1.

A patra cifră octală specifică drepturile altor utilizatori, care nu sunt în grupul fișierului, cu aceleași valori 4, 2, 1.

Fișiere și directoare

Câteva opțiuni ale comenzi 'chmod':

--reference=RFILE

folosește modul fișierului specificat de 'RFILE', în locul valorilor 'MODE'

-R, --recursive

Modifică fișiere și directoare recursiv.

Obs: Comanda 'chmod' și apelul sistem 'chmod()' nu modifica drepturile legăturilor simbolice dar aceasta nu este o problema, deoarece drepturile legăturilor simbolice nu sunt folosite niciodata.

Exemplu:

Fișiere și directoare

```
$ ls -l
total 12
-rw-r--r-- 1 dragulici users 4 2016-12-03 23:17 f1
-rw-r--r-- 1 dragulici users 6 2016-12-03 23:17 f2
-rw-r--r-- 1 dragulici users 5 2016-12-03 23:17 f3
$ chmod ug+w,o=wx f1 f2 f3
$ ls -l
total 12
-rw-rw--wx 1 dragulici users 4 2016-12-03 23:17 f1
-rw-rw--wx 1 dragulici users 6 2016-12-03 23:17 f2
-rw-rw--wx 1 dragulici users 5 2016-12-03 23:17 f3
$ chmod u+r-wx f1
$ ls -l
total 12
-r--rw--wx 1 dragulici users 4 2016-12-03 23:17 f1
-rw-rw--wx 1 dragulici users 6 2016-12-03 23:17 f2
-rw-rw--wx 1 dragulici users 5 2016-12-03 23:17 f3
$ chmod ug+o f1
$ ls -l
total 12
-rwxrwx--wx 1 dragulici users 4 2016-12-03 23:17 f1
-rw-rw--wx 1 dragulici users 6 2016-12-03 23:17 f2
-rw-rw--wx 1 dragulici users 5 2016-12-03 23:17 f3
```

Fișiere și directoare

Exemplu:

```
$ ls -l
total 12
-rw-r--r-- 1 dragulici users 4 2016-12-03 23:26 f1
-rw-r--r-- 1 dragulici users 6 2016-12-03 23:26 f2
-rw-r--r-- 1 dragulici users 5 2016-12-03 23:26 f3
$ chmod 7350 f1
$ ls -l
total 12
--wsr-s---T 1 dragulici users 4 2016-12-03 23:26 f1
-rw-r--r-- 1 dragulici users 6 2016-12-03 23:26 f2
-rw-r--r-- 1 dragulici users 5 2016-12-03 23:26 f3
$ chmod 32 f1 f2 f3
$ ls -l
total 12
-----wx-w- 1 dragulici users 4 2016-12-03 23:26 f1
-----wx-w- 1 dragulici users 6 2016-12-03 23:26 f2
-----wx-w- 1 dragulici users 5 2016-12-03 23:26 f3
```

Fișiere și directoare

`umask [-p] [-S] [mode]`

Afișază / modifică masca de drepturi a procesului shell curent pentru crearea de fișiere utilizator (atributul 'umask'). Ea va fi moștenită de procesele copii lansate prin comenzi shell și va afecta fișierele create de acestea cu 'open()', 'creat()', etc.

Dacă 'mode' lipsește, este afișată masca curentă; în acest caz:

- cu '-S', masca este afișată în formă simbolică, altfel este afișată în octal;
- cu '-p' ieșirea este într-o formă ce poate fi refolosită ca intrare.

Dacă 'mode' este prezent, masca este setată la 'mode'; în acest caz, dacă 'mode' începe cu o cifră, este interpretat ca un număr octal; altfel, este interpretat ca specificare simbolică; forma este similară celei de la comanda 'chmod'.

Fișiere și directoare

Exemplu:

```
$umask  
0002  
$umask -S  
u=rwx,g=rwx,o=rx  
$umask -S -p  
umask -S u=rwx,g=rwx,o=rx  
$umask 0765  
$umask  
0765  
$umask -S  
u=,g=x,o=w  
$umask -S uo=rw  
u=rw,g=x,o=rw  
$umask -S  
u=rw,g=x,o=rw  
$umask g=r  
$umask -S  
u=rw,g=r,o=rw
```

(observăm că am putut seta masca simbolic și cu și fără '-S').

Fișiere și directoare

`touch [OPTION]... FILE...`

Actualizează momentul ultimului acces și momentul ultimei modificări pentru fiecare fișier specificat de 'FILE...', la momentul curent din sistem sau la cel specificat prin opțiuni.

Un argument fișier ('FILE') care nu există, este creat ca fișier vid (astfel, comanda 'touch' poate fi folosită pentru crearea de fișiere vide).

Opțiuni (argumentele opțiunilor lungi sunt necesare și pentru cele scurte):

`-a`

modifică doar access time

`-m`

modifică doar modification time

`-r, --reference=FILE`

folosește momentele fișierului specificat în locul momentului curent

`-t STAMP`

folosește [[CC]YY]MMDDhhmm[.ss] în locul momentului curent

`-d, --date=STRING`

parsează și folosește stringul în locul momentului curent; stringul este în format human readable aproape liber, de exemplu:

"Sun, 29 Feb 2004 16:21:42 -0800", "2004-02-29 16:21:42",

"next Thursday"

`-c, --no-create`

nu crează nici un fișier

Fișiere și directoare

```
realpath [OPTION]... FILE...
```

Afișază pe standard output calea absolută rezolvată a fiecărui fișier indicat. Toate componente, în afară de ultima, ale acestor căi, trebuie să existe.

Fișierele indicate pot avea căi relative, cu '.'/..' în interior, și care pot conține legături simbolice; pentru fiecare, se vor rezolva legăturile simbolice, se va determina o cale către ele absolută (din rădăcină) și fără '.'/..' în interior iar aceasta se va afișa.

Cu opțiunea '--relative-to=DIR', se vor afișa căile rezolvate relative la 'DIR'.

Exemplu:

Fișiere și directoare

```
$pwd  
/home/dragulici/Desktop/work  
$ls -l  
total 12  
drwxrwxr-x 2 dragulici dragulici 4096 Nov 23 21:45 a  
drwxrwxr-x 2 dragulici dragulici 4096 Nov 23 21:46 b  
drwxrwxr-x 2 dragulici dragulici 4096 Nov 23 21:50 c  
$ls -l a  
total 0  
lrwxrwxrwx 1 dragulici dragulici 9 Nov 23 21:45 s -> ../../b  
$ls -l b  
total 4  
-rw-rw-r-- 1 dragulici dragulici 4 Nov 23 21:46 f  
$realpath a/s/f  
/home/dragulici/Desktop/work/b/f  
$realpath --relative-to=c a/s/f  
./b/f  
$realpath --relative-to=c a/s/f a/s/g a/s/h  
./b/f  
./b/g  
./b/h
```

(observăm că 'g' și 'h' nu există).

Fișiere și directoare

`pwd [OPTION] ...`

Afișază pe standard output directorul curent cu cale completă.

Opțiuni:

`-P, --physical`

Evită legăturile simbolice.

`cd [dir]`

Schimbă directorul curent al procesului shell (comandă internă).

Noul director curent va fi 'dir'; prin lipsă, se folosește variabila de environment a procesului shell 'HOME'.

Fișiere și directoare

`ls [OPTION]... [FILE]...`

Listează pe standard output informații despre fișierele 'FILE'; prin lipsă, este directorul curent. Dacă nu este specificată nici una dintre opțiunile '-cftuvSUX' sau '--sort' sortează intrările alfabetic.

Opțiuni (argumentele opțiunilor lungi sunt necesare și pentru cele scurte):

-a, --all

nu ignoră intrările care încep cu '.'

-A, --almost-all

nu listează '.' și '..' implicate

--author

cu '-l', afișază autorul fiecărui fișier

-b, --escape

afișază secvențe escape în stil C pentru caracterele nongrafcice

-B, --ignore-backups

nu listează intrările implicate care se termină cu '~~'

-c

cu '-lt': sortează și afișază după 'ctime' (momentul ultimei schimbări a caracteristicilor); cu '-l': afișază 'ctime' și sortează după nume; altfel, sortează după 'ctime', cele mai noi sunt primele

Fișiere și directoare

-C

listea intrările pe coloane

--color [=WHEN]

colorează ieșirea; 'WHEN' poate fi 'always' (valoarea implicită, dacă este omisă), 'auto', sau 'never'; folosirea culorilor pentru a distinge tipurile fișierelor este dezactivată și în mod implicit și cu '--color=never'; cu '--color=auto', comanda emite coduri de culori doar când standard output este conectat la un terminal; variabila de environment 'LS_COLORS' poate schimba setările iar ea se poate seta cu comanda 'dircolors'

-d, --directory

listea directoarele însele, nu conținuturile lor

-f

nu sortează, activează '-aU', dezactivează '-ls --color'

-F, --classify

adăugă un indicator la sfârșitul intrărilor (unul dintre '* /=>@l')

--file-type

similar, dar nu adăugă '*'

-g

ca '-l', dar nu afișază proprietarul

Fișiere și directoare

--group-directories-first

grupează directoarele înaintea fișierelor; poate fi augmentat cu o opțiune '--sort', dar orice folosire a lui '--sort=none' ('-U') dezactivează gruparea

-G, --no-group

într-o listare long, nu afișază numele grupurilor

-h, --human-readable

cu '-l' și '-s', afișază dimensiunile sub forma 1K 234M 2G etc.

-H, --dereference-command-line

urmează legăturile simbolice listate în linia de comandă

--dereference-command-line-symlink-to-dir

urmează fiecare legătură simbolică din linia de comandă

care indică un director

--hide=PATTERN

nu listează intrările implicate care se potrivesc cu masca shell 'PATTERN' (surclasat de '-a' sau '-A')

--hyperlink[=WHEN]

afișază numele fișierelor ca hyperlinkuri; 'WHEN' poate fi 'always' (valoarea implicită, dacă este omisă), 'auto', sau 'never'

-i, --inode

afișază index number-ul fiecărui fișier

Fișiere și directoare

-I, --ignore= PATTERN

nu listează intrările implicate care se potrivesc cu masca shell
'PATTERN'

-l

utilizează un format de listare long

-L, --dereference

când afișază informațiile de fișier pentru o legătură simbolică, afișază informațiile despre fișierul referit, nu despre legătura însăși

-m

umește lățimea de afișare cu o listă de intrări separate prin virgule

-n, --numeric-uid-gid

ca '-l', dar afișază user și group ID-urile numerice

-N, --literal

afișază numele intrărilor fără citare

-o

ca '-l', dar nu afișază informațiile despre grupuri

-p, --indicator-style= slash

adaugă indicatorul '/' la sfârșitul directoarelor

Fișiere și directoare

-q, --hide-control-chars

afișază '?' în locul caracterelor nongrafice

--show-control-chars

afișază caracterele nongrafice ca atare (este implicit, cu excepția cazului când programul este 'ls' iar ieșirea este un terminal)

-Q, --quote-name

include numele intrărilor între ghilimele (double quotes)

-r, --reverse

inversează ordinea de sortare

-R, --recursive

listeză recursiv subdirectoarele

-s, --size

afișază dimensiunea alocată pentru fiecare fișier, în blocuri

-S

sorteză după dimensiunea fișierelor, cele mai mari sunt primele

-t

sorteză după momentul ultimei modificări, cele mai noi sunt primele

-T, --tabsize=COLS

consideră stopuri tab la fiecare 'COLS' în loc de 8

Fișiere și directoare

-u

cu '-lt': sortează și afișază după 'access time' (momentul ultimului acces); cu '-l': afișază 'access time' și sortează după nume; altfel: sortează după 'access time', cele mai noi sunt primele

-U

nu sortează ci listează intrările în ordinea din director

-v

sortarea naturală a numerelor (de versiune) din text

-w, --width=COLS

setează lățimea de afișare la 'COLS', 0 înseamnă fără limită

-x

listează intrările pe linii, nu pe coloane

-X

sortează alfabetic după extensii

-Z, --context

afișază orice context de securitate pentru fiecare fișier

-1

listează câte un fișier pe linie; se evită '\n' cu '-q' sau '-b'

Fișiere și directoare

Cu '-l' (formatul long), comanda 'ls' afișază fiecare fișier pe câte o linie, conținând, de la stânga la dreapta:

- tipul fișierului: '-' (regular), 'd' (director), 'b' (fișier bloc), 'c' fișier special caracter), 'p' (tub), 's' (socket), 'l' (legătură simbolică);
- dreptul de citire pentru proprietar: 'r' / '-' (prezent / absent);
- dreptul de scriere pentru proprietar: 'w' / '-' (prezent / absent);
- dreptul de execuție pentru proprietar și bitul set UID:
 - 'x' (dreptul de execuție prezent, și bitul set UID ne-setat)
 - 's' (dreptul de execuție prezent, și bitul set UID setat)
 - '-' (dreptul de execuție absent, și bitul set UID ne-setat)
 - 'S' (dreptul de execuție absent, și bitul set UID setat)
- dreptul de citire pentru grup: 'r' / '-' (prezent / absent);
- dreptul de scriere pentru grup: 'w' / '-' (prezent / absent);
- dreptul de execuție pentru grup și bitul set GID:
 - 'x' (dreptul de execuție prezent, și bitul set GID ne-setat)
 - 's' (dreptul de execuție prezent, și bitul set GID setat)
 - '-' (dreptul de execuție absent, și bitul set GID ne-setat)
 - 'S' (dreptul de execuție absent, și bitul set GID setat)

Fișiere și directoare

- dreptul de citire pentru alții: 'r' / '-' (prezent / absent);
- dreptul de scriere pentru alții: 'w' / '-' (prezent / absent);
- dreptul de execuție pentru alții și bitul STICKY:
 - 'x' (dreptul de execuție prezent, și bitul STICKY ne-setat)
 - 't' (dreptul de execuție prezent, și bitul STICKY setat)
 - '-' (dreptul de execuție absent, și bitul STICKY ne-setat)
 - 'T' (dreptul de execuție absent, și bitul STICKY setat)
- numărul de legături fizice;
- numele proprietarului;
- numele grupului;
- dimensiunea;
- momentul ultimei modificări;
- numele fișierului.

Codul de return (exit status) al comenzi 'ls' poate fi:

- 0 dacă a fost OK,
- 1 dacă au fost probleme minore (ex: nu a putut fi accesat un subdirector),
- 2 dacă au fost probleme serioase (ex: nu a putut fi accesat un argument al liniei de comandă).

Fișiere și directoare

`mkdir [OPTION]... DIRECTORY...`

Crează directoarele 'DIRECTORY...', dacă nu există deja.

Opțiuni (argumentele opțiunilor lungi sunt necesare și pentru cele scurte):

`-m, --mode=MODE`

setează modul fișierului (drepturi, etc.) (ca la comanda 'chmod'),
nu 'a=rwx - umask'

`-p, --parents`

nu se semnalează erori în cazul existenței și se crează directoarele părinte așa cum este necesar

`rmdir [OPTION]... DIRECTORY...`

Șterge (remove) directoarele 'DIRECTORY...', dacă sunt goale.

Opțiuni:

`--ignore-fail-on-non-empty`

ignoră fiecare eșec care este cauzat doar de faptul că un director nu este gol

`-p, --parents`

șterge 'DIRECTORY' și ancestorii lui; ex: 'rmdir -p a/b/c' este similar cu 'rmdir a/b/c a/b a'

Fișiere și directoare

`cat [OPTION]... [FILE]...`

Comandă filtru care concatenează fișierele 'FILE...' la standard output. Dacă nu este dat nici un fișier sau dacă nu fișier este '-', citește de la standard input.

Opțiuni:

`-b, --number-nonblank`

numerotează liniile de ieșire care nu sunt goale, surclasază '-n'

`-E, --show-ends`

afișază '\$' la sfârșitul fiecărei linii

`-n, --number`

numerotează toate liniile de ieșire

`-s, --squeeze-blank`

suprimă liniile de ieșire goale repetate

`-T, --show-tabs`

afișază caracterele TAB ca '^I'

`-v, --show-nonprinting`

folosește notația '^' și 'M-', cu excepția caracterelor LFD și TAB

Exemplu: 'cat f - g' scrie conținutul lui 'f', apoi cel al intrării standard, apoi conținutul lui 'g'; 'cat' copiază standard input la standard output.

Fișiere și directoare

Obs: '^I', '^M', '^H', etc. sunt combinații de taste ('^' înseamnă Ctrl) care corespund unor caractere de control consolă - când sunt emise către un terminal, fie prin tastare, fie scrise de un proces, terminalul execută o funcție; de exemplu, '^H' corespunde caracterului BS (backspace, codul 0x08) iar la primirea lui se șterge de pe ecran ultimul caracter afișat. Pentru alte detalii, a se vedea 'man 4 console_codes'.

```
tac [OPTION]... [FILE]...
```

Comandă filtru care concatenează fișierele 'FILE...' la standard output, inversate (ultima linie prima). Dacă nu este dat nici un fișier sau dacă un fișier este '-', citește de la standard input.

Opțiuni (argumentele opțiunilor lungi sunt necesare și pentru cele scurte):

-b, --before

atașază separatorul înainte, nu după

-r, --regex

interpretează separatorul ca o expresie regulată (mască)

-s, --separator=STRING

utilizează 'STRING' ca separator, în locul newline

Fișiere și directoare

```
more [options] [file]...
```

Comandă filtru pentru afișarea paginată a unui text, un ecran o dată afișată o pagină, afișază un prompter și așteaptă (interactiv) comenzi pentru a continua (defilare încă un rând sau încă o pagină, ieșire, etc.).

Opțiunile se pot atât din linia de comandă (prevalează) cât și din variabila de environment 'MORE' - ele trebuie precedate de dash '-', de exemplu, setăm:

```
export MORE=-d'
```

Opțiuni:

-d

Afișază prompterul "[Press space to continue, 'q' to quit.]", și mesajul "[Press 'h' for instructions.]" în loc să emită un sunet (bell) atunci când este apăsată o tastă ilegală

-l

Nu face pauză după liniile ce conțin '^L' (form feed).

-f

Numără liniile logice, nu cele ale ecranului (i.e., liniile lungi nu sunt pliate)

-p

Nu defilează (scroll); în schimb, șterge tot ecranul și afișază textul.

Fișiere și directoare

-c

Nu defilează (scroll); în schimb, desenează fiecare ecran începând de sus, stergând restul fiecărei linii pe măsură ce este afișată

-s

comprimă liniile albe multiple în una singură

-u

Suprimă sublinierea

-number

Dimensiunea folosită a ecranului, în număr de linii

+number

Începe afișarea fiecărui fișier de la numărul de linie

+/string

Stringul de căutat în fiecare fișier înainte de a începe afișarea lui

Comenzi interactive (se bazează pe comenzi editorului 'vi', unele comenzi pot fi precedate de un număr zecimal 'k', '^x' înseamnă Ctrl-x):

h sau ?

Help (ajutor); afișază un sumar al comenziilor

SPACE

Afișază următoarele k linii ale textului (ex: dacă tastăm 2 urmat de SPACE, afișază încă 2 linii); implicit (nu tastăm k), k se consideră dimensiunea curentă a ecranului

Fișiere și directoare

z

Afișază următoarele k linii ale textului, implicit, k se consideră dimensiunea curentă a ecranului; argumentul devine noua valoare implicită

RETURN

Afișază următoarele k linii ale textului, implicit, k se consideră 1; argumentul devine noua valoare implicită

d sau ^D

Defilează (scroll) k linii; implicit, este dimensiunea curentă de scroll, inițial este 11; argumentul devine noua valoare implicită

q sau Q sau INTERRUPT

ieșire

s

Sare înainte k linii de text; implicit, este 1

f

Sare înainte k ecrane de text; implicit, este 1

b sau ^B

Sare înapoi k ecrane de text; implicit, este 1; funcționează doar cu fișiere care nu sunt tuburi

,

Sare la locul de unde a început ultima cautare

Fișiere și directoare

=

Afișază numărul liniei curente

/pattern

Caută a k-a apariție a unui sir care se potrivește cu expresia regulată 'pattern'; implicit, este 1

n

Caută a k-a apariție a unui sir care se potrivește cu ultima expresie regulată; implicit, este 1; deci, cu 'n' putem repeta cautarea

!command sau :!command

Execută comanda 'command' într-un proces shell copil

v

Lansează un editor la linia curentă; editorul este luat din variabila de environment

'VISUAL' dacă este definită, sau 'EDITOR', dacă 'VISUAL' nu este definită, sau

este implicit 'vi', dacă nici una dintre cele două variabile nu este definită

^L

Redesenează ecranul

:n

Salt la al k-lea fișier următor; implicit, este 1

Fișiere și directoare

:p

Salt la al k-lea fișier anterior; implicit, este 1

:f

Afișază numele fișierului și numărul liniei curente.

.

Repetă comanda anterioară

Obs: Întrucât 'more' este comandă filtru, poate fi folosit pentru afișarea paginată a ieșirii altei comenzi, ex: 'ls -l /etc | more—'.

`less [options] [file]...`

Comandă filtru pentru afișarea paginată a unui text, un ecran o dată, asemănătoare cu 'more', dar cu facilități suplimentare - multe opțiuni și comenzi, poate lucra cu multe tipuri de terminal, nu este necesar să citească tot fișierul înainte de a porni, deci pornește mai repede.

De exemplu, se poate naviga în fișier și cu taste săgeți, UP / DOWN.

Fișiere și directoare

```
cp [OPTION]... [-T] SOURCE DEST
cp [OPTION]... SOURCE... DIRECTORY
cp [OPTION]... -t DIRECTORY SOURCE...
```

Copiază fișiere și directoare; copiază 'SOURCE' în 'DEST', sau multiple 'SOURCE' în 'DIRECTORY'.

Opțiuni (argumentele opțiunilor lungi sunt necesare și pentru cele scurte):

--attributes-only

nu copiază datele fișierelor ci doar atrbutele
(fișierele copie vor fi vide)

--backup[=CONTROL]

face un backup pentru fiecare fișier destinație existent

-b

ca '--backup' dar nu acceptă un argument

--copy-contents

copiază conținutul fișierelor speciale când are loc recursie

-f, --force

dacă un fișier destinație existent nu poate fi deschis, îl șterge
(remove) și încearcă din nou (opțiunea este ignorată dacă este
și '-n'

Fișiere și directoare

-i, --interactive

cere interactiv confirmare înainte de suprascrisere (surclasă un '-n' anterior)

-H

urmează legăturile simbolice din linia de comandă în SOURCE

-l, --link

în loc să copieze fișierele, le face legături fizice (hard link)

-L, --dereference

urmează întotdeauna legăturile simbolice în SOURCE

-n, --no-clobber

nu suprascrie un fișier existent (surclasă un '-i' anterior)

-P, --no-dereference

nu urmează niciodată legăturile simbolice în SOURCE

--preserve[=ATTR_LIST]

conservează atributele specificate (implicit: mode, ownership, time-stamps),

iar dacă e posibil, atributele adiționale: context, links, xattr, all

--no-preserve=ATTR_LIST

nu conservează atributele specificate

--parents

folosește numele fișierului sursă întreg sub DIRECTORY

Fișiere și directoare

-R, -r, --recursive

copiază directoare recursiv (cu tot cu arborescența aferentă)

--remove-destination

șterge (remove) fiecare fișier destinație existent înainte de a încerca deschiderea lui (în contrast cu '--force')

--strip-trailing-slashes

elimină toate trailing slashes din fiecare argument SOURCE

-s, --symbolic-link

crează legături simbolice în loc să copieze

-S, --suffix=SUFFIX

surclasază sufixul uzual folosit la backup; sufixul uzual este '~', cu excepția cazului când este setat cu '--suffix' sau 'SIMPLE_BACKUP_SUFFIX'; metoda de control a versiunilor poate fi selectată via opțiunea '--backup' sau prin variabila de environment 'VERSION_CONTROL'; valorile pot fi:

none, off

nu face niciodată backup-uri (nici dacă este prezent '--backup')

numbered, t

face backup-uri numărate

Fișiere și directoare

existing, nil

dacă există backup-uri numărate, face backup-uri numărate,
altfel le face simple

simple, never

întotdeauna face backup-uri simple

Ca un caz special, 'cp' face un backup al lui SOURCE când sunt prezente opțiunile 'force' și 'backup' iar SOURCE și DEST sunt același fișier regular existent

-t, --target-directory=DIRECTORY

copiază toate argumentele SOURCE în interiorul DIRECTORY

-T, --no-target-directory

tratează DEST ca un fișier normal

-u, --update

copiază doar când fișierul SOURCE este mai nou decât fișierul destinație sau când fișierul destinație lipsește

-v, --verbose

explică ce se efectuează

Fișiere și directoare

Observație: Am văzut mai devreme că:

`cp fișier1 fișier2`

crează 'fișier2', dacă nu există, și îl suprascrie, dacă există

`cp fișier director`

copiază 'fișier1' în 'director', cu același nume

`cp -r director1 director2`

copiază arborescență cu originea 'director1' într-o arborescență cu originea în 'director2', dacă 'director2' nu există, sau într-un subarbore plasat în 'director2', cu același nume al originii, dacă 'director2' există.

Fișiere și directoare

```
mv [OPTION]... [-T] SOURCE DEST
mv [OPTION]... SOURCE... DIRECTORY
mv [OPTION]... -t DIRECTORY SOURCE...
```

Mută /redenumește fișiere; redenumește 'SOURCE' în 'DEST', sau mută 'SOURCE(s)' în 'DIRECTORY'; de fapt, caută să înlocuiască legături fizice, scriind în directoare (nu necesită drepturi pe fișierele numite ci doar drept de scriere pe directoarele părinte).

Opțiuni (argumentele opțiunilor lungi sunt necesare și pentru cele scurte):

-backup[=CONTROL]

face backup pentru fiecare fișier destinație existent

-b

ca '-backup' dar nu acceptă un argument

-f, --force

nu cere interactiv confirmare înainte de suprascriere

-i, --interactive

cere interactiv confirmare înainte de suprascriere

-n, --no-clobber

nu suprascrie un fișier existent

Dacă specificăm mai mult de una dintre '-i', '-f', '-n', doar cea finală are efect

Fișiere și directoare

-strip-trailing-slashes

elimină toate trailing slashes din fiecare argument SOURCE

-S, --suffix=SUFFIX

surclasază sufixul uzual folosit la backup; sufixul uzual este '~', cu excepția cazului când este setat cu '--suffix' sau 'SIMPLE_BACKUP_SUFFIX'; metoda de control a versiunilor poate fi selectată via opțiunea '--backup' sau prin variabila de environment 'VERSION_CONTROL'; valorile pot fi:

none, off

nu face niciodată backup-uri (nici dacă este prezent '--backup')

numbered, t

face backup-uri numărate

existing, nil

dacă există backup-uri numărate, face backup-uri numărate,

altfel le face simple

simple, never

întotdeauna face backup-uri simple

-t, --target-directory= DIRECTORY

mută toate argumentele SOURCE în DIRECTORY

-T, --no-target-directory

tratează DEST ca un fișier normal

Fișiere și directoare

-u, -update

mută doar când fișierul SOURCE este mai nou decât fișierul destinație sau când fișierul destinație lipsește

-v, --verbose

explică ce se efectuează

Exemplu:

```
$mkdir a  
$mkdir b  
$mkdir /tmp/c  
$echo 123 > a/f1.txt  
$echo 456 > a/f2.txt  
$echo 789 > a/f3.txt  
$ln a/f3.txt a/f4.txt  
$stat -c %m a  
/home  
$stat -c %m b  
/home  
$stat -c %m /tmp/c  
/
```

Fișiere și directoare

```
$ls -il a
total 16
921091 -rw-rw-r-- 1 dragulici dragulici 4 Nov 29 03:45 f1.txt
921097 -rw-rw-r-- 1 dragulici dragulici 4 Nov 29 03:45 f2.txt
921100 -rw-rw-r-- 2 dragulici dragulici 4 Nov 29 03:45 f3.txt
921100 -rw-rw-r-- 2 dragulici dragulici 4 Nov 29 03:45 f4.txt
$mv a/f1.txt b/g1.txt
$ls -il a
total 12
921097 -rw-rw-r-- 1 dragulici dragulici 4 Nov 29 03:45 f2.txt
921100 -rw-rw-r-- 2 dragulici dragulici 4 Nov 29 03:45 f3.txt
921100 -rw-rw-r-- 2 dragulici dragulici 4 Nov 29 03:45 f4.txt
$ls -il b
total 4
921091 -rw-rw-r-- 1 dragulici dragulici 4 Nov 29 03:45 g1.txt
```

Fișiere și directoare

```
$mv a/f2.txt /tmp/c/g2.txt
$ls -il a
total 8
921100 -rw-rw-r-- 2 dragulici dragulici 4 Nov 29 03:45 f3.txt
921100 -rw-rw-r-- 2 dragulici dragulici 4 Nov 29 03:45 f4.txt
$ls -il /tmp/c
total 4
1051689 -rw-rw-r-- 1 dragulici dragulici 4 Nov 29 03:45 g2.txt
$mv a/f3.txt /tmp/c/g3.txt
$ls -il a
total 4
921100 -rw-rw-r-- 1 dragulici dragulici 4 Nov 29 03:45 f4.txt
$ls -il /tmp/c
total 8
1051689 -rw-rw-r-- 1 dragulici dragulici 4 Nov 29 03:45 g2.txt
1051690 -rw-rw-r-- 1 dragulici dragulici 4 Nov 29 03:45 g3.txt
$diff a/f4.txt /tmp/c/g3.txt
```

Fișiere și directoare

Observații:

- directoarele 'a', 'b' sunt pe o partitură (sistem de fișiere), montată în directorul '/home', '/tmp/c' este pe altă partitură, montată în directorul '/';
- la 'mv a/f1.txt b/g1.txt', directorul destinație este pe aceeași partitură ca directorul sursă; ca atare, doar s-a înlocuit o legătură fizică cu alta (s-a operat pe directoare), fișierul nu s-a mutat ca i-nod și zonă de date
- la 'mv a/f2.txt /tmp/c/g2.txt', directoarele sursă și destinație sunt pe partituri diferite iar fișierul sursă nu are mai multe nume (legături fizice); ca atare, s-a înlocuit legătura fizică, dar s-a mutat și fișierul ca zonă de date, primind alt i-nod;
- la 'mv a/f3.txt /tmp/c/g3.txt', directoarele sursă și destinație sunt pe partituri diferite iar fișierul sursă mai are un nume (legături fizice) pe partitură sa; ca atare, s-a făcut o copie a fișierului, nu o mutare.

Fișiere și directoare

`rm [OPTION]... [FILE]...`

Șterge fișierele și directoarele specificate (remove). De fapt, caută să șteargă legături fizice ('unlink()'), ștergând intrări din directoare (nu necesită drepturi pe fișierele numite ci doar drept de scriere pe directoarele părinte). Fișierul este șters doar dacă nu mai are legături fizice și nici nu e deschis de vreun proces.

Implicit, nu șterge directoare, nici măcar goale (necessită '-r').

Opțiuni:

`-f, --force`

ignoră fișierele și argumentele inexistente, nu invită

`-i`

invită (cere confirmare) înaintea fiecărei ștergeri

`-I`

invită o dată înainte de a șterge mai mult de 3 fișiere,

sau când șterge recursiv

`--one-file-system`

când șterge recursiv, omite orice director care este într-un sistem de fișiere / partii diferit de cel al argumentului din linia de comandă corespunzător

Fișiere și directoare

--no-preserve-root

nu tratează special '/'

--preserve-root[=all]

nu șterge '/' (implicit); cu 'all', respinge orice argument din linia de comandă aflat pe un dispozitiv separat de părintele său

-r, -R, --recursive

șterge directoarele și conținuturile lor recursiv

-d, --dir

șterge directoarele goale

-v, --verbose

explică ce se execută

Pentru a șterge un fișier al cărui nume începe cu '-', de exemplu '-foo', trebuie folosită un a dintre comenziile:

```
rm -- -foo
```

```
rm ./-foo
```

Observație: dacă ștergem un fișier cu 'rm', este posibilă recuperarea unei părți a conținutului (doar se marchează zona de date ca liberă, nu mai este garantată conservarea datelor, dar zona nu este suprascrisă imediat). Pentru a ne asigura că nu se poate recupera conținutul, se poate folosi comanda 'shred' (care suprascrie conținutul).

Fișiere și directoare

`shred [OPTION]... FILE...`

Suprascrie conținutul fișierelor specificate de 'FILE' în mod repetat, cu scopul de a face datele mai greu de recuperat, și eventual șterge fișierele respective.

Dacă 'FILE' este '-', este suprascris standard output.

Opțiuni (argumentele opțiunilor lungi sunt necesare și pentru cele scurte):

`-f, --force`

schimbă drepturile pentru a permite scrierea, dacă este necesar

`-n, --iterations=N`

suprascrie de 'N' ori în loc de numărul implicit 3

`--random-source=FILE`

obține octeți aleatori din fișierul 'FILE'

`-s, --size=N`

numărul de octeți suprascrisi (sunt acceptate sufixe ca K, M, G)

`-u`

dezalocă și șterge (remove) fișierul după suprascriere;

implicit, fișierul nu este șters deoarece se obișnuiește să se opereze

pe fișiere bloc (pot corespunde unor partiții), ex. '/dev/hda'

iar aceste fișiere, de obicei nu trebuie șterse

Fișiere și directoare

--remove[=HOW]

ca '-u' dar se poate specifica cum ('HOW') se sterge intrarea din director:

'unlink' : se folosește un apel 'unlink()' standard;

'wipe' : de asemenea, se fac mai întâi neinteligibili (obfuscate) octetii numelui

'wipesync' : de asemenea, se sincronizează fiecare octet obfuscat cu discul (este modul implicit, dar poate fi costisitor)

-v, --verbose

arată progresul

-x, --exact

nu se rotunjește dimensiunea fișierelor în sus la următorul bloc
(este implicit pentru fișierele non-regular)

-z, --zero

face o suprascriere finală cu zero-uri pentru ascunde măruntearea

Observație: eficiența este limitată în sistemele de fișiere care nu suprascriu datele în locul lor original, de exemplu cele jurnalizate, care fac snapshots, care mențin în cache fișierele temporare sau care sunt comprimate. De asemenea, pot exista copii ale fișierelor în file system backups și remote mirrors, ceea ce permite recuperarea lor.

Semnale

`kill [options] <pid> [...]`

Trimite un semnal unui proces. Poate fi o comandă internă shell; comanda descrisă aici este '/bin/kill'.

Opțiuni:

`smallskip <pid> [...]`

Trimite semnal tuturor proceselor cu pid-urile `<pid>` listate.

Cu 'pid' negativ, se transmite tuturor progeselor din grupul de procese respectiv.

Cu 'pid' -1 se transmite tuturor proceselor în afară de 'init' și de 'kill' însuși.

`-<signal>`

`-s <signal>`

`--signal <signal>`

Specifică semnalul de trimis (nume - partea fără 'SIG' sau număr).

Semnalul poate fi nume, inclusiv 'SIG', nume, fără 'SIG', sau număr; prin lipsă este 'SIGTERM'.

`-l, --list [signal]`

fără argument, listează numele semnalelor;

cu un argument semnal, transformă un singur semnal între nume și muăr.

`-L, --table`

Listează numele semnalelor într-un tabel. List signal names in a nice table.



Semnale

Exemple:

```
$kill -L
 1) SIGHUP      2) SIGINT      3) SIGQUIT      4) SIGILL      5) SIGTRAP
 6) SIGABRT     7) SIGBUS      8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
$kill -1 2 SIGINT INT
INT
2
2
$gedit &
[1] 8100
$kill -9 8100
$gedit &
[2] 8114
[1] Killed           gedit
$kill -SIGKILL 8114
$gedit &
[3] 8123
[2] Killed           gedit
$kill -KILL 8123
```

```
$ (gedit &) ; (xed &)  
[3]+  Killed                  gedit  
$ps  
  PID TTY      TIME CMD  
 7791 pts/2    00:00:00 bash  
 8132 pts/2    00:00:00 gedit  
 8134 pts/2    00:00:00 xed  
 8144 pts/2    00:00:00 ps  
$kill 8132 8134
```

(în ultimul caz, s-a trimis semnalul 'SIGTERM').

Obs: câteva comenzi asemănătoare (detalii se pot afla cu 'man'): 'killall' (trimite semnale proceselor ce execută o anumită comandă), 'pkill' (trimite semnale proceselor cu anumite atribute).

Tuburi

```
mkfifo [OPTION]... NAME...
```

Creaza tuburile cu nume (FIFO) specificate de 'NAME...'.

Optiunile '[OPTION]...' pot specifica drepturile asupra tuburilor, sub forma:

`-m, --mode=MODE`

unde 'MODE' poate fi specificat simbolic ca la comanda 'chmod'; drepturile se vor seta astfel: se pornește de la drepturile implicite 'a=rw'; dacă '--mode' lipsește, se va aplica în continuare '& ~umask', unde 'umask' este masca de drepturi a shell-ului; dacă '--mode' este prezent, se va aplica în continuare 'MODE'.

Exemplu:

```
$umask  
0002  
$mkfifo tub1  
$mkfifo --mode=u=wx tub2  
$mkfifo --mode=o+wx tub3  
$ls -l tub*  
prw-rw-r-- 1 dragulici dragulici 0 Dec 27 20:01 tub1  
p-wxrw-rw- 1 dragulici dragulici 0 Dec 27 20:01 tub2  
prw-rw-rwx 1 dragulici dragulici 0 Dec 27 20:02 tub3
```

TODO: de adăugat alte comenzi ...

Cuprins

- 1 Sisteme de calcul, sisteme de operare
 - Organizarea unui sistem de calcul
 - Funcțiile sistemului de operare
- 2 Sisteme de operare uzuale
 - Incișiune în sistemul Windows
 - Incișiune în sistemul Linux
- 3 Interfața de programare C
 - Obiecte software și apeluri sistem
 - Utilizatori și grupuri
 - Fișiere și directoare
 - Procese
 - Semnale
 - Tuburi

- Biblioteca standard C (tipul 'FILE')
- 4 Interfață interactivă linie de comandă
 - Interfața Command Prompt (Windows)
 - Interfața Shell (Linux)
- 5 Interfață de scripting linie de comandă
 - Interfața Batch Scripting (Windows)
 - Interfața Shell Scripting (Linux)
- 6 Teme de laborator
 - Utilizatori și grupuri
 - Fișiere, directoare
 - Procese, semnale, tuburi

Cuprins

- 1 Sisteme de calcul, sisteme de operare
 - Organizarea unui sistem de calcul
 - Funcțiile sistemului de operare
- 2 Sisteme de operare uzuale
 - Incișiune în sistemul Windows
 - Incișiune în sistemul Linux
- 3 Interfața de programare C
 - Obiecte software și apeluri sistem
 - Utilizatori și grupuri
 - Fișiere și directoare
 - Procese
 - Semnale
 - Tuburi

- Biblioteca standard C (tipul 'FILE')
- 4 Interfață interactivă linie de comandă
 - Interfața Command Prompt (Windows)
 - Interfața Shell (Linux)
- 5 Interfața de scripting linie de comandă
 - Interfața Batch Scripting (Windows)
 - Interfața Shell Scripting (Linux)
- 6 Teme de laborator
 - Utilizatori și grupuri
 - Fișiere, directoare
 - Procese, semnale, tuburi

TODO de completat ...

Cuprins

- 1 Sisteme de calcul, sisteme de operare
 - Organizarea unui sistem de calcul
 - Funcțiile sistemului de operare
- 2 Sisteme de operare uzuale
 - Incișiune în sistemul Windows
 - Incișiune în sistemul Linux
- 3 Interfața de programare C
 - Obiecte software și apeluri sistem
 - Utilizatori și grupuri
 - Fișiere și directoare
 - Procese
 - Semnale
 - Tuburi

- Biblioteca standard C (tipul 'FILE')
- 4 Interfață interactivă linie de comandă
 - Interfața Command Prompt (Windows)
 - Interfața Shell (Linux)
- 5 Interfață de scripting linie de comandă
 - Interfața Batch Scripting (Windows)
 - Interfața Shell Scripting (Linux)
- 6 Teme de laborator
 - Utilizatori și grupuri
 - Fișiere, directoare
 - Procese, semnale, tuburi

Un shell este un interpreter de linie de comandă text care constituie o interfață utilizator pentru utilizarea SO.

Shell-ul oferă atât un limbaj de comandă interactiv cât și un limbaj de scripting; el poate fi folosit atât de utilizatorul uman cât și de SO pentru a controla execuția sistemului folosind scripturi shell.

Shell-ul are o interfață text și poate fi folosit interactiv de utilizatori de la un terminal hardware setat în mod text sau un emulator de terminal.

Într-un sistem pot fi instalate mai multe shell-uri:

Bourne Shell (sh), specificator: /bin/sh și /sbin/sh

GNU Bourne (Again) Shell (bash), specificator: /bin/bash

Korn Shell (ksh), specificator: /bin/ksh

C Shell (csh), specificator: /bin/csh

Shell-urile oferă limbaje de comandă și scripting complexe, pot exista diferențe între ele dar toate oferă posibilitatea de a folosi măști pentru specificatori de fișiere (filename wildcarding), filtrarea (piping), documente generate pe loc (here documents), substituții în linia de comandă, variabile și structuri de control pentru condiționare și iterare.

Un proces shell poate fi primit automat de un utilizator la autentificare pentru a opera interactiv dar poate fi și lansat dintr-un program (apelul 'system()') sau cu o comandă shell (ex: 'sh', 'bash') ca un proces copil, care să fie interactiv, să execute doar o singură comandă, sau să execute un script.

În cele ce urmează, vom descrie 'bash' (GNU Bourne-Again SHell), un interpreter linie de comandă compatibil cu 'sh' (POSIX Bourne Shell). El poate fi configurat să fie POSIX-conformant în mod implicit.

Din program, se poate lansa cu 'exec()' programul '/bin/bash' pentru a înlocui procesul curent. Notăm că cu 'system()' se lansează 'sh' (programul '/bin/sh') ca un proces copil care execută o singură comandă.

Din linia de comandă, se poate lansa cu:

```
bash [options] [command_string | file]
```

Lansează 'bash' pentru operare interactivă, pentru a executa o singură linie de comandă sau pentru a executa un script.

Comenzile executate de shell sunt preluate din linia de comandă (opțiunea '-c'), din fișierul 'file' sau de la standard input (dacă lipsește 'file').

Toate opțiunile shell de un caracter folosite la comanda internă 'set' pot fi folosite și la invocarea shell-ului.

În plus, 'bash' se poate invoca și cu următoarele opțiuni:

-c

comenzile sunt preluate din primul argument non-opțiune 'command_string'; dacă există argumente după 'command_string', primul este asignat lui '\$0' (numele shell-ului) iar următoarele sunt asignate parametrilor poziționali (acestea corespund argumentelor în linia de comandă ale unui script shell '\$0', '\$1', ... sau program C 'argv[0]', 'argv[1]', ...)

-i

shell-ul este interactiv

-l

shell-ul se comportă ca login shell

-s

dacă este prezentă opțiunea '-s' sau nu mai rămân argumente după procesarea opțiunilor, atunci comenzile sunt citite de la standard input; opțiunea '-s' permite setarea parametrilor poziționali când shell-ul este invocat interactiv sau când intrarea este citită dintr-un tub (pipe)

-v

afișază liniile de intrare pe măsură ce sunt citite

-x

afișază comenzi și argumentele lor pe măsură ce sunt executate

--

semnalează sfârșitul opțiunilor și dezactivează procesarea ulterioră a opțiunilor; argumentele aflate după '--' sunt tratate ca nume de fișiere și argumente.

Opțiuni multi-character - ele trebuie să apară în linia de comandă înaintea opțiunilor single-character:

--init-file file

--rcfile file

dacă shell-ul este interactiv, execută comenziile din 'file' în loc să le citească din fișierul de inițializare la nivel sistem (system wide initialization file) '/etc/bash.bashrc' și din fișierul de inițializare personal (standard personal initialization file) '~/.bashrc'

--noediting

dacă shell-ul este interactiv, nu folosește biblioteca 'GNU readline' pentru editarea liniei de comandă (de exemplu, nu se pot folosi tastele săgeată LEFT, RIGHT de deplasare cursor pentru editare)

--noprofile

nu execută comenziile din nici unul dintre fișierele '/etc/profile',
'~/.bash_profile', '~/.bash_login', '~/.profile'

--norc

nu execuță nici unul dintre '/etc/bash.bashrc', '~/.bashrc',
când shell-ul este interactiv

--posix

schimba comportamentul lui 'bash' acolo unde operația implicită
diferă de standardul POSIX, pentru a se potrivi cu standardul

Observații:

- Un proces shell este login shell dacă primul caracter al argumentului zero ('\$0') este '-' sau dacă a fost lansat cu opțiunea '-l'.
- Un proces shell este interactiv dacă a fost lansat fără argumente non-opțiune (cu excepția cazului când a fost specificat '-s') și fără opțiunea '-c', al căruia standard input și standard error sunt conectate la terminale (așa cum poate fi determinat cu 'isatty()'), sau a fost lansat cu opțiunea '-i'. La un asemenea proces shell, variabila '\$-' (care reține flag-urile ce descriu funcționarea shell-ului) conține 'i' iar aceasta permite unui script să testeze această stare.

- Un login shell execută automat la lansare scriptul '/etc/profile' și primul dintre '~/.bash_profile', '~/.bash_login', and '~/.profile' care a putut fi accesat, iar la ieșire execută '~/.bash_logout' (se poate dezactiva cu '--noprofile').
- Un shell interactiv care nu este login shell execută la lansare '/etc/bash.bashrc' și '~/.bashrc' (se poate dezactiva cu '--noprofile' sau se pot specifica alte fișiere cu '--rcfile').
- Când shell-ul este lansat ne-interactiv, el execută scriptul specificat în variabilele de environment 'BASH_ENV' (pentru căutarea fișierului script el nu folosește 'PATH').
- Dacă shell-ul este lansat cu EUID / EGID diferit de UID / GID și nu este prezentă opțiunea '-p', nu sunt executate fișiere startup, funcțiile shell nu sunt moștenite de la mediu, variabilele de environment 'SHELLOPTS', 'BASHOPTS', 'CDPATH', 'GLOBIGNORE' sunt ignorate, iar EUID este setat la UID.
Dacă este prezentă opțiunea '-p', comportamentul de startup este același dar EUID nu este resetat.
- Un fișier script pentru shell trebuie să fie un fișier text, cu dreptul de execuție setat.

Dacă rămân argumente după procesarea opțiunilor și nu este prezentă nici una dintre opțiunile '-c' sau '-s', primul argument se consideră că este numele unui script, '\$0' este setat la numele scriptului, iar parametrii poziționali sunt setați la argumentele rămase. Bash citește și execută comenzi din script, apoi se termină iar codul său de return furnizat către procesul părinte (exit status) este cel al ultimei comenzi executate din script, iar dacă nu s-au executat comenzi, codul de return este 0. Fișierul script este căutat mai întâi în directorul curent și apoi în directoarele din 'PATH'.

Exemple:

```
$bash -c 'echo $0 $1 $2'  
bash  
$bash -c 'echo $0 $1 $2' abc def ghi  
abc def ghi  
$bash -c 'echo $0 $1 $2 ; echo $0 $1 $2' abc def ghi  
abc def ghi  
abc def ghi  
$echo $0 $1 $2  
bash
```

(observăm că stringul de comandă poate conține o comandă compusă din mai multe comenzi simple iar parametrii poziționali se transmit la toate; ultimul 'echo' este executat de shell-ul inițial)

```
$bash -c 'ps -o ppid,pid,cmd'  
PPID      PID CMD  
3589      11779 bash  
11779      14003 ps -o ppid,pid,cmd  
$ps -o ppid,pid,cmd  
PPID      PID CMD  
3589      11779 bash  
11779      14005 ps -o ppid,pid,cmd
```

(observăm că comanda externă 'ps' este executată ca un copil al shell-ului inițial, ca și când procesul shell lansat de comanda 'bash' ar fi efectuat doar 'exec()', fără 'fork()')

```
$sh -c 'ps -o ppid,pid,cmd'  
PPID      PID CMD  
3589      11779 bash  
11779      13973 sh -c ps -o ppid,pid,cmd  
13973      13974 ps -o ppid,pid,cmd  
$ps -o ppid,pid,cmd  
PPID      PID CMD  
3589      11779 bash  
11779      13978 ps -o ppid,pid,cmd
```

(observăm că comanda externă 'ps' este executată ca un copil al unui copil 'sh' al shell-ului inițial, ca și când procesul shell lansat de comanda 'sh' ar fi efectuat 'fork()' și 'exec()')

```
$bash
$ps -o ppid,pid,cmd
    PPID      PID CMD
    3589      11779 bash
    11779      13926 bash
    13926      13932 ps -o ppid,pid,cmd
$exit
exit
$ps -o ppid,pid,cmd
    PPID      PID CMD
    3589      11779 bash
    11779      13934 ps -o ppid,pid,cmd
```

('bash' a lansat un copil interactiv al shell-ului inițial (similar dacă utilizam '-i'), prima comandă 'ps' și comanda 'exit' au fost introduse interactiv și executate de shell-ul copil)

```
$bash -s abc def ghi
$ps -o ppid,pid,cmd
    PPID      PID CMD
    3589      11779 bash
    11779    13900 bash -s abc def ghi
    13900    13906 ps -o ppid,pid,cmd
$echo $0 $1 $2
bash abc def
$exit
exit
$ps -o ppid,pid,cmd
    PPID      PID CMD
    3589      11779 bash
    11779    13911 ps -o ppid,pid,cmd
```

('bash' a lansat un copil interactiv al shell-ului inițial dar cu parametri poziționali - ei au inițializat '\$1', '\$2', '\$3', nu '\$0'; comenziile 'ps', 'echo' și 'exit' au fost introduse interactiv și executate de shell-ul copil)

```
$bash -c 'echo $-'  
hBc  
$bash -i  
$echo $-  
himBHs  
$exit  
exit
```

(primul 'bash' execută 'echo' neinteractiv; al doilea 'bash' este interactiv iar al doilea 'echo' și 'exit' sunt introduse interactiv; observăm flag-ul 'i' la lansarea interactivă)

```
$ls -l  
total 4  
-rwxrw-r-- 1 dragulici dragulici 34 Nov 30 00:15 myscript  
$cat myscript  
echo $0 $1 $2  
ps -o ppid,pid,cmd
```

(am pregătit un script (fișier text cu drept de execuție) pentru exemplul următor)

```
$./myscript abc def ghi
./myscript abc def
    PPID      PID CMD
    3589      11779 bash
    11779      13821 bash
    13821      13822 ps -o ppid,pid,cmd
$bash myscript abc def ghi
myscript abc def
    PPID      PID CMD
    3589      11779 bash
    11779      13823 bash myscript abc def ghi
    13823      13824 ps -o ppid,pid,cmd
$ps -o ppid,pid,cmd
    PPID      PID CMD
    3589      11779 bash
    11779      13827 ps -o ppid,pid,cmd
```

(am lansat scriptul și direct în shell-ul inițial și printr-o comandă 'bash').

TODO: de continuat ...

Cuprins

- 1 Sisteme de calcul, sisteme de operare**
 - Organizarea unui sistem de calcul
 - Funcțiile sistemului de operare
- 2 Sisteme de operare uzuale**
 - Incursiune în sistemul Windows
 - Incursiune în sistemul Linux
- 3 Interfața de programare C**
 - Obiecte software și apeluri sistem
 - Utilizatori și grupuri
 - Fișiere și directoare
 - Procese
 - Semnale
 - Tuburi

- 4 Biblioteca standard C (tipul 'FILE')**
 - Interfață interactivă linie de comandă
 - Interfața Command Prompt (Windows)
 - Interfața Shell (Linux)
- 5 Interfața de scripting linie de comandă**
 - Interfața Batch Scripting (Windows)
 - Interfața Shell Scripting (Linux)
- 6 Teme de laborator**
 - Utilizatori și grupuri
 - Fișiere, directoare
 - Procese, semnale, tuburi

Cuprins

- 1 Sisteme de calcul, sisteme de operare**
 - Organizarea unui sistem de calcul
 - Funcțiile sistemului de operare
- 2 Sisteme de operare uzuale**
 - Incursiune în sistemul Windows
 - Incursiune în sistemul Linux
- 3 Interfața de programare C**
 - Obiecte software și apeluri sistem
 - Utilizatori și grupuri
 - Fișiere și directoare
 - Procese
 - Semnale
 - Tuburi

- 4 Biblioteca standard C (tipul 'FILE')**
 - Interfață interactivă linie de comandă
 - Interfața Command Prompt (Windows)
 - Interfața Shell (Linux)
- 5 Interfața de scripting linie de comandă**
 - Interfața Batch Scripting (Windows)
 - Interfața Shell Scripting (Linux)
- 6 Teme de laborator**
 - Utilizatori și grupuri
 - Fișiere, directoare
 - Procese, semnale, tuburi

Pentru temele următoare, nu se va folosi 'system()':

1. Implementați funcția 'getpwnam()' folosind funcțiile 'setpwent()', 'getpwent()' și 'endpwent()'. Utilizați această funcție într-un program care primește ca argument în linia de comandă un username și afișază informația 'gecos' asociată.

Cuprins

- 1 Sisteme de calcul, sisteme de operare
 - Organizarea unui sistem de calcul
 - Funcțiile sistemului de operare
- 2 Sisteme de operare uzuale
 - Incursiune în sistemul Windows
 - Incursiune în sistemul Linux
- 3 Interfața de programare C
 - Obiecte software și apeluri sistem
 - Utilizatori și grupuri
 - Fișiere și directoare
 - Procese
 - Semnale
 - Tuburi

- Biblioteca standard C (tipul 'FILE')
- 4 Interfață interactivă linie de comandă
 - Interfața Command Prompt (Windows)
 - Interfața Shell (Linux)
- 5 Interfață de scripting linie de comandă
 - Interfața Batch Scripting (Windows)
 - Interfața Shell Scripting (Linux)
- 6 Teme de laborator
 - Utilizatori și grupuri
 - Fișiere, directoare**
 - Procese, semnale, tuburi

Pentru temele următoare, fișierele se pot considera oricât de mari, memoria folosită în program să fie limitată de o constantă (să fie în $O(1)$) și nu se va folosi apelul 'system()':

1. Scrieți un program care primește ca argumente în linia de comandă două fișiere 'f1' și 'f2' și realizează efectul comenzii 'grep -f f1 f2'. Se vor folosi funcții de nivel superior (tipul 'FILE').
2. Scrieți o funcție 'int eof(ind d);' care testează dacă poziția curentă din fișierul cu descriptorul 'd' este la sfârșitul fișierului, folosind 'lseek()' și 'stat()'. Funcția returnează 1 (da), 0 (nu), -1 (eroare). Utilizați funcția într-un program demonstrativ care copiază standard input la standard output (până la end of file).
3. Implementați comanda 'mv fișier1 fișier2' folosind 'link()', 'unlink()', 'symlink()'. Dacă directorul destinație este pe același disc, se înlocuiește legătura fizică; dacă este pe alt disc iar fișierul nu are alte legături fizice pe discul sursă, se mută fișierul, altfel se crează o legătură simbolică către 'fișier1'. Dacă 'fișier2' există, numele se va re-asocia.

4. Scrieți un program care compară din punct de vedere lexicografic conținutul a două fișiere (privite ca secvențe de caractere). Specificatorii fișierelor sunt dați ca argumente în linia de comandă. Se vor folosi funcții de nivel superior (tipul 'FILE').
5. Scrieți un program care determină dacă conținutul unui fișier este inclus (contiguu) în conținutul altui fișier și de câte ori (numărul aparițiilor). Se vor considera și aparițiile care se suprapun. Specificatorii fișierelor sunt dați ca argumente în linia de comandă. Se vor folosi funcții de nivel superior (tipul 'FILE').
6. Scrieți un program care sortează prin interclasare un fișier de caractere dat ca argument în linia de comandă, în maniera descrisă mai jos.

Sortarea prin interclasare presupune împărțirea sirului în două jumătăți, sortarea fiecărei părți prin aceeași metodă (deci recursiv), apoi interclasarea celor două părți (care sunt acum sortate).

Programul va lucra astfel: se împarte sirul în două, se generează două procese copil care sortează cele două jumătăți în paralel, tatal îi așteaptă să se termine, apoi interclasează jumătătile.

Nu se vor folosi fișiere auxiliare.

7. Implementați comanda 'wc [OPTION]... [FILE]...'.

Pot fi date ≥ 0 fișiere, ≥ 0 opțiuni, iar opțiunile pot lipsi sau pot fi un '-' urmat de combinații ale literelor 'c', 'w', 'l'. Se vor folosi funcții de nivel superior (tipul 'FILE').

8. Implementarea matricilor dreptunghiulare de numere reale ca fișiere binare:

O matrice va fi stocată într-un fișier binar conținând elementele sale pe linii (în format intern), plus o informație suplimentară din care să se poată deduce numărul de linii și de coloane. Scrieți funcții pentru următoarele operații:

void new(f,m,n) : crează în fișierul f o matrice m x n inițializată cu 0;

float get(f,i,j) : returnează elementul de pe poziția i,j din matricea
stocată în fișierul f;

void set(f,i,j,x) : scrie elementul real x pe poziția i,j în matricea
stocată în fișierul f.

Se vor folosi funcții de nivel superior (tipul 'FILE').

Scrieți programe care folosesc aceste funcții pentru a calcula suma și produsul a două matrici. Programele se vor apela sub forma:

sum f1 f2 f

pro f1 f2 f

unde f1, f2 sunt specificatorii fișierelor conținând matricile sursă, iar specificatorul fișierului care va conține matricea destinație. Fișierele f1 și f2 vor fi generate în prealabil cu alte programe ajutătatoare.

9. Scrieți un program care primește ca argument în linia de comandă un director și afișază arborescența de directoare și fișiere cu originea în el (similar comenzii tree /f din DOS). Se vor afișa linii din caractere '-', '+', '|' care unesc nodurile arborescenței.
10. Scrieți un program care primește ca argument în linia de comandă un director 'd' și emulează comanda 'du -sb d' (afișază dimensiunea totală în octeți a fișierelor și directoarelor din arborescența cu originea în el).
11. Scrieți un program care primește ca argumente în linia de comandă un fișier și un director și determină dacă fișierul se află ca nume și conținut în arborescența cu originea în director. În caz afirmativ, se va determina numărul de apariții.
12. Scrieți un program care primește ca argumente în linia de comandă două directoare 'd1' și 'd2' și emulează comanda 'cp -r d1 d2' (copiază recursiv arborescența cu originea în 'd1' într-o arborescență cu originea în 'd2'). Dacă 'd2' există, 'd1' și arborescența sa se vor copia ca subarbore în 'd2', cu același nume 'd1'.

13. Scrieți un program care primește ca argumente în linia de comandă două directoare 'd1' și 'd2' și emulează comanda 'diff -r d1 d2' (compară recursiv arborescențele cu originile în 'd1' și 'd2', afișând specificatorii fișierelor care apar doar într-una dintre arborescențe sau apar în ambele dar le diferă conținutul). Nu este necesar să se afișeze care sunt diferențele conținuturilor fișierelor.
14. Scrieți un program care primește ca argumente în linia de comandă două directoare 'd1' și 'd2' și șterge din arborescența cu originea în 'd2' toate fișierele și directoarele care apar cu același cale și nume și în arborescența cu originea în 'd1'.

15. Implementați comanda 'realpath specifier'.

Indicație: de la directorul său părinte al ultimei componente a lui 'specifier' se pornește ascendent, căutând i-nodul fiecărui director în directorul său părinte '..', până la rădăcina sistemului (care este propriul său părinte), reținând numele directoarelor într-o listă; apoi, lista se parcurge invers.

16. Implementați comanda 'realpath --relative-to=DIR specifier'.

Indicație: de la directorul său părinte al ultimei componente a lui 'specifier' și de la 'DIR' se pornește ascendent, căutând i-nodul fiecărui director în directorul său părinte '..', până la întâlnirea unui același director (i-nod) și reținând numele directoarelor în două liste; apoi, se compun listele, parcurgând lista lui 'DIR' direct și cea a lui 'specifier' invers.

17. Scrieți următoarele funcții C:

```
int ar(const char *arh, const char *src);  
int unar(const char *arh, const char *dst);
```

unde 'arh' este un specificator de fișier, 'src' și 'dst' sunt specificatori de fișiere sau directoare. Funcția 'ar()' arhivează sursa 'src' sub forma fișierului 'arh'; dacă 'src' este un director, se arhivează toată arborescența cu originea în el. Funcția 'unar()' dezarchivează arhiva 'arh', obținându-se destinația 'dst', care poate fi un fișier sau o arborescență. Putem presupune că arborescențele conțin doar fișiere regulate și directoare. Toți specificatorii pot conține și cale.

O arhivă este un fișier ce conține o succesiunea de înregistrări ce corespund fișierelor și directoarelor arhiveate. O înregistrare conține, în ordine:

- specificatorul fișierului / directorului arhivat (șir de caractere, terminat cu '\0'); el va conține calea relativă față de rădăcina arborelui arhivat;
- imaginea internă a unei structuri 'stat' cu informații despre fișier (ea conține și informația privind dimensiunea fișierului);
- conținutul fișierului.

Dezarchivarea va restaura pentru fiecare fișier / director extras câmpurile 'st_mode', 'st_size', 'st_atim', 'st_mtim', 'st_ctim' ale structurii 'stat' corespunzătoare.

Scrieți un program 'myar' care se poate lansa sub forma 'myar ar arh src' sau 'myar unar arh dst' ('arh', 'src', 'dst' sunt ca mai sus) și care realizează fie arhivarea fie dezarchivarea.

Cuprins

- 1 Sisteme de calcul, sisteme de operare
 - Organizarea unui sistem de calcul
 - Funcțiile sistemului de operare
- 2 Sisteme de operare uzuale
 - Incișiune în sistemul Windows
 - Incișiune în sistemul Linux
- 3 Interfața de programare C
 - Obiecte software și apeluri sistem
 - Utilizatori și grupuri
 - Fișiere și directoare
 - Procese
 - Semnale
 - Tuburi

- Biblioteca standard C (tipul 'FILE')
- 4 Interfață interactivă linie de comandă
 - Interfața Command Prompt (Windows)
 - Interfața Shell (Linux)
- 5 Interfață de scripting linie de comandă
 - Interfața Batch Scripting (Windows)
 - Interfața Shell Scripting (Linux)
- 6 Teme de laborator
 - Utilizatori și grupuri
 - Fișiere, directoare
 - Procese, semnale, tuburi**

Pentru temele următoare, nu se va folosi 'system()' :

1. Scrieți un program care primește ca argumente în linia de comandă un număr 'n' și o posibilă linie de comandă 'com arg1 ... argk' ($k \geq 0$) și realizează efectul comenzi 'nice -n com arg1 ... argk'.
2. Scrieți un program care își concatenează valoarea fiecărei variabile de environment de pe poziție pară la sfârșitul valorii variabilei pe poziția impară anterioară. Pozițiile încep de la 0 (par). Programul își va afișa environmentul înainte și după această operație.
3. Implementați un shell cu următoarele facilități:
 - acceptă comanda internă 'exit';
 - acceptă comenzi externe cu un număr oarecare de argumente;
 - acceptă rularea de script-uri; script-ul poate conține o listă de comenzi utilitare și poate accepta argumente în linia de comandă.
4. Implementați un shell cu următoarele facilități:
 - acceptă comanda internă 'exit';
 - acceptă comenzi externe cu un număr oarecare de argumente;
 - acceptă filtre de comenzi externe; filtrul poate avea un număr oarecare de componente iar comenziile pot avea un număr oarecare de argumente.

5. Scrieți un program care generează prin backtracking recursiv permutările mulțimii $\{1, \dots, n\}$ (n citit de la standard input), în aşa fel încât de fiecare dată când se completează o poziție, pentru a genera toate continuările posibile nu se intră în apel recursiv ci se lansează un proces copil care generează aceste continuări; nu se va aștepta terminarea copilului pentru a încerca o altă valoare pentru poziția curentă și astfel vectorii sunt construiți în paralel. Procesul inițial va aloca un tub anonim și toate procesele generate vor moșteni descriptorii la el; când un proces găsește o permutare, o scrie în tub, împreună cu un cap de linie; procesul inițial va citi permutările din tub și le va scrie pe standard output. Oricare dintre procese nu se termină decât după ce i s-au terminat toții copiii.

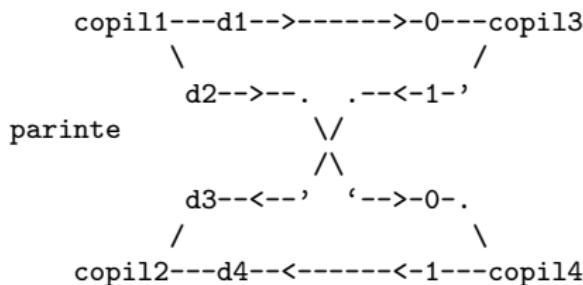
6. Scrieți un program care numără aparițiile unui sir de caractere ca subcuvânt în alt sir de caractere cu procese paralele (cele două siruri sunt date ca argumente în linia de comandă). De fiecare dată când se verifică dacă primul sir apare ca subcuvânt începând de la o poziție, verificarea se va face de către un proces copil iar procesul părinte nu va aștepta ca acesta să se termine pentru a iniția o căutare începând de la o altă poziție - astfel, verificările au loc în paralel. Fiecare proces copil furnizează părintelui răspunsul 'găsit' sub forma unui semnal SIGUSR1; handlerul asociat în părinte va incrementa un contor. După inițierea tuturor căutărilor, părintele așteaptă primirea semnalelor și terminarea tuturor copiilor, apoi afișază contorul. Se va asigura protecția la pierderea de semnale.

7. Scrieți un program care sortează prin interclasare o secvență de întregi în maniera descrisă mai jos.

Sortarea prin interclasare presupune împărțirea secvenței în două jumătăți, sortarea fiecărei părți prin aceeași metodă (deci recursiv), apoi interclasarea celor două părți (care sunt acum sortate).

Programul va citi secvența inițială de la standard input până la end of file, va scrie secvența ordonată la standard output și va lucra astfel:

- dacă survine end of file după ≤ 1 element, procesul îl scrie pe acesta (secvența este sortată) și se termină;
- dacă se citește și al doilea element, procesul alocă 4 tuburi anonime și generează 4 procese copil, care comunică astfel:



(liniile reprezintă tuburile și precizează sensul în care circulă informația și descriptorii asociați lor, d1 - d4 sunt descriptori nestandard, copiii 1 și 2 au păstrat descriptorii 0, 1 asociați la fel ca părintele);

copil1 citește secvența inițială de la standard input (este intrarea părintelui) până la end of file și scrie alternativ numerele în cele 2 tuburi, apoi se termină;

copil2 citește secvențele din cele 2 tuburi până la end of file (ele vor fi sortate), le interclasează pe măsură ce le citește și le scrie pe standard output (este ieșirea părintelui), apoi se termină;

copil3 și copil4 citesc secvențele de la standard input până la end of file, le sortează și le scriu la standard output, lucrând la fel ca și părintele (putând aloca și ei 4 tuburi și genera 4 copii), apoi se termină;

după alocarea tuburilor și generarea copiilor, părintele așteaptă terminarea lor și se termină;

procesele își vor închide descriptorii nefolosiți pe tuburi (astfel, terminarea proceselor care scriu în tuburi le va permite celor care citesc din ele să perceapă end of file).

Considerăm problema 'cautare binară':

Dorim să determinăm numărul r al aparițiilor unui număr p într-o secvență de numere n_1, \dots, n_k , $k \geq 0$, printr-o strategie de tip divide et impera:

- dacă $k = 0$, atunci $r = 0$;
- dacă $k = 1$, atunci: dacă $p = n_1$, atunci $r = 1$, altfel $r = 0$;
- altfel, se consideră jumătatea $j = (1 + k) \text{ div } 2$, se determină numerele r_1, r_2 ale aparițiilor lui p în secvențele n_1, \dots, n_j , resp. n_{j+1}, \dots, n_k , iar $r = r_1 + r_2$.

Căutarea lui p într-o (sub) secvență n_i, \dots, n_j se va face cu un proces copil.

Un asemenea proces:

- primește de la părinte p, n_i, \dots, n_j (sau p, n_1, \dots, n_k, i, j);
- furnizează către părinte r ;
- dacă $i > j$, generează 2 procese copil care numără aparițiile lui p în cele 2 subsecvențe; va furniza către părintele său suma numerelor furnizate de copii; copii vor rula în paralel, părintele va aștepta terminarea tuturor copiilor săi înainte de a se termina.

Procesul inițial va și afișa valoarea calculată.

8. Rezolvati problema 'căutare binară' a.î. fiecare proces primește de la părinte datele p , n_1 , ..., n_k ca argumente în linia de comandă și pozițiile i, j, între care să caute ca date globale (duplicate la procesele copil) și furnizează părintelui rezultatul ca și cod de return proces (valoarea returnată de 'main()' sau argument al funcției 'exit()').
9. Rezolvați problema 'cautare binară' a.î. fiecare proces primește de la părinte datele p , n_1 , ..., n_k ca argumente în linia de comandă și pozițiile i, j, între care să caute ca date globale (duplicate la procesele copil) și furnizează părintelui rezultatul r sub forma a r semnale SIGUSR1; handlerul asociat semnalului în părinte va numara semnalele cu un contor global. Se va asigura protecția la pierderea semnalelor.
10. Rezolvați problema 'cautare binară' a.î. fiecare proces primește de la părinte datele p , n_1 , ..., n_k ca argumente în linia de comandă și pozițiile i, j, între care să caute printr-un tub și furnizează părintelui r tot printr-un tub (între orice părinte - copil există câte 2 tuburi).

Considerăm următorul 'joc turnuri':

Pe masă sunt k turnuri de discuri, de dimensiuni m_1, \dots, m_k ($k \geq 0, m_i \geq 1$ diferite).

Doi jucători, 0 și 1, mută alternativ.

La o mutare, jucătorul împarte unul din turnurile existente în două, a.î. turnurile rezultate să aibă în continuare dimensiuni ≥ 1 diferite.

Pierde jucătorul care nu mai poate muta (nu mai poate rezulta o configurație cu turnuri de dimensiuni ≥ 1 diferite).

Se pune problema dacă pentru o configurație dată, jucatorul curent (cel aflat la mutare) are strategie de câștig.

Formalizare:

- configurație: (p, k, m_1, \dots, m_k) , p este 0 sau 1 (jucătorul curent), $k \geq 0, m_i \geq 1$ diferite;
- mutare: $(p, k, m_1, \dots, m_k) \rightarrow (1 - p, k + 1, n_1, \dots, n_{k+1})$ pereche de configurații, a.î. există $1 \leq i \leq k$ și $1 \leq u, v \leq k + 1$ a.î. $\{m_1, \dots, m_k\} \setminus \{m_i\} \cup \{n_u, n_v\} = \{n_1, \dots, n_{k+1}\}$ și $m_i = n_u + n_v$;
- configurația C e cu strategie de câștig pentru jucatorul curent:

$$S(C) \iff \exists K \text{ a.î. } ((C \rightarrow K) \wedge \neg S(K))$$

Obs: $\neg S(C) \iff \forall K \text{ a.î. } C \rightarrow K$, avem $S(K)$ (în particular, dacă $\neg \exists K \text{ a.î. } C \rightarrow K$ (i.e. înfrângere pentru jucatorul curent), atunci $\neg S(C)$).

Se cere un program care primește ca argumente în linia de comandă componentele numerice p, k, m_1, \dots, m_k ale unei configurații C și:

- dacă $S(C)$, atunci afișază pe standard output componentele numerice ale unei prime mutări K a.i. $C \rightarrow K$ și $\neg S(K)$;
- altfel, afișază 'NU'.

Testarea $S(C)$ se va face cu un proces copil care:

- primește de la părinte configurația C ;
- pentru toate K a.î. $C \rightarrow K$, generează un proces copil care testează $S(K)$; copiii vor rula în paralel, părintele va aștepta terminarea tuturor înainte de a se termina;
- furnizează părintelui răspunsul adevărat sau fals la testarea $S(C)$ calculat din răspunsurile copiilor (în particular, dacă nu există K a.î. $C \rightarrow K$, va furniza fals) iar în cazul răspunsului adevărat îi va furniza și configurația K transmisă unuia dintre copii care i-a furnizat adevărat.

Procesul inițial va lansa copilul care testează configurația inițială și va afișa configurația furnizată de acesta sau 'NU'.

11. Rezolvați problema 'joc turnuri' a.î. fiecare proces primește de la părinte datele p, k, m_1, \dots, m_k ca argumente în linia de comandă și furnizează părintelui răspunsul fals / adevărat ca și cod de return proces 0 / 1 (valoarea returnată de 'main()' sau argument al funcției 'exit()') și configurația următoare 1 – $p, k + 1, n_1, \dots, n_{k+1}$ printr-un fișier temporar unic creat și șters în părinte ('mkstemp()').
12. Rezolvați problema 'joc turnuri' a.î. fiecare proces primește de la părinte datele p, k, m_1, \dots, m_k ca argumente în linia de comandă și furnizează părintelui răspunsul fals / adevărat ca un semnal SIGUSR1 / SIGUSR2 și configurația următoare 1 – $p, k + 1, n_1, \dots, n_{k+1}$ printr-un fișier temporar unic creat și șters în părinte ('mkstemp()').
13. Rezolvați problema 'joc turnuri' a.î. fiecare proces primește de la părinte datele p, k, m_1, \dots, m_k printr-un tub și furnizează părintelui răspunsul fals / adevărat ca 0 / 1 și configurația următoare 1 – $p, k + 1, n_1, \dots, n_{k+1}$ printr-un alt tub (între orice părinte - copil există câte 2 tuburi).

14. Scrieți un program 'nrc' care se lansează sub forma:

nrc com

unde 'com' este o comandă externă (adică asociată unui fișier executabil de pe disc) având eventual și argumente în linia de comandă și care lansează comanda 'com', numărând cuvintele scrise de ea pe standard output. În acest scop, procesul 'nrc' crează un tub fără nume, apoi generează un proces copil ('fork()') a.î. intrarea standard a părintelui și ieșirea standard a copilului să fie în acest tub, apoi copilul se înlocuiește ('exec()') cu un proces ce execută 'com'.

15. Implementați tipul arbore binar cu vârfuri numere întregi alocat înlănțuit (cu pointeri). Implementarea trebuie să permită și arborele vid. Scrieți o funcție care primește ca parametru un arbore (pointer la rădăcina sa) și afișază vârfurile sale parcurgându-l pe niveluri.

În general, algoritmul de parcursere pe niveluri folosește o coadă în care inițial se încarcă rădăcina, apoi într-un ciclu care ține cât timp coada e nevidă se extrage un vârf, se afișază, apoi se introduc în coadă copiii lui (pointeri la rădăcinile lor). Pe post de coadă, funcția va folosi un tub fără nume (care va exista doar pe perioada apelului).

Scrieți un program ilustrativ pentru această funcție.

16. Scrieți un program '@paralel' care se lansează sub forma:

@paralel com1 @, com2 @, ... @, comn @endparalel

(tot ce e după '@paralel' sunt argumentele lui), unde 'com1', ..., 'comn' sunt comenzi externe (adică asociate unor fișiere executabile) având eventual și argumente în linia de comandă (deci 'comi' este un sir de cuvinte) și lansează n+2 procese care se desfășoară în paralel:

- câte un proces care execută fiecare dintre 'comi';
- un proces de intrare care citește caractere de la intrarea standard și multiplică fiecare caracter în n copii, pe care le trimită în niște tuburi conectate fiecare la intrarea standard a uneia din 'comi' (deci fiecare din 'comi' are intrarea standard redirectată la câte un tub);
- un proces de ieșire care citește caractere dintr-un tub și le scrie la ieșirea standard; la tubul respectiv sunt conectate ieșirile standard ale proceselor 'comi'.

Schema de funcționare va fi:



Tuburile sunt fără nume și există doar pe perioada execuției acestor procese.

Procesul de intrare are SIGPIPE anihilat, pentru a nu se termina dacă 'comi' nu se termină toate în același timp (și își închid tubul de intrare la citire).

Pentru a nu crea confuzii, în scrierea comenziilor 'comi' nu va apărea @.

Aplicați programul "@paralel" pentru a crea generalizări ale comenziilor filtru în care în loc de un lanț com1 | ... | comn avem un graf.

17. Un arbore binar poate fi descris printr-o expresie cu paranteze complete într-o din variantele următoare:

```
arbore := @  
arbore := eticheta [ arbore , arbore ]
```

unde '@' înseamnă (sub)arborele vid iar 'eticheta' este eticheta unui nod.

Scrieți un program care primește în linia de comandă expresia unui arbore de programe ('@' înseamnă un program care doar se termină, etichetele sunt specifiatorii unor fișiere executabile existente iar componentele expresiei sunt separate prin spații) și îl instanțiază ca arbore de procese care se execută în paralel; fiecare proces execută un nod, fiecare proces diferit de '@' are doi copii, cu care este conectat prin câte un tub anonim; fiecare părinte scrie în tubul copilului stâng prin standard output și în tubul copilului drept prin standard error, fiecare copil citește din tubul părintelui prin standard input. Toate procesele își vor închide descriptorii nefolosiți pe tuburi.

De exemplu, dacă programul primește în linia de comandă:

```
p1 [ @ , p2 [ p3 [ @ , @ ] , p4 [ @ , @ ] ] ]
```

Se va crea arborele de procese:

```
p1
|---2--->---0--- p2
|           |---2--->---0--- p4
|           |           |---2--->---0--- @
|           |           |---1--->---0--- @
|
|           |---1--->---0--- p3
|           |           |---2--->---0--- @
|           |           |---1--->---0--- @
|
|---1--->---0--- @
```

(p1, p2, p3, p4 sunt fișiere executabile, liniile orizontale punctate sunt tuburi, pe fiecare tub este marcat sensul de circulație a informației prin ' '> și sunt notați descriptorii prin care citesc/scriu procesele din capete).

18. Scrieți un program 'ex' care se poate lansa sub forma:

ex com1 com2 arg1 ... argk

($k \geq 0$) și care lansează comanda 'com1' dându-i ca argumente în linia de comandă cuvintele scrise la standard output de comanda 'com2 arg1 ... argk'.

Indicație: procesul 'ex' crează un tub fără nume, apoi generează un proces copil a.î. intrarea standard a părintelui și ieșirea standard a copilului să fie în acest tub, copilul se înlocuiește cu un proces ce execută 'com2 arg1 ... argk', părintele 'ex' scrie într-o zonă alocată dinamic (și redimensionată progresiv pe măsură ce este nevoie) cuvântul 'com1' și cuvintele citite din tub, apoi părintele lansează un alt proces copil care se înlocuiește cu un proces ce execută comanda cu argumente scrisă în zona alocată dinamic (cu 'exec()', nu cu 'system()'), iar în final părintele 'ex' dezaloca zona alocată dinamic, așteaptă terminarea tuturor copiilor săi și se termină.

19. Scrieți un program care generează un proces copil iar acesta trimite părintelui un număr aleator de semnale SIGUSR1; copilul număra câte a trimis, părintele număra câte a primit, apoi fiecare afișază numarul respectiv. Părintele se va termina la primirea SIGCHLD și va culege codul de return al copilului ('wait()' sau 'waitpid()'). Se va asigura protecția la pierderea unor semnale.

20. Scrieți un program care genereaza un proces copil iar acesta trimite părintelui un număr aleator de semnale != SIGKILL, SIGSTOP, SIGCONT, iar la sfârșit îi trimite un SIGTERM. Părintele face o statistică, numărând câte semnale a primit din fiecare tip, iar la SIGTERM o afișază, apoi așteaptă terminarea copilului și se termină. Se va asigura protecția la pierderea unor semnale.
21. Scrieți un program care genereaza un proces copil iar acesta trimite părintelui un număr aleator de semnale != SIGKILL, SIGSTOP, SIGCONT, SIGCHLD. Părintele determină semnalul de valoare maximă și, la terminarea copilului (SIGCHLD), îl afișază, apoi culege codul de return al copilului ('wait()') sau 'waitpid()' și se termină. Se va asigura protecția la pierderea unor semnale.
22. Scrieți un program care genereaza un proces copil iar acesta trimite părintelui un număr aleator de semnale != SIGKILL, SIGSTOP, SIGCONT, iar la sfârșit îi trimite un SIGTERM. Părintele, de fiecare dată când primește un semnal, îi scrie valoarea și timpul scurs de la începutul execuției (în secunde), apoi blochează semnalul respectiv - astfel nu sunt tratate semnalele repetate. La primirea SIGTERM, părintele așteaptă terminarea copilului și se termină. Se va asigura protecția la pierderea primului semnal trimis de fiecare tip.

23. Scrieți un program care genereaza un proces copil iar acesta trimite părintelui în continuu semnale SIGUSR1. Părintele armeaza ceasul ('alarm()') pentru 4 secunde, apoi începe să numere semnalele SIGUSR1 primite. După primirea lui SIGALRM scrie câte a primit și trimite copilului SIGKILL (așa se va termina copilului). Înainte de terminare, părintele culege codul de return al copilului ('wait()' sau 'waitpid()'). Se va asigura protecția la pierderea unor semnale.
24. Scrieți un program care genereaza un proces copil iar părintele și copilul își trimit unul altuia un număr aleator de semnale SIGUSR1, apoi un SIGINT. Fiecare proces se va termina la SIGINT, părintele va aștepta terminarea copilului, iar înainte de terminare, fiecare proces va scrie câte semnale a trimis și câte a primit. Se va asigura protecția la pierderea unor semnale.

25. Talk bazat doar pe semnale: un același program este lansat de la 2 terminale diferite de același utilizator, obținând 2 procese diferite. Fiecare proces citește de la tastatură PID-ul celuilalt. Fiecare proces are un tabel care asociază câte un semnal != SIGKILL, SIGCONT, SIGSTOP celor 26 litere, blank-ului și capului de linie. Fiecare proces citește într-un ciclu câte o linie de la tastatură, apoi o parcurge și trimite celuilalt proces semnalul asociat fiecărui caracter din ea (inclusiv blank-urile și capul de linie). De asemenea, fiecare proces, la primirea unui asemenea semnal, va afla caracterul corespunzător și-l va scrie pe ecran. Se va asigura protecția la pierderea unor semnale și se va asigura ca semnalele trimise de un proces să fie primite de celălalt în aceeași ordine (de exemplu un proces nu va emite semnalul corespunzător unui caracter decât dacă a primit confirmarea că celălalt proces a tratat semnalul pentru caracterul precedent).
26. Variantă a problemei 'Talk bazat doar pe semnale': Fiecare caracter care trebuie trimis este scris pe 8 biți iar secvența de biți este trimisă ca o succesiune de semnale SIGUSR1 și SIGUSR2.

27. Simulați funcția 'execvpe()' cu ajutorul funcțiilor 'execve()' și 'getenv()'. Funcția va căuta în lista de directoare conținută în variabila de environment 'PATH' a procesului apelant fișierele executabile al căror specificator începe cu un nume, va completa calea și va executa fișierul cu 'execve()'. Dacă procesul apelant nu are variabila 'PATH' sau ea conține sirul vid, se va căuta în directorul curent.

Scrieți un program 'run' care se poate lansa sub forma 'run lista com arg...', unde 'lista' este o listă, eventual vidă (se poate da ""), de specificatori de director, eventual siruri vide, separați prin ":" , fără spații, iar 'com arg...' este o comandă oarecare, care poate avea și argumente, și efectuează următoarele:

- generează un proces copil, pe care îl așteaptă să se termine;
- copilul își asignează variabila de environment 'PATH' cu conținutul listei 'lista', apoi se înlocuiește cu un proces ce execută comanda cu argumente, folosind noua funcție.