Universitatea “Alexandru Ioan Cuza” Iaşi
Facultatea de Informatică
Departamentul de Învăţământ la Distanţă

Cristian VIDRAŞCU

SISTEME DE OPERARE

2006

Cuprins

Prefaţă

Manualul de faţă, utilizat pentru disciplina "Sisteme de operare" la Anul 1, Semestrul II, Secţia IDD, are drept scop prezentarea unui sistem de operare concret, venind în completarea disciplinei "Arhitectura calculatoarelor şi sisteme de operare" din Anul 1 Semestrul I, care a prezentat conceptele teoretice ce stau la baza sistemelor de calcul şi a sistemelor de operare pentru operarea acestora, folosite în trecut şi în zilele noastre.

Înainte de a începe studiul acestui manual, vă recomand să recitiţi manualul disciplinei "Arhitectura calculatoarelor şi sisteme de operare" pentru a vă reîmprospăta cunoştinţele referitoare la arhitectura sistemelor de calcul şi noţiunile de bază despre sisteme de operare.

Un sistem de operare are un rol foarte important într-un sistem de calcul, el este *software*-ul care asigură interacţiunea dintre utilizatorul uman şi partea de *hardware* (*i.e.*, componentele fizice) a calculatorului exploatat de acesta, precum şi buna funcţionare a programelor de aplicaţii rulate de utilizator.

Sistemul de operare concret ales pentru prezentare în cadrul acestei discipline este un sistem de operare ce capătă în ultima vreme un tot mai pronunţat rol, şi anume `Linux`-ul.

Motivele acestei alegeri sunt multiple. În primul rînd, `Linux`-ul este unul dintre cele mai recente şi totodată venerabile sisteme de operare din lumea informaticii. Este recent, deoarece s-a născut în anul 1991 ca proiect studenţesc al celebrului de-acum *Linus Torvalds*. Este un sistem "venerabil", deoarece `Linux` moşteneşte caracteristicile sistemului de operare `UNIX`, apărut la sfîrşitul anilor '60, şi care a reprezentat un salt tehnologic spectaculos în lumea sistemelor de operare şi a informaticii în general. `UNIX`-ul a fost un catalizator pentru apariţia unor "minuni" tehnologice precum limbajul de programare C ori reţelele de calculatoare şi INTERNET-ul, a căror apariţie şi dezvoltare a fost strîns împletită cu evoluţia `UNIX`-ului.

În al doilea rînd, celălalt exemplu concret pe care l-aş fi putut alege, pe baza răspîndirii largi a acestuia, ar fi fost sistemul `MS-Windows`. Am preferat să nu fac lucrul acesta, datorită faptului că `Windows`-ul este deja un sistem binecunoscut şi utilizat de marea majoritate a utilizatorilor de calculatoare, inclusiv majoritatea disciplinelor practice de la Facultatea de Informatică folosesc ca suport sistemul `Windows` (menţionez în acest context faptul că în anii superiori veţi studia o disciplină dedicată "Programării în `Windows`", precum şi cursuri opţionale dedicate *framework*-ului `Dot NET` şi altor tehnologii MICROSOFT).

Un alt motiv serios pentru alegerea `Linux`-ului, şi care a fost şi un factor determinant în răspîndirea acestuia, este faptul că `Linux`-ul este disponibil gratuit; mai mult decît atît, este un *software open source*, adică sunt disponibile şi sursele programului, tot gratuit, un lucru important pentru programatori, care au astfel posibilitatea de a studia codul sursă, de a învăţa din el, şi de a-l adapta propriilor necesităţi.

Obiectivele generale ale cursului sunt reprezentate de o *prezentare generală* a `Linux`-ului (partea I a acestui manual) pe de o parte, şi de *programarea concurentă* în limbajul C sub `Linux` (partea II a acestui manual), pe de altă parte.

Prima parte a manualului este dedicată prezentării sistemului de operare `Linux`, şi este valabilă în general pentru toate sistemele de operare din familia `UNIX`. Această parte se vrea a fi un *ghid de utilizare* a sistemului `Linux`, fără a avea pretenţia de a fi un ghid complet. Am preferat să insist în prezentare pe conceptele fundamentale pe care se bazează acest sistem de operare, fără să obosesc cititorul cu prea multe detalii şi numeroasele opţiuni ale unor comenzi (mai mult, am preferat să nu prezint deloc opţiunile numeroase ale

meniurilor de aplicaţii grafice, din lipsă de spaţiu, şi deoarece se găsesc destule cărţi care tratează acest aspect, cum ar fi cele publicate în editura TEORA).

Ultima parte a manualului este dedicată programării concurente. Să vedem mai întîi ce înseamnă noţiunea de *programare concurentă*?

În primul semestru, la disciplina "Algoritmică şi programare" aţi învăţat *programare secvenţială*: programele scrise de dumneavoastră erau caracterizate prin faptul că aveau un singur *fir de execuţie* (adică un singur flux de instrucţiuni ce erau executate de procesor) şi rulau de sine-stătător, fără să coopereze cu alte programe (pentru realizarea unui obiectiv comun).

Acest tip de programare corespunde perioadei iniţiale din istoria sistemelor de calcul, cînd sistemele folosite erau *seriale*: la un moment dat se executa un singur program, unitatea centrală fiind acaparată de acesta din momentul începerii execuţiei lui şi pînă în momentul terminării acestuia (neputînd astfel fi folosită pentru execuţia altor programe pe toată durata de execuţie a acelui program).

Apoi sistemele de calcul au evoluat prin introducerea tehnicii de *multi-programare*, ce permitea utilizarea concomitentă a unităţii centrale pentru execuţia mai multor programe; această utilizare "simultană" a *CPU*-ului de către mai multe programe a fost posibilă datorită faptului că *hardware*-ul permitea realizarea operaţiilor de intrare/ieşire (*i.e.*, transferul de date între periferic şi memoria internă) independent de operaţiile *CPU*, şi atunci, în timp ce un program aştepta realizarea unei operaţii de intrare/ieşire (viteza de transfer cu perifericul fiind mult mai mică decît viteza de lucru a *CPU*-ului), procesorul era alocat (de către sistemul de operare) altui program pentru a executa o porţiune din instrucţiunile acestuia.

Astfel a apărut un nou tip de programare, numită uneori *programare paralelă*, denumire ce provine de la faptul că avem mai multe programe executate "în paralel" (*i.e.*, simultan, în acelaşi timp), sau *programare concurentă*, denumire mai potrivită pentru că surprinde aspectul concurenţial al execuţiei mai multor programe prin tehnica multi-programării: programele rulează concomitent şi *concurează* unele cu altele pentru resursele sistemului de calcul (procesor, memorie, periferice de intrare/ieşire), puse la dispoziţia programelor şi gestionate de către sistemul de operare.

Ca atare, scopul principal al părţii a doua a manualului este acela de a vă învăţa conceptele fundamentale ale programării concurente, utilizînd pentru aceasta sistemul `Linux` ca suport, şi C-ul ca limbaj de programare. Acesta este cel mai bun cadru de predare al programării concurente, pentru că permite concentrarea atenţiei asupra aspectelor referitoare la execuţia mai multor programe în regim concurenţial de folosire a resurselor calculatorului, fără să ne distragă atenţia aspectele referitoare la interfaţa cu utilizatorul a programelor respective.

După cum cunoaşteţi deja, în trecut sistemele de calcul erau exploatate printr-o interfaţă cu utilizatorul alfanumerică (*i.e.*, în mod text, nu grafic), care este foarte simplă de utilizat (în programele C aceasta se face prin intermediul funcţiilor din biblioteca standard de intrări/ieşiri `stdio.h`). La sfîrşitul anilor '80, s-a introdus un nou concept, interfaţa grafică cu utilizatorul, de către firma APPLE, idee preluată şi de MICROSOFT o dată cu lansarea sistemului de operare `Windows`, care a contribuit la larga răspîndire a utilizării calculatoarelor în aproape toate domeniile de activitate. Interfaţa grafică a fost introdusă şi în lumea sistemelor de tip `UNIX`, prin proiectul X WINDOW.

Această largă răspîndire s-a datorat faptului că interfaţa grafică este mult mai *prie-*

tenoasă pentru utilizatorul neprofesionist decît cea clasică, în mod text. Totuşi, pentru programatori, reversul medaliei este *dificultatea de programare* a aplicaţiilor ce folosesc o interfaţă grafică cu utilizatorul (*GUI=Graphical User Interface*), dificultate ce provine din faptul că trebuie învăţate şi utilizate mai multe tehnici noi:
i) sistemul de ferestre şi alte componente grafice utilizate de *GUI* se bazează pe o ierarhie de clase ce descriu aceste obiecte grafice, ierarhie care are de obicei sute de clase şi mii de metode;
ii) *programarea dirijată de evenimente* (*event-driven programming*) – o nouă tehnică de programare folosită pentru aplicaţiile *GUI*, ce constă în executarea anumitor operaţii la apariţia anumitor evenimente (ca, de exemplu, deplasarea sau *click*-ul *mouse*-ului, sau apăsarea unei taste pe tastatură), în funcţie de contextul apariţiei (*i.e.*, de obiectul grafic ce este activ şi primeşte acel eveniment);
iii) "ştiinţa" proiectării componentelor grafice ce vor alcătui *GUI*-ul unei aplicaţii, care se referă la aspectele estetice şi ergonomia de utilizare a acesteia (în anii superiori veţi avea un curs opţional intitulat "Interfaţa grafică cu utilizatorul", în care veţi studia aceste aspecte).

Deşi pe parcursul anilor următori vă veţi mai întîlni cu discipline care abordează problema programării concurente (cum ar fi, de exemplu, disciplina "Programare `Windows`", sau "Programare în limbajul Java", sau "Programare `Dot NET`"), deoarece aspectele referitoare la programarea concurentă, prezentate la aceste discipline, sunt strîns împletite cu cele legate de programarea interfeţei grafice cu utilizatorul, şi datorită dificultăţii acesteia din urmă, cadrul ales (*i.e.*, sistemul `Linux`, limbajul de programare C, şi interfaţa clasică, în mod text, cu utilizatorul) pentru predarea programării concurente consider că este cel mai adecvat pentru acest scop.

Acesta a fost de altfel un alt motiv serios pentru alegerea `Linux`-ului, ca exemplu de sistem de operare concret ce urma să fie prezentat în acest manual. (*Notă*: pentru sistemul de operare `Windows` este foarte dificil, dacă nu chiar imposibil, de realizat o tratare exclusivă a conceptelor legate de programarea concurentă, deoarece mecanismele prin care sunt acestea implementate nu pot fi disociate de cele referitoare la programarea interfeţei grafice cu utilizatorul.)

Observaţie: faptul că am ales folosirea în programe a interfeţei clasice (în mod text) cu utilizatorul, nu înseamnă că în `Linux` nu se poate face programare *GUI*, ci dimpotrivă, există medii grafice pentru `Linux` (despre care voi reveni cu amănunte în primul capitol din partea I a manualului) însoţite de medii de dezvoltare de aplicaţii grafice pentru acestea, ce folosesc conceptele referitoare la *GUI* şi la programarea dirijată de evenimente despre care am amintit mai sus, şi care implică o aceeaşi dificultate de programare în cazul `Linux`-ului ca şi în cazul `Windows`-ului.

În încheiere, aş dori să mai menţionez faptul că această disciplină, predată la Secţia la zi, cuprinde şi o parte teoretică, ce aprofundează unele concepte prezentate în cadrul disciplinei "Arhitectura calculatoarelor şi sisteme de operare" din Anul 1 Semestrul I. Am preferat să renunţ la această parte teoretică şi să păstrez partea practică, mult mai importantă, dedicată prezentării sistemului de operare `Linux` şi a programării concurente în `Linux`, pentru a nu încărca prea tare materia de învăţat ţinînd cont de caracterul secţiei dumneavoastră – învăţămînt deschis la distanţă. Totuşi, dacă doriţi să vă aprofundaţi şi cunoştinţele teoretice, puteţi consulta varianta în format electronic a prelegerilor susţinute de autorul acestui manual la cursurile de "Sisteme de operare" de la Secţia la zi, accesibilă

din pagina de *web* a acestuia (http://www.infoiasi.ro/~vidrascu); de asemenea, vă recomand şi următoarele două cărţi de specialitate pentru studiu individual: [9] şi [11].

Autorul doreşte pe această cale să adreseze cititorilor rugămintea de a-i semnala prin email, pe adresa `vidrascu@infoiasi.ro`, eventualele erori depistate în timpul parcurgerii acestui manual.

Convenţii utilizate în acest manual

Textul tipărit cu fontul `typewriter` este folosit pentru nume de comenzi, nume de variabile, constante, cuvinte-cheie, etc. Este, de asemenea, utilizat în cadrul exemplelor pentru a desemna conţinutul unui fişier de comenzi sau program C, ori rezultatul afişat de unele comenzi.
Textul `UNIX>` va fi folosit pentru a indica prompterul interpretorului de comenzi din `Linux` la care utilizatorul poate tasta comenzi (în interfaţa clasică, în mod text). Iar `comanda cu parametrii ei` va indica comanda ce trebuie tastată la prompter de către utilizator. În cazul în care, printre parametrii unei comenzi, apare şi text *italic*, el va trebui în general să fie înlocuit de către utilizator cu o valoare concretă.
Termenii şi denumirile păstrate în original (netraduse în română) se vor indica prin fontul *termen în engleză*.
Entităţile ce urmează se vor indica în maniera descrisă: `nume de fişiere`, `nume de utilizatori`, `nume de grupuri`, `nume (sau adrese IP) de calculatoare`, adrese de pagini web.
Construcţia [*text opţional*] folosită în cadrul sintaxei unei comenzi sau instrucţiuni specifică un text opţional, deci care nu este obligatoriu de folosit.
Caracterele ... semnifică o porţiune de text care a fost omisă pentru a nu îngreuna lizibilitatea sau pentru a reduce din spaţiul utilizat.
Textul *italic* va mai fi folosit uneori pentru a scoate în evidenţă un concept important despre care se discută în acel moment.

Partea I

Sistemul de operare Linux
Ghid de utilizare

Această primă parte a manualului cuprinde o prezentare generală a sistemelor de operare din familia `UNIX`, prezentare ce este valabilă în particular pentru sistemul de operare `Linux`.

Această parte se vrea a fi un *ghid de utilizare* a sistemului `Linux`, fără a avea pretenţia de a fi un ghid complet. Am preferat să insist în prezentare pe conceptele fundamentale pe care se bazează acest sistem de operare, fără să obosesc cititorul cu prea multe detalii şi numeroasele opţiuni ale unor comenzi (mai mult, am preferat să nu prezint deloc opţiunile numeroase ale meniurilor de aplicaţii grafice, din lipsă de spaţiu, şi deoarece se găsesc destule cărţi care tratează acest aspect).

Primul capitol al acestei părţi a manualului conţine o vedere de ansamblu asupra sistemelor de operare din familia `UNIX`, povesteşte istoricul evoluţiei `UNIX`-ului, descrie arhitectura (structura) unui sistem `UNIX` şi caracteristicile sale generale, modul de utilizare şi de conectare la un sistem `UNIX`, şi conţine o introducere despre `Linux` ca membru al acestei familii şi o descriere generală a procedurii de instalare a unei distribuţii de `Linux`.

Al doilea capitol se vrea a fi un ghid de utilizare, conţinînd o trecere în revistă a principalelor categorii de comenzi `UNIX`, o prezentare a sistemului de fişiere `UNIX` şi a interpretoarelor de comenzi `UNIX`, ce au rolul de a asigura interacţiunea sistemului cu utilizatorul, pentru a rula programele de aplicaţii dorite de acesta, precum şi de a-i pune la dispoziţie un puternic limbaj de *scripting* (pentru fişiere de comenzi).

Bibliografie utilă pentru studiu individual suplimentar: [3], [4], [8].

Capitolul 1

Introducere în UNIX

1.1 Prezentare de ansamblu a sistemelor de operare din familia UNIX

1. **Introducere**
2. **Scurt istoric al evoluţiei UNIX-ului**
3. **Vedere generală asupra sistemului UNIX**
4. **Structura unui sistem UNIX**
5. **Caracteristici generale ale unui sistem UNIX**
6. **UNIX şi utilizatorii**
7. **Conectarea la un sistem UNIX**

1.1.1 Introducere

UNIX este denumirea generica a unei largi familii de sisteme de operare *orientate pe comenzi*, *multi-user* si *multi-tasking*, dezvoltat pentru prima data in anii '70 la compania AT&T si Universitatea Berkeley. In timp, a devenit sistemul de operare cel mai raspindit in lume, atit in mediile de cercetare si de invatamint (universitare), cit si in mediile industriale si comerciale.

Ce inseamna sistem de operare *orientat pe comenzi*?
Sistemul poseda un interpretor de comenzi, ce are aceeasi sarcina ca si programul

`command.com` din `MS-DOS`, si anume aceea de a prelua comenzile introduse de utilizator, de a le executa si de a afisa rezultatele executiei acestora.

Ce inseamna sistem de operare *multi-user*?
Un astfel de sistem este caracterizat prin faptul ca exista conturi utilizator, ce au anumite drepturi si restrictii de acces la fisiere si la celelalte resurse ale sistemului (din acest motiv, se utilizeaza mecanisme de protectie, cum ar fi parolele pentru conturile utilizator). In plus, un astfel de sistem permite conectarea la sistem si lucrul simultan a mai multor utilizatori.

Ce inseamna sistem de operare *multi-tasking*?
Intr-un astfel de sistem se executa simultan (*i.e.*, in acelasi timp) mai multe programe. Programele aflate in executie sunt denumite *procese*). O asemenea executie simultana a mai multor programe mai este denumita si *executie concurenta*, pentru a sublinia faptul ca programele aflate in executie *concureaza* pentru utilizarea resurselor sistemului de calcul respectiv.

Observatie:
De fapt, cind `UNIX`-ul este utilizat pe calculatoare uni-procesor, in asemenea situatie executia simultana (concurenta) a proceselor nu este *true-parallelism* (*i.e.*, multi-procesare), ci se face tot secvential, prin multi-programare, si anume prin mecanismul de *interleaving* (întreţesere) cu *time-sharing*: timpul procesor este impartit in cuante de timp, si fiecare proces existent in sistem primeste periodic cite o cuanta de timp in care i se aloca procesorul si deci este executat efectiv, apoi este intrerupt si procesorul este alocat altui proces care se va executa pentru o cuanta de timp din punctul in care ramasese, apoi va fi intrerupt si un alt proces va primi controlul procesorului, ş.a.m.d.
Dupa cum am discutat la teoria sistemelor de operare din prima parte a acestui manual, cunoasteti deja faptul ca acest mecanism de stabilire a modului de *alocare a procesorului* proceselor existente in sistem, se bazeaza pe una din strategiile: *round-robin*, prioritati statice, prioritati dinamice, ş.a., uneori intilnindu-se si combinatii ale acestora. In cazul `UNIX`-ului, se utilizeaza strategia *round-robin* combinat cu prioritati dinamice.

Mai exista si alt tip de paralelism (concurenta), si anume multi-procesarea, ce este bazata pe arhitecturile multi-procesor sau cele distribuite. In asemenea arhitecturi avem *mai multe procesoare*, pe care se executa mai multe procese efectiv in paralel, si acestea pot comunica intre ele fie prin memorie comuna, fie prin canale de comunicatie.

Tabelul 1.1: Exemple de sisteme de operare.

Criteriul de clasificare		**numar users**	
		mono-user	*multi-user*
numar tasks	*mono-task*	`MS-DOS`, `CP/M`	*Nu există*!
	multi-task	`OS/2`, `Windows 3.x & 9x`	`UNIX` family

Observatie: sistemele de operare de retea (exemple: `Novell`, `Windows NT/2000/2003 Server`) pot fi privite ca sisteme multi-user, multi-tasking. Versiunile personale (*desktop*) de `Windows NT/2000/XP` sunt mono-user, deoarece la un moment dat un singur utilizator poate fi conectat la sistem.

Asa cum am mai spus, `UNIX`-ul este un sistem de operare multi-user si multi-tasking. Exista multe variante de `UNIX` (System V, BSD, XENIX, AT&T, SCO, AIX, Linux, ş.a.) deoarece multe companii si universitati si-au dezvoltat propria varianta de `UNIX`, nereusindu-se impunerea unui standard unic. Pentru aceste variante exista anumite diferente de implementare si exploatare, dar principiile utilizate sunt aceleasi. Mai mult, pentru utilizatorul obisnuit accesul si exploatarea sunt aproape similare (astfel, de exemplu, aceleasi comenzi sunt disponibile pe toate variantele de `UNIX`, dar unele comenzi pot avea unele dintre optiunile lor diferite de la varianta la varianta).

Important: din acest motiv, cele prezentate in manualul de fata (mai exact, in partea I si partea II a lui) sunt valabile pentru toate sistemele de tip `UNIX`, si in particular pentru `Linux`.

`Linux`-ul este o variantă de `UNIX` distribuibilă gratuit (*open-source*), pentru sisteme de calcul bazate pe procesoare Intel (80386, 80486, Pentium, etc.), procesoare Dec Alpha, şi, mai nou, şi pentru alte tipuri de procesoare (cum ar fi de exemplu cele pentru *embedded systems*). El a fost creat în 1991 de Linus Torvalds, fiind în prezent dezvoltat în permanenţă de o echipă formată din mii de entuziaşti `Linux` din lumea întreagă, sub îndrumarea unui colectiv condus de Linus Torvalds.

Calculatorul `UNIX` pe care veti lucra la laboratoare este serverul studentilor, fiind un IBM PC cu 2 procesoare, avind instalat ca sistem de operare `Linux`-ul. Numele acestui calculator este `fenrir`, si are adresa IP `193.231.30.197`, iar numele DNS complet, sub care este recunoscut in INTERNET, este: `fenrir.info.uaic.ro`, sau un nume echivalent, `fenrir.infoiasi.ro`. Vom reveni mai tirziu cu informatii referitoare la modul de conectare la el.

1.1.2 Scurt istoric al evoluţiei `UNIX`-ului

`UNIX`-ul este un sistem de operare relativ vechi, fiind creat la Bell Laboratories in 1969, unde a fost conceput si dezvoltat de Ken Thompson pentru uzul intern al unui colectiv de cercetatori condus de acesta.

Ei si-au dezvoltat sistemul de operare pornind de la citeva concepte de baza: sistem de fisiere, multi-user, multi-tasking, gestiunea perifericelor sa fie transparenta pentru utilizator, ş.a. Initial a fost implementat pe minicalculatoarele firmei DEC, seria PDP-7, fiind scris in limbaj de asamblare si Fortran.
Aparitia in 1972 a limbajului C, al carui autor principal este Dennis Ritchie de la firma Bell

Laboratories, a avut in timp un impact deosebit asupra muncii programatorilor, trecindu-se de la programarea in limbaj de asamblare la cea in C. Astfel in 1971 `UNIX`-ul este rescris impreuna cu Dennis Ritchie in C, devenind multi-tasking. In 1973, dupa o noua rescriere, devine portabil. Aceasta este versiunea 6, prima care iese in afara laboratorului Bell al firmei AT&T si care este distribuita gratuit universitatilor americane. In 1977 este implementat pe un calculator INTERDATA 8/32, primul diferit de un PDP.

Sistemul de operare `UNIX`, compilatorul C si in esenta toate aplicatiile sub `UNIX` sunt scrise in C intr-o proportie mare. Astfel, din cele 13000 linii sursa ale sistemului `UNIX`, numai 800 linii au fost scrise in limbaj de asamblare, restul fiind scrise in C. De asemenea, insasi compilatorul C este scris in C in proportie de 80%. In felul acesta limbajul C asigura o portabilitate buna pentru programele scrise in el. (Un program este *portabil* daca poate fi transferat usor de la un tip de calculator la altul.)

Portabilitatea mare a programelor scrise in C a condus la o raspindire destul de rapida a limbajului C si a sistemului de operare `UNIX`: se scria in asamblare doar un mic nucleu de legatura cu *hardware*-ul unui anumit tip de calculator, iar restul sistemului `UNIX` era scris in C, fiind acelasi pentru toate tipurile de calculatoare; mai era nevoie de un compilator de C pentru acel calculator si astfel se putea compila si instala `UNIX`-ul pe acel calculator (practic si pentru scrierea compilatorului se folosea aceeasi tehnica: era nevoie sa se scrie in limbaj de asamblare doar un nucleu, cu rol de *meta-compiler*, restul compilatorului fiind deja scris in C).

Prima versiune de referinta, `UNIX` versiunea 7 (1978), implementata pe un DEC PDP-11, are nucleul independent de *hardware*. Este prima versiune comercializata. In 1982 este elaborat `UNIX` System III pentru calculatoarele VAX 11/780, iar in 1983 `UNIX` System V. In 1980-1981 apar primele licente: `ULTRIX` (firma DEC), `XENIX` (Microsoft), `UTS` (Amdahl), etc.

Versiunea 7 a servit drept punct de plecare pentru toate dezvoltarile ulterioare ale sistemului. Plecind de la aceasta versiune, s-au nascut doua mari directii de dezvoltare:

1. dezvoltarile realizate la compania AT&T si Bell Laboratories au condus la versiunile succesive de System V `UNIX`;
2. munca realizata la Universitatea Berkeley s-a concretizat in versiunile succesive de BSD `UNIX` (acronimul BSD provine de la *Berkeley Software Distribution*).

Versiunile BSD au introdus noi concepte, cum ar fi: memoria virtuala (BSD 4.1), facilitati de retea (sc BSD 4.2), *fast file system*, schimb de informatii intre procese centralizat sau distribuit, etc.
Iar vesiunile System V au introdus drept concepte noi: semafoare, blocaje, cozi de mesaje, memorie virtuala, memorie pe 8 biti, etc.

Pe linga aceste variante majore, au fost dezvoltate si alte variante de `UNIX`, si anume: `XENIX` de catre firma Microsoft, `VENIX` de catre firma Venturecom, `UNIX SCO` de catre firma

SCO Corp., `AIX` de catre firma IBM, etc. Pe linga aceste variante, ce au fost dezvoltate plecind de la nucleul (*kernel*-ul) `UNIX` al firmei AT&T (ceea ce a necesitat cumpararea unei licente corespunzatoare), au fost dezvoltate si sisteme ne-AT&T, si anume: `MINIX` de catre Andrew Tanenbaum, `Linux` de catre Linus Torvald, `XINU` de catre Douglas Comer, `GNU` de catre FSF (acronimul FSF inseamna *Free Software Fundation*). Obiectivul fundatiei FSF este dezvoltarea unui sistem in intregime compatibil (cu cel de la AT&T) si care sa nu necesite nici o licenta de utilizare (si deci sa fie gratuit).

Aceasta multiplicare a versiunilor de `UNIX`, devenite incompatibile si facind dificila portarea aplicatiilor, a determinat utilizatorii de `UNIX` sa se regrupeze si sa propuna definirea de interfete standard: X/OPEN si POSIX (= *Standard IEEE 1003.1-1988-Portable Operating System Interface for Computer Environments*).
Aceste interfete au preluat in mare parte propunerile facute in definitia de interfata SVID (= *System V Interface Definition*) propusa de AT&T, dar influentele din celelalte variante nu sunt neglijabile.

Aceasta normalizare (standardizare) a sistemului este doar la nivel utilizator, nefiind vorba de o unicitate a nucleului: cele doua blocuri formate, Unix International si OSF (OSF = *Open Software Foundation*), continua sa-si dezvolte separat propriul nucleu, dar totusi diferentele de implementare sunt transparente pentru utilizatori.

O alta frina pentru raspindirea sistemului `UNIX`, pe linga aceasta lipsa de normalizare, a constituit-o aspectul neprietenos al interfetei utilizator, care a ramas pentru multa vreme orientata spre utilizarea de terminale alfanumerice (adica in mod text, nu grafic). Dar si pe acest plan situatia s-a imbunatatit considerabil, prin adoptarea sistemului de ferestre grafice X Window si dezvoltarea de medii grafice bazate pe acest sistem.
Sistemul X Window a fost dezvoltat in cadrul proiectului Athena de la MIT (*Massachusetts Institute of Technology*) din anii '80. Majoritatea statiilor de lucru aflate actual pe piata poseda acest sistem. Protocolul X, folosit de acest sistem, a fost conceput pe ideea distribuirii calculelor intre diferitele unitati centrale, statii de lucru ale utilizatorilor, si celelalte masini din retea pe care se executa procesele utilizatorilor. Astfel, sistemul X Window are o arhitectură client-server, ce utilizează protocolul X pentru comunicaţia prin reţea între diferitele unitati centrale şi statii de lucru. Protocolul X a fost adoptat ca standard si s-au dezvoltat o serie de biblioteci grafice, toate avind ca substrat sistemul X Window, precum ar fi: MOTIF, OPEN LOOK, etc.
În prezent, dezvoltarea sistemului X Window este administrată de organizaţia *Consorţiul X* (http://www.X.org), ce oferă şi o implementare de referinţă a sistemului X Window pe *site*-ul organizaţiei. Astfel, ultima versiune lansată este X11R6.7, la data scrierii acestor rînduri (vara anului 2004). *Notă*: X11 este numele generic al unei variante majore a protocolului, ce a fost standardizată, şi din acest motiv de multe ori apare referirea X11 în denumirea tehnologiilor ce folosesc sistemul X Window în această versiune standardizată, iar R6.7 este numărul ultimului *release*).
De asemenea, există şi o implementare *open-source* a sistemului X Window, numită XFree86 (http://www.xfree86.org). Ea furnizează o interfaţă client-server între *hardware*-ul de I/O (tastatură, *mouse*, placa video/monitor) şi mediul *desktop*, precum şi infrastructura de ferestre şi un API (*Application Programming Interface*) standardizat

pentru dezvoltarea de aplicaţii grafice X11. Pe scurt, XFree86 este o infrastructură *desktop* bazată pe X11, disponibilă gratuit (*open-source*), ultima versiune lansată fiind versiunea 4.4.0, la data scrierii acestor rînduri.

Pe parcursul anilor, punctul de vedere al cercetatorilor si dezvoltatorilor din domeniul evolutiei sistemelor, si a `UNIX`-ului in particular, referitor la dezvoltarea unui sistem distribuit fizic pe mai multe calculatoare, a evoluat de la imaginea unui sistem format din unitati separate si independente, avind fiecare propriul său sistem de exploatare si care pot comunica cu sistemele de exploatare de pe celelalte masini din sistem (acesta este cazul actual al statiilor `UNIX` dintr-o retea), la imaginea unui ansamblu de resurse a caror localizare devine transparenta pentru utilizator.
In acest sens, protocolul NFS (NFS = *Network File System*), propus de firma SUN Microsystems, a fost, cu toate incovenientele si imperfectiunile sale, prima incercare de realizare a unui astfel de sistem care a fost integrata in sistemele `UNIX` comercializate.

In ultimii ani din deceniul trecut, cercetarea s-a focalizat pe tehnologia *micro-nucleu*. *Nota*: in revista PC Report nr. 60 (din sept. 1997) puteti citi un articol ce face o comparatie intre tehnologia traditionala (*i.e.*, *nucleu monolitic*) si cea *micro-nucleu*.

1.1.3 Vedere generală asupra sistemului `UNIX`

`UNIX`-ul este un sistem de operare multi-user si multi-tasking ce ofera utilizatorilor numeroase utilitare interactive. Pe linga rolul de sistem de exploatare, scopul lui principal este de a asigura diferitelor *task*-uri (procese) si diferitilor utilizatori o repartizare echitabila a resurselor calculatorului (memorie, procesor/procesoare, spatiu disc, imprimanta, programe utilitare, accesul la retea, etc.) si aceasta fara a necesita interventia utilizatorilor.

`UNIX`-ul este inainte de toate un mediu de dezvoltare si utilizatorii au la dispozitie un numar foarte mare de utilitare pentru munca lor: editoare de text, limbaje de comanda (*shell*-uri), compilatoare, depanatoare (*debugger*-e), sisteme de prelucrare a textelor, programe pentru posta electronica si alte protocoale de acces INTERNET, si multe alte tipuri de utilitare.

Pe scurt, un sistem `UNIX` este compus din:

1. un **nucleu** (*kernel*), ce are rolul de a gestiona memoria si operatiile I/O de nivel scazut, precum si planificarea si controlul executiei diferitelor *task*-uri (procese). Este intrucitva similar BIOS-ului din `MS-DOS`.
2. un ansamblu de **utilitare de baza**, cum ar fi:
   - diferite *shell*-uri (= interpretoare de limbaje de comanda);

- comenzi de manipulare a fisierelor;
- comenzi de gestiune a activitatii sistemului (a proceselor);
- comenzi de comunicatie intre utilizatori sau intre sisteme diferite;
- editoare de text;
- compilatoare de limbaje (C, C++, Fortran, ş.a.) si un *link-editor*;
- utilitare generale de dezvoltare de programe: *debugger*-e, arhivatoare, gestionare de surse, generatoare de analizoare lexicale si sintactice, etc.
- diferite utilitare *filtru* (= programe ce primesc un fisier la intrare, opereaza o anumita transformare asupra lui si scriu rezultatul ei intr-un fisier de iesire), spre exemplu: filtru sursa Pascal→sursa C, filtru fisier text `DOS`→fisier text `UNIX` si invers, etc.
  Nota: fisierele text sub `MS-DOS` se deosebesc de fisierele text sub `UNIX` prin faptul ca sfirsitul de linie se reprezinta sub `MS-DOS` prin 2 caractere CR+LF (cu codul ASCII: 13+10), pe cind sub `UNIX` se reprezinta doar prin caracterul LF.

1.1.4 Structura unui sistem `UNIX`

Un sistem `UNIX` are o structura ierarhizata pe mai multe nivele. Mai precis exista 3 nivele, ilustrate in figura 1.1, ce urmeaza mai jos.

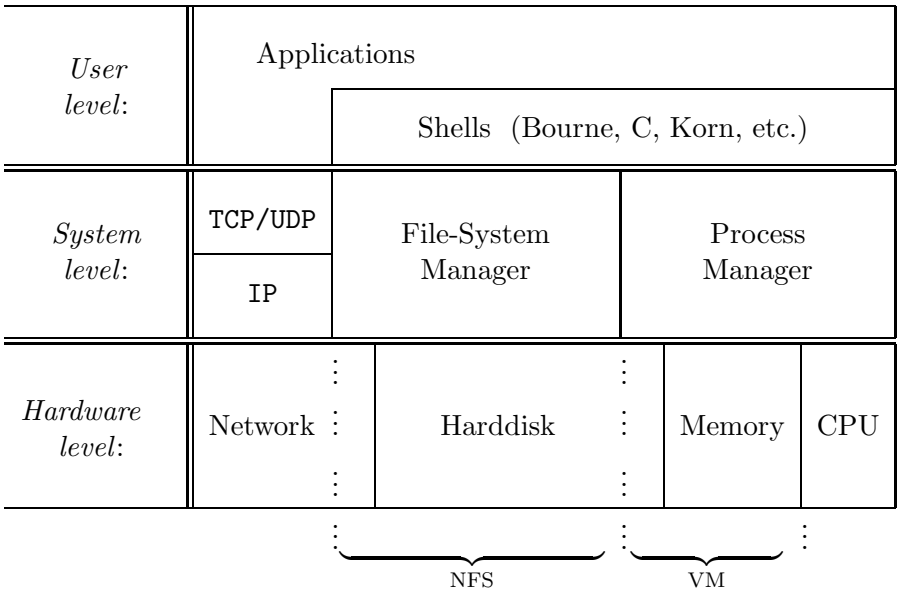

Figura 1.1: Structura unui sistem `UNIX`.

Explicatii suplimentare la figura 1.1:

1. **Nivelul hardware :** este nivelul format din componentele *hardware* ale sistemului de calcul: CPU + Memory + Harddisk + Network

2. **Nivelul system :** este reprezentat de nucleul (*kernel*-ul) sistemului `UNIX`, si are rolul de a oferi o interfata intre aplicatii (nivelul 3) si partea de *hardware* (nivelul 1), astfel încît aplicatiile sa fie portabile (independente de masina *hardware* pe care sunt rulate).
   Nucleul `UNIX` contine trei componente principale: sistemul de gestionare a fisierelor (*File-System Manager*), sistemul de gestionare a proceselor (*Process Manager*), si componenta de comunicatie in retea (comunicatia se realizeaza pe baza protocoalelor de comunicatie IP si TCP/UDP).

3. **Nivelul user :** contine limbajele de comanda (*shell*-urile) si diversele programe utilitare si aplicatii utilizator.

Nucleul (*kernel*-ul) este scris in cea mai mare parte (cca. 90%) in limbajul C. Ca urmare, functiile sistem respecta conventiile din limbajul C. Ele pot fi apelate din programele utilizator, fie direct din limbaj de asamblare, fie prin intermediul bibliotecilor din limbajul C. Aceste functii sistem, oferite de *kernel*, sunt numite in termeni `UNIX` **apeluri sistem** (*system calls*). Prin ele, functiile *kernel*-ului sunt puse la dispozitia utilizatorilor, la fel cum se face in sistemul de operare `RSX-11M` prin *directive sistem*, in sistemul `MS-DOS` prin *intreruperile software*, etc.

Fiecare nivel se bazeaza pe serviciile/resursele oferite de nivelul imediat inferior. Pe figura de mai sus, serviciile/resursele folosite de fiecare componenta sunt cele oferite de componenta sau componentele aflate imediat sub aceasta. Astfel,

- componenta de gestiune a proceselor utilizeaza, ca resurse oferite de nivelul hardware, CPU si memoria interna, plus o parte din hard-disc, sub forma discului de *swap*, pentru mecanismul de *memorie virtuala* (VM = *virtual memory*), mecanism ce utilizeaza memoria interna fizica si discul de *swap* pentru a crea o memorie interna virtuala;

- sistemul de fisiere utilizeaza restul hard-discului, plus o parte din componenta de retea, prin intermediul NFS-ului (NFS = *Network File System*);

- IP si TCP/UDP sunt protocoalele de baza pentru realizarea comunicatiei in retea, pe baza carora sunt construite toate celelalte protocoale: posta electronica, transfer de fisiere (FTP), World Wide Web (HTTP), etc.;

- interpretoarele de comenzi (*shell*-urile) utilizeaza serviciile puse la dispozitie de *Process Manager* si *File-System Manager*, iar restul aplicatiilor utilizeaza serviciile oferite de intreg nivelul system.

1.1.5 Caracteristici generale ale unui sistem UNIX

- Principalele concepte pe care se sprijina UNIX-ul sunt conceptul de **fisier** si cel de **proces**.
  Prin *fisier* se intelege o colectie de date, fara o interpretare anumita, adica o simpla secventa de octeti (modul de interpretare al lor cade in sarcina aplicatiilor care le folosesc).
  Prin *proces*, sau *task*, se intelege un program (*i.e.*, un fisier executabil) incarcat in memorie si aflat in curs de executie.

- Un **sistem de fisiere** *ierarhizat* (*i.e.*, arborescent), *i.e.* este ca un arbore (la fel ca in MS-DOS: directoare ce contin subdirectoare si fisiere propriu-zise), dar un arbore ce are o singura radacina, referita prin "/" (nu avem, ca in MS-DOS, mai multe unitati de disc logice C:, D:, etc.), iar ca separator pentru caile de subdirectoare se utilizeaza caracterul '/', in locul caracterului '\' folosit in MS-DOS.
  Numele fisierelor pot avea pina la 255 de caractere si pot contine oricite caractere '.' (nu sunt impartite sub forma 8.3, *nume.extensie*, ca in MS-DOS). Mai mult, numele fisierelor sunt *case-sensitive*, adica se face distinctie intre literele majuscule si cele minuscule.
  (Vom vedea mai multe amanunte cind vom discuta despre sistemul de fisiere in sectiunea 2.2 din capitolul 2).

- Un **sistem de procese** *ierarhizat* (*i.e.*, arborescent) si un mecanism de "*mostenire genetica*":
  Fiecare proces din sistem are un proces care l-a creat, numit proces *parinte*, (sau *tata*) si de la care "mosteneste" un anumit ansamblu de caracteristici (cum ar fi proprietarul, drepturile de acces, ş.a.), si poate crea, la rindul lui, unul sau mai multe procese *fii*.
  Fiecare proces are asignat un PID (denumire ce provine de la *Process IDentification*), ce este un numar intreg pozitiv si care este unic pe durata vietii acelui proces (in orice moment, nu exista in sistem doua procese cu acelasi PID.
  Exista un proces special, cel cu PID = 0, care este creat atunci cind este initializat (*boot*-at) sistemul UNIX pe calculatorul respectiv. Acesta nu are proces parinte, fiind radacina arborelui de procese ce se vor crea pe parcursul timpului (pina la oprirea calculatorului).
  (Vom vedea mai multe amanunte despre sistemul de procese cind vom discuta despre gestiunea proceselor in capitolul 4 din partea II).

- Un ansamblu de *puncte de acces*, din aplicatiile scrise in limbaje de nivel inalt (precum C-ul), la serviciile oferite de *kernel*, puncte de acces ce se numesc **apeluri sistem** (*system calls*).
  De exemplu, prin apelul dintr-un program C al functiei fork() putem crea noi procese.

- Este un sistem de operare **multi-user**: mai multi utilizatori pot avea acces simultan la sistem in orice moment, de la diferite terminale conectate la sistemul respectiv, terminale plasate local sau la distanta.

- Este un sistem de operare **multi-tasking**: se pot executa simultan mai multe programe, si aceasta chiar si relativ la un utilizator:
  Fiecare utilizator, in cadrul unei *sesiuni de lucru*, poate lansa in executie paralela mai multe procese; dintre acestea, numai un proces se va executa in *foreground* (planul din fata, ce are controlul asupra tastaturii si ecranului), iar restul proceselor sunt executate in *background* (planul din spate, ce nu are control asupra tastaturii si ecranului).
  In plus, fiecare utilizator poate deschide mai multe sesiuni de lucru.
  Observatie: pe calculatorul `fenrir` numarul maxim de sesiuni ce pot fi simultan deschise de un utilizator s-ar putea sa fie limitat, din considerente de supraincarcare a sistemului.

- Este un sistem de operare **orientat pe comenzi**: exista un *interpretor de comenzi* (numit uneori si *shell*) ce are aceeasi destinatie ca si in `MS-DOS`, si anume aceea de a prelua comenzile introduse de utilizator, de a le executa si de a afisa rezultatele executiei acestora.
  Daca in `MS-DOS` este utilizat practic un singur interpretor de comenzi, si anume programul `command.com` (desi teoretic acesta poate fi inlocuit de alte programe similare, cum ar fi `ndos.com`-ul), in `UNIX` exista in mod traditional mai multe interpretoare de comenzi: `sh` (*Bourne SHell*), `bash` (*Bourne Again SHell*), `csh` (*C SHell*), `ksh` (*Korn SHell*), `ash`, `zsh`, ş.a., utilizatorul avind posibilitatea sa aleaga pe oricare dintre acestea.

  Shell-urile din `UNIX` sunt mai puternice decit analogul (`command.com`) lor din `MS-DOS`, fiind asemanatoare cu limbajele de programare de nivel inalt: au structuri de control alternative si repetitive de genul `if`, `case`, `for`, `while`, etc., ceea ce permite scrierea de programe complexe ca simple *script*-uri. Un *script* este un fisier de comenzi `UNIX` (analogul fisierelor *batch* `*.bat` din `MS-DOS`).

  La fel ca in `MS-DOS`, fiecare user isi poate scrie un *script* care sa fie executat la fiecare inceput de sesiune de lucru (analogul fisierului `autoexec.bat` din `MS-DOS`), *script* numit `.profile` sau `.bash_profile` in cazul in care se utilizeaza `bash`-ul ca *shell* implicit (pentru alte *shell*-uri este denumit altfel).
  In plus, fiecare user poate avea un *script* care sa fie rulat atunci cind se deconecteaza de la sistem (adica la *logout*); acest script se numeste `.bash_logout` in cazul *shell*-ului `bash`.

  La fel ca in `MS-DOS`, exista doua categorii de comenzi: *comenzi interne* (care se gasesc in fisierul executabil al *shell*-ului respectiv) si *comenzi externe* (care se gasesc separat, fiecare intr-un fisier executabil, avind acelasi nume cu comanda respectiva).
  Forma generala de lansare a unei comenzi `UNIX` este urmatoarea:

  `UNIX>` *nume_comanda optiuni argumente* ,

  unde optiunile si argumentele pot lipsi, dupa caz. Prin conventie, optiunile sunt precedate de caracterul '-' (in `MS-DOS` este folosit caracterul '/'). Argumentele sunt cel mai adesea nume de fisiere.
  (Vom vedea mai multe amanunte cind vom discuta despre *shell*-uri in sectiunea 2.3 din capitolul 2).

- Alta caracteristica: o viziune unitara (prin intermediul unei aceleasi interfete) asupra diferitelor tipuri de operatii de intrare/iesire.
  Astfel, de exemplu, terminalul (tastatura + ecranul) are asociat un fisier special prin intermediul caruia operatiile de intrare (citirea de la tastatura) si de iesire (scrierea pe ecran) se fac similar ca pentru orice fisier obisnuit.

- Alta caracteristica: **redirectarea operatiilor I/O ale proceselor**, ce se bazeaza pe caracteristica anterioara, si a carei principala utilizare este unul dintre conceptele fundamentale ale `UNIX`-ului, si anume acela de **filtrare**.
  Ideea de baza consta in a asocia fiecarui proces din sistem un anumit numar de *fisiere logice* predefinite, numite *intrari-iesiri standard* ale procesului. Mai exact, este vorba despre `stdin` (intrarea standard), `stdout` (iesirea standard), si `stderr` (iesirea de eroare standard).
  Sistemul furnizeaza un mecanism de *redirectare* (realizat intern prin apeluri sistem specifice), care permite ca unui fisier logic a unui proces sa i se asocieze un fisier fizic particular. Astfel, `stdin` are asociata implicit tastatura, iar `stdout` si `stderr` au asociat implicit ecranul, dar li se pot asocia si alte fisiere fizice particulare de pe disc.
  Acest mecanism este repercutat la nivel extern in diversele limbaje de comanda (*shell*-uri) prin posibilitatea de a cere, la executia unei comenzi, asocierea I/O standard a procesului ce executa comanda la anumite fisiere fizice de pe disc.
  Dintre toate comenzile `UNIX`, acelea ce au proprietatea de a face o anumita prelucrare asupra datelor citite pe intrarea standard (fara a modifica fisierul fizic asociat ei) si care scriu rezultatele prelucrarii pe iesirea standard, sunt denumite traditional *filtre*.

1.1.6 `UNIX` şi utilizatorii

- Fiecare **utilizator**, pentru a putea lucra, trebuie sa posede un cont pe sistemul `UNIX` respectiv, cont caracterizat printr-un nume (*username*) si o parola (*password*), ce trebuie furnizate in momentul conectarii la sistem (operatie denumita *login*).
  Fiecare utilizator are asignat un `UID` (denumire ce provine de la *User IDentification*), ce este un numar intreg pozitiv si este unic (nu exista doi utilizatori cu acelasi `UID`).
  Exista un utilizator special, numit *root* (sau *superuser*), cu `UID` = `0`, care se ocupa cu administrarea sistemului si are drepturi depline asupra intregului sistem.

- Exista **grupuri de utilizatori**, cu ajutorul carora se gestioneaza mai usor drepturile si restrictiile de acces a utilizatorilor la resursele sistemului. Fiecare utilizator face parte dintr-un grup (si poate fi optional afiliat la alte grupuri suplimentare).
  Fiecare grup are asignat un `GID` (*Group IDentification*), ce este un numar intreg pozitiv si este unic (nu exista doua grupuri cu acelasi `GID`).

- Pentru a avea acces la sistemul `UNIX`, un nou utilizator va trebui sa obtina un **cont** nou (*i.e.*, *username* + *password*) de la administratorul sistemului. La crearea contului, acesta ii va asigna anumite drepturi si restrictii de acces la fisiere si la celelalte

resurse ale sistemului, un grup de utilizatori la care este afiliat, un director de lucru (numit *home directory*), un *shell* implicit, ş.a.
Directorul *home* este directorul curent in care este plasat utilizatorul cind se conecteaza la sistem pentru a lucra, si este, de asemenea, directorul in care isi va pastra propriile fisiere si subdirectoare.
Shell-ul implicit este interpretorul de comenzi lansat automat atunci cind utilizatorul se conecteaza la sistem.
Informatiile despre fiecare cont (*username*-ul, `UID`-ul, parola criptografiata, `GID`-ul grupului din care face parte, directorul *home*, *shell*-ul implicit, si alte informatii) sunt pastrate in fisierul de sistem `/etc/passwd`.
Un alt fisier de sistem este `/etc/group`, in care se pastreaza informatii despre grupurile de utilizatori.
Observatie: in versiunile mai noi, din motive de securitate, parolele criptografiate au fost inlaturate din fisierul `/etc/passwd`, fiind pastrate in fisierul de sistem `/etc/shaddow`, care este accesibil numai *root*-ului.

- *Atenţie*: fiecare cont trebuie utilizat doar de proprietarul lui, acesta fiind obligat sa nu-si divulge parola altor persoane, si nici sa nu dea acces prin contul lui altor persoane. Aceasta din motive de *securitate a sistemului*: se pot depista incercarile de "spargere" a sistemului si in acest caz va fi tras la raspundere proprietarul contului din care s-a facut "spargerea", indiferent daca acesta este vinovatul real sau altul s-a folosit de contul lui, cu sau fara stirea proprietarului! Modul acesta de utilizare a resurselor de calcul este stipulat si prin regulamentul facultatii/universitatii.

- Parola poate fi schimbata direct de utilizator cu ajutorul comenzii `passwd`. Din motivele expuse mai sus, va trebui sa va alegeti parole cit mai greu de "ghicit": sa fie cuvinte de minimum 7-8 litere, sa nu reprezinte numele/prenumele dumneavoastra, data nasterii, numarul de telefon sau alte date personale usor de aflat, sau combinatii ale acestora, nici alte cuvinte ce sunt oarecum simple, ca de exemplu palindroamele (= cuvinte "in oglinda", cum ar fi: ab121ba), ş.a.
  Parolele sunt criptate cu un program de criptare intr-un singur sens (nu exista metode de decriptare efective, adica in timp/spatiu rezonabile). Totusi exista programe care incearca "ghicirea" parolei prin generarea combinatiilor de litere cu probabilitate mare de a fi folosite, pe baza unui dictionar (initial existau doar pentru limba engleza, dar acum exista si pentru alte limbi). Din acest motiv programul de criptare (comanda `passwd`) nu va accepta cuvinte ce sunt usor de "ghicit" in sensul de mai sus, dar bineinteles ca nu poate sa-si dea seama daca parola tastata reprezinta vreo data personala a dumneavoastra, deci din punctul de vedere al datelor personale trebuie sa aveti singuri grija sa furnizati o parola cit mai "sigura".
  Atenţie: daca va uitati parola, nu mai puteti intra in contul dumneavoastra, si nici administratorul nu va poate afla parola, dar in schimb v-o poate sterge si astfel veti putea sa va puneti o noua parola.
  Pe anumite sisteme, utilizatorul este obligat sa-si schimbe parola periodic din motive de securitate. Astfel, pentru calculatorul `fenrir`, termenul de schimbare a parolei este setat la 2 luni, dupa care vi se blocheaza contul in caz ca nu ati schimbat-o (doar administratorul vi-l poate debloca in aceasta situatie).

1.1.7 Conectarea la un sistem UNIX

a) Sesiunea de lucru

Conectarea la sistem se realizeaza fie direct (de la consola sistemului sau alte terminale legate direct la sistem), fie de la distanta. In primul caz conectarea se face cu comanda `login`, iar pentru legarea de la distanta se utilizeaza comanda `telnet`.

Explicatie: `telnet`-ul este o aplicatie care transforma calculatorul PC pe care lucrati (sub sistemul `MS-Windows`, de obicei), in **terminal conectat, prin retea, la calculatorul** `UNIX` pe care doriti sa lucrati – in cazul de fata, calculatorul `fenrir` (*i.e.*, serverul studentilor). Comunicatia prin retea intre cele doua calculatoare se desfasoara prin protocolul TELNET. Mai nou, se foloseste si protocolul SSH, care este un TELNET criptografiat (mai exact, informatiile circula criptografiate prin retea intre cele doua calculatoare).

Sub sistemul `MS-DOS`, comanda de conectare era:

`DOS>` `telnet` *nume-calculator*

Exemple:
- `DOS>` `telnet fenrir.info.uaic.ro`
- `DOS>` `telnet 193.231.30.197`
- `DOS>` `telnet dragon.uaic.ro`

Sub sistemul `MS-Windows`, exista mai multe aplicatii client de TELNET/SSH, unele comerciale si altele *freeware*, inclusiv comanda `telnet` implicita a `Windows`-ului, care se executa intr-o fereastra `MS-DOS prompt`. Va recomand sa utilizati clientul `putty`, care este o aplicatie *open-source*, disponibila gratuit pe INTERNET, inclusiv cu codul sursa.

Atenţie: aplicatia `putty` permite conectarea folosind ambele protocoale pentru sesiuni de lucru, si cel necriptat (TELNET), si cel criptat (SSH), dar, din motive de siguranta, va sfatuiesc sa va conectati intotdeauna la serverul `fenrir` folosind protocolul SSH, indiferent de unde va conectati dumneavoastra, fie de pe un calculator dintr-un laborator al facultatii, fie de pe calculatorul personal de acasa.
Aceste motive de siguranta se refera la faptul ca informatiile de autentificare (*username*-ul si parola) circula prin retea necriptate (*i.e.*, ca si text clar) in cazul folosirii protocolului TELNET, putind fi aflate astfel de persoane rau-intentionate (cu ajutorul unor programe care “asculta” traficul prin retea, numite *sniffer*-e).

Comanda `telnet` poate fi folosita si de pe un calculator `UNIX` pentru a te conecta la un alt calculator `UNIX`. Daca cele doua calculatoare sunt de acelasi tip de sistem `UNIX`, atunci se poate folosi si comanda `rlogin` in loc de `telnet`.

In toate situatiile, indiferent de clientul de TELNET sau SSH folosit, conectarea la calculatorul `UNIX` incepe cu faza de *login* (*i.e.*, autentificarea în sistem): utilizatorului i se

cere sa furnizeze un *nume de cont* (*usename*-ul de care am vorbit mai inainte) si o *parola*. Conectarea reuseste doar daca numele si parola introduse sunt corecte (adica daca exista intr-adevar pe acel sistem un utilizator cu numele si parola specificate), si in aceasta situatie se incepe apoi o *sesiune de lucru*, adica se lanseaza interpretorul de comenzi implicit pentru acel utilizator: se afiseaza un prompter si se asteapta introducerea de comenzi de catre utilizator.
Prompterul afisat este in mod obisnuit caracterul '$' pentru utilizatorii obisnuiti, respectiv '#' pentru utilizatorul *root*, dar poate fi schimbat dupa dorinta cu comanda `prompt`, sau cu ajutorul fisierului de initializare a sesiunii de lucru (fisierul `.profile` sau `.bash_profile` in cazul *shell*-ului `bash`).
La sfirsitul sesiunii de lucru, deconectarea de la sistem se face cu comanda `logout` (se poate si cu comanda `exit`, dar numai in anumite circumstante).

b) Transferul de fişiere

In afara comenzii `telnet` care permite conectarea la un sistem `UNIX` pentru deschiderea unei sesiuni de lucru, o alta comanda utila este comanda `ftp`, care permite conectarea la un alt sistem pentru a transfera fisiere intre calculatorul pe care este executata comanda (numit *calculatorul local*) si sistemul la care se face conectarea (numit *calculatorul de la distanta*). Protocolul utilizat de aceasta comanda este protocolul FTP (abrevierea provine de la *File Transfer Protocol*).

Sub sistemul `MS-DOS`, comanda de conectare pentru transfer de fisiere era:

`DOS>` `ftp` *nume-calculator*

Exemple:
- `DOS> ftp fenrir.info.uaic.ro`
- `DOS> ftp 193.231.30.197`

Sub sistemul `MS-Windows`, exista numeroase aplicatii client de FTP, unele comerciale si altele *freeware*, inclusiv comanda `ftp` implicita a `Windows`-ului, care se executa intr-o fereastra `MS-DOS prompt`. Spre exemplu, managerul de fisiere *Windows Commander* are implementat si un client de FTP, operatie ce este disponibila din meniurile acestei aplicatii.

Comanda `ftp` incepe si ea cu o faza de *login* similara cu cea de la comanda `telnet`, dupa care urmeaza sesiunea propriu-zisa de transfer de fisiere, in care se afiseaza un prompter si se asteapta introducerea de comenzi de catre utilizator, comenzi ce sunt de tipul urmator:

1. `FTP> ls` *director*

   afişează conţinutul directorului specificat de pe calculatorul de la distanta;

2. `FTP> lls` *director*

   afişează conţinutul directorului specificat de pe calculatorul local;

3. `FTP> cd` *director*

   schimbă directorul curent în cel specificat pe calculatorul de la distanta;

4. `FTP> lcd` *director*

   schimbă directorul curent în cel specificat pe calculatorul local;

5. `FTP> get` *fisier*

   transferă fişierul specificat de pe calculatorul de la distanta pe cel local;

6. `FTP> put` *fisier*

   transferă fişierul specificat de pe calculatorul local pe cel de la distanta.

Atît programul client (*i.e.*, comanda `ftp`), cît şi programul server de FTP (*i.e.*, programul de pe calculatorul de la distanta care va raspunde la cererea de conectare adresata de client) îşi păstrează cîte un director curent de lucru propriu pe calculatorul respectiv, în raport cu care se vor considera numele de fişiere sau directoare specificate prin cale relativă în comenzile enumerate de mai sus. Operaţiile *locale* `lls` şi `lcd` se vor executa direct de către client, fără ajutorul serverului, în schimb pentru toate celelalte patru operaţii clientul va schimba informatii cu serverul pentru a putea realiza operaţia respectivă.

Observaţie: evident, sintaxa acestor comenzi difera de la un client la altul (spre exemplu, in *Windows Commander* operatiile sunt disponibile prin interfata grafica, utilizind direct *mouse*-ul), dar toate aplicatiile de acest tip ofera operatiile amintite mai sus, si altele suplimentare.

Pe linga protocolul FTP, care este necriptat, mai exista si un alt protocol, criptat, ce permite transferul de fisiere, protocol numit SCP (abrevierea provine de la *Secure Copy Protocol*), si care este practic un FTP realizat printr-un "tunel" SSH.

Atenţie: din aceleasi motive de siguranta ca la sesiunile de lucru TELNET/SSH, va sfatuiesc sa va conectati intotdeauna la serverul `fenrir` pentru transfer de fisiere folosind protocolul criptat SCP, indiferent de unde va conectati dumneavoastra, fie de pe un calculator dintr-un laborator al facultatii, fie de pe calculatorul personal de acasa.
In acest sens, va recomand sa utilizati clientul `WinSCP`, care este o aplicatie *open-source*, disponibila gratuit pe INTERNET, inclusiv cu codul sursa, pentru `Windows`, cu o interfata grafica asemanatoare celei din *Windows Commander*.

1.2 Distribuţii de `Linux`. Instalare

1. **Introducere**

1.2.1 Introducere

După cum am specificat deja la începutul acestui capitol, `Linux`-ul este o variantă de `UNIX` distribuibilă gratuit (*open-source*), pentru sisteme de calcul bazate pe procesoare Intel, procesoare Dec Alpha, şi, mai nou, şi pentru alte tipuri de procesoare (cum ar fi de exemplu cele pentru *embedded systems*). El a fost creat în 1991 de Linus Torvalds, fiind în prezent dezvoltat în permanenţă de o echipă formată din mii de entuziaşti `Linux` din lumea întreagă, sub îndrumarea unui colectiv condus de Linus Torvalds.
Mai precis, această echipă mondială se ocupă cu dezvoltarea *nucleului* sistemului de operare, care în prezent se află la versiunea 2.6.x (x-ul este numărul *minor* de versiune, incrementat la lansarea fiecărei noi versiuni a *kernel*-ului de `Linux`, iar 2.6 este numărul *major* de versiune, ce indică o familie generică de versiuni, ce se deosebeşte de precedenta prin caracteristici şi funcţionalităţi importante, introduse în nucleu o dată cu trecerea la o nouă familie de versiuni).

Ca orice sistem din familia `UNIX`-ului, şi `Linux`-ul este compus, pe lîngă *nucleul* sistemului de operare, dintr-o colecţie de utilitare de bază şi programe de aplicaţii, cum ar fi, de exemplu, diverse *shell*-uri, editoare de texte, compilatoare şi medii de dezvoltare de aplicaţii în diverse limbaje de programare, diverse utilitare filtru, programe de poştă electronică, ş.a. (le-am enumerat deja în subsecţiunea 1.1.3 de mai sus). Majoritatea sunt programe *open-source*, dar există şi aplicaţii comerciale pentru `Linux`.

Observaţie: După cum am amintit deja la începutul acestui capitol, fundaţia FSF (*Free Software Fundation*) îşi propusese să dezvolte o versiune de `UNIX` care să fie în întregime compatibilă cu varianta de `UNIX` de la AT&T, versiune numită `GNU` (acronim *recursiv* ce înseamnă *GNU's Not Unix*), şi care trebuia să fie *free-software*, deci să nu necesite nici o licenţă de utilizare (şi prin urmare să fie gratuită). Înainte de anii '90, fundaţia reuşise să realizeze deja medii de dezvoltare de aplicaţii (*i.e.*, compilator de C şi C++, depanator, link-editor, ş.a.) şi utilitarele de bază, dar îi lipsea tocmai nucleul sistemului de operare (şi nici pînă în prezent situaţia nu s-a schimbat, datorită motivului expus în continuare). Astfel, cînd în 1991 Linus Torvalds scria primele versiuni ale unui nou nucleu de tip `UNIX`, pe care l-a numit `Linux`, a luat decizia de a combina nucleul său cu mediile de dezvoltare de aplicaţii şi utilitarele de bază din familia `GNU` dezvoltate de către FSF, şi cu sistemul grafic X Window dezvoltat la MIT, pentru a forma un sistem de operare complet. Se năştea astfel un nou sistem de operare, numit `Linux`, primul sistem de operare care era disponibil în mod gratuit. De fapt, iniţial se numea `GNU/Linux`, dar s-a înrădăcinat folosirea numelui mai scurt `Linux`.

Datorită faptului că atît nucleul, cît şi uneltele `GNU` erau disponibile gratuit, diverse companii şi organizaţii, ba chiar şi unii indivizi pasionaţi de fenomenul *open-source* si `Linux`, au început să lanseze diverse variante de `Linux`, care difereau printre ele prin versiunea

nucleului ce o includeau şi prin programele (cu propriile lor versiuni) ce alcătuiau *colecţia de utilitare de bază şi programe de aplicaţii* inclusă în respectiva variantă de `Linux`. Astfel, toate includeau compilatorul `GNU C/C++` pentru limbajele C şi C++, într-o anumită versiune a acestuia. În plus, erau însoţite şi de un program de instalare a sistemului, care şi acesta diferea de la o variantă la alta de `Linux`. Diferenţele dintre aceste programe de instalare sunt mai pregnante în ceea ce priveşte modul de gestiune al *pachetelor* şi de selecţie al lor în vederea instalării, precum şi al *script*-urilor folosite pentru configurarea sistemului. În terminologia `UNIX`, prin *pachet* se înţelege un grup de programe, uneori dependente unele de altele, ce realizează o anumită sarcină (sau mai multe sarcini înrudite), împreună cu fişierele de iniţializare şi configurare aferente acestor programe.

Aceste variante de `Linux` au fost denumite *distribuţii de* `Linux`. Iniţial au fost cîteva distribuţii, ele fiind cele mai răspîndite şi în ziua de azi, cum ar fi, spre exemplu, distribuţia *Red Hat* (http://www.redhat.com), distribuţia *Slackware* (http://www.slackware.org), distribuţia *Mandrake* (http://www.mandrake.com), distribuţia *SuSE* (http://www.suse.de), distribuţia *Debian* (http://www.debian.org), ş.a. În prezent există peste o sută de distribuţii de `Linux`, adaptate pentru diverse arhitecturi, diverse scopuri de folosire a sistemului, etc. (spre exemplu, există distribuţii care pot fi rulate direct de sub `MS-Windows`, fără a fi necesară instalarea lor în partiţii `UNIX` separate, sau distribuţii care pot fi *boot*-ate direct de pe CD, fără a necesita instalarea sistemului pe harddisc, numite distribuţii "*live*"). Pe portalul oficial dedicat `Linux`-ului, accesibil la adresa web http://www.linux.org, se găsesc informaţii despre distribuţiile de `Linux` disponibile în prezent, grupate după criteriile de clasificare amintite mai sus.
Majoritatea distribuţiilor sunt disponibile pentru *download* gratuit (adresele de la care pot fi descărcate sunt indicate pe portalul amintit mai sus). Alternativ, ele pot fi comandate pe *site*-ul producătorului spre a fi trimise pe CD-uri prin poştă, sau pot fi cumpărate de la magazin sub formă de pachet *software* (ce conţine CD-urile plus manuale tipărite de instalare şi utilizare), în ambele situaţii la preţuri modice (care să acopere diversele cheltuieli implicate de multiplicare, suportul fizic folosit, transport, ş.a.).

În concluzie, o distribuţie de `Linux` constă, în principal, dintr-o anumită versiune a *kernel*-ului de `Linux` şi dintr-o anumită selecţie (specifică producătorului distribuţiei respective) a programelor, cu diverse versiuni ale lor, ce formează colecţia de utilitare de bază şi programe de aplicaţii proprie acelei distribuţii. Plus un anumit program de instalare a acelei distribuţii şi de *management* al pachetelor de programe ce alcătuiesc acea distribuţie. Diferenţele între distribuţiile provenite de la producători diferiţi constau, în principal, în ce programe au fost selectate pentru a fi incluse, distribuţiile fiind adaptate pentru diverse scopuri, precum şi, uneori, în programul de instalare şi modul de gestiune al pachetelor de către acesta. Iar în cadrul unei distribuţii provenită de la un producător oarecare, diferenţele între diferitele versiuni ale ei constau, în principal, în versiunea nucleului şi versiunile programelor incluse în respectiva versiune a distribuţiei.

1.2.2 Instalarea unei distribuţii de Linux

Fiecare producător al unei distribuţii de Linux însoţeşte acea distribuţie de manuale care descriu modul de instalare şi de utilizare a acelei distribuţii. Aceste manuale sunt disponibile în format electronic pe CD-urile cu acea distribuţie, şi eventual şi în format tipărit (în cazul în care aţi cumpărat acea distribuţie de la magazin).

Atenţie: Înainte de a vă apuca de instalarea unei distribuţii este recomandabilă citirea manualului de instalare (mai ales în cazul utilizatorilor începători, este chiar necesară citirea în prealabil a manualului de instalare).

În continuare vom prezenta paşii generali ce trebuie urmaţi în vederea instalării unei distribuţii de Linux "clasice" (*i.e.*, care trebuie instalată pe harddisc în propria partiţie de tip Linux).
Am optat pentru o prezentare generală, fără a intra prea mult în detalii, din mai multe motive: lipsă de spaţiu, diferenţele dintre diversele distribuţii în ceea ce priveşte aspectele de amănunt ale procedurii de instalare, faptul că fiecare distribuţie are un manual de instalare bine documentat, şi în plus există numeroase cărţi de specialitate dedicate UNIX-ului şi, în particular, Linux-ului, multe dintre acestea descriind şi procedura de instalare pentru una sau mai multe dintre distribuţiile Linux cele mai răspîndite (spre exemplu, se pot consulta cărţile [2] şi [3]).
În concluzie, paşii care urmează sunt doar un ghid general, pentru instalarea unei distribuţii fiind necesară studierea documentaţiei acelei distribuţii şi/sau a unei cărţi de specialitate. Aceasta cel puţin la început, în cazul utilizatorilor începători într-ale Linux-ului, căci apoi se va întîmpla exact ca în cazul Windows-ului: după efectuarea unui număr mare de instalări şi reinstalări ale sistemului, se capătă experienţă, ajungîndu-se la o simplă instalare "cu ochii închişi".

1) Pregătirea instalării

Prima etapă constă în pregătirea pentru instalarea sistemului Linux, fiind constituită din următorii paşi:

1. *Pregătirea spaţiului liber pentru stocarea sistemului de fişiere al* Linux*-ului*

   Această etapă va crea spaţiu liber pe harddisc pentru partiţiile de Linux ce vor fi create ulterior. Dacă este un sistem nou, atunci harddiscul este gol, deci nu avem nici o problemă d.p.d.v. acesta.

   Cel mai adesea însă, pe calculator avem deja instalat sistemul MS-Windows (fie 9x, fie NT/2000/XP, nu contează ce fel este). În această situaţie, dacă totuşi mai avem spaţiu nepartiţionat pe harddisc, atunci iarăşi nu avem nici o problemă. De regulă însă, fie nu mai avem deloc spaţiu liber, nepartiţionat, fie avem, dar în cantitate insuficientă. În acest caz, va trebui să eliberăm spaţiu prin *redimensionarea* (*i.e.*, micşorarea) partiţiilor existente.

Aceasta se poate face în felul următor: mai întii se defragmentează partiţia ce urmează a fi micşorată, folosind fie utilitarul de defragmentare din `MS-Windows`, fie un program de defragmentare separat (cum ar fi, spre exemplu, cel din suita *Norton Utilities*). În felul acesta blocurile ocupate de date vor fi mutate la începutul partiţiei, iar cele libere la sfîrşitul ei. Urmează apoi micşorarea efectivă a partiţiei, care se poate realiza folosind utilitarul `FIPS.EXE` (ce se găseşte pe CD-urile distribuţiei respective, de obicei în directorul `cd:\DOSUTILS`), sau o aplicaţie de partiţionare comercială (cum ar fi, spre exemplu, programul *Partition Magic*).

Atenţie: înainte de partiţionare, realizaţi copii de siguranţă ale datelor existente pe partiţia respectivă (deoarece utilizarea programelor de partiţionare comportă anumite riscuri: în cazul apariţiei unor erori – de exemplu, dacă se întrerupe alimentarea cu curent electric – în timpul desfăşurării operaţiei de (re)partiţionare, se pot pierde datele, recuperarea lor ulterioară fiind, dacă nu imposibilă, cel puţin foarte anevoioasă).

2. *Alegerea metodei de instalare*

   De obicei, sunt disponibile trei metode de instalare, după locaţia programului de instalare:

   (a) *CD-ROM.* Instalarea se va face direct de pe CD-urile ce conţin distribuţia respectivă. Pentru pornirea sistemului se poate opta fie pentru *boot*-area de pe CD (de obicei primul CD al distribuţiei este *boot*-abil), dacă *BIOS*-ul calculatorului are opţiunea de *boot*-are de pe CD-uri, fie pentru *boot*-area cu ajutorul unei dischete de *boot*, despre a cărei mod de obţinere vom vorbi mai jos.

   (b) *Harddisk.* Instalarea se va face de pe disc, după ce în prealabil conţinutul CD-urilor din care este formată distribuţia au fost copiate pe o partiţie `Linux` sau `Windows` existentă. În acest caz este necesară discheta de *boot* amintită adineaori.

   (c) *Reţea.* Instalarea se va face prin reţea, de pe un alt calculator ce conţine distribuţia de `Linux`, şi pe care o exportă în reţea prin protocolul NFS, FTP, sau HTTP. Şi în acest caz este necesară o dischetă de *boot*, care trebuie să aibă şi suport pentru reţea.

   Pe lîngă aceste metode de instalare, care toate presupun pornirea calculatorului prin *boot*-area unui sistem `Linux` minimal, fie de pe o dischetă de *boot*, fie de pe CD, mai există o posibilitate de instalare direct de pe CD-ROM, în situaţia în care pe calculator există deja instalat sistemul `MS-DOS/Windows`. Şi anume, se porneşte acest sistem şi se apelează un program special dedicat acestui scop (numit, de obicei, `AUTOBOOT.EXE` sau `AUTORUN.EXE`, şi care se găseşte pe CD-urile distribuţiei respective, de obicei în directorul `cd:\DOSUTILS`); pentru mai multe detalii despre această posibilitate, consultaţi documentaţia distribuţiei.

3. *Crearea dischetei de boot*

   După cum am amintit mai sus, este nevoie de crearea unei dischete de *boot* pentru `Linux`, care va fi folosită pentru *boot*-area unui sistem `Linux` minimal, cu ajutorul căruia se va face instalarea propriu-zisă a distribuţiei de `Linux`.

Crearea dischetei de *boot* pentru `Linux` se poate face din cadrul sistemului `MS-DOS/Windows` (folosind eventual un alt calculator ce are instalat acest sistem, în situaţia în care sistemul nostru este nou, fără nici un sistem de operare instalat pe el; sau, putem folosi o dischetă de *boot* pentru `MS-DOS`, cu suport pentru CD-ROM, pentru a porni sistemul), procedîndu-se în felul următor:
se copie pe o dischetă goală imaginea dischetei de *boot* aflată pe CD-urile distribuţiei, folosind un utilitar dedicat acestui scop (cum ar fi programul `RAWRITE.EXE`, ce se găseşte pe CD-urile distribuţiei respective, de obicei în directorul `cd:\DOSUTILS`); pentru mai multe detalii despre această operaţie, consultaţi documentaţia distribuţiei.

Observaţie: discheta de *boot* pentru `Linux` nu este necesară la instalare în situaţia în care instalarea se va face prin *boot*-area de pe CD-ul ditribuţiei, sau cînd instalarea va fi pornită din `MS-DOS/Windows`.
Totuşi, este recomandabil să creaţi o dischetă de *boot*, fie în această etapă pregătitoare, fie după instalare, pe care s-o aveţi la îndemîna dacă vreodată veţi fi în situaţia în care *boot-manager*-ul instalat de `Linux` va fi corupt (situaţie care se poate întîmpla la o reinstalare ulterioară a sistemului `MS-Windows`, deoarece programul acestuia de instalare rescrie *MBR*-ul (=*Master Boot Record*), iar acesta poate conţine *boot-manager*-ul de `Linux`); în această situaţie, după terminarea reinstalării `MS-Windows`-ului, veţi putea să porniţi sistemul cu ajutorul dischetei de *boot*, şi să refaceţi *boot-manager*-ul de `Linux` în *MBR*.

4. *Planificarea partiţiilor de* `Linux`

   Este recomandabilă crearea cel puţin a următoarelor partiţii:

   (a) o partiţie de *swap*, ce va fi folosită pentru memoria virtuală. Dimensiunea acestei partiţii trebuie să fie de minim 32 MB şi maxim 2 GB, dar recomandabil este să fie cam dublul memoriei RAM instalate in calculator (mai precis, pentru a fi optimă pentru majoritatea aplicaţiilor din ziua de azi, capacitatea memoriei virtuale se recomandă a fi aleasă astfel: de cca. 500 MB pentru o memorie RAM de 128 MB, de 256 MB pentru o memorie RAM de 256 MB, şi poate lipsi în cazul unei memorii RAM de 512 MB).

   (b) o partiţie pentru directorul `/boot`, ce va conţine nucleul `Linux` şi celelalte fişiere utilizate în timpul *boot*-ării. Ca dimensiune poate fi aleasă valoarea 32 MB.
   Observaţie importantă: aceasta fiind partiţia de pe care se *boot*-ează sistemul, există o restricţie asupra plasării sale pe harddisc. Şi anume, ea trebuie plasată la începutul harddiscului, sub limita de 1 GB (mai precis, această partiţie trebuie să aibă cilindrul de start înaintea cilindrului 1024). Această limitare se datorează modului restrictiv de acces la harddisc al programului LILO (*i.e.*, programul *boot-manager* responsabil cu pornirea sistemului). Pentru celelalte partiţii nu există nici o restricţie, ele putînd fi plasate la orice distanţă de începutul harddiscului, fie ca partiţii primare, fie ca partiţii logice în cadrul partiţiei extinse definită pe respectivul harddisc.

   (c) o partiţie de *root*, acolo unde se va afla `/` (*i.e.*, rădăcina structurii arborescente a sistemului de fişiere), şi care va conţine toate fişierele din sistem. Dimensiunea acestei partiţii trebuie aleasă astfel încît să încapă toate pachetele de aplicaţii

ce vor fi alese pentru instalare (în timpul instalării propriu-zise veţi avea posibilitatea de a alege ce aplicaţii doriţi să instalaţi dintre toate cele disponibile în distribuţia respectivă, şi vi se va comunica şi spaţiul necesar). Ţinînd cont că harddiscurile actuale au dimensiuni de zeci sau chiar sute de GB, puteţi alege fără probleme o dimensiune de cîţiva GB sau chiar mai mult pentru această partiţie.

Observaţie: în cazul în care calculatorul nu va fi folosit doar ca staţie de lucru, ci ca server `Linux`, se recomandă crearea unor partiţii suplimentare:

(a) o partiţie pentru directorul `/home`, ce va conţine fişierele utilizatorilor cu conturi pe acel server;

(b) o partiţie pentru directorul `/var`, ce va conţine fişierele cu conţinut variabil ale sistemului;

(c) o partiţie pentru directorul `/usr`, ce va conţine fişierele sistemului de operare şi aplicaţiile instalate ulterior.

În final, să amintim şi tipul sistemului de fişiere ce trebuie utilizat pentru fiecare dintre partiţiile amintite mai sus; acest tip trebuie specificat atunci cînd se realizează efectiv operaţia de partiţionare, ce este însoţită de operaţia de creare a sistemului de fişiere pe partiţia respectivă (analogul operaţiei de *formatare* utilizate în `MS-DOS/Windows`).

Pentru partiţia de *swap*, ce va avea codul numeric 82, trebuie ales tipul de sistem de fişiere *swap*. Celelalte partiţii amintite mai sus vor fi partiţii native de `Linux`, cu codul numeric 83, pentru care se va alege ca sistem de fişiere unul din tipurile următoare: *ext2* (sistemul clasic de fişiere `Linux`, compatibil cu standardele `UNIX`), *ext3* (noul sistem de fişiere `Linux`, bazat pe *ext2*, ce are adăugat suport pentru jurnalizare), sau *reiserfs* (un sistem nou de fişiere, cu suport pentru jurnalizare, ce are performanţe superioare în multe situaţii sistemelor *ext2* şi *ext3*, datorită arhitecturii interne mai eficiente).

Realizarea efectivă a partiţionării se va face într-o primă etapă a programului de instalare a distribuţiei respective. Ea se poate face şi în avans, folosind discheta de *boot*, creată la un pas amintit mai sus, pentru a porni sistemul, şi apoi se poate apela utilitarul în mod text `fdisk` pentru a crea partiţiile de `Linux`.

2) Începerea instalării propriu-zise

A doua etapă constă în instalarea propriu-zisă a sistemului `Linux`, care se începe prin pornirea (*boot*-area) sistemului cu ajutorul dischetei de *boot* amintite mai sus, sau direct folosind CD-ul *boot*-abil al distribuţiei. La sfîrşitul etapei de pornire a sistemului, se va afişa un ecran cu informaţii despre modurile de startare a instalării şi un prompter de forma

```
boot :
```

la care se aşteaptă alegerea unei opţiuni de startare a instalării.

Există două interfeţe ale programului de instalare, ce pot fi folosite la alegere în timpul instalării: o interfaţă în mod text (care se selectează de obicei tastînd comanda `text` sau `linux text` la prompterul "`boot :`" amintit anterior; comanda exactă ce trebuie tastată depinde de distribuţie, dar de obicei se oferă informaţii în acest sens chiar pe ecranul afişat în acest punct al instalării) şi o interfaţă în mod grafic, care este aleasă implicit (după scurgerea unui interval de cîteva secunde fără reacţie din partea utilizatorului, sau imediat la apăsarea tastei `ENTER` după apariţia prompterului amintit anterior). Sigur, interfaţa în mod grafic este mai "prietenoasă" pentru utilizator, dar totuşi este recomandabilă folosirea interfaţei în mod text, de exemplu în situaţia în care placa video instalată în sistem are performanţe slabe, sau dacă se doreşte un timp mai scurt de instalare, deoarece interfaţa în mod text este mai rapidă.

După alegerea interfeţei text sau grafică, programul de instalare parcurge următoarele etape de instalare (*notă*: reamintesc faptul că este doar o prezentare generală, pentru o anumită distribuţie concretă s-ar putea să apară unele mici diferenţe – etape suplimentare sau în minus, sau ordinea în care apar acestea poate fi uşor schimbată) :

1. *Selectarea limbii.*

   Limba selectată va fi utilizată pe parcursul instalării şi, implicit, şi după instalare.

2. *Selectarea tipului de instalare.*

   Se poate alege o instalare completă ("pe curat") sau o actualizare a unei instalări mai vechi (*upgrade*). În cazul instalării complete, se poate alege între o instalare *recommended*, în care opţiunile de instalare şi pachetele ce vor fi instalate sunt selectate automat de către programul de instalare, în conformitate cu un scenariu ales de utilizare a sistemului: *personal desktop* (sistem personal), *workstation* (staţie de lucru), sau *server*, şi o instalare *custom* (sau expert), care permite modificarea opţiunilor de instalare şi selecţia pachetelor dorite, oferind astfel cea mai mare flexibilitate posibilă.

3. *Configurarea tastaturii şi a mouse-ului.*

4. *Partiţionarea discului.*

   În cadrul acestei etape se definesc şi se formatează partiţiile necesare `Linux`-ului, în conformitate cu cele discutate la etapa pregătitoare a instalării.
   Există de obicei două opţiuni de partiţionare: automată şi manuală. Pentru cea din urmă se foloseşte programul de partiţionare *Disk Druid*, în cazul interfeţei grafice, respectiv cu programul clasic *fdisk*, disponibil doar pentru interfaţa în mod text.

5. *Instalarea încărcătorului de boot.*

   Pentru a putea porni sistemul `Linux` este nevoie de un încărcător de *boot* (*boot loader*), care poate porni de asemenea şi alte sisteme de operare ce sunt instalate pe disc.

   Încărcătorul clasic ce se foloşeste este programul *LILO* (acronim ce provine de la *LInux LOader*), dar mai avem şi alte două alternative: putem folosi programul

GRUB (*GRand Unified Boot loader*), sau nici un încărcător de *boot*, situaţie în care va trebui să pornim de fiecare dată sistemul `Linux` într-o altă manieră (fie cu o dischetă de *boot* pentru `Linux`, fie cu un program ce startează `Linux`-ul de sub `MS-DOS/Windows`).

Tot în această etapă se mai stabilesc modul de instalare a încărcătorului de *boot* şi celelalte sisteme de operare ce vor fi pornite de către încărcătorul de *boot*. Acesta poate fi instalat fie în *MBR* (=*Master Boot Record*), adică sectorul de boot de la începutul discului ce este încărcat automat de BIOS-ul calculatorului (se recomandă folosirea acestei opţiuni), fie în primul sector al partiţiei de *root* (în această situaţie trebuie configurat încărcătorul sistemului de operare instalat anterior pe disc pentru a şti să apeleze încărcătorul de `Linux` plasat în primul sector al partiţiei de *root* al acestuia).

Observaţii:

(a) În cazul folosirii încărcătorului clasic *LILO*, configurarea sistemelor de operare ce vor putea fi pornite prin intermediul lui, se face cu ajutorul fişierului `/etc/lilo.conf`, care este un fişier text ce poate fi editat direct pentru a specifica sistemele de operare, partiţiile de pe care vor fi pornite, şi alţi parametri opţionali de transmis *kernel*-ului `Linux` la încărcarea acestuia. După editare, activarea modificărilor efectuate se face cu comanda `lilo` (`/sbin/lilo`).
Spre exemplu, se poate pune o parolă în `/etc/lilo.conf` pentru pornirea restrictivă a `Linux`-ului – pentru orice parametru opţional transmis *kernel*-ului, se va cere parola; în acest caz, trebuie protejat fişierul astfel încît să nu fie accesibil decît *superuser*-ului (lucru ce se poate realiza cu comanda `chmod 600 /etc/lilo.conf` , efectuată de către utilizatorul *root*). Această parolă de pornire oferă protecţie faţă de atacurile de la consolă.
Cu comanda `man lilo.conf` puteţi consulta documentaţia referitoare la acest fişier de configurare a încărcătorului de *boot*.

(b) La prompterul "`lilo :`" afişat de încărcătorul *LILO*, pe lîngă comenzile implicite ce pot fi tastate, şi anume cele de selectare a sistemului de operare ce urmează a fi încărcat, se mai pot tasta o serie de alte opţiuni utile pentru utilizatorii avansaţi, ca de exemplu cu comanda `append="..."` se pot specifica o serie de parametri ce vor fi transmişi *kernel*-ului la încărcarea acestuia, sau cu comanda `linux single` se va porni sistemul în mod *single user* şi se va intra în sistem ca *root* (*i.e.*, *superuser*-ul), fără faza de autentificare (*i.e.*, nu se mai cere parola). Protecţia împotriva acestui tip de acces se poate realiza folosind o parolă în fişierul `/etc/lilo.conf`, conform celor discutate mai sus.

6. *Configurarea legăturii de reţea.*

   Se configurează placa (sau plăcile) de reţea aflată în calculator, împreună cu toate datele necesare pentru buna funcţionare în cazul legării într-o reţea de calculatoare: adresa IP, adresa de reţea, masca de reţea, numele maşinii, adresa *gateway*-ului, adresa DNS-ului, ş.a.

7. *Configurarea nivelului de securitate.*

Se configurează *firewall*-ul pe baza nivelului de securitate ales, dintre mai multe opţiuni posibile: nivel înalt, nivel mediu, nivel jos (fără *firewall*), sau opţiunea *custom*, ce permite configurarea manuală a *firewall*-ului. *Firewall*-ul este o aplicaţie foarte importantă d.p.d.v. al securităţii sistemului, ea avînd ca sarcină *filtrarea* traficului prin legătura de reţea, în funcţie de adresele IP şi porturile din pachetele de date ce o tranzitează.

8. *Configurarea utilizatorilor*.

   Se alege parola pentru *superuser* (*i.e.*, utilizatorul cu numele *root*), care posedă drepturi totale asupra sistemului. Acest utilizator trebuie folosit în mod normal doar pentru instalarea/dezinstalarea de programe şi pentru administrarea sistemului. În rest, pentru utilizarea calculatorului, se recomandă crearea unuia sau mai multor utilizatori obişnuiţi (adică, fără drepturi depline în sistem) care să fie folosiţi pentru lucrul cu calculatorul, chiar dacă acesta este folosit doar ca sistem personal (*i.e.*, acasă), deoarece o comandă greşită tastată ca *root* (*i.e.*, utilizatorul cu drepturi depline) poate cauza deteriorarea sistemului sau chiar pierderea datelor şi aplicaţiilor stocate pe disc.

9. *Configurarea autentificării în sistem*.

   Dacă calculatorul este legat în reţea, din motive de securitate este foarte important ca accesul la sistem de la distanţă (de pe un alt calculator legat la reţea, folosind protocoalele TELNET sau SSH – revedeţi discuţia despre "Conectarea la un sistem `UNIX`" din prima secţiune a acestui capitol, şi amintiţi-vă recomandarea de a folosi SSH în loc de TELNET), să fie posibil pe baza unui sistem de autentificare sigur.

   În acest sens, sunt disponibile mai multe opţiuni utile:
   • activarea/dezactivarea sistemului MD5, ce permite folosirea de parole mai sigure (cu lungimea de maxim 256 de caractere, în loc de lungimea maximă standard de 8 caractere).
   • activarea/dezactivarea sistemului *shadow* (ce permite stocarea sigură a parolelor în fişierul `/etc/shadow`, în locul variantei nesigure de păstrare în fişierul `/etc/passwd`).
   • activarea sistemului de autentificare NIS (*Network Information Service*) sau a LDAP (*Lightweight Directory Access Protocol*) – ambele mecanisme folosesc conceptul de interogare prin reţea a unui server ce conţine o bază de date de autentificare (asemănaţor cu mecanismul *Active Directory* din lumea `Windows`).
   • activarea *Kerberos* sau a SMB (*Samba*), alte două sisteme ce oferă servicii de autentificare în reţea.

10. *Selectarea şi instalarea pachetelor*.

    În această etapă, în funcţie şi de tipul de instalare selectat la al doilea pas, se pot selecta pachetele (*i.e.*, aplicaţiile) ce se doresc a fi instalate, dintre cele disponibile în distribuţia respectivă – acestea de obicei sunt de ordinul sutelor, aranjate în grupuri de pachete pe baza rolului acestora: de exemplu, aplicaţii (editoare, de calcul inginerese şi ştiiţific, suite de productivitate *office*, etc.) programe pentru *development*, programe pentru INTERNET (poştă electronică, navigatoare de *web*, etc.), programe pentru diverse servere de servicii (server de *web*, server *Samba*, server DNS, etc.), programe de sistem (pentru administrare, configurare, ş.a.), medii

grafice, ş.a.De asemenea, se oferă informaţii despre spaţiul necesar pentru instalarea pachetelor selectate.

După selectarea pachetelor, programul de instalare verifică *dependenţele* dintre pachete (anumite aplicaţii se bazează pe altele pentru a funcţiona corect) şi rezolvă lipsurile constatate pe baza interacţiunii cu utilizatorul, iar apoi are loc instalarea propriu-zisă a pachetelor.

Observaţie: Mediile grafice ce pot fi selectate folosesc sistemul de ferestre grafice X WINDOW, ce are o arhitectură de tip client-server bazată pe protocolul X11 (acest sistem a fost dezvoltat iniţtial la *MIT*, după cum am amintit în istoricul evoluţiei `UNIX`-ului, de la începutul acestui capitol). Ca medii grafice, două sunt cele mai răspîndite: *GNOME* şi *KDE*, şi se poate selecta instalarea amîndorura, numai a unuia dintre ele, sau a niciunuia (caz în care nu vom putea exploata sistemul folosind o interfaţă grafică, ci doar în interfaţa clasică în mod text). Cîteva detalii despre aceste medii:

- Mediul grafic *GNOME* (*GNU Network Object Modal Environment*) este dezvoltat de organizaţia cu acelaşi nume (http://www.gnome.org), şi este o parte a proiectului GNU. Este *free software* şi în prezent a ajuns la versiunea 2.6, disponibilă pentru `Linux` şi pentru alte variante de `UNIX` (`Solaris`, `BSD`, ş.a.).
  Pe lîngă faptul că este un mediu grafic, *GNOME* este şi o platformă de dezvoltare de aplicaţii grafice. Pentru programarea aplicaţiilor *GNOME* se foloseşte *GTK+* (http://www.gtk.org), un *toolkit* multi-platformă pentru crearea de interfeţe grafice utilizator (*GUI*s). *GTK+* face parte şi el din cadrul proiectului GNU şi este *free software*, fiind dezvoltat dintr-un proiect mai vechi de manipulare a imaginilor, *GIMP* (*GNU Image Manipulation Program*).
- Mediul grafic *KDE* (*K Desktop Environment*) este dezvoltat de organizaţia cu acelaşi nume (http://www.kde.org). Este *open source* şi în prezent a ajuns la versiunea 3.2.3, disponibilă pentru `Linux` şi pentru alte variante de `UNIX`.
  Pe lîngă faptul că este un mediu grafic, *KDE* este însoţit şi de o suită de aplicaţii de birou, numită *KOffice*, precum şi de un *framework* de dezvoltare de aplicaţii grafice. Pentru programarea aplicaţiilor *KDE* se foloseşte *Qt*, care este un *framework* general de dezvoltare de aplicaţii C++. *Qt* este *free*, fiind disponibil sub o licenţă stil *BSD*.

Voi mai menţiona faptul că mai există şi *IceWM* (*Ice Window Manager*), un *manager* de ferestre pentru sistemul X11 WINDOW. *IceWM* are avantajul că necesită mai puţine resurse decît mediile *GNOME* şi *KDE*, fiind deci util pentru cei cu calculatoare mai puţin performante.

11. *Selectarea timpului şi datelor regionale.*

    Se selectează limba, tastatura, data şi timpul curent, ţara şi fusul orar (*time zone*).

12. *Configurarea plăcii video.*

    În majoritatea cazurilor programul de instalare poate determina singur tipul plăcii video din sistem. În situaţia în care aveţi însă o placă video mai neobişnuită, s-ar putea să fiţi nevoit să-i indicaţi programului de instalare care este tipul plăcii (prin

selecţia dintr-o listă de tipuri cunoscute) şi chiar să-i căutaţi un *driver* potrivit (la distribuitor sau pe INTERNET), dacă distribuţia nu conţine *driver* pentru acel tip de placă.

13. *Configurarea monitorului şi personalizarea sistemului* X WINDOW.

    În majoritatea cazurilor programul de instalare poate determina singur tipul monitorului, altfel poate fi ajutat de utilizator prin selecţia dintr-o listă de tipuri cunoscute.

    Pentru interfaţa grafică, se selectează rezoluţia şi adîncimea culorii dorite, mediul *desktop* (GNOME sau KDE) dorit, şi dacă sistemul va porni direct în mod grafic, sau în mod consolă (*i.e.*, interfaţa text clasică).

    Observaţii:

    (a) În cazul pornirii sistemului în mod text, avem la dispoziţie 6 *terminale* la care putem deschide cîte o sesiune de lucru de la consola sistemului (*i.e.*, de la tastatura şi monitorul conectate direct la calculatorul respectiv). Bineînţeles, ele nu pot fi folosite în acelaşi timp – la orice moment de timp există doar un terminal activ, care gestionează *input*-ul de la tastatură şi *output*-ul pe ecranul monitorului – dar avem posibilitatea de a comuta între ele pentru a le folosi alternativ, această comutare realizîndu-se prin apăsarea combinaţiilor de taste `ALT+F1`, `ALT+F2`, ..., `ALT+F6` ce selectează terminalul corespunzător.

    (b) În cazul conectării la sistem de la distanţă pentru o sesiune de lucru, *input*-ul de la tastatură calculatorului de la distanţă şi *output*-ul pe ecranul monitorului de la distanţă preluate în cadrul aplicaţiei ce rulează la distanţă (clientul de SSH, spre exemplu aplicaţia PUTTY, despre care am discutat în prima parte a acestui capitol), sunt gestionate local (pe sistemul `Linux`) prin aşa-numitele *pseudo-terminale*, care au un rol asemănător cu terminalele folosite pentru conectarea de la consola sistemului.

    (c) Se cuvine menţionat faptul că se poate folosi interfaţa grafică şi în cazul conectării de la distanţă, cu observaţia că necesarul de resurse şi traficul prin reţea este mai mare în acest caz (*notă*: din acest motiv accesul studenţilor pe serverul `fenrir` al acestora, din laboratoare sau de acasă, este permis numai în mod text, nu şi în mod grafic).

14. *Configurarea celorlalte dispozitive hardware*.

    Configurarea celorlalte dispozitive periferice existente eventual în calculator, şi anume: placa de sunet, placa de reţea, imprimanta, *scanner*-ul, *tunner*-ul TV, ş.a., decurge asemănător ca pentru placa video: în majoritatea cazurilor programul de instalare poate determina automat tipul dispozitivului respectiv, sau poate fi ajutat de utilizator (prin selecţia dintr-o listă de tipuri cunoscute). În situaţia în care aveţi însă un dispozitiv periferic mai neobişnuit, s-ar putea să fiţi nevoit să-i căutaţi un *driver* potrivit (la distribuitor sau pe INTERNET), dacă distribuţia nu conţine un *driver* pentru acel tip de periferic.

15. *Crearea unei dischete de boot*.

Se oferă posibilitatea de a crea o dischetă de *boot*, ce poate fi folosită pentru a porni sistemul `Linux`. Despre această dischetă am amintit deja în prima fază, cea a pregătirii instalării, şi am explicat atunci şi de ce este recomandabil să fie creată această dischetă.

16. *Instalarea de versiuni actualizate ale pachetelor.*

    Unele distribuţii oferă opţiunea de conectare automată prin INTERNET la *site*-ul oficial al distribuţiei pentru descărcarea eventualelor versiuni mai recente ale pachetelor de programe deja instalate.

17. *Terminarea instalării.*

    La încheierea tuturor etapelor descrise mai sus, programul de instalare va cere permisiunea pentru repornirea sistemului. Se scoate eventuala dischetă de *boot* din unitatea *floppy*, sau CD-ul din unitatea CD-ROM, şi se restartează sistemul. Ca urmare, se va încărca *boot loader*-ul configurat în timpul instalării, care va încărca sistemul `Linux`, şi poate începe exploatarea acestuia.

În concluzie, cam aceştia ar fi paşii generali ce trebuie urmaţi în vederea instalării unei distribuţii de `Linux`. După cum se poate observa, sunt foarte asemănători cu paşii procedurii de instalare a unui sistem `Windows`. Chiar dacă la început aveţi impresia că procedura de instalare a `Linux`-ului este foarte anevoioasă, cu timpul (executînd multe instalări şi reinstalări) veţi acumula experienţă şi vi se va părea tot mai uşoară, la fel ca în cazul `Windows`-ului.

3) Actualizarea sistemului

După cum am discutat mai sus la tipul instalării, pe lîngă opţiunea de instalare completă a sistemului, distribuţiile de `Linux` mai oferă şi opţiunea de *upgrade* al sistemului, adică o actualizare a acestuia (*i.e.*, instalarea unei versiuni mai recente a distribuţiei respective peste una mai veche deja instalată).

De obicei, se oferă două moduri de *upgrade*: actualizarea numai a pachetelor (*i.e.*, aplicaţiilor) instalate, şi actualizarea completă, a pachetelor instalate, a *kernel*-ului `Linux`, şi al încărcătorului de *boot*.

În continuare, cîteva cuvinte despre:

Instalarea de programe

Pe parcursul exploatării sistemului, se poate ivi nevoia de a utiliza programe noi, neinstalate/inexistente în cadrul distribuţiei, obţinute din diverse surse, de obicei de pe INTERNET.

Instalarea unui program se poate face şi de către un utilizator obişnuit, în propriul director *home*, şi în acest caz programul respectiv îi va fi accesibil doar acestuia, dar cel mai recomandabil este să se facă de către utilizatorul *root*, pentru a fi accesibil tuturor utilizatorilor din sistem (unele programe nu pot fi instalate decît numai de *root*, deoarece au nevoie de privilegii extinse pentru a putea fi instalate).

În principal, se folosesc două formate pentru distribuţia programelor prin INTERNET:

1. formatul arhivă .tar.gz, *i.e.* *nume_program*`.tar.gz` (eventual poate conţine şi versiunea pe lînga numele programului);
2. formatul RPM, *i.e.* *nume_program-nr_versiune*`.rpm`, care este un format introdus de firma *Red Hat*, pentru distribuirea programelor executabile şi gestiunea pachetelor din cadrul distribuţiilor de `Linux`(pentru detalii a se consulta adresa http://www.rpm.org).

Indiferent de formatul folosit, programele sunt distribuite împreună cu manuale de instalare şi utilizare, pe care vă recomand să le citiţi înainte de a vă apuca de instalarea propriu-zisă. De asemenea, de multe ori sunt însoţite şi de arhive ce conţin sursele programului (ca în cazul programelor *open-source*).

Se cuvine să mai menţionăm modul propriu-zis de instalare pentru fiecare din cele două formate de distribuţie (*nota*: vom descrie doar paşii generali ce trebuie următi, pentru detalii rămine în sarcina cititorului să consulte documentaţia de instalare specifică fiecărui program în parte).

I) Formatul .tar.gz

Mai întii se configurează aplicaţia respectivă, cu comanda:

```
UNIX> configure [optiuni]
```

prin care se pot specifica diverite opţiuni de configurare (în lipsa specificării, se vor folosi valorile implicite pentru acestea). Spre exemplu, cu opţiunea `--prefix=`*`director-de-instalare`* se poate specifica un director în care se va instala aplicaţia respectivă. Lista tuturor opţiunilor disponibile pentru configurare se poate obţine cu comanda

```
UNIX> configure --help
```

Apoi, cu secvenţa de comenzi

```
UNIX> make
UNIX> make install
```

are loc instalarea propriu-zisă. Iar dezinstalarea se face foarte simplu, prin ştergerea directorului (inclusiv cu subdirectoarele sale) în care a fost instalată acea aplicaţie, lucru realizat prin comanda

```
UNIX> rm -r director-de-instalare
```

(în `Linux` nu există echivalentul *registry*-ului din `MS-Windows`, în care sunt salvate numeroase informaţii despre aplicaţiile instalate).

II) Formatul RPM

Aplicaţiile în acest format sunt gestionate cu comanda (programul) `rpm`. Spre exemplu, prin apelul

```
UNIX> rpm -qa
```

se pot afla informaţii despre pachetele instalate.

Instalarea unui program *nume_program-nr_versiune*`.rpm` se face prin apelul

```
UNIX> rpm -iv nume_program-nr_versiune.rpm
```

(opţiunea `-i` înseamnă instalare, iar opţiunea `-v` înseamnă modul *verbose*, adică o afişare detaliată de informaţii pe parcursul instalării).
Se poate face şi *upgrade*-ul la o versiune mai recentă a unui program deja instalat, prin apelul

```
UNIX> rpm -u nume_program-nr_versiune.rpm
```

(opţiunea `-u` înseamnă *upgrade*).
Dezinstalarea se face cu opţiunea `-e` urmată doar de numele programului respectiv:

```
UNIX> rpm -e nume_program
```

Se cuvine menţionat faptul că cele două moduri de instalare de mai sus se folosesc în modul text. Există însă şi programe de instalare în mod grafic – astfel, spre exemplu, pentru mediul *GNOME* avem disponibilă aplicaţia în mod grafic `gnorpm`, ce oferă aceleaşi funcţionalităţi ca şi programul în linie de comandă `rpm` descris mai sus.

În încheierea acestei secţiuni, voi menţiona cîteva motive pentru a vă convinge să vă instalaţi acasă o distribuţie de `Linux`.

Deşi este suficientă folosirea contului pe care-l aveţi pe serverul studenţilor `fenrir` pentru lucrul individual implicat de această disciplină (pentru a experimenta comenzile şi a lucra cu programele C la care le vom referi pe parcursul acestui manual, şi pentru a studia documentaţia `man`), totuşi acest cont este un cont de utilizator obişnuit, ce are numeroase

limitări.
Ca atare, instalîndu-vă acasă o distribuţie de `Linux`, vă veţi bucura de o flexibilitate mult sporită în folosirea sistemului, veţi putea lucra cu puteri depline, nefiind îngrădiţi de limitările impuse pe serverul studenţilor.
Una dintre aceste limitări este cea referitoare la folosirea interfeţei grafice; acasă veţi putea instala şi folosi nestingheriţi interfaţa grafică, adică un mediu grafic (*GNOME* sau *KDE*) bazat pe sistemul de ferestre X WINDOW.
În sfîrşit, un aspect deloc de neglijat este şi acela al vitezei legăturii de reţea dintre calculatorul dumneavoastră de acasă şi serverul studenţilor de la Facultatea de Informatică (mai ales dacă sunteţi conectat la *ISP*-ul dumneavoastră prin modem şi nu prin cablu), precum şi cel al preţului implicat de accesul în reţea; d.p.d.v. acesta, este evident mai convenabil să vă instalaţi `Linux` pe calculatorul de acasă şi să lucraţi pe el, decît să lucraţi pe serverul studenţilor conectat de acasă.

1.3 Exerciţii

Exerciţiul 1. Ce înseamnă `UNIX`?

Exerciţiul 2. Care sunt cele trei caracteristici majore ale `UNIX`-ului? Descrieţi-le pe scurt.

Exerciţiul 3. Din ce este compus un sistem `UNIX`?

Exerciţiul 4. Care sunt nivelele în care este structurat un sistem `UNIX`? Descrieţi-le pe scurt.

Exerciţiul 5. Descrieţi sistemul de fişiere şi cel de procese din `UNIX`.

Exerciţiul 6. Cum gestionează `UNIX`-ul utilizatorii?

Exerciţiul 7. Care este serverul studenţilor şi cum vă puteţi conecta la el pentru o sesiune de lucru? Dar pentru una de transfer de fişiere?

Exerciţiul 8. Ce este `Linux`-ul?

Exerciţiul 9. Ce înseamnă o distribuţie de `Linux`?

Exerciţiul 10. Descrieţi paşii generali ai procesului de instalare a unei distribuţii de `Linux`. Încercaţi să vă instalaţi pe calculatorul de acasă o distribuţie de `Linux`.

Capitolul 2

UNIX. Ghid de utilizare

2.1 Comenzi UNIX. Prezentare a principalelor categorii de comenzi

1. **Introducere**
2. **Comenzi de** *help*
3. **Editoare de texte**
4. **Compilatoare, depanatoare, ş.a.**
5. **Comenzi pentru lucrul cu fişiere şi directoare**
6. **Comenzi ce oferă diverse informaţii**
7. **Alte categorii de comenzi**
8. *Troubleshooting* **(Cum să procedaţi dacă se "blochează" o comandă)**

2.1.1 Introducere

La fel ca în MS-DOS, şi în UNIX există două categorii de *comenzi* (numite în acest caz şi *comenzi* UNIX, sau *comenzi shell*):

- **comenzi interne :**

  comenzi care se gasesc in fisierul executabil al *shell*-ului (i.e., interpretorului de comenzi) respectiv, ca de exemplu: `cd`, `echo`, `alias`, `exec`, `exit`, ş.a.

- **comenzi externe :**

  comenzi care se gasesc separat, fiecare intr-un fisier avind acelasi nume cu comanda respectiva.
  Acestea pot fi de doua feluri:

  1. *fisiere executabile* (adica, programe executabile obtinute prin compilare din programe sursa scrise in C sau alte limbaje), ca de exemplu: `ls`, `chmod`, `passwd`, `bash`, ş.a.;
  2. *fisiere de comenzi*, numite si *script*-uri (adica, fisiere text ce contin secvente de comenzi, analog fisierelor `*.bat` din `MS-DOS`), ca de exemplu: `.bash_profile`, `.bashrc`, ş.a.

Forma generala a unei comenzi `UNIX` este:

`UNIX`> *nume_comanda optiuni argumente* ,

unde optiunile si argumentele pot lipsi, dupa caz.
Prin conventie, optiunile sunt precedate de caracterul '-' (in `MS-DOS` este folosit caracterul '/'). Argumentele sunt cel mai adesea nume de fisiere.
Daca este vorba de o comanda externa, aceasta poate fi specificata si prin calea ei (absoluta sau relativa).
Separatorul intre numele comenzii si ceilalti parametri ai acesteia, precum si intre fiecare dintre parametri este caracterul `SPACE` sau `TAB` (unul sau mai multe spatii sau tab-uri).
O comanda poate fi scrisa pe mai multe linii, caz in care fiecare linie trebuie terminata cu caracterul '\', cu exceptia ultimei linii.

Pentru a putea executa fisierul asociat unei comenzi, utilizatorul care a lansat acea comanda trebuie sa aiba drept de executie (*i.e.*, atributul `x` corespunzator sa fie setat) pentru acel fisier (vom reveni in sectiunea urmatoare cu amanunte legate de drepturile de acces la fisiere).

In sectiunea 2.3 vom discuta mai in amanunt despre interpretoarele de comenzi din `UNIX` si modul in care executa acestea comenzile.

In continuarea acestei sectiuni vom trece in revista principalele categorii de comenzi disponibile.

2.1.2 Comenzi de *help*

Lista tuturor comenzilor interne disponibile intr-un anumit *shell* (*i.e.*, interpretor de comenzi `UNIX`) se poate obtine executind in acel *shell* comanda urmatoare:

`UNIX> help`

iar pagina de help pentru o anumita comanda interna se obtine cu comanda:

`UNIX> help` *nume_comanda_interna*

Pentru a obtine *help* despre comenzile externe sunt disponibile urmatoarele comenzi: `man`, `whatis`, `apropos`, `info`.

Comanda `man` se foloseste cu sintaxa:

`UNIX> man [`*nr_sectiune*`]` *cuvint*

si are ca efect afisarea *paginii de manual* pentru cuvintul specificat.
Cuvintul specificat reprezinta numele unei **comenzi externe** sau numele unei **functii de biblioteca C/C++** pentru care se doreste obtinerea paginii de manual.
Pagina afisata cuprinde: sintaxa comenzii/functiei, optiunile si argumentele ei cu descrierea lor, efectul acelei comenzi/functii, exemple, etc.
Argumentul optional *nr_sectiune* se utilizeaza atunci cind exista mai multe comenzi sau functii de biblioteca cu acelasi nume, pentru a afisa pagina de manual (corespunzatoare uneia dintre acele comenzi sau functii) din sectiunea de manual specificata.

Exemple:

- `UNIX> man man`

  Efect: afiseaza pagina de manual referitoare la comanda `man`.

- `UNIX> man write`

  Efect: afiseaza pagina de manual referitoare la comanda `write`, care scrie un mesaj pe terminalul utilizatorului specificat.

- `UNIX> man 2 write`

  Efect: afiseaza pagina de manual referitoare la functia de biblioteca `write`, care este apelata in programe C pentru a executa o scriere in fisierul specificat prin intermediul descriptorului de fisier deschis.

Comanda `whatis` se foloseste cu sintaxa:

`UNIX> whatis` *cuvint*

si are ca efect cautarea cuvintului specificat in baza de date WHATIS ce contine scurte descrieri despre comenzile `UNIX` si functiile de biblioteca C, si afisarea rezultatului cautarii (numai potrivirile exacte ale cuvintului cautat sunt afisate).

Exemple:

- `UNIX> man whatis`

  Efect: afiseaza pagina de manual referitoare la comanda `whatis`.

- `UNIX> whatis write`

  Efect: afiseaza urmatoarele informatii despre cele doua pagini de manual referitoare la `write`:

  ```
  write   (1)  - send a message to another user
  write   (2)  - write to a file descriptor
  ```

Comanda `apropos` se foloseste cu sintaxa:

`UNIX> apropos` *cuvint*

si are ca efect cautarea cuvintului specificat in baza de date WHATIS ce contine scurte descrieri despre comenzile `UNIX` si functiile de biblioteca C, si afisarea rezultatului cautarii (sunt afisate toate potrivirile, nu doar cele exacte, ale cuvintului cautat).

Exemple:

- `UNIX> man apropos`

  Efect: afiseaza pagina de manual referitoare la comanda `apropos`.

- `UNIX> apropos write`

  Efect: afiseaza o lista (destul de mare) cu toate paginile de manual ce contin cuvintul `write`, fie ca nume de comanda sau functie de biblioteca, fie deoarece apare in descrierea vreunei comenzi sau functii.

Comanda `info` se foloseste cu sintaxa:

`UNIX> info [`*optiuni*`] [`*cuvint* `...]`

si are ca efect afisarea *documentatiei in format* INFO pentru cuvintul sau cuvintele specificate.
Documentatia in format INFO este o alternativa la paginile de manual furnizate de comanda `man`. Formatul INFO este o ierarhie arborescenta de documente, ce contin hiper-legaturi (*i.e.*, *cross*-referinte) intre ele, asemanator cu documentele `HTML`.

Exemple:

- `UNIX> man info`

  Efect: afiseaza pagina de manual referitoare la comanda `info`.

- `UNIX> info`

  Efect: afiseaza radacina documentatiei in format INFO, din care se poate naviga in toate documentele.

- `UNIX> info write`

  Efect: afiseaza documentatia despre comanda `write`.

Atenţie: o recomandare importantă, pe care va sfatuiesc s-o urmati:

- Folositi *help*-ul furnizat de comanda `man` pentru a afla in mod amanuntit pentru ce se folosesc si cum se folosesc toate comenzile pe care le veti intilni pe parcursul acestui manual si la laboratoarele de `UNIX`.
- De asemenea, exersati diverse exemple pentru fiecare comanda pentru a intelege bine ce face si cum functioneaza acea comanda.

2.1.3 Editoare de texte

Exista mai multe editoare de fisiere text sub `UNIX`, printre care: `joe`, `pico`, `vi`, `vim`, `ed`, `mcedit`, `emacs`, `tex/latex`, ş.a.

Editorul JOE :

- Apel - se apeleaza cu comanda:

  `UNIX> joe [`*fisier*`]`

- Comenzi ale editorului JOE (sunt asemanatoare cu cele din editorul Borland Turbo Pascal):
  - Ctrl+K,H = *help* cu comenzile lui
  - Ctrl+K,B = inceput selectie bloc de text
  - Ctrl+K,K = sfirsit selectie bloc de text
  - Ctrl+K,C = copie blocul de text anterior selectat
  - Ctrl+K,M = muta blocul de text anterior selectat
  - Ctrl+K,Y = sterge blocul de text anterior selectat
  - Ctrl+K,W = scrie, in fisierul specificat, blocul de text anterior selectat
  - Ctrl+K,R = adauga, pe pozitia cursorului, continutul fisierului specificat
  - Ctrl+C = iesire fara a salva modificarile facute in fisierul editat
  - Ctrl+K,X = iesire cu salvarea modificarilor facute in fisierul editat

– ... ş.a.

Editorul PICO :

- Apel - se apeleaza cu comanda:

  `UNIX>` `pico` [*fisier*]

- Comenzi ale editorului PICO: consultati pagina de manual despre el (cu comanda `man pico`).

Editorul MCEDIT (este editorul intern din *file-manager*-ul `mc`) :

- Apel - se apeleaza cu comanda:

  `UNIX>` `mcedit` [*fisier*]

- Comenzi ale editorului MCEDIT: consultati pagina de manual despre el (cu comanda `man mcedit`).

Editorul EMACS : este un editor puternic, ce face parte din proiectul GNU, cu facilitati de mediu integrat de programare (poate apela compilatorul GNU C si depanatorul GNU).

Editorul TEX/LATEX : este un editor (de fapt, un mediu de lucru) ce permite tehnoredactarea de documente stiintifice in limbajul TEX/LATEX.

2.1.4 Compilatoare, depanatoare, ş.a.

Sub `UNIX` exista compilatoare si interpretoare pentru majoritatea limbajelor de programare existente, mai noi sau mai vechi: C, C++, Pascal, Fortran, Java, si multe altele. Dintre acestea, istoria evolutiei C-ului s-a impletit strins cu cea a `UNIX`-ului, dupa cum am discutat in primul capitol.

Compilatorul GNU C pentru limbajul C poate fi apelat cu comanda:

`UNIX>` `gcc` *sursa.c* `[-o` *executabil*`]`

care realizeaza compilarea (inclusiv *link*-editarea codului obiect) fisierului sursa specificat (*atenţie*: este obligatorie extensia .c) in fisierul executabil cu numele specificat prin optiunea `-o` (in lipsa ei, executabilul se numeste `a.out`). Pentru programe C++, compilarea se face cu comanda:

`UNIX>` `g++` *sursa.cpp* `[-o` *executabil*`]`

Ca mediu integrat de dezvoltare se poate folosi editorul `emacs`, de care am amintit mai devreme, iar pentru depanarea programelor se poate utiliza depanatorul GNU DeBugger, ce se apeleaza cu comanda `gdb`.

Vom reveni cu mai multe amanunte cind vom trata programarea concurenta in limbajul C pentru `UNIX`, in partea II a acestui manual.

2.1.5 Comenzi pentru lucrul cu fişiere şi directoare

Sunt numeroase comenzile `UNIX` care lucrează cu fişiere şi directoare. Pe acestea le vom prezenta în secţiunea următoare, dedicată sistemului de fişiere `UNIX`, după ce vom trata generalităţile legate de acesta.

2.1.6 Comenzi ce oferă diverse informaţii

1. **Comenzi ce ofera informatii despre utilizatori:**

   - informatii despre utilizatorii conectati la sistem, in diverse formate:
     - `users` = afiseaza numele utilizatorilor conectati la sistem;
     - `who` = afiseaza utilizatorii conectati la sistem si numele terminalelor;
     - `rwho` = afiseaza utilizatorii si terminalele conectate la sistem de la distanta;
     - `w` = afiseaza utilizatorii conectati la sistem, numele terminalelor, procesul curent executat in foreground pentru fiecare terminal, ş.a.;
     - `whoami` = afiseaza numele utilizatorului curent;
     - `who am i` = afiseaza numele calculatorului, numele utilizatorului curent, numele statiei, data si ora logarii, ş.a.
   - informatii personale despre un utilizator (nume real, adresa, *last login*, ş.a.):
     - `finger`
   - informatii de identificare despre un utilizator (`UID`-ul, `GID`-ul, alte grupuri de care apartine, ş.a.):
     - `id`

2. **Comenzi ce ofera informatii despre terminale:**

- pentru aflarea terminalului la care sunteti conectat:
  - `tty`

  Observaţie: fiecare sesiune de lucru (*i.e.*, conexiune la sistemul `UNIX` respectiv) are asociat un *terminal de control*, care este responsabil cu preluarea datelor de intrare generate de tastatură şi cu afişarea datelor de ieşire pe ecranul monitorului.
- pentru aflarea/schimbarea diferitelor caracteristici (ca de exemplu: viteza de transfer, secventele escape, tastele de intrerupere, ş.a.) ale terminalului asociat sesiunii de lucru:
  - `stty`

3. **Comenzi ce ofera informatii despre data, timp, ş.a.:**

- informatii despre data, timp, etc., in diverse formate:
  - `date` = afiseaza data si ora curente;
  - `cal` = afiseaza calendarul lunii curente, sau a anului specificat.
- informatii despre momentul cind a fost ultima oara pornit (*boot*-at) un calculator:
  - `uptime` = timpul ultimei porniri a calculatorului local;
  - `ruptime` = timpul ultimei porniri a calculatoarelor din retea.

4. **Comenzi ce ofera informatii despre procesele din sistem** (in diverse formate)**:**

- comanda *process status*:
  - `ps` = afiseaza informatii despre procesele existente in sistem (`PID`-ul, starea procesului, proprietarul procesului, cita memorie ocupa, cit timp procesor consuma, ş.a.);

  Dintre optiunile mai importante amintim:
  - `-a` : informatiile se referă la procesele tuturor utilizatorilor;
  - `-g` : informatiile se referă la toate procese unui grup;
  - `-l` : format lung (mai multe cîmpuri la afişare);
  - `-tx` : informaţiile se referă la procesele terminalului specificat.

  Cîmpurile afişate se referă la:
  - `PID` : identificatorul procesului;
  - `TT` : terminalul de control al procesului;
  - `TIME` : durata de executie a procesului;
  - `STAT` : starea procesului;
  - `CMD` : linia de comanda prin care a fost lansat acel proces;

  ş.a. (in functie de optiunile specificate).

Exemplu. Iata citeva exemple de optiuni ale comenzii *process status*:

`UNIX> ps`

Efect: afiseaza informatiile doar despre procesele utilizatorului curent, ce sunt rulate in *foreground*.

`UNIX> ps -ux`

Efect: afiseaza informatiile despre toate procesele utilizatorului curent, inclusiv cele ce sunt rulate in *background* sau fara terminal de control.

`UNIX> ps -aux`

Efect: afiseaza informatiile despre toate procesele din sistem, ale tuturor utilizatorilor.

- comanda pentru afisarea *job*-urilor aflate in lucru:
  - `jobs`
- comenzi pentru planificarea lansarii unor *job*-uri la anumite ore:
  - `at` = planifica lansarea unei comenzi la o anumita ora (rezultatele se primesc in mail);
  - `atq` = afiseaza lista comenzilor planificate (aflate in coada de asteptare);
  - `atrm` = stergerea din coada de asteptare a unor comenzi planificate anterior.
- alte comenzi referitoare la procese:
  - `times` = afiseaza timpii de executie ai proceselor din sistem;
  - `nice` = permite stabilirea prioritatii de executie a unui proces;
  - `nohup` = *no hang-up*: permite continuarea executiei proceselor ce folosesc intrarea/iesirea standard, si dupa incheierea sesiunii de lucru (prin deconectare de la sistem cu `logout`).

2.1.7 Alte categorii de comenzi

1. **Comenzi pentru conectarea la/deconectarea de la un calculator `UNIX`:**

- pentru *login* (*i.e.*, operatia de conectare):
  - `login` (*login* numai de la terminale locale);
  - `rlogin` (*login* de pe alte calculatoare, cu acelasi tip de `UNIX`);
  - `telnet` (*login* de pe alte calculatoare, conexiunea fiind prin protocolul necriptat TELNET);
  - `ssh` (*login* de pe alte calculatoare, conexiunea fiind prin protocolul criptat SSH).
- pentru *logout* (*i.e.*, operatia de deconectare):
  - `logout`;
  - `exit` (posibil numai din *shell*-ul de *login*).

2. **Comenzi pentru schimbarea datelor unui cont de utilizator:**

- pentru schimbarea parolei:
  - `passwd`
- pentru schimbarea *shell*-ului implicit (*i.e.*, *shell*-ul de *login*):
  - `chsh`

- pentru schimbarea "vîrstei" contului (*i.e.*, a perioadei după care expiră parola):
  - `chage`

3. **Comenzi pentru scrierea de mesaje:**

- pentru afisarea unui mesaj pe ecran (la fel ca in `MS-DOS`):
  - `echo`
- pentru trimiterea unui mesaj altui utilizator conectat la sistem:
  - `write`
- pentru activarea/dezactivarea primirii mesajelor trimise cu comanda `write` de alti utilizatori:
  - `mesg [y|n]`
- pentru stabilirea unei ferestre de comunicare de mesaje intre diferiti utilizatori conectati la sistem:
  - `talk` (pentru comunicare intre doi utilizatori)
  - `ytalk` (pentru comunicare intre mai multi utilizatori)

Exemplu. Iata citeva exemple de folosire a acestor comenzi:

`UNIX>` `echo` *mesaj*

Efect: afiseaza pe ecran mesajul specificat.

`UNIX>` `write` *user* `[`*terminal*`]` `ENTER`
Prima linie a mesajului `ENTER`
A doua linie a mesajului `ENTER`
`.....`
Ultima linie a mesajului `ENTER`
`^D` (tastele CTRL + D, ce seminifica caracterul EOF)

Efect: afiseaza mesajul specificat pe ecranul utilizatorului specificat, eventual in sesiunea de lucru specificata prin *terminal* (lucru util daca acel utilizator are deschise mai multe sesiuni de lucru in momentul respectiv).

`UNIX> mesg n`

Efect: dezactiveaza afisarea pe ecran a mesajelor trimise ulterior cu comanda `write` de catre alti utilizatori (dezactivarea este utila atunci cind afisarea acelor mesaje deranjeaza utilizatorul respectiv).

4. **Comenzi pentru arhivare/comprimare/codificare de fisiere:**

- pentru arhivarea fisierelor:
  - `tar`
- pentru comprimarea/decomprimarea unui fisier:
  - `gzip / gunzip`
  - `compress / uncompress`
- pentru codificarea/decodificarea unui fisier:
  - `encode / decode`

Exemplu. Iata citeva exemple de utilizare a comenzilor de arhivare:

`UNIX> tar cf arhiva.tar ~/work/proiect`

Efect: arhiveaza in arhiva `arhiva.tar` continutul (recursiv al) directorului `~/work/proiect`.

`UNIX> tar xf arhiva.tar`

Efect: dezarhiveaza arhiva `arhiva.tar`.

`UNIX> gzip arhiva.tar`

Efect: comprima fisierul `arhiva.tar`, producind fisierul `arhiva.tar.gz`.

`UNIX> gunzip arhiva.tar.gz`

Efect: decomprima fisierul `arhiva.tar.gz`, producind fisierul `arhiva.tar`.

`UNIX> tar zcf arhiva.tar.gz ~/work/proiect`

Efect: arhiveaza si comprima cu gzip in arhiva `arhiva.tar.gz` continutul (recursiv al) directorului `~/work/proiect`.

`UNIX> tar zxf arhiva.tar.gz`

Efect: dezarhiveaza si decomprima arhiva `arhiva.tar.gz`.

5. **Comenzi (programe) pentru diferite protocoale** INTERNET**:**

- pentru posta electronica:
  – `mail` (utilitar in linie de comanda)
  – `pine` (utilitar in mod text)
- pentru transferul de fisiere prin retea folosind protocolul FTP:
  – `ftp` = un program client de FTP, in linie de comanda
  Observatie: FTP-ul este un protocol pentru transferul de fisiere intre doua calculatoare legate in retea, ca de exemplu oricare doua calculatoare legate la INTERNET.
  – `scp` = un program client de SCP (= *Secure Copy Protocol*, este un FTP criptat), in linie de comanda. Sub `MS-Windows` este disponibila si o varianta grafica – programul `WinSCP`.
  Observatie: este recomandabil de folosit SCP-ul in locul clientilor clasici de FTP, deoarece FTP-ul nu este un protocol criptat.
- pentru transferul de pagini web din WWW folosind protocolul HTTP:
  – `lynx` = un browser WWW (*i.e.*, un program client de WWW), in mod text
  – `netscape` = un browser WWW, in mod grafic (necesita X WINDOWS)
  Observatie: WWW-ul (abreviere ce provine de la *World Wide Web*) este un protocol pentru gestionarea informatiilor in INTERNET, avind la baza protocolul HTTP si limbajul `HTML` (= *Hyper-Text Markup Language*).
- pentru regasirea informatiilor personale, folosind protocolul FINGER:
  – `finger` = un program client de FINGER, in linie de comanda
  Observatie: FINGER este un protocol pentru regasirea informatiilor personale (nume real, adresa, *last login*, *last mail read*, ş.a.) despre un utilizator de pe un calculator din reteaua INTERNET.

- pentru rezolvarea numelor simbolice in adrese `IP` sau invers, folosind serviciul DNS:
  – `nslookup` = un program in linie de comanda
  Observatie: Fiecare calculator conectat la INTERNET are asociata o adresa `IP` unica, care este un sir de 4 numere (4 octeti). Deoarece astfel de adrese sunt greu de retinut, s-au asociat si nume simbolice. Transformarea adresei `IP` a unui calculator in numele simbolic asociat acelui calculator se face prin protocolul ARP, iar transformarea inversa se face prin protocolul RARP, in ambele situatii utilizindu-se serviciul DNS (abreviere ce provine de la *Domain Name Service*).

Exemplu. Iata citeva exemple de folosire a acestor comenzi:

`UNIX> finger so`

Efect: afiseaza informatiile personale despre contul utilizator `so` de pe calculatorul local, cel pe care lucrati, adica `fenrir`.

`UNIX> finger vidrascu@thor.infoiasi.ro`

Efect: afiseaza informatiile personale despre contul utilizator `vidrascu` de pe calculatorul `thor`.

`UNIX> nslookup fenrir.infoiasi.ro`

`193.231.30.197`

Efect: afiseaza adresa `IP` a calculatorului `fenrir`.

`UNIX> nslookup 193.231.30.197`

`fenrir.info.uaic.ro`
`fenrir.infoiasi.ro`

Efect: afiseaza numele simbolice asociate calculatorului cu adresa `IP` `193.231.30.197`.

6. **Comenzi pentru cautarea de *pattern*-uri (sabloane) in fisiere:**

- pentru cautarea unui *pattern* (*i.e.*, o expresie regulata) intr-un fisier sau grup de fisiere si afisarea tuturor liniilor de text ce contin acel *pattern*:
  – `grep`

Exemplu. Comanda

`UNIX> grep vidrascu /etc/passwd`

are ca efect afisarea liniei, din fisierul `/etc/passwd`, care contine informatiile despre contul `vidrascu`.

- comenzi (mai exact, limbaje interpretate) pentru procesarea de *pattern*-uri si constructia de comenzi `UNIX`:
  – `awk`
  – `sed`
  – `perl`
- comanda *filtru* de selectare a anumitor portiuni din fiecare linie de text a fisierului specificat:
  – `cut`

- comanda *filtru* de selectare a liniilor de text unice din fisierul specificat (mai exact, efectul ei consta in pastrarea unui singur exemplar de linie de text din fiecare grup de linii consecutive ce sunt identice ca si continut):
  – `uniq`

7. **Alte comenzi:**

- comanda interna prin care se definesc/sterg *alias*-uri (*i.e.*, pseudo-comenzi), prin redenumirea vechilor comenzi sau inlantuirea mai multor comenzi:
  – `alias / unalias`

Exemplu. Efectul comenzilor urmatoare

```
UNIX> alias ll='ls -Al'
UNIX> ll tmpdir
```

consta in: prima comanda defineste *alias*-ul cu numele `ll`, iar a doua executa *alias*-ul cu numele `ll` si cu parametrii specificati, adica, in acest caz, executa comanda: `ls -Al tmpdir`.

- comanda ce copie intrarea standard (`stdin`) in iesirea standard (`stdout`) si in fisierul specificat ca parametru:
  – `tee`

Exemplu. Efectul comenzii

`UNIX>` *my_program* `| tee results.txt`

consta in: mesajele scrise pe `stdout` in timpul executiei programului specificat sunt tiparite si in fisierul `results.txt`, putind fi deci consultate dupa terminarea executiei programului.
(Aceasta facilitate este utila pentru depanarea programelor.)

- comanda pentru executia iterativa a unei comenzi specificate pentru un set de diferite linii de apel (i.e., parametri de apel pentru acea comanda):
  – `xargs`
- comanda pentru sortarea liniilor de text dintr-un fişier:
  – `sort`
- comanda pentru operaţia de *join* pe un cîmp comun a liniilor din două fişiere (este similară operaţiei de *join* a două tabele relaţionale pe care o cunoaşteţi de la disciplina *Baze de date*):
  – `join`

Observaţie: mai sunt multe comenzi, ce nu au fost amintite mai sus; o parte dintre ele le veti intilni pe parcursul sectiunilor urmatoare.

2.1.8 *Troubleshooting* (Cum să procedaţi dacă se "blochează" o comandă)

Sa presupunem ca in timpul executiei unei comenzi (un program de sistem sau un program scris de dumneavoastra), aveti impresia ca acesta s-a blocat (nu mai apare nici un mesaj pe ecran, desi ar fi trebuit, etc.).

Intr-o asemenea situatie nu trebuie sa intrati in panica, ci, pentru a opri acest program, deci pentru a obtine din nou controlul asupra prompterului *shell*-ului, urmati urmatorii pasi in ordinea in care sunt prezentati, oprindu-va la pasul la care ati reusit sa deblocati programul (adica sa apara prompterul):

1. Mai intii, asteptati un timp rezonabil, poate totusi programul nu este blocat, ci doar ocupat cu calcule laborioase.
   Daca totusi nu apare prompterul, atunci:

2. Apasati (simultan) tastele `CTRL` + `C`. Aceasta determina trimiterea semnalului de intrerupere `SIGINT` programului respectiv.
   Daca totusi nu apare prompterul, atunci:

3. Apasati (simultan) tastele `CTRL` + \. Aceasta determina trimiterea semnalului de terminare `SIGQUIT` programului respectiv.
   Daca totusi nu apare prompterul, atunci:

4. Apasati (simultan) tastele `CTRL` + `Z`. Aceasta determina suspendarea programului respectiv (*i.e.*, acel program este trecut in starea *SUSPENDED*) si afisarea prompterului.
   Mai departe, pentru a opri acel program (el este doar suspendat, nu si terminat), procedati in felul urmator. Cu comanda

   `UNIX> ps`

   aflati `PID`-ul acelui program, iar apoi dati comanda

   `UNIX> kill -9` *pid*

   unde *pid* este `PID`-ul aflat anterior. Ca urmare a acestei comenzi procesul in cauza (*i.e.*, programul blocat) este omorit (*i.e.*, terminat fortat).

In acest moment sigur ati scapat de acel program ce se blocase si aveti din nou controlul asupra prompterului *shell*-ului.

2.2 Sisteme de fişiere UNIX

1. **Introducere**
2. **Structura arborescentă a sistemului de fişiere**
3. **Montarea volumelor în structura arborescentă**
4. **Protecţia fişierelor prin drepturi de acces**
5. **Comenzi de bază în lucrul cu fişiere şi directoare**

2.2.1 Introducere

In sistemul de operare UNIX, datele si programele sunt pastrate in fisiere identificate prin nume. Numele fisierelor pot avea pina la 255 caractere si pot contine oricite caractere '.' (nu sunt impartite sub forma 8.3, *nume.extensie*, ca in sistemul de fisiere FAT din MS-DOS sau MS-Windows), singurele restrictii fiind nefolosirea caracterelor neprintabile, sau a caracterelor NULL, '/', si a spatiilor albe TAB si SPACE.

Important de retinut: spre deosebire de MS-Windows (adica de sistemele de fisiere FAT si FAT32), numele fisierelor in UNIX sunt *case-sensitive*, adica se face distinctie intre majuscule si minuscule.

UNIX nu impune nici o conventie privitoare la numirea fisierelor, dar exista sufixe utilizate in mod standard, cum ar fi spre exemplu:
• .c si .h pentru fisiere sursa in limbajul C;
• .cpp si .h pentru fisiere sursa in limbajul C++;
• .tar pentru arhive tar;
• .gz pentru arhive gzip;
• .tar.gz sau .tgz pentru arhive tar gzip;
ş.a.
Programele executabile si *script*-urile nu au in general extensie.

Sistemul de fisiere se pastreaza pe suporturi magnetice: HDD (*hard-disk*), FDD (*floppy-disk*), benzi magnetice, ş.a., sau pe suporturi optice: CD-uri (in orice format: CD-ROM, CD-R, CD-RW), DVD-uri, discuri magneto-optice, ş.a., sau poate fi emulat în memoria internă (*RAM-disk*), etc. Sistemul de fisiere poate fi păstrat doar local (*i.e.*, pe *hard-disk*-ul propriu) sau şi distribuit (de exemplu prin NFS). Orice *hard-disk* poate contine mai multe *partitii*, dintre care unele pot fi partitii de UNIX, eventual coabitînd cu partitii de MS-DOS, Novell, Windows, sau alte sisteme.

Fisierele in UNIX pot fi de urmatoarele tipuri:

- normale (ordinare);
- directoare (cataloage);
- *link*-uri (legaturi simbolice): sunt un fel de *alias*-uri pentru alte fisiere;
- fisiere speciale, in mod bloc sau in mod caracter: sunt drivere de periferice, ş.a.;
- fisiere de tip *fifo*: sunt folosite pentru comunicatia intre procese, rulate pe acelasi sistem;
- fisiere de tip *socket*: sunt folosite pentru comunicatia prin retea intre procese, rulate pe sisteme diferite.

Tipul fisierelor dintr-un director se poate afla cu comanda `ls -al`, care afiseaza toate fisierele din directorul specificat, cu diverse informatii despre ele (tipul, drepturile de acces, proprietarul, grupul proprietar, lungimea, data ultimei modificari, etc.). Tipul fisierelor este indicat de primul caracter din prima coloana a listingului afisat de comanda `ls -al`, acest caracter putind fi:

- '-' pentru fisier normal;
- 'd' pentru director;
- 'l' pentru legatura simbolica;
- 'b' sau 'c' pentru fisier special in mod bloc sau, respectiv, in mod caracter;
- 'p' pentru fisier de tip *fifo*;
- 's' pentru fisier de tip *socket*.

Exemplu. Comanda

```
UNIX> ls -al ~so
```

are ca efect: se afiseaza listingul fisierelor din directorul ~so, rezultatul aratind cam in felul urmator:

```
total 70
drwx------   6 so       users    1024 May 10 11:19 ./
drwxr-xr-x  56 root     root     1024 May  7 10:25 ../
-rw-r--r--   1 so       users     579 Apr 29 11:28 .addressbook
-rw-r--r--   1 so       users    1821 Apr 29 11:30 .addressbook.lu
-rw-r--r--   1 so       users    3597 May 10 11:19 .bash_history
drwxr-xr-x   2 so       users    1024 Apr 29 11:42 html/
drwx------   2 so       users    1024 Apr 29 11:42 mail/
-rw-r--r--   1 so       users   32640 Apr 29 11:27 fis.txt
-rw-rw-r--   1 so       users   13594 Apr 29 11:14 fis.dat
```

```
lrwxrwxrwx   1 so       users     579 Apr 29 11:28 labs -> /home/so/html/labs -rw-r--r--
1 so       users      65 Apr 29 11:13 prg1.c
prw-------   1 so       users       0 Dec 12 11:13 fifo1
```

2.2.2 Structura arborescentă a sistemului de fişiere

Sistemul de fisiere în UNIX este *ierarhizat* (arborescent), adică este ca un arbore, la fel ca in MS-DOS sau MS-Windows: directoare ce contin subdirectoare si fisiere propriu-zise. Dar cu deosebirea ca în UNIX avem un arbore ce are o singura radacina, referita prin "/" (nu avem mai multe unitati de discuri logice C:, D:, ...), iar ca separator pentru caile de subdirectoare se utilizeaza caracterul '/', in locul caracterului '\' folosit in MS-DOS sau MS-Windows.

Fisierele pot fi accesate (specificate) fie relativ la radacina "/" sistemului de fisiere (*i.e.*, specificare prin **cale absoluta**), fie relativ la directorul curent de lucru (*i.e.*, specificare prin **cale relativa la directorul curent**).

La fel ca in MS-DOS si MS-Windows, in fiecare director exista doua nume predefinite: "**.**", care reprezinta directorul curent, si "**..**", care reprezinta directorul parinte al directorului curent.

Dupa cum s-a mentionat, in UNIX fisierele sunt organizate intr-o structura ierarhica arborescenta. O astfel de structura permite o organizare eficienta si o grupare logica a fisierelor. O structura tipica de sistem de fisiere in UNIX este prezentata mai jos:

```
      /bin : cd, cat, ls, ...   (fisiere executabile - comenzi de sistem)
      /dev : lp, hdd, fdd, ...  (fisiere speciale asociate perifericelor)
root: /etc : mount, passwd, ... (fisiere de configurare si de initializare)
      /home: studs, staff, ...  (directoarele home ale utilizatorilor)
      /usr : bin, lib, ...      (aplicatii)
      /lib : ...                (biblioteci dinamice)
      /tmp : ...                (director temporar)
       ...                       si altele
```

Navigarea in structura arborescenta se poate face cu comanda pentru schimbarea directorului curent, care este comanda interna cd (la fel ca in MS-DOS), iar aflarea directorului curent se poate face cu comanda pwd.

Exemplu.

```
UNIX> pwd
/home/vidrascu

UNIX> cd /home/vidrascu/mail
UNIX> pwd
/home/vidrascu/mail
```

Fiecare utilizator are un director special numit director propriu (sau director *home*), in care se face intrarea dupa operatia de *login*. Adica imediat dupa *login* directorul curent de lucru va fi acest director *home*. De obicei, numele complet (*i.e.*, calea absoluta) a acestui director este `/home/`*`username`*, unde *`username`* este numele de cont `UNIX` al utilizatorului respectiv.

In afara de referirea la fisiere prin cale relativa la directorul curent sau prin cale absoluta, mai exista si referirea prin **cale relativa la directorul** *home*. Spre exemplu: ∼*username*`/dir1/dir2/.../file` , ceea ce este echivalent în cazul de faţă cu: `/home/`*username*`/dir1/dir2/.../file`.

Atunci cind utilizatorul se refera la propriul director *home*, in locul formei lungi de referire ∼*username*`/.../file` poate folosi forma prescurtata ∼`/.../file`. De exemplu, comanda:

```
UNIX> cd ∼/mail
```

va schimba directorul curent in subdirectorul `mail` al directorului *home* al utilizatorului care tasteaza aceasta comanda.

Revenirea in directorul *home* propriu din directorul curent se poate face simplu introducind comanda `cd` fara argumente.

2.2.3 Montarea volumelor în structura arborescentă

Sistemele de fisiere `UNIX` se pot afla pe mai multe dispozitive fizice sau in mai multe partitii ale aceluiasi disc fizic. Fiecare dintre ele are un director root "`/`" si poate fi navigat prin incarcarea sistemului de operare de pe dispozitivul respectiv.

Daca totusi sistemul de fisiere de pe un dispozitiv trebuie folosit fara incarcarea sistemului de operare de pe acel dispozitiv, exista solutia de **a monta** structura arborescenta de fisiere de pe acel dispozitiv in structura dispozitivului de pe care s-a incarcat sistemul de operare. Astfel, spre exemplu, fisierele de pe o dischetă sau de pe un CD-ROM trebuie montate pentru a putea fi accesibile.

Spre deosebire de sistemele `MS-DOS` sau `MS-Windows`, unde fisierele de pe diverse dispozitive se puteau accesa prefixate de numele discurilor logice (`A:`, `C:`, `D:`, etc.), in `UNIX` ele pot fi folosite prin *montarea* în prealabil a structurilor respective in structura arborescenta a sistemului de pe care s-a incarcat sistemul de operare, sistem al carui radacina este "`/`"-ul (echivalentul lui `C:\` din `MS-DOS` sau `MS-Windows`). Mai exact, montarea se face intr-un anumit subdirector al sistemului de fisiere "`/`", ca si cum acel subdirector ar fi identificat cu radacina structurii ce se monteaza.

Montarea se face cu comanda `mount` (ce poate fi executata doar de utilizatorul `root`), iar apoi accesarea se face cu ajutorul directorului in care s-a facut montarea.
Observatie: prin operatia de montare se inhiba accesul la eventualele fisiere ce ar exista in directorul care este punctul de montare, deci acest director trebuie sa fie de preferinta gol.

Exemplu. Comanda urmatoare monteaza discheta din prima unitate de dischetă (echivalentul discului logic `A:` din `MS-DOS`):

```
UNIX> mount /dev/fd0 /mnt/floppy
```

iar în urma execuţiei sale fisierele de pe discheta vor fi "vizibile" in subdirectorul `/mnt/floppy`. Ca urmare, putem, de exemplu, naviga prin subdirectoarele de pe discheta:

```
UNIX> cd /mnt/floppy/progs
```

Efect: noul director curent va fi subdirectorul `progs` de pe discheta.

Prin optiunea `-r` a comenzii `mount` se poate proteja la scriere noua structura atasata (adica aceasta va fi montata in modul *read-only*). Optiunea `-t` este folosita pentru a specifica tipul de sistem de fisiere ce se monteaza.

Operatia inversa montarii, numita *demontare*, se face cu comanda `umount` (ce poate fi executata, de asemenea, doar de utilizatorul `root`).

Exemplu. Comanda

```
UNIX> umount /mnt/floppy
```

are ca efect: se demonteaza discheta montata anterior in punctul `/mnt/floppy`. In continuare, vor fi din nou accesibile fisierele ce existau eventual in acest director (daca nu era gol inainte de montare).

2.2.4 Protecţia fişierelor prin drepturi de acces

Dupa cum am discutat in primul capitol, sistemul `UNIX` organizeaza utilizatorii in *grupuri de utilizatori*.
Fiecare **grup** are asociat un nume (exemplu: `studs`, `profs`, `admins`, etc.) si un numar unic de identificare a grupului, numit `GID`.
Fiecare **utilizator** face parte dintr-un anumit grup si are asociat un nume (i.e., *username* = numele contului respectiv), precum si doua numere:
• `UID`, identificatorul utilizatorului, care este un numar unic de identificare a utilizatorului;

• `GID`, identificatorul grupului din care face parte utilizatorul.
Aceste `ID`-uri ii sunt asociate atunci cind se creeaza contul acelui utilizator.

Fiecare fisier are asociat ca **proprietar**, un anumit utilizator (care, de obicei, este utilizatorul ce a creat acel fisier, dar proprietarul poate fi schimbat). Grupul acelui utilizator este **grupul proprietar** al fisierului.

Utilizatorii sunt clasificati in trei categorii in functie de relatia fata de un fisier:
• proprietarul fisierului (*owner*);
• colegii de grup ai proprietarului (*group*);
• ceilalti utilizatori (*others*).

Fiecare fisier are asociate trei tipuri de **drepturi de acces** pentru fiecare dintre cele trei categorii de utilizatori de mai sus:
• **r** (*read*) : drept de citire a fisierului;
• **w** (*write*) : drept de scriere a fisierului;
• **x** (*execute*) : drept de executie a fisierului.

In cazul directoarelor, drepturile au o semnificatie putin modificata:
• **r** (*read*) : drept de citire a continutului directorului (*i.e.*, drept de aflare a numelor fisierelor din director);
• **w** (*write*) : drept de scriere a continutului directorului (*i.e.*, drept de adaugare/stergere de fisiere din director);
• **x** (*execute*) : drept de inspectare a continutului directorului (*i.e.*, drept de acces la fisierele din director).

Pe linga cele 3 x 3 = 9 drepturi pentru fisiere (si directoare) amintite mai sus, mai exista inca trei drepturi suplimentare, ce au sens doar pentru fisierele executabile, si anume:
• **s** (*setuid bit*) : pe durata de executie a fisierului, proprietarul efectiv al procesului va fi proprietarul fisierului, si nu utilizatorul care il executa;
• **s** (*setgid bit*) : la fel, pentru grupul proprietar efectiv;
• **t** (*sticky bit*) : imaginea text (*i.e.*, codul programului) a acelui fisier executabil este salvata pe partitia de *swap* pentru a fi incarcata mai repede la executia acelui program.

Pentru modificarea drepturilor de acces, respectiv a proprietarului si grupului proprietar ale unui fisier sunt disponibile urmatoarele comenzi: `chmod`, `chown`, `chgrp`. Ele pot fi folosite numai de catre *superuser* (*i.e.*, utilizatorul `root`) sau de proprietarul fisierului.

Comanda `chmod` este folosita pentru modificarea drepturilor de acces. Pentru specificarea argumentelor ei, se poate folosi fie o notatie simbolica, fie o reprezentare in octal.
In notatia simbolica, utilizatorii sunt identificati prin:
• **u** (*user*) = proprietarul;
• **g** (*group*) = grupul proprietarului, exceptind proprietarul insusi;
• **o** (*others*) = altii (restul utilizatorilor);
• **a** (*all*) = toti utilizatorii,

prin + sau − se specifica adaugarea, respectiv eliminarea de drepturi, iar drepturile **r**,**w**,**x** sunt drepturile mentionate mai sus.

De exemplu, comanda:

```
UNIX> chmod g-rw prg1.c
```

are ca efect: se elimina drepturile de citire si scriere a fisierului `prg1.c` din directorul curent pentru grupul proprietar al fisierului (deci pentru colegii de grup ai proprietarului).

Alt exemplu:

```
UNIX> chmod go+x prg1.exe
```

are ca efect: se adauga dreptul de executie a fisierului `prg1.exe` pentru toti utilizatorii, mai putin proprietarul lui.

Observatie: celelalte drepturi, nespecificate prin notatia simbolica, ramin neschimbate.

Pentru notatia in octal, trebuie avut in vedere faptul ca **r** = 4, **w** = 2, **x** = 1, si pentru fiecare categorie de utilizatori se aduna cifrele corespunzatoare acestor optiuni, rezultind cite o cifra octala pentru fiecare categorie de utilizatori.

De exemplu, comanda:

```
UNIX> chmod 640 prg1.c
```

are ca efect: proprietarul are drept de citire si scriere (dar nu are drept de executie) asupra fisierului `prg1.c`, utilizatorii din grupul lui au doar drept de citire, iar altii nu au nici un drept.

Observatie: spre deosebire de notatia simbolica, prin notatia in octal toate cele 4 x 3 = 12 drepturi ale fisierului sunt modificate, conform specificarii din notatia in octal.

Cind un fisier este creat, ii sunt asociate atit identificatorul proprietarului, cit si cel de grup al procesului care a creat respectivul fisier. Comanda `chown` permite modificarea proprietarului unui fisier, iar comanda `chgrp` permite similar modificarea grupului de care apartine fisierul.

De exemplu, comanda:

```
UNIX> chown vidrascu prg1.c
```

are ca efect: fisierul `prg1.c` din directorul curent va avea pe utilizatorul `vidrascu` ca nou proprietar.

Iar comanda:

```
UNIX> chgrp studs prg1.c
```

are ca efect: fisierul `prg1.c` din directorul curent va avea grupul `studs` ca nou grup proprietar.

2.2.5 Comenzi de bază în lucrul cu fişiere şi directoare

1. **Comenzi pentru crearea/stergerea/listarea unui director:**

   Crearea unui director se face cu comanda `mkdir`, iar stergerea cu comanda `rmdir`. Cind se creaza un director, se insereaza automat in acesta intrarile standard "**.**" si "**..**" (este un director gol). Daca se doreste stergerea unui director, acesta trebuie sa fie gol. Este posibila insa si stergerea recursiva (directorul nu mai trebuie sa fie gol in acest caz).

   Afisarea continutului unui director se poate face cu comanda `dir`, dar cea mai cunoscuta si completa comanda este `ls`. Dintre numeroasele optiuni ale comenzii `ls`, vor fi prezentate in continuare doar cele mai importante:

   - `-l` : format lung (detaliat) de afisare a informatiilor despre continutul directorului;
   - `-t` : continutul directorului este listat sortat dupa data ultimei modificari, cele mai recente fisiere primele;
   - `-d` : se afiseaza informatii despre directoarele propriu-zise date ca parametri, in loc de a se afisa continuturile acestora;
   - `-a` : afisarea inclusiv a fisierelor a caror nume incepe cu punct;
   - `-A` : analog cu optiunea `-a`, doar ca se ignora intrarile standard "**.**" si "**..**".

   Fara optiunea `-a` sau `-A` se vor afisa doar fisierele a caror nume nu incepe cu punct. Motivul: fisierele care incep cu punct sunt, de obicei, fisiere de initializare/configurare de sistem sau a aplicatiilor, si de aceea sunt "mascate" la un listing obisnuit (i.e., un `ls` fara niciuna din aceste optiuni).
   La folosirea optiunii `-l`, pentru fisierele speciale in locul dimensiunii vor fi afisate numarul major si cel minor al dispozitivului fizic asociat.

   Comanda `dir` este similara comenzii `ls` fara optiuni.

   Spre exemplu, daca se da comanda

   ```
   UNIX> ls *
   ```

   vor fi afisate informatii despre continutul directorului curent si a subdirectoarelor din directorul curent.

2. **Comenzi pentru crearea/stergerea fisierelor:**

   Cu ajutorul comenzii generale `mknod` se pot crea orice tip de fisiere (inclusiv fisiere speciale), dar exista si comenzi particulare pentru anumite tipuri de fisiere, ca de

exemplu comanda `mkfifo` care creeaza fisiere de tip *fifo*, sau comanda `mkdir` care creeaza fisiere de tip director.

Comanda `mkfs` poate fi folosita, doar de catre *superuser*, pentru a construi un sistem de fisiere, specificind dimensiunea si dispozitivul.

Stergerea fisierelor se poate face cu comanda `rm`, in limita dreptului de scriere. Optiunea `rm -i` va interoga fiecare stergere, iar optiunea `rm -r` va sterge continutul unui director cu tot cu subdirectoarele aferente chiar daca contin fisiere (atentie deci la folosirea acestei optiuni!).

3. **Comenzi pentru copierea/mutarea/redenumirea fisierelor:**

   Comanda `mv` permite *mutarea* unui fisier sau *redenumirea* lui. Primul parametru specifica sursa, iar al doilea destinatia.

   Spre exemplu, comanda

   `UNIX> mv prg1.c p1.cc`

   are ca efect: fisierul `prg1.c` din directorul curent va fi redenumit in `p1.cc`,

   `UNIX> mv prg1.c src`

   are ca efect: fisierul `prg1.c` din directorul curent va fi mutat in subdirectorul `src` (presupunind ca acesta exista), iar

   `UNIX> mv prg1.c src/p1.cc`

   are ca efect: fisierul `prg1.c` din directorul curent va fi mutat si redenumit in `src/p1.cc`.

   Cu ajutorul comenzii `mv` se pot redenumi si directoare, dar numai in cazul cind acestea apartin de acelasi director parinte. De asemenea, se pot muta si mai multe fisiere deodata, specificind ca destinatie un director existent.

   Comanda `cp` permite *copierea* unui fisier (*i.e.*, crearea unui nou fisier ce este o copie fidela a fisierului initial). La fel ca la `mv`, primul parametru specifica sursa, iar al doilea destinatia. Daca destinatia este un director, se poate face copierea mai multor fisiere deodata.

   Spre exemplu, comanda

   `UNIX> cp prg1.c p1.cc`

   are ca efect: fisierul `prg1.c` din directorul curent va fi copiat in `p1.cc`.

   Comanda `ln` permite crearea unor *pseudonime*, *i.e. alias*-uri de nume (*link*-uri), ale unui fisier.
   Diferenta intre `cp` si `ln` este ca fisierele create prin `cp` sunt independente (o modificare a unuia nu implica si modificarea automata a celuilalt), in timp ce fisierele create cu `ln` refera acelasi fisier fizic.
   Exista doua tipuri de *alias*-uri: *link*-uri *hard* si *link*-uri simbolice (create cu optiunea `ln -i`).

   Spre exemplu, comanda:

   `UNIX> ln prg1.c p1.cc`

are ca efect: creeaza un *link hard*, cu numele `p1.cc`, catre fisierul `prg1.c` din directorul curent, iar comanda

```
UNIX> ls -i prg1.c p1.cc
2156 prg1.c
2156 p1.cc
```

afiseaza numarul *i*-nodului corespunzator celor doua nume de fisiere, si se observa ca ambele refera acelasi *i*-nod (adica acelasi fisier fizic).

Există o restrictie de utilizare a *link*-urilor *hard*: atit fisierul referit, cit si *alias*-ul creat trebuie sa se afle pe acelasi sistem de fisiere fizic (*i.e.*, pe aceeasi partitie), pe cind *link*-urile simbolice nu au aceasta restrictie, datorita faptului ca ele folosesc un mecanism de referire indirecta.

4. **Comenzi de cautare a fisierelor:**

   O comanda des utilizata pentru cautare este comanda `find`. Cu ajutorul acestei comenzi se pot localiza fisiere prin examinarea unei structuri arborescente de directoare, pe baza unor diverse criterii de cautare. Comanda `find` este foarte puternica – ea permite, spre exemplu, executia a diverse comenzi pentru fiecare aparitie gasita.

   Spre exemplu, comanda

   ```
   UNIX> find .  -name p*.* -print
   ./prg1.c
   ./src/p1.cc
   ```

   are ca efect: se vor cauta, incepind din directorul curent, fisierele a caror nume incepe cu 'p' si se vor afisa numele complete ale fisierelor gasite.

   Un alt posibil criteriu de cautare: comanda `find` poate fi folosita pentru a gasi fisierele mai mari sau mai mici fata de o anumita dimensiune:

   ```
   UNIX> find .  -size -10 -print
   ./prg1.c
   ./src/p1.cc
   ./.addressbook
   ./.addressbook.lu
   ./.bash_history
   ./.bash_profile
   ./mail/sent
   ```

   O alta comanda de cautare este comanda `which`. Ea cauta directorul in care se gaseste un fisier specificat, dar, spre deosebire de comanda `find`, il cauta numai in directoarele din variabila de mediu `PATH`.

5. **Comenzi pentru afisarea unui fisier:**

   Comenzile `cat`, `tac`, `more`, `less`, `head`, `tail`, `pg`, `lp`, `od` permit vizualizarea continutului unui fisier, dupa anumite formate.

   Comanda `cat` permite afisarea continutului unui fisier. Ea se poate folosi înlănţuită cu comanda `more` sau cu comanda `less` pentru a afisa ecran cu ecran acel fisier, sau se pot folosi in acest scop direct cele doua comenzi amintite. Iar comanda `tac` este

inversa comenzii `cat`, producind afisarea unui fisier linie cu linie de la sfirsitul lui catre inceputul acestuia.

Comenzile `head` si `tail` produc afisarea primelor, respectiv ultimelor n linii dintr-un fisier (n fiind specificat ca optiune).

Comanda `pg` face afisare cu paginare, `lp` face listare la imprimanta, iar `od` permite obtinerea imaginii memorie a unui fisier.

6. **Comenzi ce ofera diverse informatii despre fisiere:**

   Obtinerea de diverse informatii despre continutul unui fisier se poate face cu comenzile:

   - `file` : incearca sa determine tipul unui fisier;
   - `wc` : precizeaza numarul de caractere, cuvinte si linii dintr-un fisier;
   - `sum` : calculeaza suma de control a unui fisier.

   Spre exemplu:

```
UNIX> file p*.*
prg1.c:  c program text
p1.cc:  c program text
```

   Compararea a doua fisiere se poate face cu comenzile:

   - `cmp` : face o comparare binara a doua fisiere;
   - `comm` : afiseaza liniile comune a doua fisiere text;
   - `diff` : afiseaza liniile diferite a doua fisiere text.

   Obtinerea de informatii despre sistemul de fisiere se poate face cu comenzile:

   - `df` : afiseaza cit spatiu liber mai exista pe fiecare partitie;
   - `du` : afiseaza cite blocuri ocupa fiecare fisier si suma blocurilor din directorul respectiv.

   Spre exemplu, comanda:

```
UNIX> du -ks ~
```

   are ca efect: se va afisa, in kB (*kilo-bytes*), cit spatiu ocupa (in intregime, cu tot cu subdirectoare) directorul *home* al utilizatorului ce da aceasta comanda.

7. **Comenzi pentru cautarea unui sablon intr-un fisier:**

   Comanda `grep` este o comanda puternica, ce permite cautarea unui *pattern* (sablon), adica a unei expresii regulate, intr-unul sau mai multe fisiere specificate, si afisarea tuturor liniilor de text ce contin acel *pattern*.

   Spre exemplu, comanda

```
UNIX> grep vidrascu /etc/passwd
```

va afisa linia cu informatii corespunzatoare contului `vidrascu` din fisierul `/etc/passwd`, comanda

```
UNIX> grep -c main p*.*
prg1.c:  1
p1.cc:  1
```

va afişa numărul de linii pe care apare cuvîntul "main" pentru fiecare fişier ce se potriveşte specificatorului `p*.*` (opţiunea `-c` a comenzii `grep` doar contorizează numărul de apariţii, în loc să afişeze toate apariţiile), iar comanda

```
UNIX> ps -aux | grep -w so
```

are ca efect: afisarea tuturor sesiunilor deschise in acel moment ale utilizatorului `so` si a proceselor rulate in *foreground* in aceste sesiuni de catre acest utilizator (opţiunea `-w` a comenzii `grep` selectează doar potrivirile exacte ale cuvîntului "so", adică nu va selecta şi apariţiile de cuvinte ce conţin în ele particula "so" ca subcuvînt propriu).

Asadar, utilitarul `grep` este util pentru a extrage dintr-un fisier (sau grup de fisiere) acele linii care contin siruri de caractere ce satisfac un sablon dat, forma generala de apel fiind

`UNIX> grep [`*optiuni*`]` ***`sablon`*** *fisiere*

Pe linga optiunile `-c` si `-w` deja amintite, alte optiuni utile pentru `grep` ar mai fi optiunea `-i` care face selectie "*case insensitive*", si optiunea `-v` ce selecteaza liniile care nu satisfac sablonul specificat, plus multe altele pe care le puteti afla consultind pagina de manual a comenzii `grep`.

Precum am spus, pentru specificarea sablonului se pot utiliza expresii regulate. O *expresie regulata* este o secventa combinata de caractere obisnuite si de caractere speciale (ce au un rol special – ele descriu modul de interpretare al sablonului).

Caracterele speciale ce se pot folosi intr-o expresie regulata si semnificatiile lor sunt urmatoarele:

- `^` = indicator inceput de linie;
- `$` = indicator sfarsit de linie;
- `*` = caracterul anterior, repetat de oricate ori (*i.e.*, de $n \geq 0$ ori);
- `.` = orice caracter (exact un caracter);
- `[...]` = un caracter dintre cele din lista specificata (la fel ca la specificatorii de fisiere);
- `[^...]` = orice caracter cu exceptia celor din lista specificata;
- `\` = scoate din contextul uzual caracterul care urmeaza (util pentru a inhiba interpretarea caracterelor speciale);

Caracterele speciale extinse (activate prin optiunea `-E` ***`sablon`*** a comenzii `grep`) sunt urmatoarele:

- `+` = caracterul anterior, repetat cel putin o data (*i.e.*, de $n > 0$ ori);
- `?` = caracterul anterior, cel mult o aparitie (una sau zero aparitii);

- {n} sau {n,} sau {n,m} = repetarea unui caracter de un numar precizat de ori (exact de n ori in primul caz, de cel putin n ori pentru a doua forma, si de un numar de ori cuprins intre n si m, pentru a treia forma);
- *expr1* | *expr2* = lista de optiuni alternative (SAU);
- *expr1 expr2* = concatenarea expresiilor;
- (*expr*) = parantezele sunt utile pentru gruparea subexpresiilor, pentru a "scurt-circuita" precedenta operatorilor, care este urmatoarea: repetitia este mai prioritara decit concatenarea, iar aceasta este mai prioritara decit alternativa "|".

O alta comanda de tip *filtru* este comanda `cut`, ce permite selectarea anumitor portiuni din fiecare linie de text a fisierului specificat.

Portiunile selectate pot fi de tip *octet* (daca se foloseste optiunea `-b`), sau de tip *caracter* (daca se foloseste optiunea `-c`), sau de tip *cîmp* (daca se foloseste optiunea `-f`), caz in care se poate specifica si caracterul ce delimiteaza cîmpurile in cadrul fiecarei linii de text, cu ajutorul optiunii `-d` (daca nu se specifica aceasta optiune, atunci implicit se foloseste ca separator caracterul `TAB`).

Fiecare dintre cele 3 optiuni `-b`, `-c` sau `-f` este urmata de o lista de numere naturale (separate prin ',' sau prin '-' , caz in care semnifica un interval de numere), lista ce indica numerele de ordine ale portiunilor (de tipul specificat) ce vor fi selectate din fiecare linie de text a fisierului de intrare si scrise in fisierul de iesire.

Exemplu. Comanda

```
UNIX> cut -b1-8 /etc/passwd
```

va afisa numele conturilor de utilizatori existente in sistemul respectiv, iar comanda

```
UNIX> cut -f1,3-5 -d:  /etc/passwd
```

va afisa numele conturilor de utilizatori, fiecare cont fiind insotit de `UID`-ul asociat si `GID`-ul grupului din care face parte, precum si de sectiunea de comentarii (care de obicei contine numere real al utilizatorului si alte date personale).

8. **Comenzi pentru schimbarea caracteristicilor unui fisier:**
   - pentru schimbarea drepturilor de acces: `chmod`
   - pentru schimbarea proprietarului: `chown`
   - pentru schimbarea grupului proprietarului: `chgrp`

   Pe acestea le-am prezentat in subsectiunea anterioara.

2.3 Interpretoare de comenzi UNIX, partea I-a: Prezentare generală

1. **Introducere**
2. **Comenzi *shell*. Lansarea în execuţie**
3. **Execuţia secvenţială, condiţională, şi paralelă a comenzilor**
4. **Specificarea numelor de fişiere**
5. **Redirectări I/O**
6. **Înlănţuiri de comenzi (prin *pipe*)**
7. **Fişierele de configurare**
8. **Istoricul comenzilor tastate**

2.3.1 Introducere

Interpretorul de comenzi (numit uneori si *shell*) este un program executabil ce are, in UNIX, aceeasi destinatie ca si in MS-DOS, realizind doua functii de baza:

1. aceea de a prelua comenzile introduse de utilizator si de a afisa rezultatele executiei acestora, realizind astfel interfata dintre utilizator si sistemul de operare;
2. aceea de a oferi facilitati de programare intr-un limbaj propriu, cu ajutorul caruia se pot scrie *script*-uri, adica fisiere cu comenzi UNIX.

Daca in MS-DOS este utilizat practic un singur interpretor de comenzi, si anume programul command.com (desi teoretic acesta poate fi inlocuit de alte programe similare, cum ar fi ndos.com-ul), in UNIX exista in mod traditional mai multe interpretoare de comenzi: sh (*Bourne SHell*), bash (*Bourne Again SHell*), csh (*C SHell*), ksh (*Korn SHell*), tcsh (o varianta de csh), ash, zsh, ş.a., utilizatorul avind posibilitatea sa aleaga pe oricare dintre acestea.

Shell-urile din UNIX sunt mai puternice decit analogul (command.com) lor din MS-DOS, fiind asemanatoare cu limbajele de programare de nivel inalt: au structuri de control alternative si repetitive de genul if, case, for, while, etc., ceea ce permite scrierea de programe complexe ca simple *script*-uri. Un *script* este un fisier de comenzi UNIX (analogul fisierelor *batch* *.bat din MS-DOS).

In continuare ne vom referi la interpretorul de comenzi `bash` (*Bourne Again SHell*), care este instalat implicit in `Linux` drept *shell* de *login*, dar facem observatia ca majoritatea facilitatilor din `bash` sunt comune tuturor interpretoarelor de comenzi `UNIX` (chiar daca uneori difera prin sintaxa).

In cele ce urmeaza, ca si pina cum, prin cuvintul `UNIX>` vom desemna prompterul afisat de interpretor pentru introducerea de comenzi de catre utilizator, iar textul ce urmeaza dupa prompter (pina la sfirsitul liniei) va fi indicat folosind fontul `comanda de introdus` , pentru a indica comanda ce trebuie tastata de utilizator la prompter.

2.3.2 Comenzi *shell*. Lansarea în execuţie

Dupa cum am amintit deja, la fel ca in `MS-DOS`, si in `UNIX` exista doua categorii de comenzi: *comenzi interne* (care se gasesc in fisierul executabil al shell-ului respectiv) si *comenzi externe* (care se gasesc separat fiecare intr-un fisier executabil, avind acelasi nume cu comanda respectiva).

In cele ce urmeaza prin *comenzi* (sau *comenzi* `UNIX`, sau *comenzi shell*) vom intelege:

1. comenzi interne, de exemplu: `cd`, `help`, ş.a.;
   (*Observaţie*: lista tuturor comenzilor interne se poate obtine cu comanda `help`)
2. comenzi externe:
   (a) fisiere executabile (*i.e.*, programe executabile obtinute prin compilare din programe sursa scrise in C sau alte limbaje de programare), de exemplu: `passwd`, `bash`, ş.a.;
   (b) *script*-uri (*i.e.*, fisiere cu comenzi), de exemplu: `.profile`, `.bashrc`, ş.a.

Forma generala de lansare in executie a unei comenzi `UNIX` este urmatoarea:

`UNIX>` *`comanda`* `[`*`optiuni`*`]` `[`*`argumente`*`]` ,

unde optiunile si argumentele pot lipsi, dupa caz. Prin conventie, optiunile sunt precedate de caracterul '-' (in `MS-DOS` este folosit caracterul '/'). Argumentele sunt cel mai adesea nume de fisiere. Daca este vorba de o comanda externa, aceasta poate fi specificata si prin calea ei (absoluta sau relativa).

Separatorul intre numele comenzii si ceilalti parametri ai acesteia, precum si intre fiecare dintre parametri este caracterul `SPACE` sau caracterul `TAB` (unul sau mai multe spatii sau *tab*-uri).

O comanda poate fi scrisa pe mai multe linii, caz in care fiecare linie trebuie terminata cu caracterul '\', cu exceptia ultimei linii.

Exceptind cazul unei comenzi interne, sau cazul cind comanda este data prin specificarea caii sale (absoluta sau relativa), *shell*-ul va cauta fisierul executabil cu numele *comanda* in directoarele din variabila de mediu `PATH`, in ordinea in care apar in ea, si, in caz ca-l gaseste, il executa, altfel afiseaza mesajul de eroare:
`bash: comanda: command not found`

Observatie: *shell*-ul nu cauta fisierul executabil mai intii in directorul curent si apoi in directoarele din variabila de mediu `PATH`, asa cum se intimpla in `MS-DOS`; acest neajuns poate fi inlaturat in `UNIX` prin adaugarea directorului curent "**.**" la variabila `PATH`, sau prin specificarea relativa `./comanda` pentru o comanda externa avind fisierul asociat in directorul curent.

Observatie: pentru a putea executa fisierul asociat acelei comenzi, utilizatorul care a lansat acea comanda trebuie sa aiba drept de executie pentru acel fisier (*i.e.*, atributul **x** corespunzator acelui utilizator sa fie setat).

O alta posibilitate de a lansa in executie o comanda este prin apelul unui *shell*:

`UNIX> bash comanda [optiuni] [argumente]`

In sfirsit, doar pentru comenzi ce sunt *script*-uri, o a treia posibilitate este de a lansa in executie comanda prin apelul:

`UNIX> . comanda [optiuni] [argumente]`

Toate formele de lansare a unei comenzi pomenite mai sus au ca efect executia acelei comenzi in *foreground* (*i.e.*, in "planul din fata") : *shell*-ul va astepta terminarea executiei acelei comenzi inainte de a afisa din nou prompterul pentru a accepta noi comenzi din partea utilizatorului.

Exista insa si posibilitatea de a lansa o comanda in executie in *background* (*i.e.*, in "fundal") : in aceasta situatie *shell*-ul nu va mai astepta terminarea executiei acelei comenzi, ci va afisa imediat prompterul pentru a accepta noi comenzi din partea utilizatorului, in timp ce acea comanda isi continua executia (fara a putea primi *input* din partea tastaturii).
Lansarea comenzii in *background* se face adaugind caracterul '&' la sfirsitul liniei:

`UNIX> comanda [parametri] &`

Vom vedea in continuare alte posibilitati de a lansa in executie comenzi.

2.3.3 Execuţia secvenţială, condiţională, şi paralelă a comenzilor

Pina acum am vazut cum se pot lansa comenzi in executie, doar cite una o data (*i.e.*, cite o singura comanda tastata pe un rind la prompterul *shell*-ului).

Exista insa posibilitatea de a tasta pe un singur rind la prompter doua sau mai multe comenzi, caz in care comenzile trebuie separate fie prin caracterul ';', fie prin caracterul '|', sau putem avea combinatii ale acestora daca sunt mai mult de doua comenzi. Efectul lor este urmatorul:

- caracterul ';' – *executie secventiala*:

  comenzile separate prin caracterul ';' sunt executate secvential, in ordinea in care apar.

- caracterul '|' (*pipe*) – *executie paralela, inlantuita*:

  comenzile separate prin caracterul '|' sunt executate simultan (*i.e.*, in paralel, in acelasi timp), fiind inlantuite prin *pipe* (adica iesirea standard de la prima comanda este "*conectata*" la intrarea standard a celei de a doua comenzi); vom reveni mai tirziu asupra acestui aspect.

Exemplu. Secventa de comenzi

```
UNIX> ls ; cd ~so ; ls -l
```

are ca efect listarea continutului directorului curent, apoi schimbarea acestuia in directorul *home* al utilizatorului `so`, urmata apoi de listarea continutului acestuia. Iar comanda inlantuita

```
UNIX> cat /etc/passwd | grep so
```

are ca efect afisarea liniei, din fisierul `/etc/passwd`, care contine informatiile despre contul `so`.

In sfirsit, `UNIX`-ul permite si *executia conditionala* de comenzi. Mai precis, exista doua posibilitati de a conditiona executia unei comenzi de rezultatul executiei unei alte comenzi, si anume:

- prima forma este *conjuncţia* a doua comenzi:

  `UNIX>` *`comanda_1 && comanda_2`*

  Efect: intii se executa prima comanda, iar apoi se va executa a doua comanda numai daca executia primei comenzi intoarce codul de retur 0 (ce corespunde terminarii cu succes).

- a doua forma este *disjuncţia* a doua comenzi:

  `UNIX>` *`comanda_1`* `||` *`comanda_2`*

  Efect: intii se executa prima comanda, iar apoi se va executa a doua comanda numai daca executia primei comenzi intoarce un cod de retur nenul (ce corespunde terminarii cu eroare).

2.3.4 Specificarea numelor de fişiere

Referitor la specificarea fisierelor ca argumente pentru comenzi, dupa cum am amintit deja, sunt mai multe feluri de specificari:

1. prin *calea absoluta* (*i.e.*, pornind din directorul radacina), de exemplu:
   `/home/vidrascu/so/file0003.txt`
2. prin *calea relativa* (la directorul curent) (*i.e.*, pornind din directorul curent de lucru), de exemplu (presupunind ca directorul curent este `/home/vidrascu`):
   `so/file0003.txt`
3. prin *calea relativa la un director home* al unui anumit utilizator (*i.e.*, pornind din directorul *home* al acelui utilizator), de exemplu:
   `~vidrascu/so/file0003.txt`
   (*Observatie*: pentru directorul *home* propriu, *i.e.* al utilizatorului ce tasteaza respectiva comanda, se poate folosi doar caracterul ~ in loc de ~*username*.)

Toate cele de mai sus sunt specificari *unice* de fisiere (*i.e.*, fisierul se specifica prin numele sau *complet*).

La fel ca in `MS-DOS`, o alta posibilitate de a indica specificatori de fisier ca argumente in linia de apel pentru orice comanda `UNIX`, este de a specifica o lista de fisiere (un "sablon") prin folosirea caracterelor speciale de mai jos, efectul fiind ca acel sablon este inlocuit cu (*i.e.*, este "expandat" in linia de apel a comenzii prin) numele tuturor fisierelor ce se potrivesc acelei specificatii sablon (eventual nici unul), aflate in directorul specificat (toate cele trei feluri de specificare a caii de mai sus se pot folosi si pentru specificarea prin sablon):

1. caracterul '*': inlocuieste orice sir de caractere, inclusiv sirul vid;
   Exemplu: `~vidrascu/so/file*.txt`
2. caracterul '?': inlocuieste orice caracter (dar exact 1 caracter!);
   Exemplu: `~vidrascu/so/file000?.txt`

3. specificatorul multime de caractere [...] : inlocuieste exact 1 caracter, dar nu cu orice caracter, ci doar cu cele specificate intre parantezele '[' si ']', sub forma de enumerare (separate prin ',' sau nimic) si/sau interval (dat prin capetele intervalului, separate prin '-');
   Exemple: ~vidrascu/so/file000[1,2,3].txt
   ~vidrascu/so/file000[123].txt
   ~vidrascu/so/file000[1-3].txt
   ~vidrascu/so/file00[0-9][1-9].txt
   ~vidrascu/so/file000[1-3,57-8].txt

4. specificatorul multime exclusa de caractere [^...] : inlocuieste exact 1 caracter, dar nu cu orice caracter, ci doar cu cele din complementara multimii specificate intre parantezele '[' si ']' similar ca mai sus, doar cu exceptia faptului ca primul caracter de dupa '[' trebuie sa fie '^' pentru a indica complementariere (excludere);
   Exemplu: ~vidrascu/so/file000[^1-3].txt

5. caracterul '\': se foloseste pentru a inhiba interpretarea operator a caracterelor speciale de mai sus, si anume \c (unde c este unul dintre caracterele '*', '?', '[', ']', '^', '\') va interpreta acel caracter c ca text (*i.e.*, prin el insusi) si nu ca operator (*i.e.*, prin "sablonul" asociat lui in felul descris mai sus).
   Exemple: ce_mai_faci\?.txt , lectie\[lesson].txt

Alte exemple:
Presupunem ca in directorul curent avem fisierele: file01.dat, file02.dat, form.txt, probl.c, results.doc, results.bak, si test_param, ultimul fiind un fisier de comenzi. In acest caz, comanda

```
UNIX> test_param f*
```

este echivalenta cu comanda explicita

```
UNIX> test_param file01.dat file02.dat form.txt
```

De asemenea, comanda

```
UNIX> test_param m*.doc abcd
```

este echivalenta cu comanda explicita

```
UNIX> test_param abcd
```

2.3.5 Redirectări I/O

Una dintre facilitatile comune tuturor interpretoarelor de comenzi `UNIX` este cea de *redirectare* a intrarii si iesirilor standard (similar ca in `MS-DOS`).

Exista trei dispozitive logice I/O standard:
a) *intrarea standard* (`stdin`), de la care se citesc datele de intrare in timpul executiei unei comenzi;
b) *iesirea standard* (`stdout`), la care sunt scrise datele de iesire in timpul executiei unei comenzi;
c) *iesirea de eroare standard* (`stderr`), la care sunt scrise mesajele de eroare in timpul executiei unei comenzi.

In mod implicit, comunicarea directa cu utilizatorul se face prin intermediul *tastaturii* (intrarea standard) si al *ecranului* (iesirea standard si iesirea de eroare standard). Cu alte cuvinte dispozitivul logic `stdin` este atasat dispozitivului fizic tastatura (= terminalul de intrare), iar dispozitivele logice `stdout` si `stderr` sunt atasate dispozitivului fizic ecran (= terminalul de iesire).

Interpretorul de comenzi poate "forţa" un program (o comanda) ca, in timpul executiei sale, sa primeasca datele de intrare dintr-un fisier in locul tastaturii, si/sau sa trimita rezultatele intr-un fisier in locul ecranului. Realizarea redirectarilor se face in modul urmator:

1. *redirectarea intrarii standard* (`stdin`):

   `UNIX> comanda < fisier_intrare`

   Efect: datele de intrare se preiau nu de la tastatura, ci din fisierul de intrare precizat.

2. *redirectarea iesirii standard* (`stdout`):

   `UNIX> comanda > fisier_iesire`
   sau
   `UNIX> comanda >> fisier_iesire`

   Efect: rezultatele executiei nu sunt afisate pe ecran, ci sunt scrise in fisierul de iesire precizat. Diferenta intre cele doua forme este ca in primul caz vechiul continut al fisierului de iesire (daca acesta exista deja) este sters (adica se face *rewrite*), iar in al doilea caz datele de iesire sunt adaugate la sfirsitul fisierului (adica se face *append*).

3. *redirectarea iesirii de eroare standard* (`stderr`):

   `UNIX> comanda 2> fisier_iesire_err`
   sau
   `UNIX> comanda 2>> fisier_iesire_err`

   Efect: mesajele de eroare din timpul executiei nu sunt afisate pe ecran, ci sunt scrise in fisierul de iesire precizat. Diferenta intre cele doua forme este la fel ca la redirectarea `stdout`-ului (*i.e.*, *rewrite* vs. *append*).

Pentru a redirecta toate dispozitivele standard pentru o comanda, se combina formele de mai sus, ca in exemplele urmatoare:

```
UNIX>  comanda <fis1 >fis2
UNIX>  comanda >fis1 2>>fis2
UNIX>  comanda <fis1 >>fis2 2>>fis2
```

De asemenea, se poate folosi o anumita redirectare pentru mai multe comenzi, ca in exemplul urmator:

```
UNIX> (comanda1 ; comanda2) <fis1 >>fis2
```

Efect: cele doua comenzi sunt executate secvential, dar amindoua cu `stdin` redirectat din fisierul fis1 si `stdout` redirectat in fisierul fis2.

Dupa cum cunoasteti deja de la programarea in limbajul C, in programele C dispozitivele standard au asociate, in mod implicit, ca *descriptori de fisiere deschise* urmatoarele valori: `stdin = 0` , `stdout = 1` , `stderr = 2`. De aceea, se pot face redirectari spre fisiere deschise specificate nu prin numele fisierului, ci prin descriptorul de fisier asociat. Spre exemplu:

```
UNIX> comanda 2>&1
```

Efect: similar ca la specificarea fisierelor de iesire prin nume simbolice, doar ca se utilizeaza &1 pentru a specifica ca fisier de iesire acel fisier deschis asociat descriptorului de fisier 1. Un alt exemplu:

```
UNIX> echo ‘‘Mesaj de eroare’’ >&2
```

Efect: acest mesaj nu va fi afisat pe `stdin`, ci in fisierul asociat descriptorului de fisier 2 (care este de obicei `stderr`, daca nu a fost redirectata anterior).

Observatie: intr-un program C, in mod normal pentru a scrie ceva intr-un fisier se apeleaza primitiva `write(`*file-descriptor*`, ...);` , dar pentru a o putea apela in program este nevoie ca mai intii sa fie deschis fisierul respectiv, cu apelul:
`int` *file-descriptor* `= open("nume-fisier",...);` ,
iar la sfirsitul executiei programului sistemul va inchide toate fisierele deschise, chiar daca ati uitat sa le inchideti explicit prin apelul functiei `close(`*file-descriptor*`);` .

Totusi, pentru dispozitivele standard `stdin`, `stdout`, `stderr` nu mai este nevoie de apelul explicit al functiei `open` deoarece aceste dispozitive sunt deschise automat de catre sistem – terminalele de intrare/iesire sunt pastrate deschise pe toata durata sesiunii de lucru, iar eventualele fisiere folosite ca redirectari I/O prin mijloacele descrise de mai sus, sunt deschise de sistem atunci cind *shell*-ul interpreteaza linia respectiva introdusa si se pregateste s-o execute (si sunt inchise la sfirsitul executiei acelei comenzi).

2.3.6 Înlănţuiri de comenzi (prin *pipe*)

Alta dintre facilitatile comune tuturor interpretoarelor de comenzi `UNIX` este cea de inlantuire de comenzi (prin *pipe*), de care am amintit deja mai sus, cind am vorbit despre executia paralela a comenzilor.
Înlantuirea de comenzi consta in: este posibil de a conecta iesirea standard a unei comenzi la intrarea standard a alteia intr-o singura linie de comanda, eliminindu-se astfel necesitatea comunicarii intre cele doua programe prin intermediul unor fisiere temporare. Sintaxa acestei actiuni este urmatoarea:

`UNIX>` *`comanda_1`* `|` *`comanda_2`* `| ... |` *`comanda_N`*

Simbolul '|' (*pipe*) este cel care marcheaza "legarea" iesirii unei comenzi la intrarea comenzii urmatoare. Mecanismul utilizat va fi studiat ulterior, reprezentind una din formele de comunicare inter-procese specifice sistemelor de operare din familia `UNIX`. Dupa cum am spus mai sus, toate aceste comenzi sunt executate simultan, in paralel (nu secvential!), inlantuite prin *pipe*-uri (iesirea standard de la prima devine intrare standard pentru a doua, ş.a.m.d.).

Restrictie: interpretoarele de comenzi poseda o serie de comenzi interne; din motive de arhitectura a sistemului, acestea nu pot face parte din lanturi de comenzi.

Observatie: atit redirectarea intrarii si iesirii standard, cit si inlantuirea comenzilor sint posibile si in `MS-DOS`, dar sint mult mai des folosite in `UNIX`, pe de o parte datorita caracterului *multitasking* al sistemului, iar pe de alta parte datorita multitudinii de utilitare care preiau datele de intrare de la tastatura si afiseaza rezultatele pe ecran.

Exemplu. Iata citeva exemple de comenzi:

`UNIX>` *`comanda1`* `2>&1 |` *`comanda2`*
Efect: iesirile `stdout` si `stderr` de la *comanda1* sunt "conectate" la intrarea `stdin` a comenzii *comanda2*.

`UNIX> ls -Al | wc -l`
Efect: afiseaza numarul fisierelor (si subdirectoarelor) din directorul curent.

`UNIX> cat fis1 fis2 > fis3`
Efect: concateneaza primele doua fisiere in cel de-al treilea.

Sa mai vedem citeva exemple. Comanda urmatoare

`UNIX> finger | cut -f1 -d''`
va afisa lista numelor utilizatorilor conectati la sistem (prefixata de cuvintul "Login"), lista care se mai poate obtine si cu comanda:

```
UNIX> who | cut -f1 -d"" | sort -u
```
Efect: va afisa lista ordonata a numelor utilizatorilor conectati la sistem.

```
UNIX> echo "username:"; read x ; echo `who | cut -d"" -f1 | grep -c $x`
```

Efect: se afiseaza "username:" si se asteapta citirea de la tastatura a unui nume de utilizator, apoi se afiseaza numărul de sesiuni deschise de utilizatorul introdus.

Observaţie: fişierul `/etc/passwd` conţine linii de text de forma următoare:

```
username:password:uid:gid:comments:directory:shell
```

cu informatii despre conturile de utilizatori din sistem.
Iar fişierul `/etc/group` conţine linii de text de forma următoare:

```
groupname:password:gid:userlist
```

cu informatii despre grupurile de utilizatori din sistem.

```
UNIX> cat /etc/passwd | tail -20 | cut -f3,7 -d :  | less
```
Efect: se vor afişa `UID`-ul si *shell*-ul de *login* al ultimelor 20 de conturi din `/etc/passwd`.

```
UNIX>  (tail /etc/group | cut -f3-4 -d:  | less ) >/dev/null
```
Efect: nu se va afisa nimic pe ecran, datorita redirectarii iesirii standard catre fisierul *nul*. In absenta redirectarii, s-ar fi afisat `GID`-urile şi listele de utilizatori ale ultimelor 10 grupuri de utilizatori din `/etc/group`.

(*Notă*: `/dev/null` este un fisier logic cu rol de "nul": nu se poate citi nimic din el, si orice scriere in el "se pierde", *i.e.* nu are nici un efect.)

2.3.7 Fişierele de configurare

In `UNIX`, pentru fiecare *shell* exista o serie de fisiere de *initializare*, de *configurare* si de *istoric* a *shell*-ului respectiv. Numele acestor fisiere este un alt detaliu prin care se deosebesc *shell*-urile `UNIX`.

La fel ca in `MS-DOS`, fiecare utilizator isi poate scrie un fisier *script* care sa fie executat la fiecare inceput de sesiune de lucru (analogul fisierului `autoexec.bat` din `MS-DOS`), *script* numit `$HOME/.profile` sau `$HOME/.bash_profile` in cazul cind se utilizeaza `bash`-ul ca *shell* implicit (pentru alte *shell*-uri acest fisier de initializare este denumit altfel).
In plus poate avea un *script* care sa fie rulat atunci cind se deconecteaza de la sistem (adica la `logout`); acest *script* se numeste `$HOME/.bash_logout` in cazul *shell*-ului `bash`.
Dupa cum se observa, toate aceste fisiere se gasesc in directorul *home* al acelui utilizator

(`$HOME` este o variabila de mediu ce are ca valoare calea absoluta a directorului *home* al utilizatorului).

Mai exista doua fisiere de initializare valabile pentru toti utilizatorii, si anume fisierele `/etc/profile` si `/etc/environment`. La deschiderea unei noi sesiuni de lucru, mai intii sunt executate *script*-urile de sistem (*i.e.*, cele din directorul `/etc/`), si abia apoi cele particulare utilizatorului respectiv (*i.e.*, cele din `$HOME/`).

In plus, exista si niste fisiere de configurare, si anume un fisier global, numit `/etc/bashrc`, si un fisier specific fiecarui utilizator, numit `$HOME/.bashrc` (acestea sunt numele in cazul *shell*-ului `bash`).
Aceste fisiere sunt executate ori de cite ori este lansat un proces *shell* interactiv, exceptind *shell*-ul de *login*. Ca, de exemplu, atunci cind de sub *browser*-ul `lynx` se apeleaza interpretorul de comenzi printr-o anumita comanda (mai exact, apasind tasta "!"), sau dintr-un editor de texte, sau la pornirea programului `mc` (*i.e.*, a managerului de fisiere *Midnight Commander*), etc. Sau ca atunci cind, din linia de comanda, startam un nou *shell* (fara parametri).

De asemenea, mai exista un fisier de istoric, numit `$HOME/.bash_history`, care pastreaza ultimele `N` comenzi tastate (exista o variabila de mediu care specifica aceasta dimensiune `N`). Formatul de pastrare este urmatorul: fiecare linie de comanda tastata se pastreaza pe o linie de text in fisier, in ordine cronologica (ultima comanda tastata este pe ultima linie din fisier). Acest nume este specific *shell*-ului `bash` (in alte *shell*-uri se numeste altfel, sau nu exista – spre exemplu *shell*-ul `sh` nu are facilitatea de istoric).

2.3.8 Istoricul comenzilor tastate

Lista comenzilor tastate (pastrata in fisierul de istoric `$HOME/.bash_history`) poate fi vizualizata cu comanda `history`. Aceasta comanda afiseaza, in ordine cronologica, aceasta lista a comenzilor tastate anterior, fiecare linie de comanda fiind prefixata de un numar (de la 1 pina la numarul total de linii de comenzi salvate in fisierul de istoric).
Numerele afisate de comanda `history` pot fi folosite pentru a executa din nou comenzile din istoric, fara a fi nevoie sa mai fie tastate din nou. Si anume, pentru a invoca din nou o comanda, se tasteaza la prompter caracterul '!' urmat imediat de numarul asociat comenzii respective.

Exemplu. Să presupunem că la un moment dat avem următorul istoric de comenzi:

```
UNIX> history
  1  ls -Al
  2  pwd
  3  cd mail/
```

```
  4  ls -l
  5  joe inbox.txt
  .  ...
  .  ...
  .  ...
 99  cat .bash_profile
100  less .profile
```

atunci comanda

```
UNIX> !99
```

are ca efect: se executa din nou comanda `cat .bash_profile`.

Mai exista o facilitate a *shell*-ului ce permite invocarea comenzilor tastate anterior, precum si editarea lor, si anume: prin apasarea, la prompterul *shell*-ului, a tastei `UP` (*i.e.*, tasta sageata-sus) de un anumit numar de ori, se vor afisa la prompter comenzile anterioare, in ordine inversa celei in care au fost tastate, si anume cite o comanda anterioara la fiecare apasare a tastei `UP`.
Comanda astfel selectata din istoric, poate fi editata (modificata), iar apoi executata (prin apasarea tastei `ENTER`).

2.4 Interpretoare de comenzi UNIX, partea a II-a: Programare BASH

1. **Introducere**
2. **Proceduri** *shell* **(***script***-uri)**
3. **Variabile de** *shell*
4. **Structuri de control pentru** *script***-uri**
5. **Alte comenzi** *shell* **utile pentru** *script***-uri**

2.4.1 Introducere

Dupa cum am mai spus, *shell*-ul are si functia de limbaj de programare. El recunoaste toate notiunile de baza specifice oricarui limbaj de programare, cum ar fi: variabile, instructiuni de atribuire, instructiuni de control structurate (`if`, `while`, `for`, `case`), proceduri si functii, parametri. In cele ce urmeaza le vom trece pe rind in revista.

Toate aceste facilitati permit automatizarea unor actiuni pe care utilizatorul le desfasoara in mod repetat. La fel ca in `MS-DOS/Windows`, este posibila scrierea unor fisiere care sa contina comenzi, fisiere ce pot fi executate la cererea utilizatorului de catre interpretorul de comenzi. In terminologia `UNIX`, un asemenea fisier de comenzi se numeste *script*.

2.4.2 Proceduri *shell* (*script*-uri)

Procedurile *shell* sunt fisiere de comenzi `UNIX`(numite *script*-uri), analog cu fisierele batch `*.bat` din `MS-DOS`. Apelul lor se face la fel ca pentru orice comanda `UNIX`. Recuperarea in procedura *script* a argumentelor de apel se face cu ajutorul unor variabile speciale (ce vor fi deci parametrii formali in procedura *script*), variabile pe care le vom discuta mai încolo.

In fisierele *script* se foloseste caracterul '#' pentru a indica un comentariu, ce este valabil pina la sfirsitul acelei linii de text, analog cu forma *//comentariu* din limbajul C++.

Dupa cum am mai spus in sectiunea 2.1, lansarea in executie a unui fisier de comenzi se poate face in mai multe moduri:

1. prin numele lui, la fel ca orice comanda:

   `UNIX>` *nume_fisier_comenzi* `[`*parametri*`]`

   Observatie: acest mod de executie este posibil numai in cazul in care fisierul a fost facut executabil setindu-i dreptul de executie, *i.e.* atributul **x**, cu comanda `chmod`.

2. prin comanda interna `.` (sau comanda interna `source`) a interpretorului de comenzi:

   `UNIX> .` *nume_fisier_comenzi* `[`*parametri*`]`
   sau
   `UNIX> source` *nume_fisier_comenzi* `[`*parametri*`]`

3. prin apelul unui anumit *shell*, ca de exemplu al `bash`-ului:

   `UNIX> bash` *nume_fisier_comenzi* `[`*parametri*`]`

Exista unele diferente semnificative intre aceste forme de apel, dupa cum vom vedea mai jos.

Datorita existentei mai multor interpretoare de comenzi `UNIX`, este necesar un mecanism prin care sistemul de operare sa poata fi informat asupra interpretorului de comenzi pentru care a fost scris un anumit fisier de comenzi, in caz contrar acesta riscind sa nu poata fi executat.
Printr-o conventie respectata de mai multe dialecte de `UNIX` (inclusiv de catre `Linux`), prima linie a fisierului de comenzi poate contine numele interpretorului caruia ii este destinat, in forma urmatoare: `#!`*nume_interpretor_comenzi* .

Iata citeva exemple:
- `#!/bin/bash`
- `#!/bin/sh`
- `#!/bin/csh`
- `#!/bin/perl`

Se observa ca trebuie precizata si calea absoluta pentru interpretorul de comenzi cu care se doreste a fi executat acel *script*. Acest mecanism este folosit si de fisierele care sint scrise in limbaje mai puternice, dar de acelasi tip – limbaje interpretate (de exemplu limbajul Perl).

Important: apelul *shell*-ului specificat intr-un *script* in felul descris mai sus (*i.e.*, pe prima linie a fisierului), pentru a executa acel *script*, se face doar pentru prima forma de apel amintita mai sus:

`UNIX>` *nume_script* `[`*parametri*`]`

In acest caz, procesul *shell* in care se da aceasta comanda (la prompterul acestuia) va crea un nou proces ce va executa *shell*-ul specificat in fisierul *script* in felul descris mai sus (*i.e.*, pe prima linie a fisierului), sau, in cazul ca nu se specifica un *shell* in fisierul *script*, va fi o copie a *shell*-ului de comanda. In ambele situatii posibile, noul proces *shell* creat va rula

in mod *neinteractiv* (adica nu va afisa la rindul sau un prompter pentru a interactiona cu utilizatorul) si va executa linie cu linie comenzile din fisierul *script* specificat.

Pentru a doua forma de apel amintita mai sus:

`UNIX>` **`.`** *nume_script* [*parametri*]

(sau cea echivalenta cu comanda `source`), acel *script* va fi executat de fapt tot de catre *shell*-ul in care se da aceasta comanda (deci nu se mai creeaza un nou proces *shell* care sa execute acel *script*, ca la prima forma de apel).

Iar pentru a treia forma de apel:

`UNIX>` *nume_shell nume_script* [*parametri*]

acel *script* va fi executat de *shell*-ul specificat pe prima pozitie a acestui apel. Mai precis, in acest caz procesul *shell* in care se da aceasta comanda (la prompterul acestuia) va crea, asemanator ca la prima forma de apel, un nou proces ce va executa, de aceasta data, *shell*-ul specificat prin linia de comanda. Acest nou proces *shell* creat va rula in mod neinteractiv si va executa linie cu linie comenzile din *script*-ul specificat.

2.4.3 Variabile de *shell*

O alta facilitate comuna tuturor interpretoarelor de comenzi `UNIX` este utilizarea de *variabile* (similar ca in `MS-DOS`). Pentru a creste flexibilitatea sistemului este posibila definirea, citirea si modificarea de variabile de catre utilizator.
Trebuie retinut faptul ca variabilele sunt numai de tip sir de caractere (exceptie facind partial interpretorul `csh`).

Instructiunea de atribuire are forma:

`UNIX>` ***var=expr***

unde *var* este un identificator (*i.e.*, un nume) de variabila, iar *expr* este o expresie care trebuie sa se evalueze la un sir de caractere.

Atentie: in operatia de atribuire a unei valori unei variabile, caracterele spatiu puse inainte de '=' sau dupa '=' vor da eroare. Asadar, aveti grija sa nu puneti spatii!

Variabilele sunt pastrate intr-o zona de memorie a procesului *shell* respectiv, sub forma de perechi *nume* = *valoare*.

Pentru a vedea ce variabile sunt definite, se foloseste comanda:
`UNIX> set`
care va afisa lista acestor perechi.

Comanda:
`UNIX> var=`
sau
`UNIX> unset var`
are ca efect nedefinirea variabilei *var* (adica acea variabila este stearsa din memorie si orice referire ulterioara la ea va cauza afisarea unui mesaj de eroare).

Referirea la valoarea unei variabile (*i.e.*, atunci cind avem nevoie de valoarea variabilei intr-o expresie) se face prin numele ei precedat de simbolul '$', ceea ce cauzeaza *substitutia* numelui variabilei prin valoarea ei in expresia in care apare.
De exemplu, comanda

`UNIX> echo $var`

are ca efect: se va afisa valoarea variabilei *var*. De observat diferenta fata de comanda

`UNIX> echo var`

al carui efect este acela de afisare a sirului de caractere "var".

Exemplu. Iata citeva exemple de folosire a variabilelor in comenzi:

`UNIX> v=a123b`
are ca efect: variabilei cu numele `v` i se atribuie valoarea "a123b".

`UNIX> cat xy$v`
are ca efect: este echivalenta cu comanda:
`UNIX> cat xya123b`
care va afisa continutul fisierului `xya123b` din directorul curent de lucru (in caz ca exista acest fisier).

`UNIX> v=zz$v`
are ca efect: variabila `v` este modificata – mai precis, se observa ca are loc o operatie de concatenare: la valoarea anterioara a variabilei `v` se adauga prefixul "zz".

`UNIX> v=`
are ca efect: variabila `v` primeste valoarea nula (este distrusa).

Important: pentru substitutia numelui unei variabile prin valoarea ei, interpretorul de comenzi considera ca nume de variabila cel mai lung sir de caractere care incepe dupa caracterul '$' si care formeaza un identificator.
Prin urmare, pentru substitutia numelui unei variabile prin valoarea ei, vor trebui folosite

caracterele '{' si '}' pentru a indica numele variabilei atunci cind acesta nu este urmat de spatiu (*i.e.*, caracterele `SPACE` sau `TAB`) sau alt caracter care nu formeaza un identificator. Mai precis, daca se foloseste intr-o comanda o substitutie de forma: `$`{*var*}*sir* , atunci se va substitui variabila cu numele *var* si nu cea cu numele *varsir* , cum s-ar fi intimplat daca nu se foloseau acoladele.

Spre exemplu, comenzile:

```
UNIX> rad=/home/others/
UNIX> ls -l ${rad}so
```

au ca efect: se va lista continutul directorului `/home/others/so`.

Alte forme de substitutie:

a) `$`{*var*`:-`*sir*}

Efect: rezultatul expresiei este valoarea variabilei *var*, daca aceasta este definită, altfel este valoarea *sir*.

b) `$`{*var*`:-`}

Efect: rezultatul expresiei este valoarea variabilei *var*, daca aceasta este definită, altfel este afisat un mesaj standard de eroare care spune ca acea variabila este nedefinita.

c) `$`{*var*`:=`*sir*}

Efect: rezultatul expresiei este valoarea variabilei *var*, dupa ce eventual acesteia i se asigneaza valoarea *sir* (asignarea are loc doar in cazul in care *var* era nedefinita).

d) `$`{*var*`:?`*sir*}

Efect: rezultatul expresiei este valoarea variabilei *var*, daca aceasta este definită, altfel este afisat mesajul *sir* (sau un mesaj standard de eroare, daca *sir* lipseste).

e) `$`{*var*`:+`*sir*}

Efect: daca variabila *var* este definita (are o valoare), atunci i se asigneaza valoarea *sir*, altfel ramine in continuare fara valoare (deci asignarea are loc doar in cazul in care *var* era deja definita).

O alta constructie speciala recunoscuta de interpretorul de comenzi este expresia `$(`*comanda*`)`, sau echivalent `` `*comanda*` `` (unde caracterul ` este apostroful invers), al carei efect este acela de a fi substituita, in linia de comanda sau contextul in care este folosita, cu textul afisat pe iesirea standard prin executia comenzii specificate.
Spre exemplu, comanda:

```
UNIX> v=`wc -l fis.txt`
```

are ca efect: variabila `v` primeste drept valoare iesirea standard a comenzii specificate intre caracterele apostroafe inverse. In acest caz, va primi drept valoare numarul de linii ale fisierului text `fis.txt` din directorul curent de lucru.

Iata si alte exemple pentru modul de interpretare de catre *shell*-ul `bash` al caracterelor `` `...` `` sau $(...) :

```
UNIX> dir_curent=$(pwd) ; ls $dir_curent
```

are ca efect: variabila cu numele `dir_curent` va primi ca valoare iesirea standard a comenzii `pwd`, care este tocmai numele directorului curent de lucru (specificat prin cale absoluta), iar apoi se va lista continutul acestui director.
Iar succesiunea de comenzi:

```
UNIX> a=10
UNIX> a=`expr $a + 5`
UNIX> echo $a
```

are ca efect: se va afisa 15 (iar variabila `a` va avea in final valoarea 15).

Observatie: `expr` este o comanda care evalueaza expresii aritmetice ca si siruri de caractere – consultati *help*-ul pentru detalii (cu comanda `man expr`).

Pe linga comanda `set`, mai sunt si alte comenzi pentru variabile, si anume:

- Comanda de *exportare*:

  `UNIX> export` *`var [var2 var3 ... ]`*

  care are ca efect "exportul" variabilelor specificate in procesele fii ale respectivului proces *shell* (in mod obisnuit variabilele nu sunt vizibile in procesele fii, ele fiind locale procesului *shell* respectiv, fiind pastrate in memoria acestuia).
  O alta forma de apel a comenzii `export` este:

  `UNIX> export` *`var`*`=`*`valoare [var2=valoare2 ... ]`*

  care are ca efect atribuirea si exportul variabilei printr-o singura comanda.

- Comanda de *citire*:

  `UNIX> read` *`var [var2 var3 ... ]`*

  care are ca efect citirea, de la intrarea standard `stdin`, de valori si atribuirea lor variabilelor specificate.

- Comanda de *declarare read-only*:

  `UNIX> readonly` *`var [var2 var3 ... ]`*

  care are ca efect declararea variabilelor specificate ca fiind *read-only* (*i.e.* ele nu mai pot fi modificate dupa executia acestei comenzi, ci ramin cu valorile pe care le aveau cind s-a executat aceasta comanda).

In terminologia UNIX, se folosesc termenii de **variabila de** *shell* si **variabila de mediu**, cu semnificatii diferite: *variabila de shell* este o variabila accesibila doar procesului *shell* curent, pe cind *variabila de mediu* este o variabila accesibila tuturor proceselor fii ale acelui proces *shell* (*i.e.* este o variabila exportata).

Exista o serie de variabile ce sunt modificate dinamic de catre procesul *shell* pe parcursul executiei de comenzi, cu scopul de a le pastra semnificatia pe care o au. Aceste variabile dinamice, ce pot fi folosite in *script*-uri, in conformitate cu semnificatia lor, sunt urmatoarele:

1) `$0`

Semnificatia: numele procesului curent (numele *script*-ului in care este referita).

2) `$1,$2,...,$9`

Semnificatia: parametrii cu care a fost apelat procesul curent (i.e. parametrii din linia de apel in cazul unui *script*).
Observatie: un fisier de comenzi poate primi, la lansarea in executie, o serie de parametri din linia de comanda. Acestia sint referiti in corpul fisierului prin variabilele `$1,...,$9`. La fel ca in `MS-DOS`, si in `UNIX` interpretorul de comenzi pune la dispozitia utilizatorului comanda interna `shift`, prin care se poate realiza o deplasare a valorilor parametrilor din linia de comanda: vechea valoare a lui `$2` va fi referita prin `$1`, vechea valoare a lui `$3` va fi referita prin `$2`, ş.a.m.d., iar vechea valoare a lui `$1` se pierde). Comanda `shift` este utila pentru cazurile cind avem mai mult de 9 parametri in linia de comanda, pentru a putea avea acces la toti acestia.

3) `$#`

Semnificatia: numarul parametrilor din linia de comanda (fara argumentul `$0`).

4) `$*`

Semnificatia: lista parametrilor din linia de comanda (fara argumentul `$0`).

5) `$@`

Semnificatia: lista parametrilor din linia de comanda (fara argumentul `$0`).
Observatie: diferenta dintre `$@` si `$*` apare atunci cind sunt folosite intre ghilimele: la substitutie `"$*"` produce un singur cuvint ce contine toti parametrii din linia de comanda, pe cind `"$@"` produce cite un cuvint pentru fiecare parametru din linia de comanda.

6) `$$`

Semnificatia: `PID`-ul procesului curent (*i.e.*, `PID`-ul *shell*-ului ce executa acel fisier *script*).
Observatie: variabila `$$` poate fi folosita pentru a crea fisiere temporare cu nume unic, cum ar fi de exemplu numele `/tmp/err$$`.

7) `$?`

Semnificatia: codul returnat de ultima comanda *shell* executata.
Observatie: la fel ca in `MS-DOS`, si in `UNIX` fiecare comanda returneaza la terminarea ei un cod de retur, care este fie 0 (in caz de succes), fie o valoare nenula – codul erorii (in cazul unei erori).

8) `$!`

Semnificatia: `PID`-ul ultimului proces executat in *background*.

9) `$-`

Semnificatia: optiunile cu care a fost lansat procesul *shell* respectiv. Aceste optiuni pot fi:

- `-x` = modul *xtrace* de executie – afiseaza fiecare linie din *script* pe masura ce este executata (astfel se afiseaza rezultatul interpretarii liniei – adica ceea ce se executa efectiv);
- `-v` = modul *verbose* de executie – afiseaza fiecare linie citita de *shell* (astfel se afiseaza linia efectiv citita, inainte de a fi interpretata);

ş.a.

Aceste optiuni se pot folosi cu comanda `set` in felul urmator:

- `set -v ;` *script*
  Aceasta optiune este utila pentru depanare: se seteaza intii optiunea `-v` si apoi se executa fisierul *script* specificat in modul *verbose*.
- `set -x`
  Aceasta optiune este utila pentru a vedea istoricul ultimelor comenzi executate.
- `set -u`
  Aceasta optiune verifica daca variabilele au fost initializate, cauzind eroare la substitutie in cazul variabilelor neinitializate.
- `set -n`
  Aceasta optiune cauzeaza citirea comenzilor ulterioare fara a fi si executate.
- `set -i`
  Folosind aceasta optiune *shell*-ul devine interactiv (optiunea este utila pentru depanare).

Pentru dezactivarea acestor optiuni, se foloseste tot comanda `set`, cu `+` in loc de `-` , si anume: `set +`*o* , unde *o* este optiunea ce se doreste a fi dezactivata.

Comanda `set` cu parametri are ca efect atribuirea acestor parametri ca valori variabilelor `$1`, `$2`, . . . , `$9` (in functie de numarul acestor parametri). Spre exemplu, comanda:

```
UNIX> set a b c ; echo $3
```

are ca efect: determina crearea si initializarea variabilelor `$1`, `$2`, `$3` respectiv cu valorile a, b, c, iar apoi se afiseaza 'c' (*i.e.*, valoarea variabilei `$3`).

Iata un alt exemplu pentru modul de actiune al caracterelor `` `...` `` – recuperarea rezultatului unei comenzi in variabilele `$1`, `$2`, ..., `$9`:

```
UNIX> set `date` ; echo $6 $2 $3 $4
```

are ca efect: prima comanda determina initializarea variabilelor `$1`, `$2`, ..., `$9` cu cîmpurile rezultatului comenzii `date` (*notă*: cîmpurile sunt cuvintele, adica sirurile de caractere ce sunt separate prin spatii, din iesirea standard a comenzii). Iar a doua comanda va afisa anul (cîmpul 6), luna (cîmpul 2), ziua (cîmpul 3) si ora (cîmpul 4) curente.

Studiaţi toate optiunile comenzii `set`, care este o comanda interna si, prin urmare, detaliile despre ea se pot obtine cu comanda `help set` sau cu comanda `man set`.

Iata alte exemple referitoare la caracterele ce au interpretare speciala din partea *shell*-ului:

- Caracterul '&' (folosit pentru executie in *background*):

  a) `UNIX> who | grep an2&so`

  are ca efect: se va interpreta caracterul '&' drept executie in *background*.

  b) `UNIX> who | grep an2\&so`
  sau `UNIX> who | grep "an2&so"`

  au ca efect: se va interpreta caracterul '&' prin el insusi si nu ca fiind executie in *background*.

- Caracterul '$' (folosit pentru substitutie de variabila):

  a) `UNIX> who | grep "an2$PATH"`

  are ca efect: se va interpreta caracterul '$' drept substitutie de variabila, si, ca urmare, se va substitui variabila de mediu `$PATH` cu valoarea sa.

  b) `UNIX> who | grep 'an2$PATH'`

  are ca efect: se va interpreta caracterul '$' prin el insusi si nu ca substitutie de variabila.

De asemenea, exista o serie de variabile de mediu *predefinite* (*i.e.*, care au anumite semnificatii fixate, aceleasi pentru toata lumea), si anume:

1) `$HOME`

Semnificatia: directorul *home* (directorul de *login*) al acelui utilizator.

2) `$USER`

Semnificatia: *username*-ul (numele de *login*) al acelui utilizator.

3) `$LOGNAME`

Semnificatia: la fel ca `$USER`.

4) `$SHELL`

Semnificatia: numele (calea absoluta a) *shell*-ului implicit al acelui utilizator.

5) `$MAIL`

Semnificatia: numele (calea absoluta a) fisierului de posta electronica al acelui utilizator (este utilizat de *shell* pentru a ne anunta daca a fost primit un nou mesaj, necitit inca, adica acel binecunoscut mesaj "You have new mail in `$MAIL`" ce apare dupa etapa de autentificare la *login*).

6) `$PS1`

Semnificatia: sirul de caractere al prompterului principal asociat *shell*-ului.

7) `$PS2`

Semnificatia: sirul de caractere al prompterului secundar asociat *shell*-ului.
(*Nota*: prompterul secundar este prompterul folosit pentru liniile de continuare ale unei comenzi scrise pe mai multe linii.)

8) `$TERM`

Semnificatia: specifica tipul de terminal utilizat (vt100, vt102, xterm, ş.a.).

9) `$PATH`

Semnificatia: o lista de directoare in care *shell*-ul cauta fisierul corespunzator unei comenzi tastate, ce a fost specificata doar prin nume, nu si prin cale (absoluta sau relativa).

10) `$CDPATH`

Semnificatia: o lista de directoare in care *shell*-ul cauta directorul dat ca parametru comenzii `cd`, in cazul cind acesta a fost specificat doar prin nume, nu si prin cale (absoluta sau relativa); este similar ca `$PATH` pentru fisiere.

11) `$IFS`

Semnificatia: specifica multimea caracterelor ce sunt interpretate drept spatiu de catre *shell*.

Mai sunt si alte variabile de mediu (lista tuturor variabilelor definite la un moment dat este afisata de catre comanda `set` fara parametri).
Aceste variabile de mediu sunt initializate de catre procesul *shell* la deschiderea unei sesiuni de lucru, cu valorile specificate in fisierele de initializare ale sistemului (despre aceste fisiere am vorbit mai devreme). De fapt, aceste variabile sunt exportate de *shell*-ul de *login*, dupa cum se poate constata consultind aceste fisiere de initializare.

2.4.4 Structuri de control pentru *script*-uri

Fiecare interpretor de comenzi furnizeaza o serie de structuri de control de nivel inalt, si anume structurile de control alternative si repetitive uzuale din programarea structurata, ceea ce confera fisierelor de comenzi o putere mult mai mare decit este posibil in fisierele de comenzi din `MS-DOS`.

Structurile de control puse la dispozitie de interpretorul `bash` sint: `for`, `while`, `until`, `if`, si `case`. Aceste structuri sunt comenzi interne ale *shell*-ului `bash` (prin urmare se pot obtine informatii despre sintaxa si semantica lor cu comanda `help`).

I. Structurile repetitive

1) **Bucla iterativa FOR** – are urmatoarea *sintaxă*:

```
for  variabila [ in  text ]
do
   lista_comenzi
done
```

sau

```
for  variabila [ in  text ] ; do  lista_comenzi ; done
```

Semantica comenzii `for`:
Porţiunea *text* descrie o lista de valori pe care le ia succesiv *variabila*, si, pentru fiecare asemenea valoare a lui *variabila*, se executa comenzile din *lista_comenzi*.

Observatii:
a) In comenzile din corpul buclei `for` se poate folosi valoarea variabilei *variabila*.
b) Toate instructiunile, exceptind `for`, `do` si `done`, trebuie sa fie urmate de caracterul ';' sau de caracterul sfirsit de linie (`newline`).
c) Daca lipseste partea optionala (*i.e.*, partea `in` *text*), atunci ca valori pentru *variabila* se folosesc argumentele din variabila de *shell* `$*` (*i.e.*, parametrii din linia de comanda).

Exemplu. Comanda:

```
for i in `ls -t`
do
  echo $i
done
```

sau, echivalent:

```
for i in `ls -t` ; do echo $i ; done
```

are acelasi efect cu comanda `ls -t` (*i.e.*, se afiseaza continutul directorului curent, sortat dupa data ultimei modificari).

2) **Bucla repetitiva WHILE** – are urmatoarea *sintaxă*:

```
while  lista_comenzi_1
do
   lista_comenzi_2
done
```

sau

```
while  lista_comenzi_1 ; do  lista_comenzi_2 ; done
```

Semantica comenzii `while`:
Se executa comenzile din *lista_comenzi_1* si daca codul de retur al ultimei comenzi din ea este 0 (*i.e.* terminare cu succes), atunci se executa comenzile din *lista_comenzi_2* si se reia bucla. Altfel, se termina executia buclei `while`.

Observatii:
a) Bucla `while` se executa repetitiv atit timp cit codul returnat de *lista_comenzi_1* este 0 (succes).
b) Toate instructiunile, exceptind `while`, `do` si `done`, trebuie sa fie urmate de caracterul ';' sau de caracterul sfirsit de linie (`newline`).
c) Adeseori *lista_comenzi_1* poate fi comanda: `test` *argumente* sau, echivalent: `[` *argumente* `]` , care este o comanda ce exprima testarea unei conditii, dupa cum vom vedea mai încolo.

Exemplu. Comanda:

```
while true
do
  date;
  sleep 60;
done
```

sau, echivalent:

```
while true ; do  date; sleep 60; done
```

are ca efect: afiseaza in mod continuu pe ecran, din minut in minut, data si ora curenta.

3) **Bucla repetitiva UNTIL** – are urmatoarea *sintaxă*:

```
until  lista_comenzi_1
do
   lista_comenzi_2
done
```

sau

```
until  lista_comenzi_1 ; do  lista_comenzi_2 ; done
```

Semantica comenzii `until`:
Se executa comenzile din *lista_comenzi_1* si daca codul de retur al ultimei comenzi din ea este diferit de 0 (*i.e.* terminare cu eroare), atunci se executa comenzile din *lista_comenzi_2* si se reia bucla. Altfel, se termina executia buclei `until`.

Observatii:
a) Bucla `until` se executa repetitiv atit timp cit codul returnat de *lista_comenzi_1* este diferit de 0 (eroare), sau, cu alte cuvinte, bucla `until` se executa repetitiv pina cind codul returnat de *lista_comenzi_1* este 0 (succes).
b) Toate instructiunile, exceptind `until`, `do` si `done`, trebuie sa fie urmate de caracterul ';' sau de caracterul sfirsit de linie (`newline`).
c) Adeseori *lista_comenzi_1* poate fi comanda de testare a unei conditii.

II. Structurile alternative

4) **Structura alternativa IF** – are urmatoarea *sintaxă*:

```
if  lista_comenzi_1
then
   lista_comenzi_2
[ else
   lista_comenzi_3 ]
fi
```

sau

```
if  lista_comenzi_1 ; then  lista_comenzi_2 ; [ else  lista_comenzi_3 ; ] fi
```

Semantica comenzii `if`:
Se executa comenzile din *lista_comenzi_1* si daca codul de retur al ultimei comenzi din ea este 0 (*i.e.* terminare cu succes), atunci se executa comenzile din *lista_comenzi_2*. Iar altfel, se executa comenzile din *lista_comenzi_3* (daca este prezenta ramura `else`).

Observatii:
a) Ramura `else` este optionala.

b) Toate instructiunile, exceptind `if`, `then`, `else` si `fi`, trebuie sa fie urmate de caracterul ';' sau de caracterul sfirsit de linie (`newline`).
c) Adeseori *lista_comenzi_1* poate fi comanda de testare a unei conditii.

Important: Structura `if` are si o forma sintactica *imbricata*:

```
if  lista_comenzi_1
then
   lista_comenzi_2
elif  lista_comenzi_3
then
   lista_comenzi_4
elif  lista_comenzi_5
then
   lista_comenzi_6
...
else
   lista_comenzi_N
fi
```

5) **Structura alternativa CASE** – are urmatoarea *sintaxă*:

```
case expresie in
  sir_valori_1   ) lista_comenzi_1   ;;
  sir_valori_2   ) lista_comenzi_2   ;;
  ...
  sir_valori_N-1 ) lista_comenzi_N-1 ;;
  sir_valori_N   ) lista_comenzi_N
esac
```

Semantica comenzii `case`:
Daca valoarea expresiei *expresie* se gaseste in lista de valori *sir_valori_1*, atunci se executa *lista_comenzi_1* si apoi executia comenzii `case` se termina. Altfel, daca valoarea expresiei *expresie* se gaseste in lista de valori *sir_valori_2*, atunci se executa *lista_comenzi_2* si apoi executia comenzii `case` se termina. Altfel, ... ş.a.m.d.

Observatii:
a) Deci se intra in prima lista de valori cu care se potriveste, fara a se verifica unicitatea (*i.e.*, daca mai sunt si alte liste de valori cu care se potriveste). Iar apoi se iese direct afara, fara a continua cu urmatoarele linii (desi nu se pune o comanda analoaga instructiunii `break` de pe ramurile `case` ale instructiunii `switch` din limbajul C);
b) Ultima linie dinainte de `esac` nu este obligatoriu sa se termine cu caracterele ";;" precum celelalte linii;
c) Lista de valori *sir_valori_X* poate fi o enumerare de valori de forma:

valoare_1 | *valoare_2* | ... | *valoare_M*

sau poate fi o expresie regulata, ca de exemplu:

```
case $opt in
  -[ax-z]  )  comenzi ;;
  ...
esac
```

ceea ce este echivalent cu:

```
case $opt in
  -a|-x|-y|-z  )  comenzi ;;
  ...
esac
```

Exemplu. Iata si un exemplu:

```
  case $var1 in
    *.c ) var2=`basename $var1 .c` ; gcc $var1 -o$var2 ;;
    ...
  esac
  echo Source $var1 was compiled into executable $var2.
```

Efect: daca variabila `var1` are ca valoare "fisier.c" (*i.e.* numele unui fisier sursa), atunci variabila `var2` va avea ca valoare "fisier" si apoi fisierul sursa `$var1` este compilat obtinindu-se un fisier executabil cu numele `$var2`.

Observatie: comanda

UNIX> `basename` *arg1 arg2*

afiseaza in iesirea standard valoarea argumentului *arg1* dupa ce se inlatura din el sufixul *arg2*.

Important: aceste structuri de control fiind comenzi interne, puteti folosi comanda de *help* pentru comenzi interne:

UNIX> `help` *nume_structura*

pentru a afla mai multe detalii despre fiecare structura in parte.

2.4.5 Alte comenzi *shell* utile pentru *script*-uri

Comanda de testare a unei conditii este comanda (predicatul) `test`, avind forma:

`test` *`conditie`*
sau:
`[` *`conditie`* `]`

unde expresia *conditie* poate fi:

1. o comparatie intre doua siruri de caractere (utilizind simbolurile "=" si "!="):
   - `test` *`expr_1`* `=` *`expr_2`*
     Efect: returneaza *true* (*i.e.*, codul de retur 0) daca cele doua expresii au aceeasi valoare, altfel returneaza *false* (*i.e.*, cod de retur nenul);
   - `test` *`expr_1`* `!=` *`expr_2`*
     Efect: returneaza *true* (*i.e.*, codul de retur 0) daca cele doua expresii au valori diferite, altfel returneaza *false* (*i.e.*, cod de retur nenul).
2. conditii relationale: `test` *`val_1 -rel val_2`*
   Efect: returneaza true (*i.e.*, codul de retur 0) daca valoarea *val_1* este in relatia *rel* cu valoarea *val_2*, unde *rel* este unul dintre operatorii relationali urmatori:
   - `eq` : *equal* (egal, adica =);
   - `gt` : *greater-than* (mai mare decit, adica >);
   - `ge` : *greater-equal* (mai mare sau egal cu, adica $\geq$);
   - `lt` : *less-than* (mai mic decit, adica <);
   - `le` : *less-equal* (mai mic sau egal cu, adica $\leq$).
3. conditii referitoare la fisiere: `test` *`-opt nume_fisier`*
   Efect: returneaza *true* (*i.e.*, codul de retur 0) daca fisierul *nume_fisier* satisface optiunea de testare *opt* specificata, care poate fi una dintre urmatoarele:
   - `e` : testeaza daca exista un fisier de orice tip (*i.e.*, obisnuit, director, *fifo*, fisier special, etc.) avind numele *nume_fisier*;
   - `d` : testeaza daca *nume_fisier* este un director;
   - `f` : testeaza daca *nume_fisier* este un fisier obisnuit;
   - `p` : testeaza daca *nume_fisier* este un fisier de tip *fifo*;
   - `b` : testeaza daca *nume_fisier* este un fisier de tip dispozitiv in mod bloc;
   - `c` : testeaza daca *nume_fisier* este un fisier de tip dispozitiv in mod caracter;
   - `s` : testeaza daca fisierul *nume_fisier* are continut nevid (*i.e.*, are lungimea mai mare decit 0);

– `r` : testeaza daca fisierul *nume_fisier* poate fi citit de catre utilizatorul curent (*i.e.*, daca acesta are drept de citire asupra fisierului);
– `w` : testeaza daca fisierul *nume_fisier* poate fi modificat de catre utilizatorul curent (*i.e.*, daca acesta are drept de scriere asupra fisierului);
– `x` : testeaza daca fisierul *nume_fisier* poate fi lansat in executie de catre utilizatorul curent (*i.e.*, daca acesta are drept de executie asupra fisierului).

4. o expresie logica – negatie, conjunctie, sau disjunctie de conditii:

   - `test !`*`conditie_1`*
     Efect: *NOT* – negatia conditiei *conditie_1*;
   - `test` *`conditie_1`* `-a` *`conditie_2`*
     Efect: *AND* – conjunctia conditiilor *conditie_1* si *conditie_2*;
   - `test` *`conditie_1`* `-o` *`conditie_2`*
     Efect: *OR* – disjunctia conditiilor *conditie_1* si *conditie_2*,

   unde *conditie_1* si *conditie_2* sunt conditii de oricare dintre cele patru forme de mai sus.

Exemplu. *Script*-ul urmator

```
#!/bin/bash
for i in *
do
  if test -f $i
  then
    echo $i
  fi
done
```

are ca efect: se listeaza fisierele obisnuite din directorul curent.

Observatie: caracterul `*` joaca un rol special: in evaluare acest caracter se inlocuieste cu numele oricarui fisier din directorul curent, cu exceptia acelora al caror nume incepe cu caracterul punct ".". Pentru ca `*` sa nu se evalueze in acest fel, ci sa se evalueze prin el insusi, trebuie sa se utilizeze apostroafele: `'*'` ; reamintesc faptul ca am vazut mai devreme un alt exemplu (si anume: `who | grep 'an2$PATH'`) in care apostroafele determinau evaluarea caracterelor speciale prin ele insele.

Iata si alte comenzi (instructiuni) ce pot fi folosite in *script*-uri (sau la linia de comanda):

1. Comanda `break`, cu sintaxa:

   `break [`*n*`]`

   unde *n* este 1 in caz ca lipseste. Efect: se iese afara din *n* bucle `do-done` imbricate, executia continuind cu urmatoarea instructiune de dupa `done`.

2. Comanda `continue`, cu sintaxa:

   `continue [`*n*`]`

   unde n este 1 in caz ca lipseste. Efect: pentru $n = 1$ se reincepe bucla curenta `do-done` (de la pasul de reinitializare), respectiv pentru $n > 1$ efectul este ca si cum s-ar executa de n ori comanda `continue 1`.

3. Comanda `exit`, cu sintaxa:

   `exit [`*cod*`]`

   unde *cod* este 0 in caz ca lipseste. Efect: se termina (se opreste) executia *script*-ului in care apare si se intoarce drept cod de retur valoarea specificata. (*nota*: executata la prompterul *shell*-ului de *login*, va determina terminarea sesiunii de lucru.)

4. Comanda `exec`, cu sintaxa:

   `exec` *`lista_comenzi`*

   Efect: se executa comenzile specificate fara a se crea o noua instanta de *shell* (astfel *shell*-ul ce executa aceasta comanda se va "reacoperi" cu procesul asociat comenzii, deci nu este *reentrant*).

5. Comanda `wait`, cu sintaxa:

   `wait` *pid*

   Efect: se intrerupe executia *script*-ului curent, asteptindu-se terminarea procesului avind `PID`-ul specificat.

6. Comanda `eval`, cu sintaxa:

   `eval` *parametri*

   Efect: se evalueaza parametrii specificati.

7. Comanda `export`, cu sintaxa:

   `export` *variabile*

   Efect: se exporta variabilele de *shell* specificate.

8. Comanda `trap`, cu sintaxa:

   `trap` *`comanda eveniment`*

   Efect: cind se va produce evenimentul specificat (*i.e.*, cind se va primi semnalul respectiv), se va executa comanda specificata.

   Evenimente (semnale) posibile:

   - semnalul 1 = *hang-up signal*;
   - semnalul 2 = *interrupt signal* (generat prin apasarea tastelor `CTRL+C`);
   - semnalul 3 = *quit signal* (generat prin apasarea tastelor `CTRL+D`);
   - semnalul 9 = *kill signal* (semnal ce "omoara" procesul);
   - semnalul 15 = semnal de terminare normala a unui proces;

ş.a. (vom vedea mai multe detalii despre semnale `UNIX` in capitolul 4 din partea II a manualului).

Exemplu. Iata si un exemplu de folosire a comenzii `trap`:

```
UNIX> trap 'rm /tmp/ps$$ ; exit' 2
```

Efect: ulterior, cind se va primi semnalul 2 (*i.e.*, la apasarea tastelor `CTRL+C`), se va sterge fisierul temporar `/tmp/ps$$` si apoi se va termina executia procesului respectiv (in cazul de fata, fiind vorba de *shell*-ul de *login*, se va inchide sesiunea de lucru).

Important: majoritatea acestor comenzi fiind comenzi interne, puteti folosi comanda de *help* pentru comenzi interne:

```
UNIX> help comanda
```

pentru a afla mai multe detalii despre fiecare comanda in parte.

Recomandare: cititi pagina de manual a interpretorului de comnezi `bash`, ce este accesibila cu comanda `man bash`.

Exemplu. Sa scriem un *script* care sa afiseze lista subdirectoarelor din directorul curent. O posibila solutie este urmatoarea:

```
#!/bin/bash
for dir in *
do
  if [ -d $dir ]
  then
    echo $dir
  fi
done
```

Exemplu. Iata un alt exemplu de *script* al carui scop este acela de a apela programul de dezarhivare potrivit pe baza sufixului din numele unui fisier arhivat, dat ca parametru:

```
#!/bin/bash
#Script pentru dezarhivare diverse arhive

if [ $# != 1 ] ; then
  echo -e "\a\nUsage: $0 <arh-file>\n"
fi

if [ ! -f $1 ] ; then
  echo "Error: $1 is not a file or does not exist at the given location."
  exit
```

```
fi

case $1 in
  *.tar         ) tar  -xf $1 ;;
  *.tgz | *.taz ) tar -zxf $1 ;;
  *.gz  | *.Z   ) gunzip   $1 ;;
  *.zip         ) unzip    $1 ;;
  *.Z           ) uncompress $1 ;;
  *             ) exit
esac
```

In continuare vom mai vedea citeva exemple de *script*-uri.

Exemplu. *Script*-ul urmator are scopul de a "face curatenie": sterge fisierele bak (care au numele terminat cu caracterul ~, si reprezinta o versiune mai veche a unui fisier) dintr-un director dat ca parametru:

```
#!/bin/bash
# Utilitar de stergere a fisierelor bak (de forma *~) dintr-un director dat

if test $# = 0 ; then
  echo Usage: $0 "<directory>"
  exit
fi

# optiunea -rec este folosita de acest script pentru apel recursiv
if test $1 = -rec ; then
  shift
else
  if test ! -d $1 ; then
    echo "Error: $1 is not a directory !"
    exit
  fi
  echo -ne "\a\nDeleting bak files from directory $1 recursively (y/n) ? "
  read rasp
  if test $rasp != y ; then
    echo "Cancelled !"
    exit
  fi
fi

# bucla de stergere
for i in  $1/*~  $1/.*~
do
  if test -f $i ; then
    echo -n "Deleting bak file $i (y/n) ? "
    read rasp
    if test $rasp = y ; then
      rm $i  &&  echo "$i was deleted..."
    fi
```

```
  fi
done

# apelul recursiv
for i in  $1/*  $1/.*
do
  if test  "$i" = "$1/."  -o  "$i" = "$1/.."  ; then
    continue
  fi

  if test -d $i ; then
    $0 -rec $i    # apel recursiv
  fi
done
```

Exemplu. Putem afisa numele de cont si numele real, sortate alfabetic, ale tuturor utilizatorilor care au primit mesaje noi prin posta electronica cu urmatorul *script*:

```
#!/bin/bash
#Listeaza toti userii care au new mail, sortati dupa numele de cont

for username in `cut -f1 -d:  /etc/passwd` ;
do finger -pm $username | grep "New mail" -B 15 | head -1 ;
done | sort | less
```

O alta facilitate a *shell*-ului `bash` este aceea de definire de functii *shell*, ce se apeleaza apoi prin numele lor, ca orice comanda.
Iata un exemplu de definire a unei functii *shell*:

```
#De adaugat rindurile de mai jos in fisierul personal .bash_profile
#Efectul:  cind veti parasi mc-ul, directorul curent va fi ultimul director in
#care ati navigat in mc, in loc sa fie directorul din care ati intrat in mc.

function mc ()
{
MC=`/usr/bin/mc -P "$@"`
[ -n "$MC" ] && cd "$MC";
unset MC ;
}

declare -x mc
```

Observatie: datorita utilitatii acestei functii, ea a fost inclusa in fisierele de initializare globale de catre administratorii serverului studentilor, `fenrir`, astfel încît nu mai trebuie adaugat in fisierul de initializare personal.

Iata inca un exemplu de *script* ce foloseste o functie:

```
#!/bin/bash
function cntparm ()
{
 echo -e "$# params: $*\n"
}
cntparm "$*"
cntparm "$@"
```

Daca apelam acest script cu urmatoarea linie de comanda:

```
UNIX> ./script a b c
1 parms:  a b c
3 parms:  a b c
```

mesajele afisate pe ecran ne demonstreaza diferenta dintre modul de evaluare al variabilelor `$*` si `$@` atunci cind sunt cuprinse intre ghilimele.

2.5 Exerciţii

Exerciţiul 1. De cîte tipuri sunt comenzile `UNIX`?

Exerciţiul 2. Ce este un *shell* `UNIX`?

Exerciţiul 3. Care sunt comenzile de *help*? Ce conţin paginile de manual `UNIX`?

Exerciţiul 4. Ce editoare de texte pentru `UNIX` cunoaşteţi? Dar compilatoare pentru limbajele C şi C++?

Exerciţiul 5. Care sunt comenzile ce oferă informaţii despre utilizatori?

Exerciţiul 6. Care sunt comenzile ce oferă informaţii despre terminale? Dar despre dată şi timp?

Exerciţiul 7. Care sunt comenzile ce oferă informaţii despre procesele din sistem?

Exerciţiul 8. Care sunt comenzile ce permit conectarea la un sistem `UNIX` pentru o sesiune de lucru? Dar pentru deconectare?

Exerciţiul 9. Cu ce comenzi putem scrie mesaje pe ecran, eventual destinate altor utilizatori?

Exerciţiul 10. Care sunt comenzile de arhivare a fişierelor?

Exerciţiul 11. Care sunt comenzile pentru poştă electronică, transfer de fişiere, navigare pe WWW, şi alte programe de INTERNET?

Exerciţiul 12. Care sunt comenzile de căutare de şabloane în fişiere?

Exerciţiul 13. Studiaţi cu ajutorul comenzii `man` opţiunile şi efectul tuturor comenzilor `UNIX` amintite în secţiunea 2.1.

Exerciţiul 14. Ce caracteristici au sistemele de fişiere din `UNIX`, ce le deosebesc de cele din `MS-DOS` şi `MS-Windows`?

Exerciţiul 15. De cîte tipuri sunt fişierele în `UNIX`?

Exerciţiul 16. Care este structura sistemului de fişiere `UNIX`?

Exerciţiul 17. Cu ce comandă se navighează prin sistemul de fişiere? Dar pentru vizualizarea unui director?

Exerciţiul 18. Cîte moduri de specificare a numelor de fişiere există în `UNIX`?

Exerciţiul 19. Ce înseamnă operaţie de montare a unui sistem de fişiere şi la ce este ea utilă? Care sunt comenzile pentru montare şi demontare de sisteme de fişiere?

Exerciţiul 20. Care sunt comenzile ce oferă informaţii despre terminale? Dar despre dată şi timp?

Exerciţiul 21. Cum se realizează protecţia accesului la fişiere în `UNIX`? Explicaţi drepturile de acces la fişiere.

Exerciţiul 22. Care sunt comenzile ce permit modificarea drepturilor de acces, a proprietarului şi a grupului proprietar ale unui fişier?

Exerciţiul 23. Care sunt comenzile pentru directoare?

Exerciţiul 24. Care sunt comenzile pentru operaţiile uzuale asupra fişierelor (*i.e.*, creare, ştergere, copiere, mutare, etc.) ?

Exerciţiul 25. Care sunt comenzile de căutare pentru fişiere?

Exerciţiul 26. Care sunt comenzile de afişare a fişierelor?

Exerciţiul 27. Cu ce comenzi putem obţine diverse informaţii despre fişiere?

Exerciţiul 28. Care sunt comenzile ce permit arhivarea şi comprimarea fişierelor?

Exerciţiul 29. Care sunt comenzile de căutare de şabloane în fişiere?

Exerciţiul 30. Studiaţi cu ajutorul comenzii `man` opţiunile şi efectul tuturor comenzilor `UNIX` amintite în secţiunea 2.2.

Exerciţiul 31. Ce rol are interpretorul de comenzi (*shell*-ul) `UNIX`?

Exerciţiul 32. Cîte *shell*-uri `UNIX` cunoaşteţi?

Exerciţiul 33. Care sunt formele generale de lansare în execuţie a unei comenzi? Dar pentru execuţie în *background*?

Exerciţiul 34. Care sunt formele de lansare în execuţie secvenţială, paralelă şi condiţională a unui grup de comenzi?

Exerciţiul 35. Care sunt cele trei forme de specificare a unui nume de fişier?

Exerciţiul 36. Care sunt şabloanele ce pot fi folosite pentru a specifica grupuri de nume de fişiere?

Exerciţiul 37. Care sunt perifericele I/O standard, şi ce se înţelege prin redirectarea acestora? Cum se realizează redirectarea lor pentru o anumită comandă?

Exerciţiul 38. Ce înseamnă înlănţuirea de comenzi, şi cum se realizează practic?

Exerciţiul 39. Care sunt fişierele de iniţializare, de configurare, şi respectiv de istoric ale *shell*-ului `bash`? Cum se pot refolosi informaţiile salvate în istoric?

Exerciţiul 40. Avem in directorul curent un fisier "tst", continind 2 linii de text:

```
!@#$%^&*()
({}<>$^***
```

Ce se va afisa pe iesirea standard in urma executiei comenzii:
`UNIX> cat tst|cut -d^ -f2|cut -d$ -f1|tail -1`

Exerciţiul 41. Avem in directorul curent trei fisiere cu numele 1, 2 si 3, continind fiecare cite o linie de text specificata mai jos. Ce va afisa pe iesirea standard comanda urmatoare:

```
UNIX> cat 2 2 1 2 3 3 | grep '#$@'| cut -f1 -d$ | tail 2 | cut -f1 -d@
```

Fisierul 1:	Fisierul 2:	Fisierul 3:
`%#^%#$@`	`#$%@^&#`	`#%#%@$%`

Exerciţiul 42. Să se scrie comanda care afişează numele şi `UID`-urile tuturor utilizatorilor sistemului.
(*Indicaţie*: folosiţi informaţiile din fişierul `/etc/passwd`.)

Exerciţiul 43. Să se scrie comanda care afişează numele şi `GID`-urile tuturor grupurilor de utilizatori ai sistemului.
(*Indicaţie*: folosiţi informaţiile din fişierul `/etc/group`.)

Exerciţiul 44. Să se scrie comanda care va scrie în fişierul `users-logati.txt`, *username*-urile tuturor utilizatorilor prezenţi în sistem la momentul execuţiei comenzii, în ordine alfabetică (şi unică).

Exerciţiul 45. Să se scrie comanda care afişează (în mod unic) toate *shell*-urile folosite de utilizatorii sistemului.
(*Indicaţie*: folosiţi informaţiile din fişierul `/etc/passwd`.)

Exerciţiul 46. Să se scrie comanda care verifică dacă există mai mult de 2 sesiuni deschise ale unui utilizator specificat şi în caz afirmativ să se afişeze un mesaj informativ.

Exerciţiul 47.* Testaţi ce se întîmplă cînd aveţi în directorul *home* ambele fişiere de iniţializare `.profile` şi `.bash_profile`. Cum sunt ele executate, în ce ordine sau care dintre ele? Ce se întîmplă dacă aveţi doar fişierul `.profile`, nu şi `.bash_profile`, sau invers?

Exerciţiul 48.* Care vor fi efectiv cele 3 dispozitive I/O standard în timpul execuţiei unei comenzi (*i.e.*, care vor fi efectele redirectărilor), pentru fiecare din liniile de comandă de mai jos?

```
UNIX> comanda 2>&1 >fisier
UNIX> comanda 2>>&1 >fisier
UNIX> comanda 2>&1 >>fisier
UNIX> comanda 2>>&1 >>fisier
UNIX> comanda >fisier 2>&1
UNIX> comanda >fisier 2>>&1
UNIX> comanda >>fisier 2>&1
UNIX> comanda >>fisier 2>>&1
```

Ce se întîmplă de fapt în fiecare caz? Se pierde ceva, şi dacă da, atunci ce anume?

Exerciţiul 49. Ce sunt procedurile *shell* (*script*-urile) ? Care sunt modurile de lansare a lor în execuţie?

Exerciţiul 50. Ce sunt variabilele de *shell*? Cum li se atribuie valori?

Exerciţiul 51. Cum se realizează referirea la valoarea unei variabile? Cîte forme de substituţie există?

Exerciţiul 52. Cum interpretează *shell*-ul construcţiile `` `...` `` sau `$(...)` ?

Exerciţiul 53. Care sunt variabilele de *shell* dinamice cu semnificaţie specială?

Exerciţiul 54. Ce înseamnă variabilă de mediu? Cum se realizează exportul variabilelor?

Exerciţiul 55. Care sunt variabilele de mediu predefinite?

Exerciţiul 56. Descrieţi sintaxa şi semantica structurilor repetitive (`for`, `while` şi `until`).

Exerciţiul 57. Descrieţi sintaxa şi semantica structurilor alternative (`if` şi `case`).

Exerciţiul 58. Care sunt principalele opţiuni ale comenzii `test`?

Exerciţiul 59. Ce realizează comenzile `break`, `continue`, `exit`, `wait`, `exec`, şi `trap`?

Exerciţiul 60. Scrieţi un *script* care să afişeze permisiunile tuturor fişierelor şi subdirectoarelor (recursiv) din directorul dat ca argument.
(*Indicaţie*: folositi comenzile `find` si `stat`.)

Exerciţiul 61. Ce se va afişa pe iesirea standard în urma execuţiei *script*-ului de mai jos?

```
#!/bin/bash
a=3
b=5
c=$a+$b
echo 'suma este $c' </dev/null
```

Exerciţiul 62. Ce se va afişa pe iesirea standard în urma execuţiei *script*-ului de mai jos?

```
#!/bin/true
arhiva=YES
case $1 in
  *.tar ) tar -vxf $1;;
  *.tgz|*.taz ) tar -vzxf $1;;
  *.gz|*.Z ) gunzip $1;;
  * ) arhiva=NO
esac
[ $arhiva != NO ] && echo "Ok" || echo 'Nu'
```

Exerciţiul 63. Fie scriptul următor:

```
#!/bin/shellulmeu
gata=0
until [ $gata != 0 -o $# == 0 ]
do
  echo $1 ; ls -lA $1
  echo Continuati? (y/n)
  read x
  [ x == n ] && gata=1
  shift
done
```

Ce va produce acest *script* dacă este apelat cu linia următoare:
`UNIX> cd ; .` *script* `mail /etc ./html`

Exerciţiul 64. Care este efectul *script*-ului de mai jos?

```
#!/bin/bash
F=0
for f in $(ls -Al /bin/f*)
do
  [ -f $f ] && F=$(expr $F + 1)
done
echo $F
```

Exerciţiul 65. Care este efectul *script*-ului următor?

```
#/bin/bash
nr=0
while [ $1 ]
do
   nr=$(expr $nr + 1)
   echo ${nr}: $1
   shift
done
echo "-----"
```

Exerciţiul 66. Fie *script*-ul `bash` de mai jos.

```
while test $# -ge 1 ; do
  case $1 in
    n ) N=$2; shift;;
    u ) U=$2; shift;;
  esac
  shift
done
echo -n $N && echo $U
```

Ce va afisa acest *script* pe iesirea standard daca se apeleaza cu linia urmatoare:
`UNIX> ./script n u u i`

Exerciţiul 67. Scrieţi un *script* care să efectueze calculul puterii n la m.

Exerciţiul 68. Scrieţi un *script* care să efectueze calculul recursiv al factorialului.

Exerciţiul 69. Scrieţi un *script* care să afişeze utilizatorii din anul 2 care au pagini *web* locale.
(*Indicaţie*: testaţi existenţa şi vizibilitatea pentru toată lumea a unuia dintre fişierele `index.htm` sau `index.html` sau `index.php`, care trebuie să se afle în subdirectorul `html` din directorul *home* al utilizatorului.)

Exerciţiul 70. Scrieţi un *script* care să simuleze comanda `du` : să afişeze suma (în octeţi şi în megaocteţi) a dimensiunilor fişierelor obişnuite din directorul curent.

Exerciţiul 71. Scrieţi un *script* care să decidă dacă un şir oarecare de caractere este număr.
(*Indicaţie*: folosiţi comanda `grep` cu expresii regulate ca argument.)

Exerciţiul 72. Scrieţi un *script* care să producă acelaşi efect cu comanda `ps`, dar fără a utiliza comanda `ps`; nu este necesară implementarea tuturor opţiunilor comenzii `ps`.
(*Indicaţie*: utilizati informatiile din directorul `/proc`.)

Exerciţiul 73. Script pentru automatizarea procesului de dezvoltare de programe C*
Scrieţi un *script* care să vă ajute la scrierea programelor în C, prin care să se automatizeze ciclul:
modificare sursă $\longrightarrow$ compilare $\longrightarrow$ testare (execuţie).

Cu alte cuvinte, *script*-ul va trebui să lanseze editorul de texte preferat pentru fişierul *program*`.c` specificat ca parametru în linia de comandă (sau citit de la tastatură, în caz contrar), apoi să interogheze utilizatorul dacă doreşte lansarea compilatorului şi în caz afirmativ s-o facă (fişierul executabil să aibă numele *program*, deci fără sufixul "`.c`"), apoi dacă sunt erori de compilare (lucru observabil prin erorile de compilare afişate de compilator) să reia ciclul de la editare (bineînţeles cu o pauză pentru ca utilizatorul să aibă timp să citească erorile afişate pe ecran), iar dacă nu sunt erori la compilare, să interogheze utilizatorul dacă doreşte testarea (execuţia) acelui program şi în caz afirmativ să execute acel fişier executabil rezultat prin compilare.
Deci la fiecare pas să fie o interogare a utilizatorului dacă doreşte să continue cu următorul pas!

(*Observaţie*: acest *script* îl veţi putea folosi la următoarele lecţii, cînd vom trece la programarea în C sub `Linux`.)

Partea II

Programare concurentă în Linux

În această a doua parte a manualului vom aborda programarea de sistem în limbajul de programare C pentru sistemele de operare din familia `UNIX`, în particular pentru `Linux`.

Mai precis, obiectivul principal al acestei părţi a manualului este acela de a vă învăţa conceptele fundamentale ale *programării concurente / paralele*, adică al scrierii de programe ce vor fi executate mai multe "în paralel" (*i.e.* concomitent, în acelaşi timp), într-un mediu concurenţial: pe parcursul execuţiei lor, programele *concurează* unele cu altele pentru resursele sistemului de calcul (procesor, memorie, periferice de intrare–ieşire), puse la dispoziţia lor şi gestionate de către sistemul de operare.

Am ales ca variantă de tratare a acestui subiect, al programării concurente, să utilizez sistemul `Linux` ca suport, limbajul C ca limbaj de programare, şi utilizarea în programe a interfeţei clasice (*i.e.*, în mod text) cu utilizatorul, deoarece, conform celor explicate în prefaţa manualului, acesta este cel mai adecvat cadru de predare al programării concurente, pentru că permite concentrarea atenţiei asupra aspectelor referitoare la execuţia mai multor programe în regim concurenţial de folosire a resurselor calculatorului, fără să ne distragă atenţia aspectele referitoare la interfaţa cu utilizatorul a programelor respective.

Primul capitol al acestei părţi a manualului tratează gestiunea fişierelor prin program şi accesul în mod concurent sau exclusiv la fişiere.

Al doilea capitol tratează gestiunea proceselor, pe lîngă conceptul de fişier, procesul fiind celălalt concept fundamental al `UNIX`-ului, ce se referă la execuţia unui program. Sunt prezentate operaţiile de bază cu procese: crearea proceselor, sincronizarea lor, "reacoperirea" lor (*i.e.*, încărcarea pentru execuţie a unui fişier executabil) şi tratarea excepţiilor prin intermediul semnalelor `UNIX`.

Al treilea capitol abordează mecanismele disponibile în `UNIX` pentru comunicaţia între procese, descriind amănunţit două astfel de mecanisme: canalele de comunicaţie interne (*pipe*-urile) şi canalele de comunicaţie externe (*fifo*-urile).

Bibliografie utilă pentru studiu individual suplimentar: [1], [5], [6], [7], [10].

Dezvoltarea aplicaţiilor C sub `Linux`

Introducere

Sistemul de operare `Linux`dispune de utilitare performante destinate dezvoltarii de aplicaţii C. Dintre acestea un rol aparte îl au:

- compilatorul de C, `gcc` (acronim ce provine de la *GNU C Compiler*);
- *link*-editorul, `ld` ;
- depanatorul, `gdb` (acronim ce provine de la *GNU DeBugger*) ;
- bibliotecarul (= utilitarul pentru construcţia bibliotecilor), `ar` ;
- utilitarul pentru construirea fisierelor proiect, `make` ;
- editorul `emacs` (tot o aplicaţie din proiectul GNU), cu puternice facilităţi de integrare cu compilatorul `gcc` şi depanatorul `gdb`.

Limbajul C oferă suportul cel mai adecvat dezvoltarii aplicaţiilor sub `Linux`. Acest lucru este motivat în principal de modul convenabil (din punctul de vedere al programatorului) în care se face accesul la serviciile sistemului de operare – funcţiile (apelurile) sistem. Aceste funcţii permit o multitudine de operaţii referitoarela lucrul cu fişiere, alocarea memoriei, controlul proceselor, etc. Mai mult, fiind scrise in C, funcţiile sistemului de operare au un mecanism de apelare comod, similar oricărei rutine scrisă de utilizator.

Crearea unei aplicaţii C simple

Dezvoltarea unei aplicaţii C presupune parcurgerea următoarelor etape:

1. *editarea textului sursă*

   Pentru aceasta se poate folosi un editor de texte de tipul `joe`, `vim`, `pico` sau `mcedit` (editorul intern al programului `mc`).
   Fişierul creat trebuie să aibă extensia "`.c`", extensie care prin convenţie este atribuită surselor scrise în limbajul C.

   Să presupunem că editǎm următorul fişier sursă C:

```
/*hello.c*/
#include <stdio.h>
int main()
{
  printf("Hello world!\n");
  return 0;
}
```

După editarea textului sursă, îl vom salva într-un fişier denumit `hello.c`.

2. *compilarea programului şi editarea de legături*

   Se face cu compilatorul `gcc` astfel:

   `UNIX> gcc hello.c`

   se va obţine fisierul executabil cu numele implicit `a.out`. Sau, cu

   `UNIX> gcc -o hello hello.c`

   se va obţine fisierul executabil cu numele `hello`.

3. *lansarea în execuţie*

   Se face fie specificînd numele complet (*i.e.*, calea absolută) a executabilului, de exemplu `$HOME/programe_c/hello`, fie specificînd numele prin cale relativă al executabilului, de exemplu, din directorul în care se găseşte ca director curent de lucru, îi putem specifica doar numele:

   `UNIX> ./hello`

Observaţie: comanda `gcc hello.c -o hello` semnifică compilarea şi *link*-editarea (*i.e.*, rezolvarea apelurilor de funcţii) fisierului sursă `hello.c`, generându-se un executabil al carui nume este specificat prin optiunea `-o`. De fapt, se execută în mod transparent următoarea secvenţă de operaţii:

- preprocesorul expandează *macro*-urile şi include fişierele *header* corespunzătoare;
- prin compilare, se generează cod obiect (*i.e.*, fişiere obiect, cu extensia "`.o`");
- editorul de legături `ld` rezolvă apelurile de funcţii şi creează executabilul din fişierele obiect.

Atenţie: aşadar compilatorul `gcc` apelează automat şi editorul de legături, de aceea etapa de editare a legăturilor poate fi transparentă pentru utilizator, însă acest lucru nu este obligatoriu.

Într-adevăr, secvenţa de mai sus poate fi parcursă şi manual:

`UNIX> gcc -E hello.c -o hello.cpp`
Efect: se execută doar preprocesarea, generându-se fişierul intermediar `hello.cpp`; examinaţi-i conţinutul!

`UNIX> gcc -x cpp-output -c hello.cpp -o hello.o`
Efect: se generează un fişier obiect utilizând un fişier sursă deja preprocesat.

`UNIX> gcc hello.o -o hello`
Efect: fişierul obiect este *link*-editat şi se creează executabilul.

Utilizarea compilatorului

Principalele funcţii ale compilatorului de C sunt următoarele :

- verifică corectitudinea codului sursă C;
- generează instrucţiuni în limbaj maşină pentru codul sursă C;
- grupează instrucţiunile într-un modul obiect care poate fi prelucrat de editorul de legături.

Lansarea compilatorului se fac cu comanda `gcc`, a cărei sintaxă este:

`UNIX> gcc -`*`optiuni`* *fisier_sursa1* `[` *fisier_sursa2* `... ]`

Efectul ei constă în următoarele:
Compilatorul creează câte un fişier obiect pentru fiecare fişier sursă, iar editorul de legături construieşte automat fişierul executabil din fişierele obiect create (dacă nu au fost erori la compilare şi dacă nu s-a specificat altfel).
Numele fişierelor obiect sunt aceleaşi cu numele surselor, dar au extensia "`.o`".
Numele implicit al fişierului executabil este `a.out`, dar aşa cum am văzut el poate fi modificat (cu opţiunea `-o`).

Cele mai frecvent utilizate opţiuni ale compilatorului sunt prezentate în tabelul de mai jos:

Opţiune	**Efect**
`-o` *nume*	fişierul executabil se va numi *nume* şi nu `a.out`
`-c`	suprimă editarea de legături, se creează doar fişiere obiect
`-w`	suprimă toate avertismentele şi mesajele de informare
`-Wall`	afişează toate avertismentele şi mesajele de informare
`-std`	generează avertismente pentru cod care nu este standard ANSI C

Alte opţiuni utile ale compilatorului:

1. *Includerea de fişiere header şi de biblioteci*:

Opţiune	Efect
`-I`*director*	fişierele *header* sunt căutate şi în directorul *director*
`-L`*director*	bibliotecile sunt căutate şi în directorul *director*
`-l`*nume*	*link*-editează biblioteca *nume*
`-static`	*link*-editează static

Exemplu:

```
UNIX> gcc myapp.c -I/home/so/include -L/home/so/libs -lmy \
-static -o myapp
```

Observaţie: în mod implicit, compilatorul `gcc` *link*-editează *dinamic* (*i.e.*, la executie), utilizând biblioteci partajate (*shared libraries*); extensia librariilor partajate este `.so`.
Prin comanda de mai sus, se cere *link*-editarea în mod *static* a bibliotecii cu numele `libmy.so`. Se garantează astfel executarea aplicaţiei şi pe un sistem pe care nu este instalată biblioteca în cauză, cu preţul creşterii dimensiunii finale a executabilului.

2. *Optimizarea codului*:
(optimizarea codului constituie o încercare de a îmbunătăţi performanţele programului. *Atenţie*: este vorba aici de "*compiler magic*" şi nu de eficienţa algoritmilor utilizaţi în faza de proiectare a aplicaţiei!)

Opţiune	Efect de optimizare
`-O0`	nu optimizează codul generat
`-O`n	specifica un nivel n de optimizare, cu $n = 1, 2, 3$

Pentru uzul general se recomandă utilizarea opţiunii `-O2`.

3. *Depanarea programului*

Opţiune	Efect
`-g`	include în executabil informaţii standard utile la depanare
`-ggdb`	include în executabil informaţii utilizabile doar de `gdb`

Interpretarea mesajelor compilatorului

Mesajele compilatorului au următoarea formă:

nume_fisier: nr_linie : severitate : mesaj

Severitatea mesajului specifică gravitatea erorii. Ea poate fi:

- *informativă* : pentru o acţiune benignă;
- *avertisment* : pentru un cod care nu este incorect dar poate fi îmbunătăţit;

- *eroare* : pentru un cod incorect; compilarea continuă, dar nu este creat fişier obiect;
- *eroare fatală* : pentru un cod a cărui incorectitudine nu poate fi tolerată; compilarea se opreşte şi bineînţeles nu este creat fişierul obiect.

Depanatorul GNU

Depanatorul `gdb` (*GNU db*) este programul de depanare utilizat în `Linux`.

Depanarea necesită compilarea codului sursă cu optiunea `-g` (creându-se astfel o tabelă de simboluri îmbogăţită); informatii specifice `gdb` se includ in executabil, la compilare, cu opţiunea `-ggdb`.

Pornirea depanatorului se realizează cu comanda:

`UNIX>` `gdb -q` *myprog* `[`*corefile*`]`

Utilizarea fisierului *core* este optionala; el imbunatateste funcţionalităţile depanatorului `gdb`. Optiunea `-q` (*quiet*) inlatura mesajele privind licentierea. O optiune utila este si `-d`, cu care se poate stabili directorul de lucru.

După ce am pornit depanatorul cu comanda de mai sus, el afişează un prompter de forma `(gdb)`, la care aşteaptă introducerea de comenzi.

Pentru a porni sesiunea de depanare, se utilizeaza comanda *run* (accepta si argumente, pe care le paseaza programului):

`(gdb) run [`*parametri*`]`

În plus, valoarea variabilei `$SHELL` determina valorile variabilelor de mediu in care se executa programul. Modificarea argumentelor din linia de comandă a programului şi a valorilor variabilelor de mediu, dupa ce a fost pornita sesiunea de depanare, se realizeaza cu comenzile:

`(gdb) set args` *arg1 arg2 ...*

şi respectiv

`(gdb) set environment` *env1 env2 ...*

Pentru a inspecta secvenţa de execuţie a funcţiilor programului, se foloseşte comanda `backtrace`.

Cu comanda `list [m,n]` se afiseaza portiuni din codul sursă.

Valoarea unei expresii legale este afisata cu comanda `print`; de exemplu:

```
(gdb) print i          //valoarea varabilei i
(gdb) print array[i]   //valoarea componentei i din tabloul array
(gdb) print array@5    //adresele de memorie ale primelor 5 locatii
(gdb) print array[0]@5 //valorile pastrate in primele 5 locatii
```

Observaţie: daca `print` afiseaza valoarea unei expresii, eventualele efecte colaterale (*side-effects*) ale expresiei se regasesc in programul depanat.

Valoarea unei variabile se poate modifica cu comanda:

```
(gdb) set variable i=10
```

Tipul unei expresii este afisat de catre comanda `whatis`; de exemplu:

```
(gdb) whatis i
(gdb) whatis array
(gdb) whatis function_name
```

Notă: `whatis` afiseaza tipul unui pointer; informatii despre tipul de baza al unui pointer se obtin cu comanda `ptype`.

Comanda `break` creează un punct de întrerupere (*breakpoint*) al execuţiei programului:

```
(gdb) break <filename:line_number>
(gdb) break <filename:function_name>
```

Se pot crea şi puncte de întrerupere condiţionată de valoarea (nenulă) a unei expresii:

```
(gdb) break 25 if i==15
```

Continuarea executiei, dupa atingerea unui punct de intrerupere, se realizeaza cu comanda `continue`. Pentru a sterge un punct de intrerupere se utilizeaza `delete`; punctul de intrerupere poate fi doar dezafectat daca se utilizeaza `disable` (respectiv `enable` pentru reactivare). Informatii despre punctele de intrerupere setate se obtin cu comanda `info breakpoints`.

Comanda `step` executa cate o instructiune. La intalnirea unui apel de functie, `step` paseste in respectiva functie; pentru a executa intr-un pas o functie se utilizeaza comanda `next`.

Comanda `call` *function_name*`(`*arguments*`)` apeleaza si executa o functie. Comanda `finish` termina executia functiei curente si afiseaza valoarea returnata, iar comanda `return` *value* opreste executia functiei curente, returnând valoarea precizata.

Capitolul 3

Gestiunea fişierelor

3.1 Primitivele I/O pentru lucrul cu fişiere

1. **Introducere**
2. **Principalele primitive I/O**
3. **Funcţiile I/O din biblioteca standard de C**

3.1.1 Introducere

In `UNIX`/`Linux` functiile utilizate de sistemul de gestiune a fisierelor pot fi clasificate in urmatoarele categorii:

1. primitive de creare de noi fisiere: `mknod`, `creat`, `mkfifo`, etc.;
2. primitive de accesare a fisierelor existente: `open`, `read`, `write`, `lseek`, `close`;
3. primitive de manipulare a *i*-nodului: `chdir`, `chroot`, `chown`, `chmod`, `stat`, `fstat`;
4. primitive de implementare a *pipe*-urilor: `pipe`, `dup`;
5. primitive de extindere a sistemului de fisiere: `mount`, `umount`;
6. primitive de schimbare a structurii sistemului de fisiere: `link`, `unlink`.

In literatura de specialitate despre `UNIX`, aceste functii sunt denumite **apeluri sistem** (*system calls*).

O observatie generala este aceea ca in caz de eroare toate aceste primitive returneaza valoarea `-1`, precum si un numar de eroare in variabila globala `errno`.

3.1.2 Principalele primitive I/O

In continuare vom trece in revista principalele apeluri sistem referitoare la fisiere:

1) Crearea de fisiere ordinare:

Se face cu ajutorul functiei `creat`. De asemenea, se mai poate face si cu functia `open`, folosind un anumit parametru. Interfata functiei `creat` este urmatoarea:

`int creat(char*` *nume_cale*`, int` *perm_acces*`)`

unde:
– *nume_cale* = numele fisierului ce se creeaza;
– *perm_acces* = drepturile de acces pentru fisierul ce se creeaza;
iar valoarea `int` returnata este descriptorul de fisier deschis, sau `-1` in caz de eroare.

Efect: in urma executiei functiei `creat` se creeaza fisierul specificat (sau, in caz ca deja exista acel fisier, atunci el este trunchiat la zero, pastrindu-i-se drepturile de acces pe care le avea) si este deschis in scriere.

2) Transmiterea mastii drepturilor de acces la crearea unui fisier:

Se face cu ajutorul functiei `umask`.

3) Controlul dreptului de acces la un fisier:

Se face cu ajutorul functiei `access`. Interfata functiei `access` este urmatoarea:

`int access(char*` *nume_cale*`, int` *drept*`)`

unde:
– *nume_cale* = numele fisierului;
– *drept* = dreptul de acces ce se verifica, si anume poate lua urmatoarele valori:
- 1 - se verifica daca este setat dreptul de executie;
- 2 - se verifica daca este setat dreptul de scriere;
- 4 - se verifica daca este setat dreptul de citire;
- 0 - se verifica daca fisierul exista;

iar valoarea `int` returnata este `0`, daca accesul verificat este permis, respectiv `-1` in caz de eroare.

4) Deschiderea unui fisier:

Se face cu ajutorul functiei `open`. Interfata functiei `open` este urmatoarea:

`int open(char*` *nume_cale*`, int` *tip_desch*`, int` *perm_acces*`)`

unde:
– *nume_cale* = numele fisierului ce se deschide;
– *perm_acces* = drepturile de acces pentru fisier (utilizat numai in cazul crearii acelui fisier);
– *tip_desch* = specifica tipul deschiderii, fiind o combinatie (*i.e.*, disjunctie logica pe biţi) a urmatoarelor constante simbolice:

- `O_RDONLY` - deschidere pentru citire;
- `O_WRONLY` - deschidere pentru scriere;
- `O_RDWR` - deschidere pentru citire si scriere;
- `O_APPEND` - pozitioneaza cursorul la sfirsitul fisierului, astfel incit orice scriere in el se adauga la sfirsitul lui;
- `O_CREAT` - creare fisier (daca deja exista, este trunchiat la zero);
- `O_TRUNC` - daca fisierul exista, este trunchiat la zero;
- `O_EXCL` - open exclusiv: daca fisierul exista si este setat `O_CREAT`, atunci se returneaza eroare;

iar valoarea `int` returnata este descriptorul de fisier deschis, sau `-1` in caz de eroare.

5) Inchiderea unui fisier:

Se face cu ajutorul functiei `close`. Interfata functiei `close` este urmatoarea:

`int close(int` *df*`)`

unde:
– *df* = descriptorul de fisier (cel returnat de functia `open`);
iar valoarea `int` returnata este `0`, daca inchiderea a reusit, respectiv `-1` in caz de eroare.

6) Citirea dintr-un fisier:

Se face cu ajutorul functiei `read`.
Interfata functiei `read` este urmatoarea:

`int read(int` *df*`, char*` *buffer*`, unsigned` *nr_oct*`)`

unde:
– *df* = descriptorul fisierului din care se citeste (cel returnat de functia `open`);
– *buffer* = adresa de memorie la care are loc depunerea octetilor cititi;

– *nr_oct* = numarul de octeti de citit din fisier;
iar valoarea `int` returnata este numarul de octeti efectiv cititi, daca citirea a reusit (chiar si partial), si `-1` in caz de eroare.

Observatie: numarul de octeti efectiv cititi poate fi inclusiv 0, daca la inceputul citirii fisierul este pe pozitia `EOF`.

Efect: prin executia functiei `read` se incearca citirea a *nr_oct* octeti din fisier incepind de la cursor (*i.e.*, pozitia curenta in fisier). Cererea de citire se termina intr-una din situatiile urmatoare:
a) s-au citit efectiv *nr_oct* octeti din fisier;
b) fisierul nu mai contine date;
c) apare o eroare in citirea datelor din fisier sau in copierea lor la adresa de memorie specificata.
La sfirsitul citirii cursorul va fi pozitionat pe urmatorul octet dupa ultimul octet efectiv citit.

7) Scrierea intr-un fisier:

Se face cu ajutorul functiei `write`. Interfata functiei `write` este urmatoarea:

`int write(int` *df*`, char*` *buffer*`, unsigned` *nr_oct*`)`

unde:
– *df* = descriptorul fisierului din care se citeste (cel returnat de functia `open`);
– *buffer* = adresa de memorie al carei continut se scrie in fisier;
– *nr_oct* = numarul de octeti de scris in fisier;
iar valoarea `int` returnata este numarul de octeti efectiv scrisi, daca scrierea a reusit (chiar si partial), si `-1` in caz de eroare.

Efect: prin executia functiei `write` se incearca scrierea a *nr_oct* octeti incepind de la cursor (*i.e.*, pozitia curenta in fisier).
La sfirsitul scrierii cursorul va fi pozitionat pe urmatorul octet dupa ultimul octet efectiv scris.

8) Pozitionarea cursorului (*i.e.*, ajustarea deplasamentului) intr-un fisier:

Se face cu ajutorul functiei `lseek`. Interfata functiei `lseek` este urmatoarea:

`long lseek(int` *df*`, long` *val_ajust*`, int` *mod_ajust*`)`

unde:
– *df* = descriptorul fisierului ce se pozitioneaza;
– *val_ajust* = valoarea de ajustare a deplasamentului;
– *mod_ajust* = modul de ajustare, indicat dupa cum urmeaza:
- 0 - ajustarea se face fata de inceputul fisierului;

- 1 - ajustarea se face fata de deplasamentul curent;
- 2 - ajustarea se face fata de sfirsitul fisierului;

iar valoarea `int` returnata este noul deplasament in fisier (întotdeauna, fata de inceputul fisierului), sau `-1` in caz de eroare.

Maniera uzuală de prelucrare a unui fişier constă în următoarele: deschiderea fişierului, o buclă de parcurgere a lui cu citire/scriere, şi apoi închiderea sa (toate aceste operaţii se fac cu ajutorul apelurilor sistem amintite mai sus). Pentru exemplificare, vom ilustra în continuare un program C care transformă un fişier text `MS-DOS` în fişier text `UNIX`.

Exemplu. Se cunoaşte faptul că fişierele text din `MS-DOS` diferă de cele din `UNIX` prin modul de reprezentare a caracterului `newline` (sfîrşit de linie): în `MS-DOS` acesta se reprezintă printr-o secvenţă de 2 caractere `ASCII` (*i.e.*, 2 octeţi): primul este caracterul `CR` (ce are codul 13), iar al doilea este caracterul `LF` (ce are codul 10), pe cînd în `UNIX` el se reprezintă doar printr-un singur caracter `ASCII` (*i.e.*, 1 octet): caracterul `LF` (cu codul 10).
Ne propunem să scriem un filtru `MS-DOS`→`UNIX`, adică un program C care transformă un fişier text `MS-DOS` în fişier text `UNIX`.
Vom proceda în felul următor: vom parcurgem fişierul de intrare specificat ca parametru în linia de comandă şi vom înlocui fiecare apariţie a secvenţei de caractere `CR+LF` cu caracterul `LF`, salvînd rezultatul într-un fişier de ieşire specificat tot ca parametru în linia de comandă.
Un program care realizează acest lucru este următorul:

```
/*
  File: d2u.c  (efect: filtru dos 13,10 -> unix 10)
*/
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>

extern int errno;

int main(int argc,char *argv[])
{
 char bufin[512],bufout[600];
 int i,m,in,out;
 char *pin,*pout;

 if (argc!=3)
   { printf("\nUsage:  %s  file[dos]  outfile\n\n",v[0]);
     exit(1);
   }

 in=open(argv[1],O_RDONLY);
 printf("Outputfile: %s  \n",argv[2]);
 out=creat(argv[2], O_WRONLY | 0644 );

 while ( (m=read(in,bufin,512)) !=0)
```

```
 {
    pin=bufin;
    pout=bufout;
    for(i=0; i<m; i++,pin++)
    {    /* if (*pin=='\015' && *(pin+1)=='\012')
        { pin++;   *pout='\012';}            */
     if (*pin!='\015')
   { *pout=*pin;  pout++; }
    }
    *pout='\0';
    i=write(out,bufout,strlen(bufout));
    if (i==-1)
      { printf("errno = %d ->",errno); perror(0);
      }
  }
  printf("Terminat\n");
  close(in);
  close(out);
  return 0;
}
```

9) Duplicarea unui descriptor de fisier:

Se face cu ajutorul functiei `dup`. Mai exista inca o functie asemanatoare, si anume functia `dup2`.

10) Controlul operatiilor I/O:

Se face cu ajutorul functiei `fcntl`.

11) Obtinerea de informatii continute de *i*-nodul unui fisier:

Se face cu ajutorul functiilor `stat` sau `fstat`.

12) Stabilirea/eliberarea unei legaturi pentru un fisier:

Se face cu ajutorul functiei `link`, respectiv `unlink`.

13) Schimbarea drepturilor de acces la un fisier:

Se face cu ajutorul functiei `chmod`.

14) Schimbarea proprietarului unui fisier:

Se face cu ajutorul functiei `chown`.

15) Crearea fisierelor *pipe* (*i.e.*, canale fara nume):

Se face cu ajutorul functiei `pipe`.

16) Crearea fisierelor *fifo* (*i.e.*, canale cu nume):

Se face cu ajutorul functiei `mkfifo`. Interfata functiei `mkfifo` este urmatoarea:

`int mkfifo(char*` *nume_cale*`, int` *perm_acces*`)`

unde:
– *nume_cale* = numele fisierului *fifo* ce se creeaza;
– *perm_acces* = drepturile de acces pentru acesta;
iar valoarea `int` returnata este `0` in caz de succes, sau `-1` in caz de eroare.

Efect: in urma executiei functiei `mkfifo` se creeaza fisierul *fifo* specificat, cu drepturile de acces specificate.

17) Montarea/demontarea unui sistem de fisiere:

Se face cu ajutorul functiei `mount`, respectiv `umount`.

18) Crearea/stergerea unui director:

Se face cu ajutorul functiei `mkdir`, respectiv `rmdir`.

19) Aflarea directorului curent de lucru:

Se face cu ajutorul functiei `getcwd`.

20) Schimbarea directorului curent:

Se face cu ajutorul functiei `chdir`.

21) Prelucrarea fisierelor dintr-un director:

Lucrul cu directoare decurge asemanator ca cel cu fisiere, tot o bucla de forma: deschidere, citire/scriere, inchidere. Se folosesc structurile de date si functiile urmatoare:

```
DIR  * dd; // descriptor de director deschis
struct dirent * de; // intrare in director
// deschiderea directorului
if( ( dd = opendir( nume_director)) == NULL)
{
 ...  // trateaza eroarea
}
// prelucrarea secventiala a tuturor intrarilor din director
while( ( de = readdir( dd)) != NULL)
```

```
{
 ...  // prelucreaza intrarea curenta, avind numele dat de cimpul: de->d_name
}
// inchiderea directorului
closedir(dd);
```

22) Crearea de fisiere speciale:

Se face cu ajutorul functiei `mknod`. Interfata functiei `mknod` este urmatoarea:

`int mknod(char*` *nume_cale*`, int` *perm_acces*`, int` *disp*`)`

unde:
– *nume_cale* = numele fisierului ce se creeaza;
– *perm_acces* = tipul fisierului si drepturile de acces pentru acesta;
– *disp* = dispozitivul specificat printr-un numar intreg (numarul minor si numarul major al dispozitivului);
iar valoarea `int` returnata este descriptorul de fisier deschis, sau `-1` in caz de eroare.

Efect: in urma executiei functiei `mknod` se creeaza fisierul cu tipul specificat (sau, in caz ca deja exista acel fisier, atunci el este trunchiat la zero, pastrindu-i-se drepturile de acces pe care le avea) si este deschis in scriere.

Observatie: aceasta este o functie generala ce permite crearea oricarui tip de fisier `UNIX/Linux`. De fapt, toate celelalte functii de creare de fisiere amintite mai sus, *i.e.* functiile `creat` (sau `open(...O_CREAT...)`), `mkdir`, `mkfifo`, etc., apeleaza la rindul lor functia `mknod` cu un parametru corespunzator tipului de fisier dorit a se crea.

3.1.3 Funcţiile I/O din biblioteca standard de C

Pe lingă apelurile sistem amintite mai sus care permit prelucrarea unui fişier în maniera uzuală: deschidere, buclă de parcurgere cu citire/scriere, şi apoi închidere (*i.e.*, primitivele `open`, `read`, `write`, `close`), mai există un set de funcţii I/O din biblioteca standard de C (cele din `stdio.h`), care permit şi ele prelucrarea unui fişier în maniera uzuală:
– `fopen` = pentru deschidere;
– `fread`, `fwrite` = pentru citire, respectiv scriere binară;
– `fscanf`, `fprintf` = pentru citire, respectiv scriere formatată;
– `fclose` = pentru închidere;

Nu voi mai aminti prototipul acestor funcţii de bibliotecă, deoarece le cunoaşteţi deja de la limbajul C din anul I. Vă voi reaminti doar faptul că aceste funcţii de bibliotecă lucrează *buffer*-izat, cu *stream*-uri I/O, iar descriptorii de fişiere utilizaţi de ele nu mai sunt de tip `int`, ci sunt de tip `FILE*`.

Practic, aceste funcţii I/O de nivel înalt folosesc un *buffer* (*i.e.*, o zonă de memorie cu rol de *cache*) pentru operaţiile de citire/scriere.
Un apel de scriere nu va scrie direct pe disc, ci doar va copia datele de scris în *buffer*. Acesta va fi "golit" (*i.e.*, conţinutul său va fi scris pe disc) abia în momentul în care se umple, sau dacă se întîlneşte caracterul `newline`. O altă posibilitate de a forţa "golirea" *buffer*-ului pe disc înainte de a se umple, este de a apela funcţia de bibliotecă `fflush`.
Observaţie: scrierea efectivă pe disc a conţinutului *buffer*-ului se face cu ajutorul apelului sistem `write`.
Asemănător se petrec lucrurile şi la citire: o operaţie de citire va citi date direct din *buffer*, dacă sunt îndeajuns, sau, în caz contrar, dacă *buffer*-ul s-a golit, el va fi "umplut" printr-o singură operaţie de citire de pe disc, chiar dacă funcţia de bibliotecă solicitase citirea unei cantităţi mai mici de date. *Observaţie*: citirea efectivă de pe disc în *buffer* se face cu ajutorul apelului sistem `read`.

Deosebirea între aceste perechi de funcţii constă în faptul că primele (*i.e.*, `open`, ... etc.) sunt de nivel scăzut, lucrînd la nivelul nucleului sistemului de operare (sunt apeluri sistem, implementate în nucleu), pe cînd ultimele (*i.e.*, `fopen`, ... etc.) sunt de nivel înalt, fiind funcţii de bibliotecă (implementate cu ajutorul celor de nivel scăzut).

3.2 Accesul concurent/exclusiv la fişiere în `UNIX`: blocaje pe fişiere

1. **Introducere**
2. **Blocaje pe fişiere. Primitivele folosite**
3. **Fenomenul de interblocaj. Tehnici de eliminare a interblocajului**

3.2.1 Introducere

`UNIX`-ul fiind un sistem multi-tasking, in mod uzual este permis accesul concurent la fisiere, adica mai multe procese pot accesa "simultan" in citire şi/sau scriere un acelasi fisier, sau chiar o aceeasi inregistrare dintr-un fisier.

Acest acces concurent ("simultan") la un fisier de catre procese diferite poate avea uneori efecte nedorite (ca de exemplu, distrugerea integritatii datelor din fisier), si din acest motiv s-au implementat in `UNIX` mecanisme care sa permita accesul exclusiv (adica un singur proces are permisiunea de acces) la un fisier, sau chiar la o anumita inregistrare dintr-un fisier.

3.2.2 Blocaje pe fişiere. Primitivele folosite

In continuare vom exemplifica prin citeva programe modul de acces concurent/exclusiv la fisiere, si vom trece in revista apelurile sistem folosite pentru accesul exclusiv la fisiere.

Urmatorul program este un exemplu de asemenea acces concurent la fisiere:

Exemplu. Iată sursa programului `access1.c` (programul acces versiunea 1.0):

```
/*
  File: access1.c (versiunea 1.0)
*/
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

extern int errno;

int main(int argc, char* argv[])
{
  int fd;
  char ch;

  if(argv[1] == NULL)
  {
    fprintf(stderr,"Trebuie apelat cu cel putin un parametru.\n");
    exit(1);
  }

  if( (fd=open("fis.dat", O_RDWR)) == -1)
  {  /* trateaza cazul de eroare ... */
    fprintf(stderr,"Nu pot deschide fisierul fis.dat: %s\n",strerror(errno));
    exit(2);
  }

  /* parcurgerea fisierului caracter cu caracter pina la EOF */
  while( read(fd,&ch,1) != 0)
  {
    if(ch == '#')
    {
      lseek(fd,-1L,1);
      sleep(10);
      write(fd,argv[1],1);
      printf("Terminat. S-a inlocuit primul # gasit [ProcesID:%d].\n",getpid());
      return 0;
    }
  }

  printf("Terminat. Nu exista # in fisierul dat [ProcesID:%d].\n",getpid());
  return 0;
}
```

Compilarea programului se va face cu comanda:
`UNIX> gcc access1.c -o access1`

Apoi creati un fisier `fis.dat` care sa contina urmatoarea linie de text:

```
abc#def#ghi#jkl
```

acest lucru putindu-l realiza, de exemplu, cu comanda:
`UNIX> echo abc#def#ghi#jkl > fis.dat`

Observatie: sa folositi aceasta comanda inaintea fiecarei executii a programului de mai sus, deoarece fisierul `fis.dat` va fi modificat la executia programului si va trebui sa faceti mai multe executii pe acelasi fisier original.

Dupa cum se observa din codul sursa, efectul acestui program este urmatorul: înlocuieşte prima aparitie a caracterului '#'pe care o gaseste în fisierul `fis.dat` cu primul caracter din primul argument din linia de comanda.
Spre exemplificare, executati comenzile:

```
UNIX> access1 500 6 salut
Terminat.  S-a inlocuit primul # gasit [ProcesID:9055].
UNIX> cat fis.dat
abc5def#ghi#jkl
```

Observatie: Nu uitati sa refaceti originalul `fis.dat` inaintea fiecarei executii, cu comanda `echo` de mai sus.

Sa vedem acum ce se intimpla daca lansam concurent două procese care sa execute acest program, lucru realizat prin comanda:

```
UNIX> access1 1 & access1 2 &
Terminat.  S-a inlocuit primul # gasit [ProcesID:9532].
Terminat.  S-a inlocuit primul # gasit [ProcesID:9533].
```

iar apoi putem vizualiza rezultatul execuţiei cu comanda:
`UNIX> cat fis.dat`

Probabil va asteptati ca dupa executie fisierul `fis.dat` sa arate cam asa:

`abc1def2ghi#jkl` sau `abc2def1ghi#jkl`

în funcţie de care dintre cele 2 procese a apucat primul sa suprascrie primul caracter '#'din acest fisier, celuilalt proces raminindu-i al doilea caracter '#'pentru a-l suprascrie.

Cu toate acestea, oricite executii s-ar face, intotdeauna se va obtine rezultatul urmator:

`abc1def#ghi#jkl` sau `abc2def#ghi#jkl`

Explicatie: daca cititi cu atentie sursa programului `access1.c`, veti constata ca exista o asteptare de 10 secunde intre momentul depistarii primei inregistrari din fisier care este '#'si momentul suprascrierii acestei inregistrari cu alt caracter (si anume cu primul caracter din primul argument din linia de comanda). Din acest motiv ambele procese se vor opri pe aceeasi inregistrare pentru a o suprascrie (dupa 10 secunde de la depistarea ei). Procesul care suprascrie ultimul va fi deci cel care determina rezultatul final.

Concluzie: acest exemplu ilustreaza ce se intimpla cind se acceseaza un fisier de catre mai multe procese in mod concurent (fara blocaj).

Tocmai pentru a evita asemenea fenomene ce nu sunt de dorit in anumite situatii, sistemul `UNIX` pune la dispozitie un mecanism de *blocare* (*i.e.* de punere de "*lacăte*") pe portiuni de fisier pentru acces exclusiv. Prin acest mecanism se defineste o zona de *acces exclusiv* la fisier (sau o "*secţiune critică*", cum mai este denumita in limbajul programarii paralele). O asemenea portiune nu va putea fi accesata in mod concurent de mai multe procese pe toata durata existentei blocajului.

Pentru a pune un blocaj (*lacat*) pe fisier trebuie utilizată urmatoarea structura de date:

```
struct flock  /* este definita in fisierul header fcntl.h */
{
  short l_type;    /* indica tipul blocarii */
  short l_whence;  /* indica pozitia relativa (originea) */
  long l_start;    /* indica pozitia in raport cu originea */
  long l_len;      /* indica lungimea portiunii blocate */
  int l_pid;
}
```

unde:
– cîmpul `l_type` indică tipul blocarii, putînd avea ca valoare una dintre constantele:
- `F_RDLCK` : blocaj in citire;
- `F_WRLCK` : blocaj in scriere;
- `F_UNLCK` : deblocaj (se inlatura lacatul);

– cîmpul `l_whence` indică pozitia relativa (originea) in raport cu care este interpretat cimpul `l_start`, putînd avea ca valoare una dintre constantele:
- `SEEK_SET = 0` : originea este BOF(=begin of file);
- `SEEK_CUR = 1` : originea este CURR(=current position in file);
- `SEEK_END = 2` : originea este EOF(=end of file);

– cîmpul `l_start` indică pozitia (*i.e.*, *offset*-ul in raport cu originea `l_whence`) de la care incepe zona blocata. *Observatie*: trebuie sa fie negativ pentru `l_whence = 2`.
– cîmpul `l_len` indică lungimea in octeti a portiunii blocate
– cîmpul `l_pid` este gestionat de functia `fcntl` care pune blocajul, fiind utilizat pentru a memora `PID`-ul procesului proprietar al acelui lacat. *Observatie*: are sens consultarea acestui cîmp doar atunci cind functia `fcntl` se apeleaza cu parametrul `F_GETLK`.

Pentru a pune lacatul pe fisier, dupa ce s-au completat cimpurile structurii de mai sus, trebuie apelată functia `fcntl`. Interfata functiei `fcntl` este urmatoarea:

```
#include <fcntl.h>

int fcntl(int fd, int mod, struct flock* sfl)
```

unde:
– *fd* = descriptorul de fisier deschis pe care se pune lacatul;
– *sfl* = adresa structurii `flock` ce defineste acel lacat;
– *mod* = indica modul de punere, putind lua una dintre valorile:

- `F_SETLK` :

  permite punerea unui lacat pe fisier (în citire sau în scriere, functie de tipul specificat in structura `flock`). In caz de esec se seteaza variabila `errno` la valoarea `EACCES` sau la `EAGAIN`;

- `F_GETLK` :

  permite extragerea informatiilor despre un lacat pus pe fisier;

- `F_SETLKW` :

  permite punerea/scoaterea blocajelor in mod "blocant", adica se asteapta (*i.e.*, functia nu returneaza) pina cind se poate pune blocajul. Posibile motive de asteptare: se incearca blocarea unei zone deja blocate de un alt proces, ş.a.

Indicaţie: a se citi neaparat paginile de manual despre functia `fcntl` si structura `flock`.

Observaţii:

1. Cîmpul `l_pid` din structura `flock` este actualizat de functia `fcntl`;
2. Blocajul este scos automat daca procesul care l-a pus fie inchide fisierul, fie isi termina executia;
3. Scoaterea (deblocarea) unui segment dintr-o portiune mai mare anterior blocata poate produce doua segmente blocate.
4. Blocajele (lacatele) nu se transmit proceselor fii in momentul crearii acestora cu functia `fork`. Motivul: fiecare lacat are in structura `flock` asociata `PID`-ul procesului care l-a creat (si care este deci proprietarul lui), iar procesele fii au, bineinteles, `PID`-uri diferite de cel al parintelui.

Revenind la exemplul nostru, pentru a obtine ca dupa executia celor doua procese fisierul `fis.dat` sa arate în felul următor:

`abc1def2ghi#jkl` sau `abc2def1ghi#jkl`

va trebui sa rescriem programul pentru a folosi acces exclusiv la fisier prin intermediul lacatelor, obtinind astfel a doua versiune a programului nostru.

Iată sursa programului `access2.c` (programul acces versiunea 2.0):

```
/*
  File: access2.c (versiunea 2.0)
*/
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

extern int errno;

int main(int argc, char* argv[])
{
  int fd,codblocaj;
  char ch;
  struct flock lacat;

  if(argv[1] == NULL)
  {
    fprintf(stderr,"Trebuie apelat cu cel putin un parametru.\n");
    exit(1);
  }

  if( (fd=open("fis.dat", O_RDWR)) == -1)
  {  /* trateaza cazul de eroare ... */
    perror("Nu pot deschide fisierul fis.dat deoarece ");
    exit(2);
  }

  /* pregateste lacat pe fisier */
  lacat.l_type   = F_WRLCK;
  lacat.l_whence = SEEK_SET;
  lacat.l_start  = 0;
  lacat.l_len    = 1; /* aici se poate pune orice valoare, inclusiv 0,
    deoarece pentru problema noastra nu conteaza lungimea zonei blocate.*/

  /* Incercari repetate de punere a lacatului pina cind reuseste */
  while( ((codblocaj=fcntl(fd,F_SETLK,&lacat)) == -1)
         && ((errno==EACCES)||(errno==EAGAIN))         )
  {
    fprintf(stderr, "Blocaj imposibil [ProcesID:%d].\n", getpid());
    perror("\tMotivul");
    sleep(6);
  }
  if(codblocaj == -1)
  {
    fprintf(stderr,"Eroare unknown la blocaj [ProcesID:%d].\n", getpid());
    perror("\tMotivul");
```

```
    exit(3);
  }
  else
    printf("Blocaj reusit [ProcesID:%d].\n", getpid());


  /* parcurgerea fisierului caracter cu caracter pina la EOF */
  while( read(fd,&ch,1) != 0)
  {
    if(ch == '#')
    {
      lseek(fd,-1L,1);
      sleep(10);
      write(fd,argv[1],1);
      printf("Terminat. S-a inlocuit primul # gasit [ProcesID:%d].\n",getpid());
      return 0;
    }
  }

  printf("Terminat. Nu exista # in fisierul dat [ProcesID:%d].\n",getpid());
  return 0;
}
```

Compilarea programului se va face cu comanda:
`UNIX> gcc access2.c -o access2`

Dacă faceţi mai multe executii, lansand concurent două procese care sa execute acest program, lucru realizat prin comanda:
`UNIX> access2 1 & access2 2 &`
(nu uitati sa refaceti fisierul original `fis.dat` la fiecare executie), veti observa ca obtinem rezultatul scontat.

Explicatie: dintre cele doua procese, doar unul va reusi sa puna lacatul, iar celalalt proces va astepta pina cind va reusi sa puna lacatul. Aceasta se va intimpla abia cind primul proces se va termina (si deci va fi scos lacatul pus de catre el). Mai mult, daca va uitati la pauzele puse in program prin apelurile functiei `sleep`, veti intelege de ce la fiecare executie va apare exact de 2 ori mesajul:
`Blocaj imposibil [Proces:...].`
(*Motivul*: $2 = \min\{k \text{ numar intreg} \mid 6*k > 10\}$. Ginditi-va la justificarea acestei formule!!!)

Observaţie: în programul anterior apelul de punere a lacatului era neblocant (*i.e.*, cu parametrul `F_SETLK`). Se poate face si un apel blocant, *i.e.* functia `fcntl` nu va returna imediat, ci va sta in asteptare pina cind reuseste sa puna lacatul.

Iată sursa programului `access2w.c` (programul acces versiunea 2.0 cu apel blocant):

```
/*
  File: access2w.c (versiunea 2.0 cu lacat pus in mod blocant)
```

```
*/
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

extern int errno;

int main(int argc, char* argv[])
{
  int fd;
  char ch;
  struct flock lacat;

  if(argv[1] == NULL)
  {
    fprintf(stderr,"Trebuie apelat cu cel putin un parametru.\n");
    exit(1);
  }

  if( (fd=open("fis.dat", O_RDWR)) == -1)
  {  /* trateaza cazul de eroare ... */
    perror("Nu pot deschide fisierul fis.dat deoarece ");
    exit(2);
  }

  /* pregateste lacat pe fisier */
  lacat.l_type   = F_WRLCK;
  lacat.l_whence = SEEK_SET;
  lacat.l_start  = 0;
  lacat.l_len    = 1; /* aici se poate pune orice valoare, inclusiv 0,
     deoarece pentru problema noastra nu conteaza lungimea zonei blocate.*/


  /* O singura incercare de punere a lacatului in mod blocant
     (intra in asteptare pina cind reuseste) */
  printf("Incep punerea blocajului in mod blocant [Proces:%d].\n",getpid());
  if( fcntl(fd,F_SETLKW,&lacat) == -1)
  {
    if(errno == EINTR)
      fprintf(stderr,"Apelul fcntl a fost intrerupt [ProcesID:%d].\n",getpid());
    else
      fprintf(stderr,"Eroare unknown la blocaj [ProcesID:%d].\n",getpid());
    perror("\tMotivul");
    exit(3);
  }
  else
    printf("Blocaj reusit [ProcesID:%d].\n",getpid());


  /* parcurgerea fisierului caracter cu caracter pina la EOF */
  while( read(fd,&ch,1) != 0)
  {
```

```
    if(ch == '#')
    {
      lseek(fd,-1L,1);
      sleep(10);
      write(fd,argv[1],1);
      printf("Terminat. S-a inlocuit primul # gasit [ProcesID:%d].\n",getpid());
      return 0;
    }
  }

  printf("Terminat. Nu exista # in fisierul dat [ProcesID:%d].\n",getpid());
  return 0;
}
```

Lansînd concurent două procese care să execute acest program, veţi observa că obţinem acelaşi rezultat ca şi în cazul variantei neblocante.

Observaţie importantă:
Se poate constata faptul ca versiunea 2.0 a programului nostru (ambele variante, şi cea neblocantă, şi cea blocantă) nu este optima: *practic*, cele doua procese îşi fac treaba *secvenţial*, unul după altul, şi nu concurent, deoarece de abia dupa ce se termina acel proces care a reusit primul sa puna lacatul pe fisier, va putea incepe si celalalt proces sa-si faca treaba (*i.e.*, parcurgerea fisierului si inlocuirea primului caracter '#'intilnit).

Aceasta observatie ne sugereaza ca putem imbunatati timpul total de executie, permitind celor doua procese sa se execute intr-adevar concurent, pentru aceasta fiind nevoie sa punem lacat doar pe un singur caracter (si anume pe primul caracter '#'intilnit), in loc sa blocam tot fisierul.

Exerciţiu. Scrieti versiunea 3.0 a acestui program, cu blocaj la nivel de caracter.

Ideea de rezolvare: programul va trebui sa faca urmatorul lucru: cind intilneste primul caracter '#'in fisier, pune lacat pe el (*i.e.*, pe exact un caracter) si apoi il rescrie.

Rezolvare: daca nu ati reusit sa scrieti singuri programul, atunci iată cum ar trebui să arate programul `access3w.c` (programul acces versiunea 3.0, varianta cu apel blocant):

```
/*
  File: access3w.c (versiunea 3.0, cu lacat pus in mod blocant)
*/
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

extern int errno;

int main(int argc, char* argv[])
{
```

```
  int fd;
  char ch;
  struct flock lacat;

  if(argv[1] == NULL)
  {
    fprintf(stderr,"Trebuie apelat cu cel putin un parametru.\n");
    exit(1);
  }

  if( (fd=open("fis.dat", O_RDWR)) == -1)
  {  /* trateaza cazul de eroare ... */
    perror("Nu pot deschide fisierul fis.dat deoarece ");
    exit(2);
  }

  /* pregateste lacat pe caracterul de la pozitia curenta */
  lacat.l_type   = F_WRLCK;
  lacat.l_whence = SEEK_CUR;
  lacat.l_start  = 0;
  lacat.l_len    = 1;

  /* parcurgerea fisierului caracter cu caracter pina la EOF */
  while( read(fd,&ch,1) != 0)
  {
    if(ch == '#')
    {
      lseek(fd,-1L,1);

      /* O singura incercare de punere a lacatului in mod blocant */
      printf("Pun blocant lacatul pe #-ul gasit deja [Proces:%d].\n",getpid());
      if( fcntl(fd,F_SETLKW,&lacat) == -1)
      {
        fprintf(stderr,"Eroare la blocaj [ProcesID:%d].\n", getpid());
        perror("\tMotivul");
        exit(3);
      }
      else
        printf("Blocaj reusit [ProcesID:%d].\n", getpid());

      sleep(5);
      write(fd,argv[1],1);
      printf("Terminat. S-a inlocuit primul # gasit [ProcesID:%d].\n",getpid());
      return 0;
    }
  }

  printf("Terminat. Nu exista # in fisierul dat [ProcesID:%d].\n",getpid());
  return 0;
}
```

Observaţie: ideea de rezolvare expusa mai sus (aplicata in programul `access3w.c`) nu este intrutotul corecta, in sensul ca nu se va obtine intotdeauna rezultatul scontat, deoarece între momentul primei depistari a '#'-ului si momentul reusitei blocajului exista posibilitatea ca acel '#'sa fie suprascris de celalalt proces (tocmai pentru a forţa apariţia unei situaţii care cauzează producerea unui rezultat nedorit, am introdus în program acel apel `sleep(5)` între punerea blocajului pe caracterul '#'si rescrierea lui).
Aceasta idee de rezolvare se poate corecta astfel: dupa punerea blocajului, se verifica din nou daca acel caracter este intr-adevar '#'(pentru ca intre timp s-ar putea sa fi fost rescris de celalalt proces), si daca nu mai este '#', atunci trebuie scos blocajul si reluata bucla de cautare a primului '#'din fisier.
Incercati singuri sa adaugati aceasta corectie la programul `access3w.c` (daca nu reusiti, atunci consultati în anexa B programul `access4w.c`).

Observaţie: funcţionarea corectă a lacătelor se bazează pe *cooperarea* proceselor pentru asigurarea accesului exclusiv la fisiere, *i.e.* toate procesele care vor să acceseze mutual exclusiv un fişier (sau o porţiune dintr-un fişier) vor trebui să folosească lacăte pentru accesul respectiv. Altfel, dacă un proces scrie direct un fişier (sau o porţiune dintr-un fişier), apelul său de scriere nu va fi împiedicat de un eventual lacăt pus pe acel fişier (sau acea porţiune de fişier) de către un alt proces.
Cu alte cuvinte, lacătele sunt nişte *semafoare* pentru accesul exclusiv la (porţiuni din) fişiere, spre deosebire de *semafoarele* clasice cunoscute din teoria sistemelor de operare, care asigură accesul exclusiv la variabile de memorie şi/sau bucăţi de cod, adică la (porţiuni din) memoria internă.
În continuare vom exemplifica cele spuse mai sus.

Exemplu. Dacă lansăm concurent două procese, unul care să execute programul acces versiunea 2.0 (aceea care blochează imediat fişierul `fis.dat` şi apoi începe căutarea primului '#'din fişier), iar altul care să suprascrie fişierul `fis.dat`, lucru realizat prin comanda:
`UNIX> access2w 1 & echo "text-fara-diez">fis.dat &`
vom constata că suprascrierea fişierului de către comanda `echo` reuşeşte, indiferent de faptul că programul `acces2w` blochează în scriere fişierul respectiv, acesta nereuşind să mai găsească nici un '#'ˆın fişier.

3.2.3 Fenomenul de interblocaj. Tehnici de eliminare a interblocajului

Observatie: la fel ca în cazul utilizării *semafoarelor* clasice cunoscute din teoria sistemelor de operare, prin folosirea lacătelor pe fişiere poate apare fenomenul de interblocaj.

Exemplu. Sa consideram doua procese `P1` si `P2` care au nevoie sa blocheze pentru acces exclusiv aceleasi doua resurse `R1` si `R2` (de exemplu doua inregistrari intr-un fisier), acapararea resurselor facindu-se in ordine inversa:

Pasii executati de `P1`:

```
1. Acaparare resursa R1
2. Acaparare resursa R2
3. Utilizare resurse
4. Eliberare resurse
```

Pasii executati de `P2`:

```
1. Acaparare resursa R2
2. Acaparare resursa R1
3. Utilizare resurse
4. Eliberare resurse
```

Daca cele doua procese sunt executate concurent, poate surveni situatia de interblocaj: `P1` a reusit sa blocheze `R1` (a executat pasul `1`) si `P2` a reusit sa blocheze `R2` (a executat pasul `1`) si acum ambele procese incearca sa execute pasul `2` : `P1` asteapta eliberarea resursei `R2` pentru a o acapara, iar `P2` asteapta eliberarea resursei `R1` pentru a o acapara. Deci ambele procese vor astepta la infinit, fiind vorba de un interblocaj.

Exista trei modalitati de tratare a interblocajului:

a) tehnici de prevenire;

b) tehnici de evitare;

c) tehnici de reacoperire (eliminare).

In continuare sa vedem doua tehnici de tratare a interblocajului:

1) Tehnica punctelor de retur ("*rollback*")
Tipul ei: eliminarea interblocajului.

Ideea consta in: pentru exemplul anterior, cind procesul `P1` a reusit blocarea resursei `R1` si a esuat pe resursa `R2`, va face urmatoarul lucru: elibereaza resursa deja ocupata `R1`, asteapta un anumit interval de timp aleator (pentru a impiedica sincronizarea proceselor) si apoi reia executia de la pasul `1`. Acelasi algoritm (eliberare resurse in caz de esec, si reluare dupa o asteptare) il va executa si procesul `P2`.

2) Tehnica ordonarii resurselor
Tipul ei: prevenirea interblocajului.

Ideea consta in: se defineste o relatie de ordine pe multimea resurselor. Atunci cind un proces vrea sa acapareze mai multe resurse, va trebui sa le ocupe in ordinea crescatoare a acestora (ordinea de deblocare nu conteaza, dar pentru eficienta ar trebui tot in ordine crescatoare); daca nu reuseste, atunci asteapta pina se elibereaza respectiva resursa. Datorita acestei ordini a resurselor si algoritmului de acaparare a resurselor, nu mai este nevoie de retur pentru eliminarea interblocajului.

Cum se poate defini relaţia de ordine a resurselor?
Exemplul 1: pentru un fisier, putem lua ca ordine ordinea inregistrarilor din fisier (adica *offset*-ul in fisier).
Exemplul 2: pentru un arbore, putem aplica blocarea de la radacina spre frunze; alteori se poate utiliza ordinea de la frunze spre radacina.

O alta situatie care poate apare in programarea concurenta ar fi urmatoarea: sa consideram secventa de cod

```
if (E)
then I1
else I2
```

Se poate intimpla ca expresia `E` sa fie evaluata la `true`, dar pina cind se executa instructiunea `I1`, un alt proces poate face ca `E` sa devina `false` (si deci am vrea de fapt sa se execute instructiunea `I2`).
Din acest motiv, in programarea concurenta se utilizeaza "*sectiuni critice*" (implementate prin blocaje, semafoare, monitoare, etc.) pentru asemenea portiuni de cod ce trebuie executate exclusiv.

3.3 Exerciţii

Exerciţiul 1. Care sunt principalele apeluri de sistem pentru lucrul cu fişiere?

Exerciţiul 2. Studiaţi cu ajutorul comenzii `man` prototipurile tuturor apelurilor de sistem pentru lucrul cu fişiere amintite în secţiunea 3.1.

Exerciţiul 3.* Ce efect are fragmentul de cod C următor?

```
char file[]="a.txt";
int rval;
rval=access(file, F_OK);
if (rval)
  printf("%s exista!\n", file);
else
  printf("Eroare!\n");
```

Exerciţiul 4.* Ce efect are fragmentul de cod C următor?

```
struct stat* info;
char file[] = "a.txt";
stat(file,info);
if ((info->st_mode & S_IFMT) == S_IFREF)
  printf("fisier obisnuit!\n");
else
  printf("alt tip de fisier!\n");
```

*Exerciţiul 5**. Dacă nu sunteţi utilizatorul *root*, ce se va afişa pe ecran în urma execuţiei programului următor?

```
#include <stdio.h>
#include <fcntl.h>
main() {
  int f = open("/etc/passwd",O_RDWR);
  switch(f) {
    case 0x1 : printf("0x01"); break;
    case 0xFF: printf("0xFF"); break;
    case 0x0 : printf("0x00"); break;
    case 0x2 : printf("0x02"); break;
    default: printf("descriptor=%d",f);
  }
}
```

*Exerciţiul 6**. Ce se poate obţine pe ecran în urma execuţiei programului următor?

```
#include<stdio.h>
#include<sys/stat.h>
int get_file_size(char *path) {
   struct stat file_stats;
   if(stat(path,&file_stats))
     return file_stats.st_size;
   else return 0;
}
int main() {
  printf("%d\n",get_file_size("/etc/passwd"));
}
```

*Exerciţiul 7**. Considerăm programului următor:

```
#include ...
main(){
  int fd;  char *s="Linux";
  unlink("fis2");
  fd = open("fis1", O_CREAT|O_WRONLY|O_TRUNC,0700);
  write(fd,s,strlen(s));
  link("fis1","fis2");
}
```

Presupunînd că sunt incluse fişierele *header* necesare, şi că aveţi suficiente drepturi în directorul în care veţi executa acest program, ce va afisa comanda `cat fis1 fis2` executată după execuţia acestui program?

Exerciţiul 8. Scrieţi un program C care să simuleze comanda `tac`.

Exerciţiul 9. Scrieţi un program C care să simuleze comanda `head`.

Exerciţiul 10. Scrieţi un program C care să afiseze toate intrarile dintr-un director dat ca parametru.

Exerciţiul 11. Scrieţi un program C care să adauge la sfirsitul unui fisier continutul altui fisier.

Exerciţiul 12. Scrieţi un program C care să afiseze permisiunile tuturor fisierelor si subdirectoarelor (recursiv) din directorul dat ca argument.

Exerciţiul 13.* Scrieţi un program C care să afişeze încontinuu într-o buclă numele directorului curent de lucru şi următorul meniu de operaţii posibile:
– [**M**]kdir = creeaza directorul specificat (ca subdirector in directorul curent)
– [**R**]mdir = sterge directorul curent
– [**C**]hdir = schimba directorul curent in cel specificat
– [**L**]ist = listeaza continutul directorului curent
– [**S**]elect = selecteaza fisierul specificat din directorul curent si aplica-i urmatorul submeniu de operatii:
- [**C**]opy = copie-l in fisierul specificat
- [**D**]elete = sterge-l
- [**R**]ename = redenumeste-l cu numele specificat
- [**V**]iew = afiseaza-l pe ecran
– [**Q**]uit = terminare program
Deci intr-o bucla se va afisa directorul curent si acest meniu, apoi se va citi cite o tasta din meniul anterior si se va executa operatia asociata ei, iar apoi se reia bucla (pina cind se va da Quit).

Exerciţiul 14. Scrieţi un *script* `bash` care să realizeze aceleaşi operaţii ca şi programul C de la exerciţiul precedent.

Exerciţiul 15. Cîte tipuri de blocaje pe fişiere există? Care sunt deosebirile dintre ele?

Exerciţiul 16. Care sunt structurile de date şi apelurile sistem utilizate pentru lucrul cu blocaje pe fişiere?

Exerciţiul 17.* Scrieţi programul `access4w.c` care să corecteze neajunsul versiunii 3.0 a programului `access`, dată în secţiunea 3.2.

Exerciţiul 18. În ce constă fenomenul de interblocaj? Cum poate fi el tratat?

Exerciţiul 19. Realizaţi o implementare practică a exemplului cu cele două procese `P1` şi `P2` şi cele două resurse `R1` şi `R2` amintit în secţiunea 3.2, şi trataţi interblocajul. *Indicaţie*: procesele vor fi două programe C, resursele două fişiere de date, iar acapararea/eliberarea unei resurse va însemna punerea/scoaterea blocajului pe fişierul respectiv.

Exerciţiul 20. Să se implementeze un semafor binar folosind lacătele pe fişiere.

Capitolul 4

Gestiunea proceselor

4.1 Procese UNIX. Introducere

1. **Noţiuni generale despre procese**
2. **Primitive referitoare la procese**

4.1.1 Noţiuni generale despre procese

Termenul de *program* specifică de obicei un *fişier executabil* (evident, obţinut prin compilare dintr-un fişier sursă), aflat pe un suport de memorare extern (*i.e.*, *harddisk*). Un program este încărcat în memorie şi executat de nucleul sistemului de operare prin intermediul primitivei `exec` (despre care vom vorbi mai încolo).

O instanţă a unui program aflat în execuţie poartă numele de *proces*. Acesta este o entitate gestionată de nucleul sistemului de operare, entitate ce conţine imaginea în memorie a fişierului executabil (zonele de cod, date şi stivă), precum şi resursele utilizate în momentul execuţiei (registre, fişiere deschise, ş.a.). Prin urmare,

DEFINIŢIE: "Un *proces* este un program în curs de execuţie."

Mai sunt si alte definitii ale notiunii de proces, dar aceasta este mai potrivita deoarece face referire si la timp, adica la caracterul temporal al procesului.

Deci un *proces* este executia unui program, fiind caracterizat de: o durata de timp (perioada de timp in care se executa acel program), o zona de memorie alocata (zona de cod + zona de date + stiva), timp procesor alocat, ş.a.

Evident, la un moment dat de timp pot exista in curs de executie doua procese (adica executii) diferite ale aceluiasi program (*i.e.*, fişier executabil).
Facind o analogie cu programarea orientata obiect, am avea corespondenta:

```
program ⟷ conceptul de clasa
proces  ⟷ conceptul de obiect (i.e., instanta a unei clase)
```

UNIX-ul fiind un sistem de operare *multi-tasking*, aceasta înseamnă că, la un moment dat, există o listă de procese aflate în evidenţa sistemului de operare pentru execuţie. Fiecare proces este identificat de un număr întreg unic numit PID (*Process IDentifier*), iar lista proceselor poate fi aflată cu comanda ps (ce a fost prezentată în capitolul 2 din partea I a acestui manual).

Practic, nucleul sistemului de operare gestionează această listă de procese prin intermediul unei *tabele a proceselor* (alocată în spaţiul de memorie al nucleului). Această tabelă conţine cîte o intrare pentru fiecare proces existent în sistem, intrare referită prin *identificatorul de proces* (*i.e.*, PID-ul) acelui proces, şi care conţine o serie de informaţii despre acel proces.

Aşadar, în UNIX fiecare proces este caracterizat de PID-ul său (*i.e.*, un *identificator de proces* unic), şi, în plus, are un unic proces *părinte* (sau *tată*), şi anume acel proces care l-a creat (crearea se face prin intermediul primitivei fork, despre care vom vorbi mai încolo). Un proces poate crea *oricîte* procese (evident, în limita resurselor sistemului), procese care se vor numi procese *fii* (sau *copii*) ai procesului respectiv care le-a creat. Pe baza relaţiei părinte–fiu, procesele sunt organizate într-o ierarhie arborescentă de procese, a cărei rădăcină este procesul cu PID-ul 0.
De asemenea, fiecare proces are un *proprietar*, acel utilizator care l-a lansat în execuţie, şi un *grup proprietar*, şi anume grupul utilizatorului care este proprietarul acelui proces.

Procese speciale (ale sistemului de operare):

- procesul "*swapper*", cu PID=0 :

  este *planificatorul de procese*, un proces de sistem ce are rolul de a planifica toate procesele existente în sistem. El este creat la incarcarea sistemului de operare de catre *boot-loader*, devenind radacina arborelui de procese (din el se nasc toate celelalte procese, pe baza apelului fork despre care vom discuta mai încolo);

- procesul "*init*", cu PID=1 :

  este procesul de initializare invocat de procesul "*swapper*" la terminarea incarcarii sistemului de operare;

- procesul "*pagedaemon*", cu PID=2 :

  este procesul care se ocupa de paginarea memoriei.

Observaţie: pe parcursul exploatării sistemului, procesele existente in sistem evolueaza dinamic: se nasc procese si sunt distruse (la terminarea lor) in functie de programele

rulate de utilizatori. Ca atare, ierarhia arborescenta a proceselor din sistem, despre care am vorbit mai devreme, precum si, implicit, structura de date a nuclelului ce gestioneaza aceasta ierarhie (*i.e.*, tabela de procese), nu sunt statice, ci au un caracter dinamic – sunt intr-o continua evolutie, in functie de programele rulate de utilizatori.
Totusi, radacina ierarhiei (*i.e.*, procesul "*swapper*" cu `PID`=0) este fixa, in sensul ca acest proces nu se termina *niciodata* (mai exact, se termina atunci cind este inchis sistemul de calcul, sau doar resetat). La fel se intimpla si cu alte citeva procese – *procesele de sistem* (*i.e.*, procesele componente ale nucleului sistemului de operare), dintre care trei le-am amintit mai sus.

Revenind la *tabela proceselor* gestionată de nucleu, fiecare intrare din ea, corespunzătoare unui anumit proces, conţine o serie de informaţii despre acel proces (dintre care o parte au fost deja amintite mai sus), şi anume:

- `PID`-ul = identificatorul de proces – este un intreg pozitiv, de tipul `pid_t` (tip definit in *header*-ul `sys/types.h`);
- `PPID`-ul = identificatorul procesului parinte;
- terminalul de control
- `UID`-ul proprietarului = identificatorul utilizatorului care executa procesul;
- `GID`-ul proprietarului = identificatorul grupului din care face parte utilizatorul ce executa procesul;
- `EUID`-ul si `EGID`-ul = `UID`-ul si `GID`-ul proprietarului *efectiv*, adică acel utilizator ce determina drepturile procesului de acces la resurse (pe baza bitilor *setuid bit* si *setgid bit* din masca de drepturi de acces a fisierului executabil asociat acelui proces);
  (*notă*: reamintesc faptul ca, daca bitul *setuid* este 1, atunci, pe toata durata de executie a fisierului respectiv, proprietarul efectiv al procesului va fi proprietarul fisierului, si nu utilizatorul care il executa; similar pentru *setgid bit*.)
- *starea procesului* – poate fi una dintre următoarele:
  - *ready* = pregatit pentru executie;
  - *run* = in executie;
  - *wait* = in asteptarea producerii unui eveniment (ca, de exemplu, terminarea unei operatii I/O);
  - *finished* = terminare normala.
- linia de comanda si mediul (*i.e.*, variabilele de mediu transmise de parinte);

ş.a.

Accesul in program la parametrii din linia de comanda prin care s-a lansat in executie programul respectiv, precum si la variabilele de mediu transmise acestuia de catre parinte la crearea sa, se poate realiza declarind functia *main* a programului in felul urmator:

`int main (int` *argc*`, char*` *argv*`[], char*` *env*`[])`

unde variabila *argv* este un vector de pointeri catre parametrii din linia de comanda (sub forma de siruri de caractere), ultimul element al tabloului fiind pointerul `NULL`, iar variabila `argc` contine numarul acestor parametri. Similar, variabila *env* este un vector de pointeri catre variabilele de mediu (care sunt siruri de caractere), ultimul element al tabloului fiind pointerul `NULL`.

Accesul la variabilele de mediu se poate realiza si prin intermediul functiilor `getenv()` si `setenv()` din biblioteca `stdlib.h`. Iata un exemplu de cod C prin care se poate afla valoarea unei variabile de mediu:

```
char* path;
path=getenv("PATH");
printf("The value of variable PATH is %s\n", path ? path : "not set!");
```

În continuare vom prezenta cîteva apeluri sistem cu ajutorul cărora putem afla aceste informaţii.

4.1.2 Primitive referitoare la procese

În continuare vom trece în revistă cîteva primitive (*i.e.*, apeluri sistem) referitoare la procese:

1) Primitive pentru aflarea `PID`-urilor unui proces si a parintelui acestuia: `getpid`, `getppid`. Interfetele acestor functii sunt urmatoarele:

```
#include <unistd.h>
pid_t getpid(void)
pid_t getppid(void)
```

Efect: functia `getpid` returneaza `PID`-ul procesului apelant, iar `getppid` returneaza `PID`-ul parintelui procesului apelant.

Exemplu. Următorul program exemplifică apelul acestor primitive:

```
/*
  File: exemplu1.c
*/
#include <stdio.h>
#include <unistd.h>

int main(void)
{
  printf("\n\nProcesul: %d , avind parintele: %d\n",getpid(),getppid());
  return 0;
}
```

2) Primitive pentru aflarea `ID`-urilor proprietarului unui proces si a grupului acestuia: `getuid`, `getgid` si `geteuid`, `getegid`. Interfetele acestor functii sunt urmatoarele:

```
#include <unistd.h>
uid_t getuid(void)
gid_t getgid(void)
uid_t geteuid(void)
gid_t getegid(void)
```

Efect: functia `getuid` returneaza `UID`-ul (*i.e.*, `User ID`-ul) proprietarului, adică al utilizatorului care a lansat in executie procesul apelant, iar functia `getgid` returneaza `GID`-ul (*i.e.*, `Group ID`-ul) grupului proprietar, adică al grupului utilizatorului care a lansat in executie procesul apelant.
Functia `geteuid` returneaza `Effective User ID`-ul, adică `UID`-ul proprietarului *efectiv*, iar functia `getegid` returneaza `Effective Group ID`-ul, adică `GID`-ul grupului proprietarului *efectiv*.

3) Alte primitive ce ofera diverse informatii, sau modifica diverse atribute ale proceselor, ar mai fi urmatoarele primitive (definite tot in *header*-ul `unistd.h`): `setuid()`, `setgid()`, `getpgrp()`, `getpgid()`, `setpgrp()`, `setpgid()`. (Consultaţi paginile de manual electronic corespunzătoare acestor primitive pentru a afla detalii despre ele).

4) Primitive de suspendare a executiei pe o durata de timp specificata: `sleep` si `usleep`. Interfetele acestor functii sunt urmatoarele:

```
void sleep(int nr_sec)
void usleep(int nr_msec)
```

Efect: functia `sleep` suspendă executia procesului apelant timp de *nr_sec* secunde, iar functia `usleep` suspendă executia procesului apelant timp de *nr_msec* milisecunde.
Observaţie: după cum vom vedea în lecţia despre semnale `UNIX`, apelul `sleep`, respectiv `usleep`, este întrerupt în momentul cînd procesul primeşte un semnal, *i.e.* apelul returnează imediat, fără a aştepta scurgerea intervalului de timp specificat.

Exemplu. Următorul program exemplifică apelul acestor primitive:

```
/*
  File: exemplu2.c
*/
#include <stdio.h>
#include <unistd.h>

void main(void)
{
  printf("\n\nProcesul: %d , avind parintele: %d\n",getpid(),getppid());
  printf("\n\nProprietarul procesului: UID=%d, GID=%d\n",getuid(),getgid());
  printf("\n\nProprietarul efectiv: UID=%d, GID=%d\n",geteuid(),getegid());
  printf("Start of sleeping for 3 seconds...\n");
  sleep(3);
  printf("Finish of sleeping for 3 seconds.\n");
  return 0;
}
```

5) Primitiva de terminare a executiei: `exit`. Interfata acestei functii este urmatoarea:

`void exit(int` *cod_retur*`)`

Efect: functia `exit` termină executia procesului apelant si returneaza sistemului de operare codul de terminare specificat ca argument.
Observatie: acelasi efect are si instructiunea `return` *cod_retur*`;` , dar numai daca apare in functia *main* a programului.

6) Functia `system` permite lansarea de comenzi `UNIX` dintr-un program C, printr-un apel de forma:

`system(`*comanda*`);`

Efect: se creează un nou proces, în care se incarcă *shell*-ul implicit, ce va executa comanda specificată.

Iată şi două exemple:
• `system("ps");`
Efect: se execută comanda `ps`, ce afişează procesele curente ale utilizatorului;
• `system("who | cut -b 1-8 > useri-logati.txt");`
Efect: se execută comanda înlănţuită specificată, care creează un fişier cu numele `useri-logati.txt`, în care se vor găsi numele utilizatorilor conectaţi la sistem.

4.2 Crearea proceselor şi terminarea lor

1. **Crearea proceselor – primitiva** `fork()`
2. **Terminarea proceselor**

4.2.1 Crearea proceselor – primitiva `fork()`

Singura modalitate de creare a proceselor in `UNIX/Linux` este cu ajutorul apelului sistem `fork`. Prototipul lui este urmatorul:

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

Efect: prin acest apel *se creeaza o copie a procesului apelant*, si ambele procese – cel nou creat si cel apelant – isi vor continua executia cu urmatoarea instructiune (din programul executabil) ce urmează dupa apelul functiei `fork`.
Singura diferenta dintre procese va fi valoarea returnata de functia `fork`, precum si, bineinteles, `PID`-urile proceselor.
Procesul apelant va fi *parintele* procesului nou creat, iar acesta va fi *fiul* procesului apelant (mai exact, unul dintre procesele fii ai acestuia).

Observaţie referitoare la această "copiere":
Memoria ocupata de un proces poate fi impartita in urmatoarele zone:

`CODE`	$\longrightarrow$	zona de cod
`DATA`	$\longrightarrow$	zona de date
`ENV`	$\longrightarrow$	zona de mediu
`FILE`	$\longrightarrow$	zona descriptorilor de fisiere

Zona de cod contine instructiunile programului, zona de date contine variabilele programului (datele statice/dinamice, stiva, registrii, etc.), zona de mediu contine variabilele de mediu (ce sunt primite la crearea procesului, de la procesul parinte), iar zona descriptorilor de fisiere contine descriptorii de fisiere deschise.

Aceasta impartire a memoriei ocupate de un proces corespunde organizarii logice a memoriei calculatorului, ce este impartita in *pagini logice* de memorie. Fiecarei pagini logice ii corespunde o *pagina fizica* de memorie in memoria calculatorului. Aceasta corespondenta poate varia pe parcursul executiei procesului respectiv. (Este vorba aici despre tehnicile de gestionare a memoriei in `UNIX` - vezi cursul de sisteme de operare din anul I.)
Totalitatea paginilor fizice la un moment formeaza memoria ocupata de acel proces la momentul respectiv, cu observatia evidenta ca memoria fizica ocupata nu este obligatoriu o zona contigua de memorie (doar memoria logica ocupata este intotdeauna o zona contigua de memorie).

Important: practic, noul proces creat cu primitiva `fork` va avea aceeasi zona de cod fizica ca si procesul parinte, doar zonele de date (i.e., zonele `DATA`, `ENV`, `FILE`) vor fi, fizic, diferite. Insa, imediat dupa executia functiei `fork`, aceste zone vor contine aceleasi valori, deoarece, în cursul apelului, se face copierea zonelor de date ale procesului parinte in zonele de date, fizice, ale fiului.
Din acest motiv – deoarece se duplica zona de date, deci inclusiv registrii, deci inclusiv registrul PC ("*Program Counter*") – executia procesului fiu va continua din acelasi punct ca

si a parintelui: se va executa instructiunea imediat urmatoare dupa apelul `fork`. *Atenţie*: este vorba de următoarea instrucţiune în limbaj maşină, a nu se confunda cu limbajul sursă a programului respectiv, care de obicei este limbajul C.
Tot din acest motiv, imediat dupa apelul `fork` procesul fiu va avea aceleasi valori ale variabilelor din program si aceleasi fisiere deschise ca si procesul parinte. Mai departe insa, *fiecare proces va lucra pe zona sa de memorie*. Deci, daca fiul modifica valoarea unei variabile, aceasta modificare nu va fi vizibila si in procesul tata (si nici invers). În concluzie, nu avem memorie partajata (*shared memory*) între procesele tata si fiu.

Valoarea returnată: funcţia `fork` returneaza valoarea `-1`, in caz de eroare (daca nu s-a putut crea un nou proces), iar in caz de succes, returneaza respectiv urmatoarele valori in cele doua procese, tata si fiu:

- n, in procesul tata, unde n este `PID`-ul noului proces creat;
- `0`, in procesul fiu.

Observaţii:

1. `PID`-ul unui nou proces nu poate fi niciodata `0`, deoarece procesul cu `PID`-ul `0` nu este fiul nici unui proces, ci este radacina arborelui proceselor (arbore ce descrie relatiile parinte-fiu dintre toate procesele existente in sistem). Mai mult, procesul cu `PID`-ul `0` este singurul proces din sistem ce nu se creeaza prin apelul `fork`, ci el este creat atunci cind se *boot*-ează sistemul `UNIX/Linux` pe calculatorul respectiv.

2. Procesul nou creat poate afla `PID`-ul tatalui cu ajutorul primitivei `getppid`, pe cind procesul tata nu poate afla `PID`-ul noului proces creat, fiu al lui, prin alta maniera decit prin valoarea returnata de apelul `fork`. Nu s-a creat o primitivă pentru aflarea `PID`-ului fiului deoarece, spre deosebire de părinte, fiul unui proces nu este unic – un proces poate avea zero, unul, sau mai mulţi fii la un moment dat.

Exemplu. Urmatorul program exemplifica crearea unui nou proces cu primitiva `fork` si ilustreaza unele diferente dintre cele doua procese, părinte si fiu:

```
/*
  File: fork.c  (exemplu de creare a unui fiu)
*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
  pid_t pid;
  int   a=0;

  if( (pid=fork() ) == -1)
```

```
  {
    perror("Eroare la fork");
    exit(1);
  }

  if (pid == 0)
  { /* fiu */
    printf("Procesul fiu id=%d, cu parintele: id=%d\n",getpid(),getppid());
    printf("Procesul fiu: dupa fork, variabila a=%d\n", a);
    a = 5;
    printf("Procesul fiu: dupa modificare, variabila a=%d\n", a);
  }
  else
  { /* parinte */
    printf("Procesul tata id=%d, cu parintele: id=%d si fiul: id=%d\n",
           getpid(),getppid(),pid);
    sleep(2);
    printf("Procesul tata: variabila a=%d\n", a);
  }

  /* zona de cod comuna */
  printf("Zona de cod comuna, executata de %s.\n", pid==0?"fiu":"tata");
  return 0;
}
```

4.2.2 Terminarea proceselor

Procesele se pot termina în două moduri:

1. *Terminarea normală*:

   se petrece în momentul întîlnirii în program a apelului primitivei `exit`, ce a fost prezentată în secţiunea anterioară, sau la sfîrşitul funcţiei *main* a programului, sau la întîlnirea instrucţiunii `return` în funcţia *main*.

   Ca efect, procesul este trecut în starea *finished*, se închid fişierele deschise (şi se salvează pe disc conţinutul *buffer*-elor folosite), se dealocă zonele de memorie alocate procesului respectiv, ş.a.m.d.

   Codul de terminare (furnizat de primitiva `exit` sau de instrucţiunea `return`) este salvat în intrarea corespunzătoare procesului respectiv din tabela proceselor; intrarea respectivă nu este dealocată ("ştearsă") imediat din tabelă, astfel încît codul de terminare a procesului respectiv să poată fi furnizat procesului părinte la cererea acestuia (ceea ce se face cu ajutorul primitivei `wait` despre care vom discuta mai tîrziu); de abia după ce s-a furnizat codul de terminare părintelui, intrarea este "ştearsă" din tabelă.

2. *Terminarea anormală*:

   se petrece în momentul primirii unui semnal `UNIX` (mai multe detalii vom vedea mai tîrziu, cînd vom discuta despre semnale `UNIX`).

   Şi în acest caz se dealocă zonele de memorie ocupate de procesul respectiv, şi se păstrează doar intrarea sa din tabela proceselor pîna cînd părintele său va cere codul de terminare (reprezentat în acest caz de numărul semnalului ce a cauzat terminarea anormală).

4.3 Sincronizarea proceselor

1. **Introducere**
2. **Primitiva `wait`**

4.3.1 Introducere

In programarea concurenta exista notiunea de *punct de sincronizare* a doua procese: este un punct din care cele doua procese au o executie simultana (*i.e.*, este un punct de asteptare reciproca).

Punctul de sincronizare nu este o notiune dinamica, ci una statica (o notiune fixa): este precizat in algoritm (*i.e.*, program) locul unde se gaseste acest punct de sincronizare.

Citeva caracteristici ale punctului de sincronizare ar fi urmatoarele:
• procesele isi reiau executia simultan dupa acest punct;
• punctul de sincronizare este valabil pentru un numar fixat de procese (nu neaparat doar pentru doua procese), si nu pentru un numar variabil de procese.

Primitiva `fork` este un exemplu de punct de sincronizare: cele doua procese – procesul apelant al primitivei `fork` si procesul nou creat de apelul acestei primitive – isi reiau executia simultan din acest punct (*i.e.*, punctul din program in care apare apelul functiei `fork`).

Un exemplu de utilizare a punctului de sincronizare:

Exemplu. Sa consideram problema calculului maximului unei secvente de numere: un proces *master* imparte secventa de numere la mai multe procese *slave*, fiecare dintre acestea va calcula maximul subsecventei primite, si apoi se va sincroniza cu procesul *master* printr-un punct de sincronizare pentru ca sa-i transmita rezultatul partial obtinut. Dupa primirea tuturor rezultatelor partiale, procesul *master* va calcula rezultatul final.

4.3.2 Primitiva `wait`

Un alt exemplu de sincronizare, des intilnita in practica, ar fi următorul:

Procesul parinte poate avea nevoie de valoarea de terminare returnata de procesul fiu. Pentru a realiza aceasta facilitate, trebuie stabilit un punct de sincronizare intre sfirsitul programului fiu si punctul din programul parinte in care este nevoie de acea valoare, si apoi transferata acea valoare de la procesul fiu la procesul părinte.
Aceasta situatie a fost implementata in `UNIX` printr-o primitiva, numita `wait`.

Apelul sistem `wait` este utilizat pentru a astepta un proces fiu sa se termine. Interfata acestei functii este urmatoarea:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int* stat_loc)
```

Efect: apelul functiei `wait` suspendă executia procesului apelant pina în momentul cind unul (oricare) dintre fiii lui se termină sau este stopat (*i.e.*, terminat anormal) printr-un semnal. Dacă există deja vreun fiu care s-a terminat sau a fost stopat, atunci functia `wait` returneaza imediat.

Functia `wait` returneaza ca valoare `PID`-ul acelui proces fiu, iar in locatia referita de pointerul *stat_loc* este salvata urmatoarea valoare:
• valoarea de terminare a acelui proces fiu (si anume, in octetul *high* al acelui `int`), daca functia `wait` returneaza deoarece s-a terminat vreun proces fiu;
• codul semnalului (si anume, in octetul *low* al acelui `int`), daca functia `wait` returneaza deoarece un fiu a fost stopat de un semnal.
Daca procesul apelant nu are procese fii, atunci functia `wait` returneaza valoarea `-1`, iar variabila `errno` este setata in mod corespunzator pentru a indica eroarea (`ECHILD` sau `EINTR`).

Observaţie importantă: daca procesul parinte se termina inaintea vreunui proces fiu, atunci acestui fiu i se va atribui ca parinte procesul `init` (ce are `PID`-ul `1`), iar acest lucru se face pentru toate procesele fii neterminate in momentul terminarii parintelui lor.
Iar daca un proces se termina inaintea parintelui lui, atunci el devine *zombie* – procesul se termina in mod obisnuit (*i.e.*, se inchid fisierele deschise, se elibereaza zona de memorie ocupata de proces, ş.a.m.d.), dar se pastreaza totusi intrarea corespunzatoare acelui proces in tabela proceselor din sistem, pentru ca aceasta intrare va pastra codul de terminare a procesului, cod ce va putea fi consultat, eventual, de catre parintele procesului prin intermediul functiei `wait`.
(Aceasta observatie este valabila intotdeauna, nu doar numai in cazul folosirii functiei `wait`.)

Pe lîngă primitiva `wait`, care asteapta terminarea oricarui fiu, mai exista inca o primitivă, numita `waitpid`, care va astepta terminarea unui anumit fiu, mai exact a procesului fiu avind `PID`-ul specificat ca argument.

Indicaţie: a se citi neaparat paginile de manual despre functiile `wait` si `waitpid`.

Exemplu. Iată un exemplu simplu ce ilustreaza cazul terminarii normale a fiului:

```
/*
  File: wait-ex1.c  (terminare normala a fiului)
*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
  if (fork() == 0)
  { /* fiu */
    printf("Proces fiu id=%d\n", getpid());
    exit(3);
  }
  else
  { /* parinte */
    int pid_fiu, cod_term;
    pid_fiu = wait(&cod_term);
    printf("Tata: sfirsit fiul %d cu valoarea %d\n", pid_fiu, cod_term);
  }
}
```

In urma executiei acestui program, se va afisa valoarea `768` (adica `3*256`), deoarece este o terminare normala a fiului si deci valoarea de terminare se depune in octetul *high* al locaţiei date ca argument apelului `wait`.

Observatie: exista unele *macro*-uri ce fac conversia valorii de terminare (a se vedea *help*-ul de la functia `wait`).

Exemplu. Iată şi un alt exemplu, ce ilustreaza cazul terminarii anormale a fiului:

```
/*
  File: wait-ex2.c  (terminare anormala a fiului)
*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
  if (fork() == 0)
  { /* fiu */
    printf("Proces fiu id=%d\n", getpid());
    for(;;);
  }
```

```
  else
  { /* parinte */
    int pid_fiu, cod_term;
    pid_fiu = wait(&cod_term);
    printf("Tata: sfirsit fiul %d cu valoarea %d\n", pid_fiu, cod_term);
  }
}
```

Executati acest program in *background*, prin comanda:
`UNIX> wait-ex2 &`

Se observa ca procesele nu se opresc niciodata (intrucit procesul fiu executa o bucla infinita, iar procesul tata il asteapta cu `wait`). Ca atare, pentru a le opri, aflati `PID`-ul fiului (cu comanda `ps`) si apoi omoriti-l (cu comanda `kill -9` *pid_fiu*); astfel de fapt ii transmiteti semnalul cu numarul `9`. Ca urmare, cele doua procese se vor termina, iar in urma executiei lor se va afisa valoarea `9`, adica numarul semnalului care l-a stopat pe fiu, deoarece este o terminare anormala a fiului si deci numarul semnalului se depune in octetul *low* al locaţiei date ca argument apelului `wait`.

4.4 Reacoperirea proceselor

1. **Introducere**
2. **Primitivele din familia** `exec`

4.4.1 Introducere

Dupa cum am vazut deja, singura modalitate de a crea un nou proces in `UNIX` este prin apelul functiei `fork`. Numai ca in acest fel se creeaza o copie a procesului apelant, adica o noua instanta de executie a aceluiasi fisier executabil.

Si atunci, cum este posibil sa executam un alt fisier executabil decit cel care apeleaza primitiva `fork`?
Raspuns: printr-un alt mecanism, acela de "*reacoperire a proceselor*", disponibil in `UNIX` prin intermediul primitivelor de tipul `exec`.

4.4.2 Primitivele din familia `exec`

In `UNIX/Linux` exista o familie de primitive `exec` care transforma procesul apelant intr-un alt proces specificat (prin numele fisierului executabil asociat) ca argument al apelului `exec`.
Noul proces se spune ca "*reacoperă*" procesul ce a executat apelul `exec`, si el moşteneşte caracteristicile acestuia (inclusiv `PID`-ul), cu exceptia citorva dintre ele (vom reveni mai jos cu lista acestora).

Observaţie: in caz de succes, *apelul* `exec` *nu returnează* !!!, deoarece nu mai exista procesul apelant. Prin urmare, `exec` este singurul exemplu de functie (cu exceptia primitivei `exit`) al carei apel nu returneaza inapoi in programul apelant.

Exista in total 6 functii din familia `exec`. Ele difera prin nume si prin lista parametrilor de apel, si sunt impartite in 2 categorii (ce difera prin forma in care se dau parametrii de apel):
• a) numarul de parametri este variabil;
• b) numarul de parametri este fix.

1. Prima pereche de primitive `exec` este perechea `execl` si `execv`, ce au interfetele urmatoare:
   a$_1$) `int execl(char*` *ref*`, char*` *argv0*`, ..., char*` *argvN*`)`
   b$_1$) `int execv(char*` *ref*`, char*` *argv*`[])`

   Argumentul *ref* reprezinta numele procesului care va reacoperi procesul apelant al respectivei primitive `exec`. El trebuie sa fie un nume de fisier executabil care sa se afle in directorul curent (sau sa se specifice si directorul in care se afla, prin cale absoluta sau relativa), deoarece nu este cautat in directoarele din variabila de mediu `PATH`.
   Argumentul *ref* este obligatoriu, celelalte argumente pot lipsi; ele exprima parametrii liniei de comanda pentru procesul *ref*.
   Ultimul argument *argvN*, respectiv ultimul element din tabloul *argv*`[]`, trebuie sa fie pointerul `NULL`.
   Prin conventie *argv0*, respectiv *argv*`[0]`, trebuie sa coincida cu *ref* (deci cu numele fisierului executabil). Aceasta este insa doar o conventie, nu se produce eroare in caz ca este incalcata. De fapt, argumentul *ref* specifica numele real al fisierului executabil ce se va incarca si executa, iar *argv0*, respectiv *argv*`[0]`, specifica numele afisat (de comenzi precum `ps`, `w`, ş.a.) al noului proces.

2. A doua pereche de primitive `exec` este perechea `execle` si `execve`, ce au interfetele urmatoare:
   a$_2$) `int execle(char*` *ref*`, char*` *argv0*`, ..., char*` *argvN*`, char*` *env*`[])`
   b$_2$) `int execve(char*` *ref*`, char*` *argv*`[], char*` *env*`[])`

   Efect: similar cu perechea anterioara, doar ca acum ultimul parametru permite transmiterea catre noul proces a unui *environment* (*i.e.*, un mediu: o multime de

şiruri de caractere de forma `variabila=valoare`).
La fel ca pentru *argv*`[]`, ultimul element din tabloul *env*`[]` trebuie sa fie pointerul `NULL`.

3. A treia pereche de primitive `exec` este perechea `execlp` si `execvp`, ce au interfetele urmatoare:
   a3) `int execlp(char*` *ref*`, char*` *argv0*`, ..., char*` *argvN*`)`
   b3) `int execvp(char*` *ref*`, char*` *argv*`[])`

   Efect: similar cu perechea `execl` si `execv`, cu observatia ca fisierul *ref* este cautat si in directoarele din variabila de mediu `PATH`, in cazul in care nu este specificat impreuna cu calea, relativa sau absoluta, pina la el.

In caz de esec (datorita memoriei insuficiente, sau altor cauze), toate primitivele `exec` returneaza valoarea `-1`. Altfel, functiile `exec` nu mai returneaza, deoarece procesul apelant nu mai exista (a fost reacoperit de noul proces).

Caracteristicile procesului după `exec`:

Noul proces mosteneste caracteristicile vechiului proces (are acelasi `PID`, aceeasi prioritate, acelasi proces parinte, aceeasi descriptori de fisiere deschise, etc.), cu exceptia citorva dintre ele.
Si anume, diferitele caracteristici ale procesului sunt conservate in timpul reacoperirii cu oricare dintre functiile din familia `exec`, cu exceptia urmatoarelor caracteristici, in conditiile specificate:

Caracteristica	**Condiţia în care nu se conservă**
Proprietarul efectiv	Daca este setat bitul *setuid* al fisierului incarcat, proprietarul acestui fisier devine proprietarul efectiv al procesului.
Grupul efectiv	Daca este setat bitul *setgid* al fisierului incarcat, grupul proprietar al acestui fisier devine grupul proprietar efectiv al procesului.
Handler-ele de semnale	Sunt reinstalate *handler*-ele implicite pentru semnalele corupte (interceptate).
Descriptorii de fisiere	Daca bitul `FD_CLOEXEC` de inchidere automata in caz de `exec`, al vreun descriptor de fisier a fost setat cu ajutorul primitivei `fcntl`, atunci acest descriptor este inchis la `exec` (ceilalti descriptori de fisiere ramin deschisi).

Exemplul urmator ilustreaza citeva dintre aceste proprietati.

Exemplu. Consideram următoarele doua programe, primul este `before_exec.c` care apeleaza `exec` pentru a se reacoperi cu al doilea, numit `after_exec.c`:

```
/*
  File: before_exec.c
```

```
*/
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

char tab_ref[1000];

void main()
{
  printf("Caracteristici inainte de exec\n");
  printf("------------------------------\n");
  printf("ID-ul procesului : %d\n",getpid());
  printf("ID-ul parintelui : %d\n",getppid());
  printf("Proprietarul real : %d\n",getuid());
  printf("Proprietarul efectiv : %d\n",geteuid());
  printf("Directorul de lucru : %s\n\n",getcwd(tab_ref,1000));

  /* cerere de inchidere a intrarii standard la reacoperire */
  fcntl(STDIN_FILENO, F_SETFD, FD_CLOEXEC);

  /* reacoperire */
  execl("after_exec","after_exec",NULL);
}


/*
  File: after_exec.c
*/
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

char tab_ref[1000];

void main()
{
  int nrBytesRead;
  char ch;
  printf("Caracteristici dupa exec\n");
  printf("------------------------\n");
  printf("ID-ul procesului : %d\n",getpid());
  printf("ID-ul parintelui : %d\n",getppid());
  printf("Proprietarul real : %d\n",getuid());
  printf("Proprietarul efectiv : %d\n",geteuid());
  printf("Directorul de lucru : %s\n\n",getcwd(tab_ref,1000));

  nrBytesRead = read(STDIN_FILENO, &ch, 1);
  printf("Numarul de caractere citite: %d\n",nrBytesRead);
  if( nrBytesRead = -1 )
    perror("Error reading stdin (because it is closed !) ");
}
```

Compilati cele doua programe, dindu-le ca nume de executabil numele sursei fara extensia `.c`, si apoi lansati-l in executie pe primul dintre ele. El va fi reacoperit de cel de-al doilea, iar in urma executiei veti constata ca variabila `nrBytesRead` are valoarea `-1`, motivul fiind că intrarea standard `stdin` este inchisa in procesul `after_exec`.

Exemplu. Iată şi un alt exemplu – un program care se reacopera cu el insusi, dar la al doilea apel isi modifica parametrii de apel pentru a-si da seama ca este la al doilea apel si astfel sa nu intre intr-un apel recursiv la infinit.

```
/*
  File: exec-rec.c  (exemplu de apel recursiv prin exec)
*/
#include <stdio.h>
#include <errno.h>
extern int errno;

void main(int argc, char* argv[], char* env[])
{
  char **r, *s, *w[5];
  printf("PID=%d, PPID=%d, OWNER=%d\n",getpid(),getppid(),getuid());
  printf("ENVIRONMENT:\n");
  r=env;
  while( s=*r++)
  { printf("%s\n",s); }
  putchar('\n');

  env[0]="Salut.";
  env[1]=NULL;

  w[0]=argv[0]; /* numele executabilului ! */
  w[1]="2nd call";
  w[2]=NULL;

  if( (argv[1] != NULL) && (argv[1][0] == '2') )
  {
    exit(0); /* oprire recursie la al doilea apel ! */
  }
  else
  {
    printf("Urmeaza apelul primitivei exec.\n");
    if( execve(argv[0], w, env) == -1)
    {
      printf("Error on exec: err=%d\n", errno);
      exit(1);
    }
  }
}
```

Observaţii:

1. Apelul `exec` consuma mai multa memorie decit apelul `fork`.

2. Comportamentul in cazul fisierelor deschise in momentul apelului primitivelor `exec`: daca s-au folosit instructiuni de scriere *buffer*-izate (ca de exemplu funcţiile `fprintf`, `fwrite` ş.a. din biblioteca standard de C), atunci *buffer*-ele nu sunt scrise automat in fisier pe disc în momentul apelului `exec`, deci informatia din ele se pierde.
   Comentariu: in mod normal *buffer*-ul este scris in fisier abia in momentul cind s-a umplut, sau la intilnirea caracterului '\n'(*newline*). Dar se poate forţa scrierea *buffer*-ului in fisier cu ajutorul functiei `fflush` din biblioteca standard de C.

Iată şi un exemplu referitor la ultima observatie:

Exemplu. Sa consideram urmatoarele trei programe, `com-0.c`, `com-1.c` şi `com-2.c`, dintre care primele doua se reacopera fiecare cu al treilea, şi care vor fi executate in maniera specificata mai jos:

```
/*
  File: com-0.c
*/
#include <stdio.h>

void main()
{
  int fd;
  fd=creat("fis.txt",0666);
  close(1);  /* inchid stdout */
  dup(fd);   /* duplic fd cu 1 (primul gasit liber) */
  close(fd); /* inchid fd */
  /* practic astfel am redirectat stdout in fisierul fis.txt */
  write(1,"Salut",5);
  execl("com-2","com-2",NULL);
}
```

```
/*
  File: com-1.c
*/
#include <stdio.h>

void main()
{
  printf("Salut");
  fflush(stdout);
  execl("com-2","com-2",NULL);
}
```

```
/*
  File: com-2.c
```

```
*/
#include <stdio.h>

void main()
{
  write(1," la toti!",9);
}
```

Compilati cele trei programe, dindu-le ca nume de executabil numele sursei fara extensia `.c`, si apoi lansati-le in executie astfel:

```
UNIX> com-1
Salut la toti!
```

Obsevatie: Daca eliminam apelul `fflush` din programul `com-1.c`, atunci pe ecran se va afisa doar mesajul "`la toti!`", deoarece "`Salut`" se pierde prin `exec`, *buffer*-ul nefiind golit pe disc.

```
UNIX> com-0
UNIX> cat fis.txt
Salut la toti!
```

Deci programul `com-0` a scris mesajul in fisierul `fis.txt` si nu pe ecran.
Concluzie: descriptorii de fisiere deschise din `com-0` s-au "moştenit" prin `exec` în `com-2`.

Observaţie: functia `dup(`*fd*`)` cauta primul descriptor de fisier nefolosit si il redirecteaza catre descriptorul primit ca parametru (a se consulta *help*-ul acestei functii pentru detalii suplimentare). Cu ajutorul ei, în programul `com-0` am redirectat ieşirea standard `stdout` a programului în fisierul `fis.txt`.

4.5 Semnale `UNIX`

1. **Introducere**
2. **Categorii de semnale**
3. **Tipurile de semnale predefinite ale `UNIX`-ului**
4. **Cererea explicită de generare a unui semnal – primitiva `kill`**
5. **Coruperea semnalelor – primitiva `signal`**
6. **Definirea propriilor *handler*-ere de semnal**
7. **Blocarea semnalelor**
8. **Aşteptarea unui semnal**

4.5.1 Introducere

Semnalele `UNIX` reprezinta un mecanism fundamental de manipulare a proceselor si de comunicare intre procese, ce asigură tratarea evenimentelor asincrone apărute în sistem.

Un *semnal* `UNIX` este o *intrerupere software* generată în momentul producerii unui anumit eveniment şi transmisa de sistemul de operare unui anumit proces (deci este intrucitva similar intreruperilor din `MS-DOS`).

Un semnal este *generat* de aparitia unui eveniment exceptional (care poate fi o eroare, un eveniment extern sau o cerere explicita). Orice semnal are asociat un *tip*, reprezentat printr-un numar intreg pozitiv, si un proces *destinatar* (*i.e.*, procesul caruia ii este destinat acel semnal). Odata generat, semnalul este pus in *coada de semnale* a sistemului, de unde este extras si transmis procesului destinatar de catre sistemul de operare.

Transmiterea semnalului destinatarului se face imediat dupa ce semnalul a ajuns in coada de semnale, cu o exceptie: daca primirea semnalelor de tipul respectiv a fost *blocată* de catre procesul destinatar (vom vedea mai tirziu cum anume se face acest lucru), atunci transmiterea semnalului se va face abia in momentul cind procesul destinatar va debloca primirea acelui tip de semnal.

In momentul in care procesul destinatar primeste acel semnal, el isi *intrerupe executia* si va executa o anumita actiune (*i.e.*, o functie de tratare a acelui semnal), functie numita *handler de semnal* si care este atasata tipului de semnal primit, dupa care procesul isi va relua executia din punctul in care a fost intrerupt (cu anumite exceptii: unele semnale vor cauza terminarea sau intreruperea acelui proces).

In concluzie, fiecare tip de semnal are asociat o actiune (un *handler*) specifica acelui tip de semnal.

4.5.2 Categorii de semnale

In general, evenimentele ce genereaza semnale se impart in trei categorii: erori, evenimente externe si cereri explicite.

1) O **eroare** inseamna ca programul a facut o operatie invalida si nu poate sa-si continue executia.
Nu toate erorile genereaza semnale, ci doar acele erori care pot apare in orice punct al programului, cum ar fi: impartirea la zero, accesarea unei adrese de memorie invalide, etc.
Exemplu de erori ce nu genereaza semnale: erorile in operatiile I/O – apelul de functie respectiv va intoarce un cod de eroare, de obicei `-1`.

2) **Evenimentele externe** sunt in general legate de operatiile I/O sau de actiunile altor procese, cum ar fi: sosirea datelor (pe un *socket* sau un *pipe*, de exemplu), expirarea intervalului de timp setat pentru un *timer* (o alarma), terminarea unui proces fiu, sau suspendarea/terminarea programului de catre utilizator (prin apasarea tastelor `CTRL+Z` sau `CTRL+C`).

3) O **cerere explicita** inseamna generarea unui semnal de catre un proces, prin apelul functiei de sistem `kill`, a carei sintaxa o vom discuta mai tirziu.

Important: semnalele pot fi generate *sincron* sau *asincron*.
Un semnal **sincron** este un semnal generat de o anumita actiune specifica in program si este livrat (daca nu este blocat) in timpul acelei actiuni. Evenimentele care genereaza semnale sincrone sunt: erorile si cererile explicite ale unui proces de a genera semnale pentru el insusi.
Un semnal **asincron** este generat de un eveniment din afara zonei de control a procesului care il receptioneaza, cu alte cuvinte, un semnal ce este receptionat, in timpul executiei procesului destinatar, la un moment de timp ce nu poate fi anticipat. Evenimentele care genereaza semnale asincrone sunt: evenimentele externe si cererile explicite ale unui proces de a genera semnale destinate altor procese.

Pentru fiecare tip de semnal exista o actiune implicita de tratare a acelui semnal, specifica sistemului de operare `UNIX` respectiv. Aceasta actiune este denumita *handler-ul de semnal implicit* atasat acelui tip de semnal.

Atunci cind semnalul este livrat procesului, acesta este intrerupt si are trei posibilitati de comportare: fie sa execute aceasta actiune implicita, fie sa ignore semnalul, fie sa execute o anumita functie *handler* utilizator (*i.e.*, scrisa de programatorul respectiv).
Setarea unuia dintre cele trei comportamente se face cu ajutorul apelului primitivelor `signal` sau `sigaction`, despre care vom vorbi mai tirziu. Asadar, la fiecare primire a unui anumit tip de semnal, se va executa acea actiune (comportament) ce a fost setata la ultimul apel al uneia dintre cele doua primitive, apel efectuat pentru acel tip de semnal.

Observatie: daca actiunea specificata pentru un anumit tip de semnal este de a-l ignora, atunci orice semnal de acest tip este inlaturat din coada de semnale imediat dupa primire, chiar si in cazul in care acel tip de semnal este blocat pentru procesul respectiv (vom discuta mai tirziu despre blocarea semnalelor).

4.5.3 Tipurile de semnale predefinite ale `UNIX`-ului

In fisierul *header* `signal.h` se gaseste lista semnalelor `UNIX` predefinite, mai exact numarul intreg asociat fiecarui tip de semnal, impreuna cu o constanta simbolica, cu observatia ca in programe se recomanda folosirea constantelor simbolice in locul numerelor (deoarece numerele asociate semnalelor pot diferi de la o versiune de `UNIX` la alta).

Aceasta lista poate fi obtinuta si cu comanda urmatoare:

```
UNIX> kill -l
```

iar pagina de manual ce contine descrierea semnalelor poate fi obtinuta astfel:

```
UNIX> man 7 signal
```

Aceste tipuri predefinite de semnale se pot clasifica in mai multe categorii:

1. semnale standard de eroare: `SIGFPE`, `SIGILL`, `SIGSEGV`, `SIGBUS`;
2. semnale de terminare: `SIGHUP`, `SIGINT`, `SIGQUIT`, `SIGTERM`, `SIGKILL`;
3. semnale de alarma: `SIGALRM`, `SIGVTALRM`, `SIGPROF`;
4. semnale asincrone I/O: `SIGIO`, `SIGURG`;
5. semnale pentru controlul proceselor: `SIGCHLD`, `SIGCONT`, `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, `SIGTTOU`;
6. alte tipuri de semnale: `SIGPIPE`, `SIGUSR1`, `SIGUSR2`.

In continuare, sa trecem in revista categoriile de semnale amintite mai sus.
Observatie: semnalele notate cu * mai jos au urmatorul comportament: procesul destinatar, cind se intrerupe la primirea semnalului, provoaca crearea unui fisier *core* (ce contine imaginea memoriei procesului in momentul intreruperii), care poate fi inspectat pentru depanarea programului; fisierul *core* este scris pe disc in directorul de unde a fost lansat acel proces.

1. *Semnale standard de eroare*:
   - `SIGFPE` * = *signal floating point error*, semnal sincron generat in caz de eroare aritmetica fatala, cum ar fi impartirea la zero sau *overflow*-ul.
   - `SIGILL` * = *signal illegal instruction*, semnal sincron generat cind se incearca executarea unei instructiuni ilegale, adica programul incearca sa execute zone de date (in loc de functii), situatie ce poate apare daca fisierul executabil este stricat, sau daca se paseaza un pointer la o data acolo unde se asteapta un pointer la o functie, sau daca se corupe stiva prin scrierea peste sfirsitul unui *array* de tip automatic.
   - `SIGSEGV` * = *signal segmentation violation*, semnal sincron generat in caz de violare a segmentului de memorie, adica procesul incearca sa acceseze o zona de memorie care nu ii apartine (care nu ii este alocata – apartine altor procese, etc.).
     Cauze de producere a acestui eveniment: folosirea pentru acces la memorie a unui pointer `NULL` sau neinitializat, ori folosirea unui pointer pentru parcurgerea unui *array*, fara a verifica depasirea sfirsitului *array*-ului.

- `SIGBUS` * = *signal bus error*, semnal sincron generat in caz de eroare de magistrala, ce poate apare tot atunci cind se utilizeaza un pointer `NULL` sau neinitializat, numai ca, spre deosebire de `SIGSEGV` care raporteazaun acces nepermis la o zona de memorie valida (existenta), `SIGBUS` raporteaza un acces nepermis la o zona de memorie invalida: adresa inexistenta, sau un pointer dezaliniat, cum ar fi referirea la un intreg (reprezentat pe 4 octeti), la o adresa nedivizibila cu 4 (fiecare tip de calculator si sistem de operare are o anumita politica de aliniere a datelor).

  Observatie: toate aceste semnale au drept actiune implicita terminarea procesului (cu afisarea unui mesaj de eroare specific) si crearea acelui fisier *core*.

2. *Semnale de terminare*: ele sunt folosite pentru a indica procesului sa-si termine executia, intr-un fel sau altul. Motivul pentru care se folosesc este acela de a putea "face curat" inainte de terminarea propriu-zisa: se pot salva date in fisiere, sterge fisierele temporare, restaura vechiul tip de terminal in caz ca el a fost modificat de catre program, etc.

   - `SIGHUP` = *signal hang-up*, semnal generat in momentul deconectarii terminalului (datorita unei erori in retea, sau altor cauze), ori la terminarea procesului de control a terminalului, si este trimis proceselor asociate cu acea sesiune de lucru, avind ca efect deconectarea efectiva a acestor procese de terminalul de control. Actiunea implicita constă în terminarea procesului.
   - `SIGINT` = *signal program interrupt*, semnal de intrerupere, generat atunci cind utilizatorul foloseste caracterul `INTR` (adica: apasa tastele `CTRL+C`) pentru a termina executia programului.
   - `SIGQUIT` = *signal quit*, semnal de intrerupere, generat cind utilizatorul foloseste caracterul `QUIT` (de obicei, tastele `CTRL+\`), fiind asemanator cu semnalul `SIGINT`: provoaca terminarea procesului.
   - `SIGTERM` = semnal generic, folosit pentru terminarea proceselor; spre deosebire de `SIGKILL`, acest semnal poate fi blocat, ignorat, sau sa i se asigneze un *handler* propriu.
   - `SIGKILL` = *signal kill*, semnal utilizat pentru terminarea imediata a proceselor; el nu poate fi blocat, ignorat, sau sa i se asigneze un *handler* propriu, deci are o comportare fixa – terminarea procesului, de aceea se spune ca este un semnal fatal. Poate fi generat doar de evenimentul cerere explicita, folosind apelul `kill()`.

3. *Semnale de alarma*: ele indica expirarea timpului pentru *timer*-e si alarme, care pot fi setate prin apelul primitivelor `alarm()` si `setitimer()`. Comportamentul implicit al acestor semnale este terminarea procesului, de aceea este indicata asignarea de *handler*-e proprii pentru ele.

   - `SIGALRM` = *signal time alarm*, semnal emis la expirarea timpului pentru un *timer* care masoara timpul "real" (*i.e.*, intervalul de timp scurs intre inceputul si sfirsitul procesului).

- `SIGVTALRM` = *signal virtual time alarm*, semnal emis la expirarea timpului pentru un *timer* care masoara timpul "virtual" (*i.e.*, timpul in care procesul utilizeaza efectiv *CPU*-ul).
- `SIGPROF` = semnal emis la expirarea timpului pentru un *timer* care masoara timpul in care procesul utilizeaza efectiv *CPU*-ul si timpul in care *CPU*-ul asteapta indeplinirea unor conditii (cum ar fi,de exemplu, terminarea unor cereri de I/O) pentru acel proces. Acest tip de semnal se utilizeaza pentru a implementa facilitati de optimizare a codului programelor.

4. *Semnale asincrone I/O*: ele sunt utilizate impreuna cu facilitatile I/O ale sistemului; trebuie apelata explicit functia `fcntl()` asupra unui descriptor de fisiere pentru a se putea ajunge in situatia de a se genera aceste semnale.

   - `SIGIO` = semnal folosit pentru a indica ca un anumit descriptor de fisiere este gata de a realiza operatii I/O; doar descriptorii asociati unui *socket* sau unui *pipe* pot genera acest tip de semnal; semnalul este generat in momentul cind, spre exemplu, se receptioneaza niste date pe un *socket*, pentru a indica programului ca trebuie sa faca un *read()* pentru a le citi.
   - `SIGURG` = semnal transmis atunci cind date "urgente" (asa-numitele *out-of-band data*) sunt receptionate pe un *socket*.

5. *Semnale pentru controlul proceselor*:

   - `SIGCHLD` = *signal child*, semnal trimis procesului parinte atunci cind procesul fiu (*i.e.*, emitatorul semnalului) isi termina executia.
     In general, este util ca sa se asigneze un *handler* propriu pentru acest tip de semnal, in care sa se utilizeze apelurile `wait()` sau `waitpid()` pentru a accepta codul de terminare al proceselor fii.
     Observatie: astfel *kernel*-ul va elibera intrarea corespunzatoare acelui fiu din tabela proceselor; in caz contrar acest lucru se petrece abia la terminarea procesului tata.
   - `SIGCONT` = *signal continue*, semnal transmis pentru a cauza continuarea executiei unui proces, care a fost anterior suspendat prin semnalul `SIGSTOP` sau prin celelate semnale ce suspenda procese.
   - `SIGSTOP` = *signal stop*, semnal utilizat pentru suspendarea executiei unui proces. La fel ca si *SIGKILL*, acest semnal are o comportare fixa, neputind fi blocat, ignorat, sau sa i se asigneze un *handler* propriu.
   - `SIGTSTP` = semnal interactiv de suspendare a executiei unui proces, generat prin tastarea caracterului `SUSP` (de obicei, tastele `CTRL+Z`). Spre deosebire de `SIGSTOP`, el poate fi blocat, ignorat, sau sa i se asigneze un *handler* propriu.
   - `SIGTTIN` = semnal transmis unui proces, ce ruleaza in *background*, in momentul in care incearca sa citeasca date de la terminalul asociat. Actiunea sa implicita este de a suspenda executia procesului.
   - `SIGTTOU` = semnal transmis unui proces, ce ruleaza in *background*, in momentul in care incearca sa scrie date la terminalul asociat, sau sa schimbe tipul terminalului. Actiunea sa implicita este de a suspenda executia procesului.

Observaţii:

i) Atunci cind procesele sunt suspendate, acestora nu li se mai pot transmite semnale, cu exceptia semnalelor `SIGKILL` si `SIGCONT`. Semnalul `SIGKILL` nu poate fi corupt, si duce la terminarea procesului; desi semnalul `SIGCONT` poate fi corupt (*i.e.*, blocat sau ignorat), el va duce oricum la reluarea executiei procesului.

ii) Transmiterea unui semnal `SIGCONT` unui proces duce la eliminarea din coada de semnale a tuturor semnalelor `SIGSTOP` destinate acelui proces (deci care inca nu au fost transmise procesului).

iii) Daca un proces dintr-un grup de procese orfane (adica procesul parinte al grupului si-a terminat executia inaintea proceselor fii) primeste unul dintre semnalele `SIGTSTP`, `SIGTTIN` sau `SIGTTOU`, si nu are un *handler* propriu asignat pentru acel semnal, atunci il va ignora, deci nu-si suspenda executia (motivul fiind ca, acest proces fiind orfan, nu exista posibilitatea sa-si reia executia).

6. *Alte tipuri de semnale*: sunt utilizate pentru a raporta alte conditii ce pot apare.

   - `SIGPIPE` = semnal emis in caz de tentativa de scriere intr-un *pipe* din care nu mai are cine sa citeasca.
     Motivul: cind se folosesc *pipe*-uri (sau *fifo*-uri), aplicatia trebuie astfel construita incit un proces sa deschida *pipe*-ul pentru citire inainte ca celalalt sa inceapa sa scrie. Daca procesul care trebuie sa citeasca nu este startat, sau se termina in mod neasteptat, atunci scrierea in *pipe* cauzeaza generarea acestui semnal. Daca procesul blocheaza sau ignora semnalele `SIGPIPE`, atunci scrierea in *pipe* esueaza cu `errno=EPIPE`.
     Actiunea implicita a acestui semnal este terminarea procesului si afisarea unui mesaj de eroare corespunzator ("Broken pipe").
   - `SIGUSR1` si `SIGUSR2` = semnale furnizate pentru ca programatorul sa le foloseasca dupa cum doreste. Sunt utile pentru comunicatia inter-procese.
     Actiunea implicita a acestor semnale fiind terminarea procesului, este necesara asignarea unor *handler*-e proprii pentru aceste semnale.
     Singurul eveniment care genereaza aceste semnale este cererea explicita, folosind apelul `kill()`.

Alte semnale `UNIX`:

- `SIGTRAP` * = *signal trap*, semnal emis dupa executia fiecarei instructiuni, atunci cind procesul este executat in modul de depanare.
- `SIGIOT` * = *signal I/O trap*, semnal emis in caz de probleme *hardware* (de exemplu, probleme cu discul).
- `SIGSYS` * = semnal emis in caz de apel sistem cu parametri eronati.

ş.a.

Observaţie: o parte din aceste tipuri de semnale depind si de suportul oferit de partea de *hardware* a calculatorului respectiv, nu numai de partea sa de *software* (*i.e.*, sistemul de operare de pe acel calculator). Din acest motiv, exista mici diferente in implementarea acestor semnale pe diferite tipuri de arhitecturi de calculatoare, adica unele semnale se poate sa nu fie implementate deloc, sau sa fie implementate (adica, actiunea implicita asociata lor) cu mici diferente.
Exemple de semnale ce pot diferi de la un tip de arhitectura la altul: cele generate de erori, cum ar fi `SIGBUS`, etc. Astfel, in `Linux` nu este implementat semnalul `SIGBUS`, deoarece *hardware*-ul Intel386 pentru care a fost scris `Linux`-ul, nu permite detectarea acelui eveniment descris mai sus, asociat semnalului `SIGBUS`.

4.5.4 Cererea explicită de generare a unui semnal – primitiva `kill`

Cererea explicita de generare a unui semnal se face apelind primitiva `kill`, ce are urmatoarea interfata:

`int kill (int` *pid*`, int` *id-signal*`);`

Argumente:
– *pid* = `PID`-ul procesului destinatar;
– *id-signal* = tipul semnalului (*i.e.*, constanta simbolica asociata).
Valoarea returnata: `0`, in caz de reusita, si `-1`, in caz de eroare.

Observatie: daca al doilea argument este `0`, atunci nu se trimite nici un semnal, dar este util pentru verificarea validitatii `PID`-ului respectiv (*i.e.*, daca exista un proces cu acel `PID` in momentul apelului, sau nu): apelul `kill(`*pid*`,0);` returneaza `0` daca `PID`-ul specificat este valid, sau `-1`, in caz contrar.

Pentru cererea explicita de generare a unui semnal se poate folosi si comanda `kill` (la prompterul *shell*-ului):

`UNIX>` `kill -`*nr-semnal pid*

cu observatia ca trebuie dat numarul semnalului, nu constanta simbolica asociata (astfel, pentru semnalul `SIGKILL`, numarul este 9). Se pot specifica mai multe `PID`-uri, sau nume de procese. Consultati *help*-ul (cu comanda `man kill`) pentru detalii.

Un proces poate trimite semnale către sine însuşi folosind funcţia `raise`, ce are următorul format:

`int raise(int` *id-signal*`)`

Efect: prin apelul `raise(`*id-signal*`);` un proces isi auto-expediaza un semnal de tipul specificat; este echivalent cu apelul `kill(getpid(),`*id-signal*`);` .

4.5.5 Coruperea semnalelor – primitiva `signal`

Specificarea actiunii la receptia semnalelor se poate face cu apelurile de sistem `signal()` sau `sigaction()`, functii ale caror prototipuri se gasesc in fisierul *header* `signal.h`, fisier in care mai sunt definite si constantele simbolice: `SIG_DFL` (*default*), `SIG_IGN` (*ignore*), si `SIG_ERR` (*error*), al caror rol va fi explicat mia jos.

Dupa cum am mai spus, actiunea asociata unui semnal poate fi una dintre urmatoarele trei:
– o actiune implicita, specifica sistemului de operare respectiv;
– ignorarea semnalului;
– sau un *handler* propriu, definit de programator.

Se utilizeaza termenul de *corupere a unui semnal* cu sensul de: setarea unui *handler* propriu pentru acel tip de semnal. Uneori, se mai foloseste si termenul de *tratare a semnalului*.

Observatie: dupa cum am mai spus, semnalele `SIGKILL` si `SIGSTOP` nu pot fi corupte, ignorate sau blocate!

Primitiva `signal()`, utilizata pentru specificarea actiunii la receptia semnalelor, are urmatorul prototip:

`sighandler_t signal (int` *id-signal*`, sighandler_t` *action*`);`

Apelul functiei `signal()` stabileste ca, atunci cind procesul receptioneaza semnalul *id-signal*, sa se execute functia (*handler*-ul de semnal) *action*.

Argumentul *action* poate fi numele unei functii definite de utilizator, sau poate lua una dintre urmatoarele valori (constante simbolice definite in fisierul *header* `signal.h`):
– `SIG_DFL` : specifica actiunea implicita (cea stabilita de catre sistemul de operare) la receptionarea semnalului.
– `SIG_IGN` : specifica faptul ca procesul va ignora acel semnal. Sistemul de operare nu permite sa se ignore sau corupa semnalele `SIGKILL` si `SIGSTOP`, de aceea functia `signal()` va returna o eroare daca se face o asemenea incercare.

Observatie: in general nu este bine ca programul sa ignore semnalele (mai ales pe acelea care reprezinta evenimente importante). Daca nu se doreste ca programul sa receptioneze semnale in timpul executiei unei anumite portiuni de cod, solutia cea mai indicata este sa se blocheze semnalele, nu ca ele sa fie ignorate.

Functia `signal()` returneaza vechiul *handler* pentru semnalul specificat, deci astfel poate fi apoi restaurat daca este nevoie.
In caz de esec (daca, spre exemplu, numarul *id-signal* nu este numar valid de semnal, sau se incearca coruperea semnalelor `SIGKILL` sau `SIGSTOP`), functia `signal()` returneaza ca valoare constanta simbolica `SIG_ERR`.

In cazul cind argumentul *action* este numele unei functii definite de utilizator, aceasta functie trebuie sa aiba prototipul `sighandler_t`, unde tipul `sighandler_t` este definit astfel:

```
typedef void (*sighandler_t)(int);
```

adica este tipul "functie ce intoarce tipul `void`, si are un argument de tip `int`".

La momentul executiei unui *handler* de semnal, acest argument va avea ca valoare numarul semnalului ce a determinat executia acelui *handler*. In acest fel, se poate asigna o aceeasi functie ca *handler* pentru mai multe semnale, in corpul ei putind sti, pe baza argumentului primit, care dintre acele semnale a cauzat apelul respectiv.

Exemplu. Sa scriem un program care sa ignore intreruperile de tastatura, adica semnalul `SIGINT` (generat de tastele `CTRL+C`) si semnalul `SIGQUIT` (generat de tastele `CTRL+\`).

a) Iată o primă versiune a programului, fara ignorarea semnalelor:

```
/*
  File: sig-ex1a.c  (fara ignorarea semnalelor)
*/
#include <stdio.h>

void main()
{
  printf("Inceput bucla infinita; poate fi oprita cu ^C sau ^\\ !\n");
  for(;;);
  printf("Sfirsit program.\n");
}
```

b) Iată a doua versiune a programului, cu ignorarea semnalelor:

```
/*
  File: sig-ex1b.c  (cu ignorarea semnalelor)
*/
#include <stdio.h>
#include <signal.h>

void main()
{
  /* ignorarea semnalelor SIGINT si SIGQUIT */
```

```
  signal( SIGINT, SIG_IGN);
  signal(SIGQUIT, SIG_IGN);
  printf("Inceput bucla infinita in care ^C si ^\\ sunt ignorate...\n");
  for(;;);
  printf("Sfirsit program.\n");
}
```

Acest program va rula la infinit, neputind fi oprit cu `CTRL+C` sau cu `CTRL+\`. Pentru a-l termina, va trebui sa-l suspendati (cu `CTRL+Z`), sa-i aflati `PID`-ul si apoi sa-l omoriti cu comanda `kill -9` *pid* (sau: `CTRL+Z` si apoi comanda `kill %`).

Exemplu. Sa modificam exemplul anterior in felul urmator: corupem semnalele sa execute un *handler* propriu, care sa afiseze un anumit mesaj. Iar apoi refacem comportamentul implicit al semnalelor.

```
/*
  File: sig-ex2.c  (cu coruperea semnalelor)
*/
#include <stdio.h>
#include <signal.h>

/* handler-ul propriu de semnal pentru SIGINT si SIGQUIT */
void my_handler(int nr_sem)
{
  /* actiuni dorite de utilizator */
  printf("Nu mai tasta CTRL+%s caci nu are efect.\n",
            (nr_sem==SIGINT ? "C":"\\"));
}

int main()
{
  int i;
  /* coruperea semnalelor SIGINT si SIGQUIT */
  signal( SIGINT, my_handler);
  signal(SIGQUIT, my_handler);

  /* portiune de cod pe care ^C si ^\ sunt corupte */
  printf("Inceput portiune de cod pe care ^C si ^\\ sunt corupte...\n");
  for(i=0;i<10;i++) { printf("Portiune corupta...\n"); sleep(1); }
  printf("Sfirsit portiune.\n");

  /* refacerea comportamentului implicit pentru cele doua semnale */
  signal(SIGINT, SIG_DFL);
  signal(SIGQUIT, SIG_DFL);

  /* portiune de cod pe care ^C si ^\ nu sunt corupte */
  for(i=0;i<10;i++) { printf("Portiune necorupta...\n"); sleep(1); }
  printf("Sfirsit program.\n");
  return 0;
}
```

Cealalta primitiva utilizata pentru specificarea actiunii la receptia semnalelor este functia `sigaction()`. Ea are, in principiu, aceeasi utilizare ca si functia `signal()`, dar permite un control mai fin al comportamentului procesului la receptionarea semnalelor. Consultati *help*-ul pentru detalii suplimentare despre functia `sigaction`.

4.5.6 Definirea propriilor *handler*-ere de semnal

Un *handler* de semnal propriu este o functie definita de programator, care se compileaza deci impreuna cu restul programului; in loc insa de a apela direct aceasta functie, sistemul de operare este instruit, prin apelul functiilor `signal` sau `sigaction`, sa o apeleze atunci cind procesul receptioneaza semnalul respectiv.

Ultimul exemplu de mai sus ilustreaza folosirea unui *handler* de semnal propriu.

Exista doua strategii principale care se folosesc in *handler*-ele de semnal:

1. Se poate ca *handler*-ul sa notifice primirea semnalului prin setarea unei variabile globale si apoi sa returneze normal, urmind ca in bucla principala a programului, acesta sa verifice periodic daca acea variabila a fost setata, in care caz va efectua operatiile dorite.
2. Se poate ca *handler*-ul sa termine executia procesului, sau sa transfere executia intr-un punct in care procesul poate sa-si recupereze starea in care se afla in momentul receptionarii semnalului.

Atentie: trebuie luate masuri speciale atunci cind se scrie codul pentru *handler*-ele de semnal, deoarece acestea pot fi apelate asincron, deci la momente imprevizibile de timp. Spre exemplu, in timp ce se executa *handler*-ul asociat unui semnal primit, acesta poate fi intrerupt prin receptia unui alt semnal (al doilea semnal trebuie sa fie de alt tip decit primul; daca este acelasi semnal, el va fi blocat pina cind se termina tratarea primului semnal).

Important: prin urmare, primirea unui semnal poate intrerupe nu doar executia programului respectiv, ci chiar executia *handler*-ului unui semnal anterior primit, sau poate intrerupe executia unui apel de sistem efectuat de program in acel moment.

Apelurile de sistem ce pot fi intrerupte de semnale sunt urmatoarele:
`close`, `fcntl` [operatia `F_SETLK`], `open`, `read`, `recv`, `recvfrom`, `select`, `send`, `sendto`, `tcdrain`, `waitpid`, `wait` si `write`.
In caz de intrerupere, aceste primitive returneaza valoarea `-1` (mai putin `read` si `write`, care returneaza numarul de octeti cititi, respectiv scrisi cu succes), iar variabila `errno` este setata la valoarea `EINTR`.

4.5.7 Blocarea semnalelor

Blocarea semnalelor inseamna ca procesul spune sistemului de operare sa nu ii transmita anumite semnale (ele vor ramine in coada de semnale, pina cind procesul va debloca primirea lor).

Nu este recomandat ca un program sa blocheze semnalele pe tot parcursul executiei sale, ci numai pe durata executiei unor parti critice ale codului lui. Astfel, daca un semnal ajunge in timpul executiei acelei parti de program, el va fi livrat procesului dupa terminarea acesteia si deblocarea acelui tip de semnal.

Blocarea semnalelor se realizeaza cu ajutorul functiei `sigprocmask()`, ce utilizeaza structura de date `sigset_t` (care este o masca de biti), cu semnificatia de set de semnale ales pentru blocare.
Iar cu ajutorul functiei `sigpending()` se poate verifica existenta, in coada de semnale, a unor semnale blocate, deci care asteapta sa fie deblocate pentru a putea fi livrate procesului.
Consultati *help*-ul pentru detalii suplimentare despre aceste functii.

4.5.8 Aşteptarea unui semnal

Daca aplicatia este influentata de evenimente externe, sau foloseste semnale pentru sincronizare cu alte procese, atunci ea nu trebuie sa faca altceva decit sa astepte semnale. Functia `pause`, cu prototipul:

```
int pause();
```

are ca efect suspendarea executiei programului pina la sosirea unui semnal.

Daca semnalul duce la executia unui *handler*, atunci functia `pause` returneaza valoarea `-1`, deoarece comportarea normala este de a suspenda executia programului tot timpul, asteptind noi semnale. Daca semnalul cauzeaza terminarea executiei programului, apelul `pause()` nu returneaza.

Simplitatea acestei functii poate ascunde erori greu de detectat. Deoarece programul principal nu face altceva decit sa apeleze `pause()`, inseamna ca cea mai mare parte a activitatii utile in program o realizeaza *handler*-ele de semnal. Insa, cum am mai spus, codul acestor *handler*-e nu este indicat sa fie prea lung, deoarece poate fi intrerupt de alte semnale.

De aceea, modalitatea cea mai indicata, atunci cind se doreste asteptarea unui anumit semnal (sau o multime fixata de semnale), este de a folosi functia `sigsuspend()`, ce are prototipul:

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

Functia aceasta are ca efect: se inlocuieste masca de semnale curenta a procesului cu cea specificata de parametrul *set* si apoi se suspenda executia procesului pina la receptionarea unui semnal, de catre proces (deci un semnal care nu este blocat, adica nu este cuprins in masca de semnale curenta).

Masca de semnale ramine la valoarea setata (*i.e.*, valoarea lui *set*) numai pina cind functia `sigsuspend()` returneaza, moment in care este reinstalata, in mod automat, vechea masca de semnale.

Valoarea returnata: `0`, in caz de succes, respectiv `-1`, in caz de esec (iar variabila `errno` este setata in mod corespunzator: `EINVAL`, `EFAULT` sau `EINTR`).

Exemplu. Sa scriem un program care sa-si suspende executia in asteptarea semnalului `SIGQUIT` (generat de tastele `CTRL+\`), fara a fi intrerupt de alte semnale.

```
/*
  File: sig-ex5.c  (asteptarea unui semnal)
*/
#include <signal.h>

void main()
{
  sigset_t mask, oldmask;

  /* stabileste masca de semnale ce vor fi blocate:
     toate semnalele, exceptind SIGQUIT */
  sigfillset(&mask);
  sigdelset (&mask, SIGQUIT);

  printf("Suspendare program pina la apasarea tastelor CTRL+\\.\n");

  /* se asteapta sosirea unui semnal SIGQUIT, restul fiind blocate */
  sigsuspend(&mask);

  printf("Sfirsit program.\n");
}
```

In incheiere, as dori sa va recomand consultarea unei variante mai detaliate a acestei lectii despre semnale `UNIX`, folosita pentru studentii de la Sectia la zi, si care este disponibila la adresa *web* http://fenrir.infoiasi.ro/∼so.

4.6 Exerciţii

Exerciţiul 1. Care sunt apelurile de sistem care oferă informaţii despre un proces, şi ce informaţii oferă fiecare dintre acestea?

Exerciţiul 2. Care este modalitatea prin care se pot crea noi procese în `UNIX`?

Exerciţiul 3. Ce efect are apelul `fork`? Ce valoare returnează?

Exerciţiul 4. Cum putem deosebi părintele de fiu în urma creării acestuia din urmă prin apelul `fork`?

Exerciţiul 5. Pot comunica părintele şi fiul prin intermediul variabilelor de memorie? Justificaţi răspunsul.

Exerciţiul 6.* Ce se poate obţine pe ecran în urma execuţiei programului următor?

```
#include <unistd.h>
#include <stdio.h>
void main() {
  int pid;
  printf("A");
  pid=fork();
  if(pid<0) printf("err");
  printf("B");
}
```

Exerciţiul 7.* Cîte procese fiu sunt create în urma execuţiei programului următor, dacă toate apelurile `fork` reuşesc?

```
#include <unistd.h>
void main() {
  int contor;
  for(contor=1; contor<=7; ++contor)
     if(contor & 1) fork();
}
```

Exerciţiul 8. **Lista de procese**: scrieti un program C care sa creeze o lista de procese de lungime n (valoare citita de la tastatura). Si anume, procesul $\mathtt{P}_1$ va avea ca fiu pe procesul $\mathtt{P}_2$, acesta la rindul lui il va avea ca fiu pe procesul $\mathtt{P}_3$, ş.a.m.d. pina la procesul $\mathtt{P}_n$, care nu va avea nici un fiu.
Incercati o rezolvare nerecursiva si una recursiva a acestei probleme.

Exerciţiul 9. **Arborele de procese**: scrieti un program C care sa creeze un arbore k-ar complet cu n nivele, de procese (valorile k si n vor fi citite de la tastatura). Si anume, unicul proces $\mathtt{P}_{1,1}$ de pe nivelul `1` al arborelui (*i.e.*, radacina arborelui) va avea k procese fii, si anume procesele $\mathtt{P}_{2,1},\ldots,\mathtt{P}_{2,k}$ de pe nivelul `2` al arborelui, fiecare dintre acestea la rindul lui va avea k procese fii pe nivelul `3` al arborelui, ş.a.m.d. pina la cele 2^{n-1} procese de pe nivelul n al arborelui, care nu vor avea nici un fiu.
Incercati o rezolvare nerecursiva si una recursiva a acestei probleme.

Exerciţiul 10. Ce înseamnă noţiunea de punct de sincronizare?

Exerciţiul 11. Cum se poate realiza o sincronizare între un proces părinte şi terminarea unui fiu al acestuia?

Exerciţiul 12. Ce efect are apelul `wait`? Ce valoare returnează şi cînd anume?

Exerciţiul 13.* **Suma distribuită**: scrieti un program C care sa realizeze urmatoarele: un proces `P0` citeste numere de la tastatura si le trimite la doua procese fii `P1` si `P2`, acestea calculeaza sumele si le trimit inapoi la parintele `P0`, iar `P0` aduna cele doua sume partiale si afiseaza rezultatul final.

Indicaţie de rezolvare: pentru comunicatia intre procese puteti folosi fisiere obisnuite – procesul `P0` scrie numerele citite in fisierele `f1i` si `f2i`, de unde sunt citite de procesele `P1`, respectiv `P2`, care le aduna si scriu sumele partiale in fisierele `f1o` si `f2o`, de unde sunt citite de procesul `P0` si adunate.
Observaţie: va trebui sa rezolvati si unele probleme de sincronizare ce apar la comunicatiile intre cele trei procese. Mai precis, apar doua tipuri de probleme:
– *o sincronizare la fişierele de intrare*: procesul fiu `P1`, respectiv `P2`, va trebui sa citească datele din fişierul de intrare corespunzător, `f1i` si respectiv `f2i`, abia după ce părintele `P0` a terminat de scris datele în fişierul respectiv;
– *o sincronizare la fişierele de ieşire*: procesul părinte `P0` va trebui sa citească datele din fişierul de ieşire corespunzător, `f1o` si respectiv `f2o`, abia după ce fiul `P1`, respectiv `P2`, a terminat de scris datele în fişierul respectiv.

Exerciţiul 14. Ce înseamnă noţiunea de "reacoperire" a proceselor?

Exerciţiul 15. Ce apel sistem permite încărcarea şi execuţia unui fişier executabil?

Exerciţiul 16. Ce efect are apelul `exec`? Ce valoare returnează şi cînd anume?

Exerciţiul 17. Cîte primitive de tipul `exec` există şi prin ce se deosebesc ele? Care este semnificaţia argumentelor pentru fiecare primitivă în parte?

Exerciţiul 18. Scrieti un program C care sa execute comanda `ls -a -l`. La sfîrşitul executiei trebuie sa fie afisat textul: "Comanda a fost executata ...".
Indicaţie: folositi primitiva `execlp`.

Exerciţiul 19. Rezolvaţi exerciţiul precedent utilizînd primitiva `execl`.

Exerciţiul 20.* De cîte ori va afişa pe ecran şirul "Hello!" programul de mai jos?

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

int main(){ int fd;
  fd=creat("a",O_WRONLY|0606);
  close(2);
  dup(fd); close(fd);
  fprintf(stderr,"Hello!");
  execlp("cat","cat","a",NULL);
  if(fork() != -1)
    printf("Hello!");
}
```

Exerciţiul 21. Rescrieţi programul cu suma distribuită de la exerciţiul 11*, folosind `wait` in procesul *master* pentru a astepta terminarea celor două procese *slave* şi, în plus, rescrieti procedura *slave* într-un program C separat, care să fie apelat prin `exec` din procesul fiu corespunzător creat prin `fork` de procesul *master*, iar numele fisierelor de intrare/iesire sa-i fie trasferate ca argumente in linia de comanda.

Exerciţiul 22. Studiaţi cu ajutorul comenzii `man` toate apelurile de sistem pentru gestiunea proceselor amintite în secţiunile precedente.

Exerciţiul 23. Ce sunt semnalele `UNIX`?

Exerciţiul 24. Ce categorii de evenimente generează semnale?

Exerciţiul 25. Ce se întîmplă cînd se generează un semnal?

Exerciţiul 26. Ce se înţelege prin *handler*-ul asociat unui semnal?

Exerciţiul 27. Descrieţi categoriile de semnale `UNIX`.

Exerciţiul 28. Ce efect are apelul `kill`?

Exerciţiul 29. Ce înseamnă coruperea (sau tratarea) unui semnal, şi cum se realizează?

Exerciţiul 30. Ce înseamnă *handler* propriu de semnal?

Exerciţiul 31. Ce înseamnă blocarea unui semnal, şi cum se realizează?

Exerciţiul 32. Cum se realizează aşteptarea unui semnal?

Exerciţiul 33. Ce semnale nu pot fi tratate sau ignorate?

Exerciţiul 34. Apăsarea căror taste generează semnalul `SIGINT`? Dar pentru `SIGQUIT`?

*Exerciţiul 35**. De cîte ori va afişa pe ecran şirul "death of child!" programul de mai jos, presupunînd că apelul `fork` reuşeşte de fiecare dată ?

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

void my_handler(int signal){
  printf("death of child!\n"); fflush(stdout);
}
int main(){
  int i;
  for(i=1; i<=3; ++i)
    if(fork() == 0)
      signal(SIGCHLD, my_handler);
  while (wait(NULL) > 0) continue;
  return 0;
}
```

Exerciţiul 36. Scrieţi un program C care creează un proces nou ce scrie într-o buclă infinită numere întregi consecutive pornind de la 0; după 3 secunde procesul tată îi va trimite un semnal de oprire procesului fiu.

*Exerciţiul 37**. **Hi-ho**: scrieţi două programe, primul sa scrie pe ecran "HI-" in mod repetat, iar al doilea sa scrie "HO, " in mod repetat, si sa se foloseasca semnale `UNIX` pentru sincronizarea proceselor, astfel ca, atunci cînd sunt lansate în paralel cele două programe, pe ecran să fie afişată exact succesiunea:

HI-HO, HI-HO, HI-HO, ...

si nu alte combinatii posibile de interclasare a mesajelor afişate de cele două procese.

Capitolul 5

Comunicaţia inter-procese

5.1 Introducere. Tipuri de comunicaţie între procese

După cum se cunoaşte de la teoria sistemelor de operare, există două categorii principale de comunicaţie între procese, şi anume:

- comunicaţia prin memorie partajată ("*shared-memory communication*"):

  procesele trebuie să partajeze (*i.e.*, să aibă acces în comun la) o zonă de memorie, comunicaţia realizîndu-se prin scrierea şi citirea de valori la adrese de memorie din cadrul zonei partajate;

- comunicaţia prin schimb de mesaje ("*message-passing communication*"):

  procesele nu mai trebuie să aibă o zonă de memorie comună, ci ele comunică prin schimbul de informaţii (*mesaje*), care *circulă* de la un proces la altul prin intermediul unor *canale de comunicaţie*.

  La rîndul ei, comunicaţia prin schimb de mesaje se poate clasifica în mai multe subcategorii, în funcţie de tipul şi caracteristicile canalelor de comunicaţie utilizate. Astfel, o primă clasificare grosieră a canalelor de comunicaţie utilizate în comunicaţia prin schimb de mesaje ar consta în următoarele:

  1. *comunicaţie locală*:

  se utilizează canale ce permit comunicarea doar între procese aflate în execuţie pe un acelaşi calculator;

  2. *comunicaţie la distanţă*:

  se utilizează canale ce permit comunicarea între procese aflate în execuţie pe calculatoare diferite (conectate prin intermediul unei reţele de calculatoare).

Sistemele `UNIX/Linux` oferă mecanisme pentru toate categoriile de comunicaţie între procese enumerate mai sus. În continuare, le vom studia doar pe acelea care permit comunicaţia locală, prin schimb de mesaje. Este vorba despre aşa-numitele canale de comunicaţie `UNIX` (numite şi *pipes*, de la termenul în limba engleză).

Practic, un canal de comunicaţie `UNIX`, sau *pipe*, este o "conductă" prin care pe la un capăt se scriu mesajele (ce constau în şiruri de octeţi), iar pe la celălalt capăt acestea sunt citite – deci este vorba despre o structură de tip coadă, adica o listă FIFO (*First-In,First-Out*). Această "conductă" FIFO poate fi folosită pentru comunicare de către două sau mai multe procese, pentru a transmite date de la unul la altul.

Canalele de comunicaţie `UNIX` se împart în două subcategorii:

- canale interne:

  aceste "conducte" sunt create in memoria interna a sistemului `UNIX` respectiv;

- canale externe:

  aceste "conducte" sunt fisiere de un tip special, numit *fifo*, deci sunt pastrate in sistemul de fisiere (aceste fisiere *fifo* se mai numesc si *pipe*-uri cu nume).

5.2 Comunicaţia prin canale interne

1. **Introducere**
2. **Canale interne. Primitiva `pipe`**

5.2.1 Introducere

In aceasta secţiune a manualului ne vom referi la canalele interne, iar in următoarea secţiune le vom trata pe cele externe.

Canalele interne mai sunt numite şi *canale fără nume*, fiind, după cum am specificat deja, un fel de "conducte" create în memoria internă a sistemului, ce sunt folosite pentru comunicaţia locală între procese, prin schimb de mesaje. Datorită faptului că ele sunt *anonime* (*i.e.*, nu au nume), pot fi utilizate pentru comunicaţie doar de către procese "înrudite" prin `fork/exec`, după cum vom vedea mai încolo.

5.2.2 Canale interne. Primitiva `pipe`

Deci un canal intern este un canal de comunicaţie aflat în memorie, prin care pot comunica local două sau mai multe procese.

Crearea unui canal intern se face cu ajutorul apelului sistem `pipe`. Interfata functiei `pipe` este urmatoarea:

`int pipe(int *`p`)`

unde:
– p = parametrul efectiv de apel trebuie sa fie un tablou `int[2]` ce va fi actualizat de functie astfel:

- p`[0]` va fi descriptorul de fisier deschis pentru capătul de citire al canalului;
- p`[1]` va fi descriptorul de fisier deschis pentru capătul de scriere al canalului;

iar valoarea `int` returnata este `0`, in caz de succes (*i.e.*, daca s-a putut crea canalul), sau `-1`, in caz de eroare.

Efect: in urma executiei primitivei `pipe` se creeaza un canal intern si este deschis la ambele capete – in citire la capatul referit prin p`[0]`, respectiv in scriere la capatul referit prin p`[1]`.

Dupa crearea unui canal intern, scrierea in acest canal si citirea din el se efectueaza la fel ca pentru fisierele obisnuite.
Si anume, citirea din canal se va face prin intermediul descriptorului p`[0]` folosind functiile de citire uzuale pentru fisierele obisnuite (*i.e.*, primitiva `read`, sau se pot folosi functiile I/O din biblioteca standard de C, respectiv `fread`, `fscanf`, ş.a.m.d., dar pentru aceasta trebuie să se folosească un descriptor de tip `FILE*`, asociat descriptorului p`[0]` printr-o modalitate pe care o vom vedea mai încolo).
Iar scrierea în canal se va face prin intermediul descriptorului p`[1]` folosind functiile de scriere uzuale pentru fisierele obisnuite (*i.e.*, primitiva `write`, sau se pot folosi functiile I/O din biblioteca standard de C, respectiv `fwrite`, `fprintf`, ş.a.m.d., daca se foloseste un descriptor de tip `FILE*`, asociat descriptorului p`[1]`).

Observaţie importantă:
Pentru ca doua (sau mai multe) procese sa poata folosi un canal intern pentru a comunica, ele trebuie sa aiba la dispozitie cei doi descriptori p`[0]` si p`[1]` obtinuti prin crearea canalului, deci procesul care a creat canalul prin apelul `pipe`, va trebui sa le "transmita" cumva celuilalt proces.
De exemplu, in cazul cind se doreste sa se utilizeze un canal intern pentru comunicarea intre doua procese de tipul parinte-fiu, atunci este suficient sa se apeleze primitiva `pipe` de creare a canalului inaintea apelului primitivei `fork` de creare a procesului fiu. In acest

fel in procesul fiu avem la dispozitie cei doi descriptori necesari pentru comunicare prin intermediul acelui canal intern.
La fel se procedeaza si in cazul apelului primitivelor `exec` (deoarece descriptorii de fisiere deschise se mostenesc prin `exec`).
De asemenea, trebuie retinut faptul ca daca un proces isi inchide vreunul din capetele unui canal intern, atunci nu mai are nici o posibilitate de a redeschide ulterior acel capat al canalului.

Caracteristici si restrictii ale canalelor interne:

1. Canalul intern este un canal unidirectional, adica pe la capatul p`[1]` se scrie, iar pe la capatul p`[0]` se citeste.
   Însă toate procesele pot să scrie la capatul p`[1]`, şi să citeasca la capatul p`[0]`.

2. Unitatea de informatie pentru canalul intern este octetul. Adica, cantitatea minima de informatie ce poate fi scrisa in canal, respectiv citita din canal, este de 1 octet.

3. Canalul intern functioneaza ca o coada, adica o lista FIFO (*First-In, First-Out*), deci citirea din canal se face cu distrugerea (*i.e.*, consumul) din canal a informatiei citite.
   Asadar, citirea dintr-un canal difera de citirea din fisiere obisnuite, pentru care citirea se face fara consumul informatiei din fisier.

4. Capacitatea canalului intern este limitata la o anumita dimensiune maxima (4 Ko, 16 Ko, etc.), ce difera de la un sistem `UNIX` la altul.

5. Citirea dintr-un canal intern (cu primitiva `read`) funcţionează în felul următor:

   - Apelul `read` va citi din canal si va returna imediat, fara sa se blocheze, numai daca mai este suficienta informatie in canal, iar in acest caz valoarea returnata reprezinta numarul de octeti cititi din canal.
   - Altfel, daca canalul este gol, sau nu contine suficienta informatie, apelul de citire `read` va ramine blocat pina cind va avea suficienta informatie in canal pentru a putea citi cantitatea de informatie specificata, ceea ce se va intimpla in momentul cind alt proces va scrie in canal.
   - Alt caz de exceptie la citire, pe linga cazul golirii canalului:
     daca un proces incearca sa citeasca din canal si nici un proces nu mai este capabil sa scrie in canal vreodata (deoarece toate procesele si-au inchis deja capatul de scriere), atunci apelul `read` returneaza imediat valoarea `0` corespunzatoare faptului ca a citit `EOF` din canal.
     În concluzie, pentru a se putea citi `EOF` din canal, trebuie ca mai intii toate procesele sa inchida canalul in scriere (adica sa inchida descriptorul p`[1]`).

   Observaţie: la fel se comportă la citirea din canale interne si functiile de citire de nivel înalt (`fread`, `fscanf`, etc.), doar că acestea lucrează *buffer*-izat.

6. Scrierea intr-un canal intern (cu primitiva `write`) funcţionează în felul următor:

- Apelul `write` va scrie in canal si va returna imediat, fara sa se blocheze, numai daca mai este suficient spatiu liber in canal, iar in acest caz valoarea returnata reprezinta numarul de octeti efectiv scrisi in canal (care poate sa nu coincida intotdeauna cu numarul de octeti ce se doreau a se scrie, caci pot apare erori I/O).
- Altfel, daca canalul este plin, sau nu contine suficient spatiu liber, apelul de scriere `write` va ramine blocat pina cind va avea suficient spatiu liber in canal pentru a putea scrie informatia specificata ca argument, ceea ce se va intimpla in momentul cind alt proces va citi din canal.
- Alt caz de exceptie la scriere, pe linga cazul umplerii canalului:
  daca un proces incearca sa scrie in canal si nici un proces nu mai este capabil sa citeasca din canal vreodata (deoarece toate procesele si-au inchis deja capatul de citire), atunci sistemul va trimite acelui proces semnalul `SIGPIPE`, ce cauzeaza intreruperea sa si afisarea pe ecran a mesajului "Broken pipe".

Observaţie: la fel se comportă la scrierea în canale interne si functiile de citire de nivel înalt (`fwrite`, `fprintf`, etc.), doar că acestea lucrează *buffer*-izat. Acest fapt cauzează uneori erori dificil de depistat, datorate neatenţiei programatorului, care poate uita uneori aspectele legate de modul de lucru *buffer*-izat al funcţiilor de scriere din biblioteca standard de C, *i.e.* poate uita să forţeze "golirea" *buffer*-ului în canal cu ajutorul funcţiei `fflush`, imediat după apelul funcţiei de scriere propriu-zise.

Observaţie:
Cele afirmate mai sus, despre blocarea apelurilor de citire sau de scriere in cazul canalului gol, respectiv plin, corespund comportamentului implicit, **blocant**, al canalelor interne.
Insa, exista posibilitatea modificarii acestui comportament implicit, intr-un comportament **neblocant**, situatie in care apelurile de citire sau de scriere nu mai ramin blocate in cazul canalului gol, respectiv plin, ci returneaza imediat valoarea `-1`, si seteaza corespunzator variabila `errno`.

Modificarea comportamentului implicit in comportament **neblocant** se realizeaza prin setarea atributului `O_NONBLOCK` pentru descriptorul corespunzator acelui capat al canalului intern pentru care se doreste modificarea comportamentului, cu ajutorul primitivei `fcntl`.
Spre exemplu, apelul

```
fcntl(p[1],F_SETFL,O_NONBLOCK);
```

va seta atributul `O_NONBLOCK` pentru capatul de scriere al canalului intern referit de variabila *p*.

Atenţie: după cum am mai spus, functiile I/O de nivel inalt (*i.e.*, `fread`/`fwrite`, `fscanf`/`fprintf`, şi celelalte din biblioteca standard de C) lucreaza *buffer*-izat. Ca atare, la scrierea intr-un canal folosind functiile `fwrite`, `fprintf`, etc., informatia nu ajunge imediat in canal in urma apelului, ci doar in *buffer*-ul asociat acelui descriptor, iar "golirea" *buffer*-ului in canal se va intimpla abia in momentul umplerii *buffer*-ului, sau la intilnirea

caracterului '\n'(*newline*) in specificatorul de format al functiei de scriere, sau la apelul functiei `fflush` pentru acel descriptor.
Prin urmare, daca se doreste garantarea faptului ca informatia ajunge in canal imediat in urma apelului de scriere cu functii de nivel inalt, trebuie sa se apeleze, imediat dupa apelul de scriere, si functia `fflush` pentru descriptorul asociat capatului de scriere al acelui canal.

Exemplu. Urmatorul program exemplifica modul de utilizare a unui canal intern pentru comunicatia intre doua procese, cu observatia ca programul foloseste primitivele `read` si `write` (*i.e.*, functiile I/O de nivel scazut, ne-*buffer*-izate) pentru a citi din canal, respectiv pentru a scrie in canal.

```
/*
  File: pipe-ex1.c

  Exemplu de utilizare a unui pipe intern pentru comunicatia intre
  doua procese, folosind functii I/O de nivel scazut (nebufferizate).
*/

#include<stdio.h>
#include<errno.h>
extern int errno;

#define NMAX 1000

int main(void)
{
  int pid, p[2];
  char ch;

  /* creare pipe intern */
  if(pipe(p) == -1)
  {
    fprintf(stderr,"Error: can't open a channel, errno=%d\n",errno);
    exit(1);
  }

  /* creare proces fiu */
  if( (pid=fork()) == -1)
  {
    fprintf(stderr,"Error: can't create a child!\n");
    exit(2);
  }

  if(pid)
  { /* in tata */

    /* tatal isi inchide capatul Read */
    close(p[0]);
```

```
    /* citeste caractere de la tastatura,
       pentru terminare: CTRL+D (i.e. EOF in Unix),
       si le transmite doar pe acelea care sunt litere mici */
    while( (ch=getchar()) != EOF)
      if((ch>='a') && (ch<='z'))
        write(p[1],&ch,1);

    /* tatal isi inchide capatul Write,
       pentru ca fiul sa poata citi EOF din pipe */
    close(p[1]);

    /* asteapta terminarea fiului */
    wait(NULL);
  }
  else
  { /* in fiu */
    char buffer[NMAX];
    int nIndex = 0;

    /* fiul isi inchide capatul Write */
    close(p[1]);

    /* fiul citeste caracterele din pipe si salveaza in buffer,
       pina depisteaza EOF, apoi afiseaza continutul bufferului. */

    while( read(p[0],&ch,1) != 0)
      if(nIndex < NMAX)
        buffer[nIndex++] = ch;

    buffer[ (nIndex==NMAX) ? NMAX-1 : nIndex ] = '\0';
    printf("Fiu: am citit buffer=%s\n",buffer);

    /* fiul isi inchide capatul Read */
    close(p[0]);
        /* Obs: nici nu mai era nevoie de acest close explicit, deoarece
        oricum toti descriptorii sunt inchisi la terminarea programului.*/
  }
  return 0;
}
```

Efectul acestui program:
Procesul tata citeste un sir de caractere de la tastatura, sir terminat cu combinatia de taste `CTRL+D` (*i.e.*, caracterul `EOF` in `UNIX`), si le transmite procesului fiu, prin intermediul canalului, doar pe acelea care sunt litere mici. Iar procesul fiu citeste din canal caracterele trasmise de procesul parinte si le afiseaza pe ecran.

Exemplu. Programul urmator este un alt exemplu de utilizare a unui canal intern pentru comunicatia intre doua procese, cu observatia ca programul foloseste functiile de bibliotecă `fscanf` si `fprintf` (*i.e.*, funcţiile I/O de nivel inalt, *buffer*-izate) pentru a citi din canal, respectiv pentru a scrie in canal.

```
/*
  File: pipe-ex2.c

  Exemplu de utilizare a unui pipe intern pentru comunicatia intre
  doua procese, folosind functii I/O de nivel inalt (bufferizate).
*/
#include<stdio.h>
#include<errno.h>
extern int errno;

int main(void)
{
  int pid, nr, p[2];
  FILE *fin,*fout;

  if(pipe(p) == -1)
  {
    fprintf(stderr,"Error: can't open channel, err=%d\n",errno);
    exit(1);
  }

  /* atasare descriptori de tip FILE* la cei de tip int */
  fin = fdopen(p[0],"r"); /* capatul de citire  */
  fout= fdopen(p[1],"w"); /* capatul de scriere */

  /* creare proces fiu */
  if( (pid=fork()) == -1)
  {
    fprintf(stderr,"Error: can't create a child!\n");
    exit(2);
  }

  if(pid)
  { /* in tata */

    /* tatal isi inchide capatul Read */
    fclose(fin);

    /* citeste numere de la tastatura,
       pentru terminare: CTRL+D (i.e. EOF in Unix),
       si le transmite prin pipe procesului fiu.
       OBSERVATIE: in pipe numerele sunt scrise formatat, nu binar, si
       de aceea trebuie separate printr-un caracter care nu-i cifra
       (in acest caz am folosit '\n') pentru a nu se "amesteca"
       cifrele de la numere diferite atunci cind sunt citite din pipe! */
    while(scanf("%d",&nr) != EOF)
    {
      fprintf(fout,"%d\n",nr);
      fflush(fout);
    }

    /* tatal isi inchide capatul Write,
```

```
      pentru ca fiul sa poata citi EOF din pipe */
    fclose(fout);

    /* asteapta terminarea fiului */
    wait(NULL);
  }
  else
  { /* in fiu */

    /* fiul isi inchide capatul Write */
    fclose(fout);

    /* fiul citeste numerele din pipe si le afiseaza pe ecran,
       pina depisteaza EOF in pipe.
       OBS: conform celor de mai sus, caracterul '\n' este folosit
       ca separator de numere ! */
    while(fscanf(fin,"%d",&nr) != EOF)
    {
      printf("%d\n",nr);
      fflush(stdout);
    }

    /* fiul isi inchide capatul Read */
    fclose(fin);
      /* Obs: nici nu mai era nevoie de acest fclose explicit, deoarece
       oricum toti descriptorii sunt inchisi la terminarea programului.*/
  }
  return 0;
}
```

Efectul acestui program:
Procesul tata citeste un sir de numere de la tastatura, sir terminat cu combinatia de taste `CTRL+D` (*i.e.*, caracterul `EOF` in `UNIX`), si le transmite procesului fiu, prin intermediul canalului. Iar procesul fiu citeste din canal numerele trasmise de procesul parinte si le afiseaza pe ecran.

Observaţie: deoarece acest program foloseste functiile I/O de nivel inalt, este necesara conversia descriptorilor de fisiere de la tipul `int` la tipul `FILE*`, lucru realizat cu ajutorul funcţiei de bibliotecă `fdopen`.

Altă observaţie: în acest exemplu, numerele au fost scrise in canal ca text (*i.e.*, ca secventa a cifrelor care le compun) si nu ca reprezentare binara. Din acest motiv ele trebuie separate printr-un caracter care nu este cifra (in program s-a folosit caracterul '\n', dar poate fi folosit oricare altul), cu scopul ca aceste numere sa poata fi citite la destinatar in mod corect, fara ca cifrele lor sa se "amestece" intre ele.

5.3 Comunicaţia prin canale externe

1. **Introducere**
2. **Canale externe (fişiere *fifo*)**
3. **Aplicaţie: implementarea unui semafor**
4. **Aplicaţie: programe de tip client-server**

5.3.1 Introducere

In aceasta secţiune a manualului ne vom referi la canalele externe, dupa ce în secţiunea precedentă le-am tratat pe cele interne.

Canalele externe, numite si *canale cu nume*, sunt tot un fel de "conducte" pentru comunicaţia locală între procese prin schimb de mesaje, la fel ca si canalele interne, doar ca in cazul canalelor externe avem de a face cu niste "conducte" ce sunt fisiere `UNIX` de un tip special, numit *fifo*, deci sunt pastrate in sistemul de fisiere (aceste fisiere *fifo* se mai numesc si *pipe*-uri cu nume).

5.3.2 Canale externe (fişiere *fifo*)

Deci un canal extern este un canal de comunicaţie prin care pot comunica local două sau mai multe procese, comunicaţia făcîndu-se în acest caz printr-un fisier `UNIX` de tip *fifo*.

Comunicaţia între procese prin intermediul unui canal *fifo* poate avea loc daca acele procese cunosc numele fisierului *fifo* respectiv, deci nu mai avem restricţia de la canale interne, aceea că procesele trebuiau să fie "înrudite" prin `fork/exec`.

Modul de utilizare al unui fisier *fifo* este similar ca la fisierele obisnuite: mai intii se deschide fisierul, apoi se scrie in el si/sau se citeste din el, iar la sfirsit se inchide fisierul.

Crearea unui fisier *fifo* se face cu ajutorul primitivei `mkfifo` (sau, echivalent, cu primitiva `mknod` apelată cu *flag*-ul `S_IFIFO`). Urmatorul program exemplifica modul de creare a unui fisier *fifo*.

```
/*
  File: mkf.c (creare fisier fifo)
*/
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<errno.h>
extern int errno;

int main(int argc, char** argv)
{
  if(argc != 2)
  {
    fprintf(stderr,"Sintaxa apel: mkf nume_fifo\n");
    exit(1);
  }

  if( mkfifo(argv[1], 0666) == -1 )
  /* sau, echivalent: if( mknod(argv[1], S_IFIFO | 0666, 0) == -1 ) */
  {
    if(errno == 17)   // 17 = errno for "File exists"
    {
       fprintf(stdout,"Note: fifo %s exista deja !\n",argv[1]);
       exit(0);
    }
    else
    {
       fprintf(stderr,"Eroare: creare fifo imposibila, errno=%d\n",errno);
       perror(0);
       exit(2);
    }
  }
  return 0;
}
```

Alternativ, un fisier *fifo* poate fi creat si direct de la prompterul *shell*-ului, folosind comenzile `mkfifo` sau `mknod`.

Restul operatiilor asupra canalelor *fifo* se fac la fel ca la fisiere obisnuite, fie cu primitivele I/O de nivel scazut (*i.e.*, `open`, `read`, `write`, `close`), fie cu functiile I/O de nivel înalt din biblioteca standard de C (*i.e.*, `fopen`, `fread`/`fscanf`, `fwrite`/`fprintf`, `fclose`, ş.a.).

Prin urmare, deschiderea unui fisier *fifo* se poate face cu apelul functiei `open` sau `fopen`, intr-unul din urmatoarele trei moduri posibile:
• *read &write* (deschiderea ambelor capete ale canalului),
• *read-only* (deschiderea doar a capatului de citire),
• sau *write-only* (deschiderea doar a capatului de scriere),
modul de deschidere fiind specificat prin parametrul transmis functiei de deschidere.

Observaţie importantă:
În mod implicit, deschiderea se face in mod *blocant*: o deschidere *read-only* trebuie sa se "sincronizeze" cu una *write-only*. Cu alte cuvinte, daca un proces incearca o deschidere a unui capat al canalului extern, apelul functiei de deschidere ramine blocat (*i.e.*, functia nu returneaza) pina cind un alt proces va deschide celalalt capat al canalului extern.

Si in cazul canalelor externe apar restrictiile si problemele de la canale interne si anume:

Caracteristici si restrictii ale canalelor externe:

1. Canalul extern este un canal unidirectional, adica pe la un capat se scrie, iar pe la capatul opus se citeste.
   Insa toate procesele pot să scrie la capatul de scriere, şi să citeasca la capatul de citire.

2. Unitatea de informatie pentru canalul extern este octetul. Adica, cantitatea minima de informatie ce poate fi scrisa in canal, respectiv citita din canal, este de 1 octet.

3. Canalul extern functioneaza ca o coada, adica o lista FIFO (*First-In, First-Out*), deci citirea din canal se face cu distrugerea (*i.e.*, consumul) din canal a informatiei citite.
   Asadar, citirea dintr-un canal extern (*i.e.*, fisier *fifo*) difera de citirea din fisiere obisnuite, pentru care citirea se face fara consumul informatiei din fisier.

4. Capacitatea canalului extern este limitata la o anumita dimensiune maxima (4 Ko, 16 Ko, etc.), ce difera de la un sistem `UNIX` la altul.

5. Citirea dintr-un canal extern (cu primitiva `read`) funcţionează în felul următor:

   - Apelul `read` va citi din canal si va returna imediat, fara sa se blocheze, numai daca mai este suficienta informatie in canal, iar in acest caz valoarea returnata reprezinta numarul de octeti cititi din canal.
   - Altfel, daca canalul este gol, sau nu contine suficienta informatie, apelul de citire `read` va ramine blocat pina cind va avea suficienta informatie in canal pentru a putea citi cantitatea de informatie specificata, ceea ce se va intimpla in momentul cind alt proces va scrie in canal.
   - Alt caz de exceptie la citire, pe linga cazul golirii canalului:
     daca un proces incearca sa citeasca din canal si nici un proces nu mai este capabil sa scrie in canal vreodata (deoarece toate procesele si-au inchis deja capatul de scriere), atunci apelul `read` returneaza imediat valoarea `0` corespunzatoare faptului ca a citit `EOF` din canal.
     În concluzie, pentru a se putea citi `EOF` din canalul *fifo*, trebuie ca mai intii toate procesele sa inchida canalul in scriere (adica sa inchida descriptorul corespunzator capatului de scriere).

   Observaţie: la fel se comportă la citirea din canale interne si functiile de citire de nivel înalt (`fread`, `fscanf`, etc.), doar că acestea lucrează *buffer*-izat.

6. Scrierea intr-un canal intern (cu primitiva `write`) funcţionează în felul următor:

   - Apelul `write` va scrie in canal si va returna imediat, fara sa se blocheze, numai daca mai este suficient spatiu liber in canal, iar in acest caz valoarea returnata reprezinta numarul de octeti efectiv scrisi in canal (care poate sa nu coincida intotdeauna cu numarul de octeti ce se doreau a se scrie, caci pot apare erori I/O).
   - Altfel, daca canalul este plin, sau nu contine suficient spatiu liber, apelul de scriere `write` va ramine blocat pina cind va avea suficient spatiu liber in canal pentru a putea scrie informatia specificata ca argument, ceea ce se va intimpla in momentul cind alt proces va citi din canal.
   - Alt caz de exceptie la scriere, pe linga cazul umplerii canalului:
     daca un proces incearca sa scrie in canal si nici un proces nu mai este capabil sa citeasca din canal vreodata (deoarece toate procesele si-au inchis deja capatul de citire), atunci sistemul va trimite acelui proces semnalul `SIGPIPE`, ce cauzeaza intreruperea sa si afisarea pe ecran a mesajului "Broken pipe".

   Observaţie: la fel se comportă la scrierea în canale interne si functiile de citire de nivel înalt (`fwrite`, `fprintf`, etc.), doar că acestea lucrează *buffer*-izat. Acest fapt cauzează uneori erori dificil de depistat, datorate neatenţiei programatorului, care poate uita uneori aspectele legate de modul de lucru *buffer*-izat al funcţiilor de scriere din biblioteca standard de C, *i.e.* poate uita să forţeze "golirea" *buffer*-ului în canal cu ajutorul funcţiei `fflush`, imediat după apelul funcţiei de scriere propriu-zise.

Observaţie:
La fel ca la canale interne, cele afirmate mai sus, despre blocarea apelurilor de citire sau de scriere in cazul canalului gol, respectiv plin, corespund comportamentului implicit, **blocant**, al canalelor externe.
Insa, exista posibilitatea modificarii acestui comportament implicit, intr-un comportament **neblocant**, situatie in care apelurile de citire sau de scriere nu mai ramin blocate in cazul canalului gol, respectiv plin, ci returneaza imediat valoarea `-1`, si seteaza corespunzator variabila `errno`.

Modificarea comportamentului implicit in comportament **neblocant** se realizeaza prin setarea atributului `O_NONBLOCK` pentru descriptorul corespunzator acelui capat al canalului extern pentru care se doreste modificarea comportamentului, ceea ce se poate face fie direct la deschiderea canalului, fie dupa deschidere cu ajutorul primitivei `fcntl`. Spre exemplu, apelul

fd_out `= open("`*canal_fifo", O_WRONLY | O_NONBLOCK);*

va seta la deschidere atributul `O_NONBLOCK` pentru capatul de scriere al canalului extern cu numele `canal_fifo` din directorul curent de lucru. Iar secvenţa de cod

```
fd_out = open("canal_fifo",O_WRONLY);
...
fcntl(fd_out,F_SETFL,O_NONBLOCK);
```

va seta, dupa deschidere, atributul `O_NONBLOCK` pentru capatul de scriere al canalului extern cu numele `canal_fifo` din directorul curent de lucru.

Atenţie: după cum am mai spus şi la canale interne, functiile I/O de nivel inalt (*i.e.*, `fread`/`fwrite`, `fscanf`/`fprintf`, şi celelalte din biblioteca standard de C) lucreaza *buffer*-izat. Ca atare, la scrierea intr-un canal extern folosind functiile `fwrite`, `fprintf`, etc., informatia nu ajunge imediat in canal in urma apelului, ci doar in *buffer*-ul asociat acelui descriptor, iar "golirea" *buffer*-ului in canal se va intimpla abia in momentul umplerii *buffer*-ului, sau la intilnirea caracterului '\n'(*newline*) in specificatorul de format al functiei de scriere, sau la apelul functiei `fflush` pentru acel descriptor.
Prin urmare, daca se doreste garantarea faptului ca informatia ajunge in canal imediat in urma apelulului de scriere cu functii de nivel inalt, trebuie sa se apeleze, imediat dupa apelul de scriere, si functia `fflush` pentru descriptorul asociat capatului de scriere al acelui canal.

Deosebiri ale canalelor externe faţă de cele interne:

1. Functia de creare a unui canal extern nu produce si deschiderea automata a celor doua capete, acestea trebuie dupa creare sa fie deschise explicit prin apelul unei funcţii de deschidere a unui fişier.

2. Un canal extern poate fi deschis, la oricare din capete, de orice proces, indiferent daca acel proces are sau nu vreo legatura de rudenie (prin `fork`/`exec`) cu procesul care a creat canalul extern.
   Aceasta este posibil deoarece un proces trebuie doar sa cunoasca numele fisierului *fifo* pe care doreste sa-l deschida, pentru a-l putea deschide. Bineinteles, procesul respectiv mai trebuie sa aiba si drepturi de acces pentru acel fisier *fifo*.

3. Dupa ce un proces inchide un capat al unui canal *fifo*, acel proces poate redeschide din nou acel capat pentru a face alte operatii I/O asupra sa.

Pentru mai multe detalii despre canalele *fifo*, va recomand citirea paginii de *help* a comenzii de creare a unui canal extern:
`UNIX> man 3 mkfifo`
si a paginii generale despre canale externe:
`UNIX> man fifo`

5.3.3 Aplicaţie: implementarea unui semafor

În continuare vom da un exemplu de utilizare a canalelor cu nume, şi anume ele pot fi folosite pentru implementarea unui semafor.

Semaforul este o structura de control clasica in programarea concurenta, ce permite executia exclusiva a unei secvente de cod, *i.e.* controleaza accesul intr-o *sectiune critica*.

Executia exclusiva a unei secvente de cod se bazeaza pe doua rutine exportate de tipul de data semafor: `sem_enter` si `sem_exit` (ele mai sunt întîlnite în literatura de specialitate şi sub numele de `wait(`*semafor*`)` şi respectiv `signal(`*semafor*`)`).
Acestea vor fi apelate in programe in ordinea urmatoare:

```
sem_enter();
...
sem_exit();
```

Portiunea de cod dintre cele doua apeluri se numeste *portiune critica*, ea mai fiind numita si *zona critica* sau *sectiune critica*.

Prin folosirea semaforului avem certitudinea ca portiunea critica se va executa in exclusivitate de primul proces care a executat cu succes apelul `sem_enter`.
Deci primul proces care executa cu succes `sem_enter`, trece de punctul de sincronizare si capata acces in portiunea critica. Celelalte procese care au ajuns la punctul de sincronizare (*i.e.*, la apelul `sem_enter`), intra intr-o coada de asteptare. Dupa ce acel proces a iesit din portiunea critica (*i.e.*, a executat `sem_exit`), este deblocat primul proces din coada de asteptare, adica i se permite accesul in portiunea critica, ş.a.m.d.

Cind apare nevoia utilizarii unei zone critice?
In programarea paralela sunt situatii cind două sau mai multe procese (ce executa fie acelasi cod, fie programe diferite), executate in paralel, trebuie sa partajeze (*i.e.*, sa utilizeze in comun) o resursa nepartajabila (*i.e.*, o resursa care poate fi utilizata de cel mult un proces la un moment dat), sau sa comunice intre ele, sau sa execute exclusiv o secventa de cod, etc., deci situatii in care apar probleme de sincronizare intre procese.

Exemplu. Sa presupunem, spre exemplu, ca avem de implementat o structura de tip lista simplu inlantuita, utilizabila concurent de mai multe procese:

$\texttt{CapLista} \longrightarrow (I_1,L_1) \longrightarrow (I_2,L_2) \longrightarrow \ldots \longrightarrow (I_n,L_n)$

unde un element al listei este o pereche de forma `E=(I,L)`, fiind formată din `I`=informatia propriu-zisa, si `L`=legatura (pointerul) catre urmatorul element din lista.

Operatia de inserare in capul listei al unui element $E_0=(I_0,L_0)$ s-ar face prin urmatoarea secventa de pasi:

1. L_0 := CapLista //leaga $E_0 \longrightarrow E_1$
2. CapLista := E_0 //rupe legatura CapLista $\longrightarrow E_1$ si leaga CapLista $\longrightarrow E_0$

Daca doua (sau mai multe) procese ar executa concurent operatia de inserare in capul listei, se pot obtine erori – prin anumite amestecari ale secventelor de pasi executate de

cele doua procese se obtin date eronate.
Din acest motiv este nevoie ca operatia de inserare sa se execute in mod *atomic* (*i.e.*, exclusiv, adică să fie neintreruptibilă). Pentru aceasta, trebuie sa se trateze prin exclusivitate accesul in read si write la variabila `CapLista`.

Rutina de inserare in capul listei ar trebui sa arate cam în felul următor:

```
semafor s;     // declarare data de tip semafor
...
alloc(E0);     // alocare memorie pentru elementul E0
I0:= info;     // asignare informatie in cimpul I0
s.sem_enter(); // punctul de sincronizare (punctul de intrare)
L0:=CapLista;  // modificare cimpul L0
CapLista:=E0;  // modificare variabila CapLista
s.sem_exit();  // punctul de iesire
```

unde `s` este o data de tip semafor comuna acelor procese care vor executa concurent operatii de inserare in lista, iniţializat cu valoarea `1` (deci un *semafor binar*).

Observaţie importantă:
Apelurile `sem_enter` si `sem_exit` trebuie sa fie perechi, pe orice fir posibil de execuţie! Daca un proces executa `sem_exit` fara sa fi executat anterior `sem_enter`, iar un al doilea proces este in portiunea critica, atunci executia acestei portiuni de catre al doilea proces nu mai este exclusiva (un alt proces poate intra in portiunea critica, in timp ce al doilea proces nu a parasit-o inca).

O alta situatie de pereche punct intrare – punct iesire este urmatoarea:

```
semafor s;
...
s.sem_enter();
if( TEST )
{
  s.sem_exit();
  ...
}
else
{
  s.sem_exit();
  ...
}
```

Deci daca avem nevoie de executie exclusiva doar pentru testul din instructiunea `if`, nu si pentru actiunile de pe cele doua ramuri `then-else`, atunci este indicat sa se puna apelul `sem_exit` pe cele doua ramuri imediat dupa test.

Concluzie: deblocarea semaforului trebuie facuta cit mai repede, imediat ce acest lucru este posibil (pentru a permite cit mai repede celorlalte procese sa intre in portiunea critica).

Atenţie: daca se folosesc doua semafoare "imbricate", ca mai jos, poate apare fenomenul de interblocaj (*deadlock*):

Procesul P1:

```
...
s1.sem_enter();
...
s2.sem_enter();
...
{s1,s2}.sem_exit();
...
```

Procesul P2:

```
...
s2.sem_enter();
...
s1.sem_enter();
...
{s1,s2}.sem_exit();  // nu conteaza ordinea
...                  // apelurilor sem_exit()
```

O secventa de executie posibila este urmatoarea: procesul P1 intra in zona sa critica controlata de semaforul `s1`, P2 intra in zona sa critica controlată de `s2`, apoi P1 incearca sa intre in zona sa critica controlată de `s2` (dar ramine blocat, deoarece semaforul `s2` este "ocupat" de procesul P2), iar P2 incearca sa intre in zona sa critica controlată de `s1` (dar ramine blocat, deoarece semaforul `s1` este "ocupat" de procesul P1); deci a aparut o situatie de interblocaj.

Observaţie: din acest motiv in unele sisteme ce utilizeaza semafoare nu este permis ca in sectiunea critica a unui semafor sa se apeleze intrarea intr-un alt semafor.

Indicaţii generale de folosire a semafoarelor:
– portiunea critica sa fie cit mai scurta;
– in portiunea critica sa nu se intimple erori (cicluri infinite, etc.), deoarece acestea duc la blocarea intregului sistem de procese ce utilizeaza acel semafor.

Şi acum, să vedem cum am putea implementa un semafor folosind canale *fifo*.

Ideea este foarte simplă: iniţializarea semaforului ar consta în crearea unui fişier *fifo* de către un proces cu rol de *supervizor* (poate fi oricare dintre procesele ce vor folosi acel semafor, sau poate fi un proces separat); iniţial acest proces *supervizor* va scrie în canal 1 caracter oarecare, dacă e vorba de un seminar binar (respectiv n caractere oarecare, dacă e vorba de un seminar general n-ar), şi va păstra deschise ambele capete ale canalului pe toată durata de execuţie a proceselor ce vor folosi acel semafor (cu scopul de a nu se pierde pe parcurs informaţia din canal datorită inexistenţei la un moment dat pentru fiecare capăt a măcar unui proces care să-l aibă deschis).

Operaţia `sem_enter` va consta în citirea unui caracter din fişierul *fifo* (mai precis, întîi deschiderea lui, urmată de citirea efectivă a unui octet, şi apoi eventual închiderea fişierului). Citirea se va face în modul implicit, *blocant*, ceea ce va asigura aşteptarea procesului la punctul de intrare în zona sa critică în situaţia cînd semaforul este "pe roşu", adică dacă canalul *fifo* este gol.
Operaţia complementară, `sem_exit`, va consta în scrierea unui caracter în fişierul *fifo* (mai precis, întîi deschiderea lui, urmată de scrierea efectivă a unui octet, şi apoi eventual închiderea fişierului).

5.3.4 Aplicaţie: programe de tip client-server

În continuare vom da un alt exemplu de utilizare a canalelor cu nume, şi anume ele pot fi folosite pentru implementarea programelor de tip client-server.

O *aplicaţie de tip client-server* este compusă din două componente:

- **serverul**:

  este un program care dispune de un anumit număr de *servicii* (*i.e.* funcţii/operaţii), pe care le pune la dispoziţia clienţilor.

- **clientul**:

  este un program care "interoghează" serverul, solicitindu-i *efectuarea unui serviciu* (dintre cele puse la dispoziţie de acel server).

Exemplu. *Browser*-ele pe care le folosiţi pentru a naviga pe INTERNET (cum ar fi, spre exemplu, MS Internet Explorer-ul, Netscape Navigator/Mozilla, sau Opera) sunt un exemplu de program client, care se conectează la un program server, numit *server de web*, solicitîndu-i transmiterea unei pagini *web*, care apoi este afişată în fereastra grafică a *browser*-ului.

Folosirea unei aplicaţii de tip client-server se face în felul următor:
Programul server va fi rulat în *background*, şi va sta în aşteptarea cererilor din partea clienţilor, putînd servi mai mulţi clienţi simultan.
Iar clienţii vor putea fi rulaţi mai mulţi simultan, din acelaşi cont sau din conturi utilizator diferite, şi se vor conecta la serverul rulat în *background*.
Deci putem avea la un moment dat mai multe procese client, care încearcă, fiecare independent de celelalte, să folosească serviciile puse la dispoziţie de procesul server.

Observaţie: în realitate, programul server este rulat pe un anumit calculator, iar clienţii pe diverse alte calculatoare, conectate la INTERNET, comunicaţia realizîndu-se folosind *socket*-uri, prin intermediul reţelelor de calculatoare. Însă putem simula aceasta folosind comunicaţie prin canale externe (*fifo*-uri) şi executînd toate procesele (*i.e.*, serverul şi clienţii) pe un acelaşi calculator.

Tipurile de servere existente în realitate, d.p.d.v. al servirii "simultane" a mai multor clienţi, se împart în două categorii:

- server *iterativ*

  Cît timp durează efectuarea unui serviciu (*i.e.*, rezolvarea unui client), serverul este blocat: nu poate răspunde cererilor venite din partea altor clienţi. Deci nu poate rezolva mai mulţi clienţi în acelaşi timp!

- server *concurent*

  Pe toată durata de timp necesară pentru efectuarea unui serviciu (*i.e.*, rezolvarea unui client), serverul nu este blocat, ci poate răspunde cererilor venite din partea altor clienţi. Deci poate rezolva mai mulţi clienţi în acelaşi timp!

Detalii legate de implementare:

- Pentru comunicarea între procesele client şi procesul server este necesar să se folosească, drept canale de comunicaţie, canalele externe (adică *pipe*-urile cu nume), sau se mai pot utiliza *socket*-uri, aşa cum se întîmplă în realitate, după cum am discutat mai sus. *Atenţie*: nu se pot folosi *pipe*-uri interne, deoarece procesul server şi procesele clienţi nu sunt înrudite (prin `fork/exec`).

- Mai mult, trebuie avut grijă la gestiunea drepturilor de acces la fişierele *fifo* folosite pentru comunicaţie, astfel încît să se poată rula procesele client simultan, din diferite conturi utilizator (deci, de pe terminale de lucru diferite, conectate - evident - la acelaşi calculator, în cazul de faţă la serverul `UNIX` al studenţilor, `fenrir`, pe care lucraţi dumneavoastră).

- Un alt aspect legat tot de comunicaţie: serverul nu cunoaşte în avans clienţii ce se vor conecta la el pentru a le oferi servicii, în schimb clientul trebuie să cunoască serverul la care se va conecta pentru a beneficia de serviciul oferit de el. Ce înseamnă aceasta d.p.d.v. practic?
  Serverul va crea un canal *fifo* cu un nume fixat, cunoscut în programul client, şi va aştepta sosirea informaţiilor pe acest canal. Un client oarecare se va conecta la acest canal *fifo* cunoscut şi va transmite informaţii de identificare a sa, care vor fi folosite ulterior pentru realizarea efectivă a comunicaţiilor implicate de serviciul solicitat (s-ar putea să fie nevoie de canale suplimentare, particulare pentru acel client, ca să nu se amestece comunicaţiile destinate unui client cu cele destinate altui client conectat la server în acelaşi timp cu primul).

- Referitor la modul de servire "simultană" a mai multor clienţi, există diferenţe de implementare a serverului, în cazul serverelor iterative faţă de cele concurente. Mai precis, pentru serverele de tip iterativ este suficient un singur proces `UNIX`, pe cînd pentru serverele de tip concurent este nevoie de mai multe procese `UNIX`: un proces *master*, care aşteaptă sosirea cererilor din partea clienţilor, şi la fiecare cerere sosită, el va crea un nou proces fiu, un *slave* care va fi responsabil cu rezolvarea propriu-zisă a clientului respectiv, iar *master*-ul va relua imediat aşteptarea unei noi cereri, fără să aştepte terminarea procesului fiu.
  Observaţie: ca o analogie cu execuţia comenzilor tastate la prompterul oferit de *shell*-urile `UNIX`, serverul iterativ corespunde execuţiei de comenzi în *foreground*, iar serverul concurent corespunde execuţiei de comenzi în *background* (situaţie în care prompterul este reafişat imediat, fără să se aştepte terminarea execuţiei comenzii).

5.4 Alte mecanisme pentru comunicaţia inter-procese

Pe lîngă *pipe*-urile interne şi externe, studiate deja, mai există şi alte mecanisme utile pentru comunicaţia între procese:

- *Socket*-uri:

  Este vorba despre o altă categorie de canale de comunicaţie, diferită de *pipe*-urile interne şi externe, categorie ce are avantajul de a putea fi folosită pentru transmisia de date între procese aflate în execuţie pe calculatoare diferite, conectate între ele printr-o reţea de calculatoare. Această categorie este reprezentată de aşa-numitele *socket*-uri, introduse pentru prima dată în varianta `BSD` de `UNIX`.

  Socket-urile sunt tot un tip special de fişiere `UNIX`, la fel ca şi fişierele *fifo*, putînd fi exploatate şi ele prin intermediul interfeţei standard a primitivelor de lucru cu fişiere (`read`, `write`, etc.), principala deosebire faţă de fişierele *fifo* (*i.e.*, canalele externe) fiind aceea că pot fi folosite pentru comunicaţia între procese aflate în execuţie pe calculatoare diferite, spre deosebire de canalele externe (şi cele interne), ce pot fi folosite doar pentru comunicaţia între procese aflate în execuţie pe un acelaşi calculator.

  Particularităţile folosirii *socket*-urilor, şi în general întreaga problematică a comunicaţiei între procese printr-o reţea de calculatoare, nu constituie subiectul acestui manual, ele urmînd a fi abordate în cadrul unei discipline pe care o veţi studia ulterior, intitulată "Reţele de calculatoare".

- biblioteca `IPC`:

  Un alt mecanism de comunicaţie inter-procese (de fapt mai multe) este pus la dispoziţie de biblioteca `IPC` (care este o prescurtare de la *Inter-Process Communication*). Această bibliotecă oferă trei facilităţi utile: memorie partajată, semafoare, şi cozi de mesaje. Din lipsă de spaţiu, biblioteca `IPC` nu este tratată în cadrul acestui manual, rămînînd ca studiu individual pentru cei care doresc să-şi însuşească mai multe mecanisme utile în programarea concurentă sub `UNIX`/`Linux`. Ca punct de plecare în studiul individual, vă recomand citirea paginii de manual a bibliotecii `IPC` de pe sistemul `UNIX` pe care lucraţi.

5.5 Şabloane de comunicaţie între procese

În continuare vom trece în revistă cîteva situaţii ce pot apare la comunicatia prin canale interne şi externe, şi care pot ridica anumite probleme la implementarea lor.

După cum am văzut, un canal (intern sau extern) este o "conductă" unidirecţională prin care "curge" informaţia de la un anumit capăt (*i.e.*, cel de scriere) către celălalt capăt (*i.e.*, cel de citire). Însă, la un moment dat, pot exista mai multe procese cu rol de

"scriitori" (*i.e.*, care scriu date în acel canal, pe la capătul de scriere), şi pot exista mai multe procese cu rol de "cititori" (*i.e.*, care citesc date din acel canal, pe la capătul de citire); bineînţeles, o parte dintre procese pot fi atît "scriitori", cît şi "cititori".

După numărul de procese "scriitori" şi "cititori" ce utilizează un canal (intern sau extern) pentru a comunica între ele, putem diferenţia următoarele *pattern*-uri (*i.e.*, şabloane) de comunicaţie inter-procese:

- comunicaţie *unul la unul*:

  atunci cînd canalul este folosit de un singur proces "scriitor" pentru a transmite date unui singur proces "cititor";

- comunicaţie *unul la mulţi*:

  atunci cînd canalul este folosit de un singur proces "scriitor" pentru a transmite date mai multor procese "cititori";

- comunicaţie *mulţi la unul*:

  atunci cînd canalul este folosit de mai multe procese "scriitori" pentru a transmite date unui singur proces "cititor";

- comunicaţie *mulţi la mulţi*:

  atunci cînd canalul este folosit de mai multe procese "scriitori" pentru a transmite date mai multor procese "cititori".

Primul caz, cel al comunicaţiei *unul la unul*, este cel mai simplu, neridicînd probleme deosebite. Exemplele de programe date anterior în secţiunea 5.2 despre canale interne, se încadrează în acest şablon de comunicaţie.

Celelalte cazuri ridică anumite probleme, de care trebuie să se ţină cont la implementarea lor. Să le trecem pe rînd în revistă.

În cazul comunicaţiei *unul la mulţi* pot interveni două categorii de factori ce pot genera anumite probleme, cum ar fi aceea de *corupere* a mesajelor:

1. O primă categorie se referă la *lungimea* mesajelor:

   - mesaje de lungime *constantă*

     Dacă toate mesajele transmise au o lungime constantă (cunoscută de toţi "cititorii"), atunci nu sunt probleme deosebite – fiecare mesaj poate fi citit *atomic* (*i.e.*, dintr-o dată, printr-un singur apel `read`); pentru aceasta, la fiecare citire din canal (*i.e.*, fiecare apel `read`) se vor citi exact atîţia octeţi cît este constanta ce indică lungimea mesajelor.

- mesaje de lungime *variabilă*

  Dacă însă mesajele transmise au lungimi variabile (necunoscute deci de "cititorii"), atunci pot apare probleme deoarece mesajele nu mai pot fi citite *atomic* (*i.e.*, dintr-o dată, printr-un singur apel `read`); soluţia în acest caz este să se folosească mesaje formatate astfel:

  MESAJ = HEADER + MESAJUL PROPRIU-ZIS ,

  partea de *header* fiind un mesaj de lungime fixă ce contine lungimea părţii de mesaj propriu-zis.

  Pentru ca citirea mesajelor din canal să se poată face *atomic*, fiecare proces "cititor" va trebui să respecte următorul protocol: întîi se citeşte un *header* (deci un mesaj de lungime fixă), şi apoi se citeşte mesajul propriu-zis, cu lungimea determinată din *header*-ul abia citit. În plus, deoarece este nevoie de două apeluri *atomice* `read` prin acest protocol pentru a citi un mesaj în întregime, trebuie asigurat faptul că nici un alt proces nu va face citiri din canal între momentele corespunzătoare celor două apeluri `read`. Cu alte cuvinte, trebuie garantat accesul exclusiv la canalul intern (sau extern). Aceasta se poate rezolva folosind un semafor (implementat, de exemplu, folosind blocaje pe fişiere).

2. O a doua categorie se referă la *destinatarul* mesajelor:

   - mesaje cu destinatar *oarecare*

     Dacă mesajele transmise de "scriitor" nu sunt pentru un anumit destinatar specific, atunci nu sunt probleme deosebite din acest punct de vedere – fiecare mesaj poate fi citit şi prelucrat de oricare dintre procesele "cititori".

   - mesaje cu destinatar *specificat*

     Dacă însă mesajele transmise de "scriitor" sunt pentru un anumit destinatar specific, atunci trebuie asigurat faptul că mesajul este citit exact de către "cititorul" căruia îi era destinat. Soluţia în acest caz este să se folosească mesaje formatate astfel:

     MESAJ = HEADER + MESAJUL PROPRIU-ZIS ,

     partea de *header* conţinînd un identificator al destinatarului (apoi, la mesajul formatat astfel, se aplică tehnicile discutate mai sus referitoare la lungimea mesajelor).

     Pentru ca citirea mesajelor din canal să se poată face corect, fiecare proces "cititor" va trebui să respecte următorul protocol: dacă a citit un mesaj care nu-i era destinat lui, îl va scrie înapoi în canal, şi apoi va face o pauză aleatoare (*i.e.*, îsi va suspenda execuţia pentru un timp aleator) înainte de a încerca să citească din nou din canal.

Evident, ambele categorii de factori, atît lungimea mesajelor, cît şi destinatarul lor, depind de *logica aplicaţiei* (*i.e.* de funcţionalităţile implementate în programele care folosesc acel canal pentru a comunica între ele).

Observaţie: uneori, din motive de uşurinţă de scriere a codului şi de eficienţă a execuţiei codului, se poate prefera înlocuirea unui singur canal folosit pentru comunicaţie *unul la*

mulţi, cu mai multe canale folosite pentru comunicaţie *unul la unul*, cîte un canal pentru fiecare proces "cititor" existent.

În cazul comunicaţiei *mulţi la unul* pot interveni următoarele două categorii de factori ce pot genera anumite probleme:

1. O primă categorie se referă la *lungimea* mesajelor:

   - mesaje de lungime *constantă*
   - mesaje de lungime *variabilă*

   Această situaţie se tratează similar ca la comunicaţia *unul la mulţi*, cu singura observaţie că nu mai este necesară folosirea unui semafor pentru accesul exclusiv la canal, deoarece avem un singur "cititor".
   (*Observaţie*: spre deosebire de citirea din canal, scrierea în canal nu ridică probleme d.p.d.v. al lungimii mesajelor, indiferent dacă avem mai mulţi "scriitori", ca în acest caz, sau doar unul singur, ca în cazul precedent, datorită faptului că se poate realiza în mod *atomic*, printr-un singur apel `write`, deoarece "scriitorul" cunoaşte dinainte lungimea mesajului pe care urmează să-l scrie în canal.)

2. O a doua categorie se referă la *expeditorul* mesajelor:

   - mesaje cu expeditor *oarecare*
     Dacă mesajele recepţionate de "cititor" nu trebuie tratate ca venind de la un anumit expeditor specific, atunci nu sunt probleme deosebite din acest punct de vedere – fiecare mesaj poate fi citit şi prelucrat în acelaşi fel indiferent de la care proces "scriitor" provine.
   - mesaje cu expeditor *specificat*
     Dacă însă mesajele recepţionate de "cititor" trebuie tratate ca venind de la un anumit expeditor specific, atunci trebuie asigurat faptul că fiecare mesaj îi indică "cititorului" care este exact "scriitorul" care i l-a trimis. Soluţia în acest caz este să se folosească mesaje formatate astfel:

     MESAJ = HEADER + MESAJUL PROPRIU-ZIS ,

     partea de *header* conţinînd un identificator al expeditorului (apoi, la mesajul formatat astfel, se aplică tehnicile discutate mai sus referitoare la lungimea mesajelor).

Evident, şi în acest caz ambele categorii de factori, atît lungimea mesajelor, cît şi expeditorul lor, depind de *logica aplicaţiei* (*i.e.* de funcţionalităţile implementate în programele care folosesc acel canal pentru a comunica între ele).

Observaţie: şi în acest caz se poate prefera uneori înlocuirea unui singur canal folosit pentru comunicaţie *mulţi la unul*, cu mai multe canale folosite pentru comunicaţie *unul la unul*, cîte un canal pentru fiecare proces "scriitor" existent.

În sfîrşit, în cazul comunicaţiei *mulţi la mulţi* pot interveni toate categoriile de factori pe care le-am văzut la comunicaţiile *unul la mulţi* şi *mulţi la unul*:

1. *lungimea* mesajelor
2. *expeditorul* mesajelor
3. *destinatarul* mesajelor

Tratarea acestora se face prin combinarea soluţiilor prezentate mai sus la comunicaţiile *unul la mulţi* şi *mulţi la unul*.

Observaţie: aceste şabloane de comunicaţie prezentate mai sus pot apare nu numai la comunicaţia prin canale interne şi externe, ci la orice fel de comunicaţie prin schimb de mesaje (prin reţea – folosind *socket*-uri, prin fişiere obişnuite, etc.).

În final, un ultim aspect care trebuie menţionat este următorul: în discuţia de pînă acum am presupus că mediul de comunicaţie nu impune vreo limită asupra lungimii maxime pe care o pot avea mesajele transmise prin el. Ce facem însă în situaţia cînd vrem să transmitem mesaje nelimitate printr-un canal de comunicaţie care permite doar mesaje de lungime limitată? Acest aspect este foarte important, de exemplu, în cazul comunicaţiei prin reţea – folosind *socket*-uri –, unde există o limită de cca. 1500 octeţi. După cum aţi văzut, şi *pipe*-urile, interne şi externe, sunt limitate, dar în cazul lor limita nu pune prea mari probleme, dacă se folosesc operaţiile de citire şi scriere în modul implicit, *blocant* (deoarece, atunci cînd se umple canalul, apelul `write` va fi întrerupt pînă cînd se va face loc în canal printr-o citire).

Aşadar, în situaţia în care mediul de comunicaţie impune o anumită limitare a lungimii maxime pe care o pot avea mesajele transmise prin acel mediu, şi se doreşte totuşi transmiterea unor mesaje de lungime mai mare decît limita admisibilă, soluţia este de a "împărţi" mesajele în *pachete* (= mesaje de lungime fixă, cel mult egală cu limita admisibilă), şi atunci:

MESAJ = SECVENŢĂ ORDONATĂ DE PACHETE ,

PACHET = HEADER + CONŢINUTUL PROPRIU-ZIS AL PACHETULUI ,

partea de *header* conţinînd un identificator al mesajului şi un identificator al pachetului în cadrul mesajului din care face parte. Identificatorul mesajului este necesar pentru "separaţia mesajelor" (*i.e.*, pentru a putea deosebi pachetele aparţinînd unui mesaj de pachetele aparţinînd altui mesaj). Identificatorul pachetului este necesar pentru refacerea mesajului din pachete la destinaţie, deoarece ordinea de emisie a pachetelor nu este obligatoriu să coincidă cu ordinea în care sunt recepţionate pachetele la destinaţie (în situaţia cînd mediul de comunicaţie poate cauza inversarea ordinii pachetelor în timpul tranzitării lor prin el).

5.6 Exerciţii

Exerciţiul 1. Cîte categorii de comunicaţie între procese există?

Exerciţiul 2. Ce înseamnă noţiunea de canal de comunicaţie?

Exerciţiul 3. De cîte tipuri sunt canalele de comunicaţie în `UNIX`?

Exerciţiul 4. Ce efect are apelul `pipe`? Ce valoare returnează el?

Exerciţiul 5. În ce condiţii pot comunica între ele două procese folosind un canal intern?

Exerciţiul 6. Cum ne putem referi la capetele canalului creat de apelul `pipe`, şi în ce scop le folosim?

Exerciţiul 7. Cum funcţionează citirea dintr-un canal intern, şi care sunt situaţiile de excepţie?

Exerciţiul 8. Cum funcţionează scrierea într-un canal intern, şi care sunt situaţiile de excepţie?

Exerciţiul 9. Care este diferenţa dintre comportamentul blocant şi cel neblocant al canalelor interne? Cum se poate schimba comportamentul implicit?

Exerciţiul 10.* Ce efect va avea programul următor (ce se va afişa pe ecran la execuţie) ?

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

int main(){ int p[2]; char c=0;
  pipe(p);
  switch(fork()) {
    case -1: fprintf(stderr,"eroare la fork"); exit(1);
    case  0: close(1); dup(p[1]);
             system("echo 'test message' root");
             break;
    default: close(p[1]);
             while(0!=read(p[0],&c,1)) printf("%c",c);
             wait(NULL);
  }
  return 0;
}
```

Exerciţiul 11.* Scrieti un program C care sa determine capacitatea canalelor interne pentru sistemul `UNIX` pe care il utilizati.

Indicaţie de rezolvare: programul va crea un canal intern si apoi va scrie in mod repetat in el, fara a citi nimic, si contorizind numarul de octeti scrisi efectiv in canal, pina cind apelul de scriere va esua datorita umplerii canalului (*atenţie*: trebuie sa setati atributul neblocant pentru capatul de scriere al canalului, pentru ca apelul de scriere sa nu ramina blocat in momentul umplerii canalului).

*Exerciţiul 12**. Rescrieţi programul din exerciţiul 12* din capitolul 3 care calcula o sumă distribuită folosind, pentru comunicatia intre procese, canale interne in loc de fisiere obisnuite.

Exerciţiul 13. Ce sunt fişierele *fifo*?

Exerciţiul 14. În ce condiţii pot comunica între ele două procese folosind un canal extern? Care ar fi deosebirea faţă de canalele interne?

Exerciţiul 15. Ce efect are apelul `mkfifo`? Ce parametri trebuie transmişi acestuia?

Exerciţiul 16. Cum se realizează deschiderea capetelor unui canal extern, şi ce probleme pot apare?

Exerciţiul 17. Cum funcţionează citirea dintr-un canal extern, şi care sunt situaţiile de excepţie?

Exerciţiul 18. Cum funcţionează scrierea într-un canal extern, şi care sunt situaţiile de excepţie?

Exerciţiul 19. Care este diferenţa dintre comportamentul blocant şi cel neblocant al canalelor externe? Cum se poate schimba comportamentul implicit?

Exerciţiul 20. Care sunt principalele diferenţe dintre canalele interne şi cele externe?

*Exerciţiul 21**. Ce efect va avea programul următor?

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

int main()
{
  int fd; char sir[300];
  mknod("a.txt", S_IFIFO|0666, 0);
  fd=open("a.txt", O_WRONLY);
  while(gets(sir), !feof(stdin))
    write(fd, sir, strlen(sir));
  return 0;
}
```

*Exerciţiul 22**. Scrieti un program C care sa determine capacitatea canalelor externe pentru sistemul `UNIX` pe care il utilizati.

Indicaţie de rezolvare: la fel ca la canale interne – programul va crea un canal intern si apoi va scrie in mod repetat in el, fara a citi nimic, si contorizind numarul de octeti scrisi efectiv in canal, pina cind apelul de scriere va esua datorita umplerii canalului (*atenţie*: trebuie sa setati atributul neblocant pentru capatul de scriere al canalului, pentru ca apelul de scriere sa nu ramina blocat in momentul umplerii canalului).

Exerciţiul 23. Rescrieţi programele `pipe-ex1.c` si `pipe-ex2.c` date ca exemplu în secţiunea 5.2 a acestui manual folosind, pentru comunicatia intre procese, canale externe în locul celor interne.

*Exerciţiul 24**. Rescrieţi programul din exerciţiul 12* din capitolul 3 care calcula o sumă distribuită folosind, pentru comunicatia intre procese, canale externe.

Exerciţiul 25. **Hi-ho**: scrieţi două programe, primul sa scrie pe ecran "HI-" in mod repetat, iar al doilea sa scrie "HO, " in mod repetat, si sa se foloseasca canale *fifo* pentru sincronizarea proceselor, astfel ca, atunci cînd sunt lansate în paralel cele două programe, pe ecran să fie afişată exact succesiunea:

HI-HO, HI-HO, HI-HO, ...

si nu alte combinatii posibile de interclasare a mesajelor afişate de cele două procese.

Exerciţiul 26. La ce sunt utile semafoarele?

Exerciţiul 27. Scrieţi un program în care să implementaţi un semafor binar folosind canale *fifo*.

Exerciţiul 28. Ce înseamnă aplicaţii de tip client-server?

Exerciţiul 29. De cîte tipuri sunt serverele?

Exerciţiul 30. **Mini-FTP**: Scrieţi o aplicaţie de tip client-server pentru transferul de fişiere. Clientul va oferi utilizatorului o interfaţă text (*i.e.*, un prompter de genul `FTP>`) la care va putea tasta următoarele 6 comenzi:

1. `FTP> ls` *director*

   va afişa conţinutul directorului specificat de pe "calculatorul" server;

2. `FTP> lls` *director*

   va afişa conţinutul directorului specificat de pe "calculatorul" client;

3. `FTP> cd` *director*

   va schimba directorul curent în cel specificat pe "calculatorul" server;

4. `FTP> lcd` *director*

   va schimba directorul curent în cel specificat pe "calculatorul" client;

5. `FTP> get` *fisier*

   va transfera fişierul specificat de pe "calculatorul" server pe "calculatorul" client;

6. `FTP> put` *fisier*

   va transfera fişierul specificat de pe "calculatorul" client pe "calculatorul" server.

Evident, atît programul client, cît şi programul server îşi vor păstra cîte un director curent de lucru propriu, în raport cu care se vor considera numele de fişiere sau directoare specificate prin cale relativă în comenzile de mai sus. Operaţiile *locale* `lls` şi `lcd` se vor executa direct de către client, fără ajutorul serverului, în schimb pentru toate celelalte patru operaţii clientul va contacta serverul pentru a realiza operaţia respectivă cu ajutorul acestuia din urmă.

Cerinţă: pentru programul server, încercaţi întîi scrierea unui server iterativ, şi apoi a unuia concurent.

Exerciţiul 31. Ce şabloane de comunicaţie pot apare la comunicaţia inter-procese prin intermediul canalelor de comunicaţie?

Exerciţiul 32. Care sunt factorii de care trebuie să se ţină cont la implementare, pentru a nu apare diverse probleme (gen "coruperea" mesajelor) la folosirea acestor şabloane de comunicaţie?

Exerciţiul 33. Ce alte mecanisme de comunicaţie inter-procese mai sunt disponibile în `UNIX`, pe lîngă canalele interne şi cele externe?

Bibliografie

[1] D. Acostăchioaie: *Programare C şi C++ pentru Linux*, editura Polirom, Iaşi, 2002.

[2] D. Acostăchioaie: *Administrarea şi configurarea sistemelor Linux*, ediţia a doua, editura Polirom, Iaşi, 2003.

[3] D. Acostăchioaie, S. Buraga: *Utilizare Linux. Noţiuni de bază şi practică*, editura Polirom, Iaşi, 2004.

[4] B. Ball, S. Smoogen: *Teach yourself Linux in 24 hours*, editura SAMS Publishing, Indianapolis, 1998.

[5] I. Ignat, E. Muntean, K. Pusztai: *UNIX: Gestionarea fişierelor*, editura MicroInformatica, Cluj-Napoca, 1992.

[6] I. Ignat, A. Kacso: *UNIX: Gestionarea proceselor*, editura MicroInformatica, Cluj-Napoca, 1995.

[7] M. Mitchell, J. Oldham, A. Samuel: *Advanced Linux Programming*, editura New Riders Publishing, Indianapolis, 2001.

[8] R.G. Sage: *UNIX pentru profesionişti*, Editura de Vest, Timişoara, 1993.

[9] A. Silberschatz, J. Peterson, P. Galvin: *Operating Systems Concepts*, editura Addison-Wesley, Reading MA, 2001.

[10] R. Stevens: *Advanced UNIX Programming in the UNIX Environments*, editura Addison-Wesley, Reading MA, 1992.

[11] A. Tanenbaum: *Modern Operating Systems*, editura Prentice Hall International, 2001.

[12] ***, *The Linux Documentation Project*: http://metalab.unc.edu/LDP/

Anexa A

Rezolvare exerciţii din partea I

În această anexă vom prezenta rezolvarea celor mai dificile exerciţii (indicate prin semnul * adăugat numărului exerciţiului în textul manualului) din partea I.
Observaţie: din lipsă de spaţiu, şi pentru a nu răpi cititorului plăcerea de a încerca singur să le rezolve, am omis rezolvarea exerciţiilor considerate a fi mai simple, precum şi a tuturor exerciţiilor de tip întrebare, al căror răspuns se poate afla direct din textul manualului de faţă, sau din paginile de manual `UNIX`, disponibile prin comanda `man` pe sistemul `UNIX` pe care lucraţi.

Exerciţiul 47* **din capitolul 2**.

Indicaţii de rezolvare. Pentru a testa modul de execuţie a fişierelor de iniţializare, folosiţi comenzi de afişare pe ecran a unor mesaje. De exemplu, adăugaţi în fiecare fişier la sfîrşit cîte o linie de forma:
`echo "Executing file .profile"`
şi, respectiv,
`echo "Executing file .bash_profile"`
(dacă vreunul dintre fişiere nu există, atunci creaţi-l). Apoi deschideţi o nouă sesiune de lucru şi urmăriţi mesajele afişate pe ecran.
Pentru a doua parte a exerciţiului, ştergeţi cîte unul din cele două fişiere (sau doar redenumiţi-l altfel, ca să nu-i pierdeţi conţinutul) şi repetaţi experimentul.

Exerciţiul 48* **din capitolul 2**.

Indicaţii de rezolvare. Pentru a testa o comandă care să scrie mesaje pe ambele ieşiri standard (şi pe `stdout`, şi pe `stderr`), puteţi folosi următorul program C (după ce-l compilaţi):

```
/*
  File: prg.c
  (un mic program care scrie mesaje si pe stdout, si pe stderr)
  Compilare: gcc prg.c -ocomanda
*/
```

```
#include <stdio.h>
void main()
{
  fprintf(stdout,"prg.c -> message on stdout\n");
  fprintf(stderr,"prg.c -> message on stderr\n");
  fprintf(stdout,"prg.c -> 2nd message on stdout\n");
}
```

Modul de testare:
`UNIX>` `comanda` *redirectari* `; cat fisier`
pentru fiecare din cele 8 linii de comandă specificate în enunţul exerciţiului.

Exerciţiul 73* **din capitolul 2**.

Rezolvare. Iată *script*-ul pentru automatizarea procesului de dezvoltare de programe C:

```
#!/bin/bash
#
# Efect: editeaza un fisier sursa C , apoi il compileaza si
#        afiseaza erorile de compilare , iar apoi il executa.

clear

echo "*************** Begin of script for Edit-Compile-Execute ***********"

if test $# -gt 1
then
  echo " Usage:  $0  [  namefile[.c]  ] "
  sleep 1
  echo "*************** End of script for Edit-Compile-Execute *************"
  exit 0
fi

if test $# -eq 1
then
  file=$1
else
  echo "************************** Source filename ************************"
  echo -n " Input the file's name for editing/compiling/testing : "
  read file
fi
fileexit=`basename $file .c`
file=${fileexit}.c
if test -f fileexit
then
  mv $fileexit $file
  echo " The source file $fileexit was renamed in $file !"
  sleep 3
fi
sleep 1
```

```
while true
do

  echo "************************** Editing **********************************"
  while true
  do
    echo -n " Do you want to edit the file $file ? Y/N  "
    read rasp
    case $rasp in
      n | N ) break 2 ;;
      y | Y ) # pico $file
              joe $file
              break ;;
    esac
  done


  echo "************************** Compiling ********************************"
  while true
  do
    echo -n " Do you want to compile the file $file ? Y/N  "
    read rasp
    case $rasp in
      n | N ) break 2 ;;
      y | Y ) gcc -o ${fileexit}.exe  $file 2> ${fileexit}.err
              break ;;
    esac
  done

  if test -s ${fileexit}.err
  then
    #joe ${fileexit}.err
    echo "************************** Print compiling errors *****************"
    head ${fileexit}.err
    continue # reluarea buclei de la editare
  else
    rm ${fileexit}.err
  fi


  echo "************************** Testing **********************************"
  while true
  do
    echo -n " Do you want to execute the file ${fileexit}.exe ? Y/N  "
    read rasp
    case $rasp in
      n | N ) break 2 ;;
      y | Y ) ${fileexit}.exe
              break ;;
    esac
```

```
  done

  sleep 2
  echo "************************* Finish ***********************************"
  echo "The execution of the file ${fileexit}.exe was finished!"
  sleep 1
  echo "Now, you may want to edit again the source for more modifications!"
  sleep 2
  continue # reluarea buclei de la editare

done

echo "*************** End of script for Edit-Compile-Execute *************"
sleep 1
```

Anexa B

Rezolvare exerciţii din partea II

În această anexă vom prezenta rezolvarea celor mai dificile exerciţii (indicate prin semnul * adăugat numărului exerciţiului în textul manualului) din partea II.
Observaţie: din lipsă de spaţiu, şi pentru a nu răpi cititorului plăcerea de a încerca singur să le rezolve, am omis rezolvarea exerciţiilor considerate a fi mai simple, precum şi a tuturor exerciţiilor de tip întrebare, al căror răspuns se poate afla direct din textul manualului de faţă, sau din paginile de manual `UNIX`, disponibile prin comanda `man` pe sistemul `UNIX` pe care lucraţi.

Exerciţiul 3^* **din capitolul 3**.

Rezolvare. Răspunsul corect: se testează dacă există fişierul "a.txt" în directorul curent; mesajul afişat este tocmai pe dos. (*Justificare*: datorită faptului că funcţia `access` returnează valoarea 0 pentru test îndeplinit.)

Exerciţiul 4^* **din capitolul 3**.

Rezolvare. Răspunsul corect: va da eroare la execuţie, afişînd mesajul "Segmentation fault". (*Justificare*: programul "crapă" la apelul `stat` datorită faptului că pointerul `info` nu este alocat.)

Exerciţiul 5^* **din capitolul 3**.

Rezolvare. Răspunsul corect: se va afişa `descriptor=-1`. (*Justificare*: deoarece va fi eroare la *open*, un utilizator obişnuit neavînd drept de scriere asupra fişierului `/etc/passwd`.)

Exerciţiul 6^* **din capitolul 3**.

Rezolvare. Răspunsul corect: se va afişa valoarea 0, indiferent dacă programul este executat de un utilizator obişnuit sau de utilizatorul *root*. (*Justificare*: deoarece apelul `stat` returnează 0 pentru succes şi -1 pentru eroare.)

Exerciţiul 7* din capitolul 3.

Rezolvare. Răspunsul corect: se va afişa şirul "LinuxLinux". (*Justificare*: deoarece programul creează pe "fis2" drept *hard-link* către fişierul ordinar "fis1".)

Exerciţiul 13* din capitolul 3.

Indicaţii de rezolvare. Iata citeva sfaturi pentru implementarea operatiilor din meniu:
– pentru creare director: utilizati functia `mkdir` ;
– pentru stergere director: utilizati functia `rmdir` ;
– pentru schimbare director: utilizati functia `chdir` ;
– pentru listare director: utilizati sablonul descris mai sus pentru parcurgerea secventiala a intrarilor dintr-un director ;
– pentru operatia de copiere (`copy fis_src fis_dest`): scrieti o functie care sa execute urmatoarele operatii:

1. deschide fisierele `fis_src` si `fis_dest`
2. repeat
3. citeste din `fis_src` si scrie ce a citit in `fis_dest`
4. until `EOF` on `fis_src`
5. inchide cele doua fisiere

– pentru stergere fisier: utilizati functia `unlink` ;
– pentru redenumire fisier: utilizati functia `rename` ;
– pentru vizualizare fisier: in mod asemanator ca la copiere, scrieti o functie care sa execute urmatoarele operatii:

1. deschide fisierul
2. repeat
3. citeste din fisier si scrie ce a citit `stdout`
4. until `EOF` on fisier
5. inchide fisierul

Exerciţiul 17* din capitolul 3.

Rezolvare. Iată sursa programului `access4w.c` (programul acces versiunea 4.0, varianta cu apel blocant):

```
/*
  File: access4w.c (versiunea 4.0, cu lacat pus in mod blocant)
*/
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

extern int errno;

int main(int argc, char* argv[])
{
  int fd;
  char ch;
  struct flock lacat;

  if(argv[1] == NULL)
  {
    fprintf(stderr,"Trebuie apelat cu cel putin un parametru.\n");
    exit(1);
  }

  if( (fd=open("fis.dat", O_RDWR)) == -1)
  {  /* trateaza cazul de eroare ... */
    perror("Nu pot deschide fisierul fis.dat deoarece ");
    exit(2);
  }

  /* pregateste lacat pe caracterul dinaintea pozitiei curente */
  lacat.l_type   = F_WRLCK;
  lacat.l_whence = SEEK_CUR;
  lacat.l_start  = -1;
  lacat.l_len    = 1;


  /* parcurgerea fisierului caracter cu caracter pina la EOF */
  while( read(fd,&ch,1) != 0)
  {
    if(ch == '#')
    {
      /* O singura incercare de punere a lacatului in mod blocant */
      printf("Pun blocant lacatul pe #-ul gasit deja [Proces:%d].\n",getpid());
      if( fcntl(fd,F_SETLKW,&lacat) == -1)
      {
        fprintf(stderr,"Eroare la blocaj [ProcesID:%d].\n", getpid());
        perror("\tMotivul");
        exit(3);
      }
      else
        printf("Blocaj reusit [ProcesID:%d].\n", getpid());

      sleep(5);
```

```
      /* Inainte de a face suprascrierea, dar dupa blocare reusita,
         mai trebuie verificat inca o data daca caracterul curent
         este '#', deoarece intre momentul primei depistari a '#'-ului
         si momentul reusitei blocajului exista posibilitatea ca
         acel '#' sa fie suprascris de celalalt proces !  */
      lseek(fd,-1L,1);
      read(fd,&ch,1);
      if(ch == '#')
      {
        lseek(fd,-1L,1);
        write(fd,argv[1],1);
        printf("Terminat. S-a inlocuit primul # gasit [ProcesID:%d].\n",getpid());
        return 0;
      }
      else
      {
        printf("Caracterul # pe care-l gasisem a fost deja suprascris. "
               "Voi cauta altul [ProcesID:%d].\n",getpid());

        /* Deblocarea acelui caracter (care intre timp fusese schimbat
               din '#' in altceva de catre celalalt proces). */
        lacat.l_type=F_UNLCK;
        fcntl(fd,F_SETLK,&lacat);
        lacat.l_type=F_WRLCK;

        /* Urmeaza reluarea buclei de cautare a '#'-ului ... */
      }
    }/*endif*/
  }/*endwhile*/

  printf("Terminat. Nu exista # in fisierul dat [ProcesID:%d].\n",getpid());
  return 0;
}
```

Exerciţiul 6* din capitolul 4.

Rezolvare. Răspunsul corect: se va afişa şirul "ABAB" dacă apelul `fork` reuşeşte, respectiv "AerrB" în caz contrar (dacă e eroare la `fork`). (*Justificare*: datorită faptului că apelul `printf` lucrează *buffer*-izat, iar în momentul creării procesului fiu, acesta va "moşteni" conţinutul *buffer*-ului.)

Exerciţiul 7* din capitolul 4.

Rezolvare. Răspunsul corect: se vor crea în total 15 procese fii (*i.e.*, fara procesul initial). *Justificare*: doar pentru valorile impare ale contorului se apeleaza `fork`, iar in fiecare fiu creat se continua bucla `for` cu valoarea curenta a contorului. Mai precis:
– procesul initial P_0 creeaza 4 procese fii: P_1 (pentru `contor`=1), P_2 (pentru `contor`=3), P_3 (pentru `contor`=5), si P_4 (pentru `contor`=7);
– fiul P_1 creeaza 3 fii: P_5 (pentru `contor`=3), P_6 (pentru `contor`=5), si P_7 (pentru

contor=7) ;
– fiul P_2 creeaza 2 fii: P_8 (pentru contor=5), si P_9 (pentru contor=7) ;
– fiul P_3 creeaza 1 fiu: P_{10} (pentru contor=7) ;
– fiul P_4 nu creeaza nici un fiu ;
– fiul P_5 creeaza 2 fii: P_{11} (pentru contor=5), si P_{12} (pentru contor=7) ;
– fiul P_6 creeaza 1 fiu: P_{13} (pentru contor=7) ;
– fiul P_7 nu creeaza nici un fiu ;
– fiul P_8 creeaza 1 fiu: P_{14} (pentru contor=7) ;
– fiii P_9, P_{10} nu creeaza nici un fiu ;
– fiul P_{11} creeaza 1 fiu: P_{15} (pentru contor=7) ;
– fiii P_{12}, P_{13}, P_{14}, P_{15} nu creeaza nici un fiu .

Exerciţiul 13* din capitolul 4.

Rezolvare. Iată sursa programului `suma.c` care rezolva problema sumei distribuite:

```
/*
  File: suma.c  (suma distribuita cu 3 procese)
*/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>

extern int errno;

void master_init();
void master_finish();
void slave_work(char* fi, char* fo);

int main()
{
  int pid1,pid2;

  printf("\n\n\n\n\n");

  /* Curata rezultatele executiilor anterioare -> necesar !!! */
  remove("f1o"); remove("f2o");

  /* Citire numere de la tastatura si scriere in fisierele f1i si f2i */
  master_init();

  /* Am citit numerele inainte de crearea fiilor deoarece este nevoie de
   sincronizare (fiul trebuie sa astepte tatal sa scrie in fisier pentru a
   avea ce citi), iar citirea de la tastatura poate dura oricit de mult */

  /* Creare primul proces slave */
  if( (pid1=fork()) == -1)
  {
    fprintf(stderr,"Error fork la fiul 1 !\n");
```

```
    exit(2);
  }

  if(pid1 == 0)
  { /* sunt in procesul fiu 1 */

    slave_work("f1i","f1o");
    return 0;
    /* sfirsit executie fiu 1 */
  }
  /* else sunt in procesul master, executat in paralel cu if-ul de mai sus */

  /* Creare al doilea proces slave */
  if( (pid2=fork()) == -1)
  {
    fprintf(stderr,"Error fork la fiul 2 !\n");
    exit(2);
  }

  if(pid2 == 0)
  { /* sunt in procesul fiu 2 */

    slave_work("f2i","f2o");
    return 0;
    /* sfirsit executie fiu 2 */
  }
  /* else sunt in procesul master, executat in paralel cu cei doi fii */

  /* Citeste cele 2 sume partiale si afiseaza suma lor. */
  master_finish();
  return 0;
}

/*********************************************************************/

void master_init()
{
  FILE *f1,*f2;
  int nr, flag;

  if( (f1=fopen("f1i","wt")) == NULL)
  {
    fprintf(stderr,"Error opening file f1i, err=%d\n",errno); exit(3);
  }
  if( (f2=fopen("f2i","wt")) == NULL)
  {
    fprintf(stderr,"Error opening file f2i, err=%d\n",errno); exit(3);
  }

  printf("Introduceti numerele (0 pentru terminare):\n");
  flag=1;
  do
```

```
  {
    scanf("%d", &nr);
    fprintf( (flag==1?f1:f2), "%d ", nr);
        /* Atentie: spatiul din format este necesar! */
    flag=3-flag;

  }while(nr!=0);

  fclose(f1); fclose(f2);
}

/*********************************************************************/

void master_finish()
{
  /* Aici mai apare o sincronizare: master-ul trebuie sa citeasca sumele
  partiale abia dupa ce acestea au fost calculate si scrise in fisierele
  f1o si f2o de catre procesele slave.
  Rezolvare: incercare repetata de citire cu pauza intre incercari.
  (sau: se poate astepta terminarea celor 2 fii folosind primitiva wait)*/

  FILE *f1,*f2;
  int sp1,sp2, cod;

  /* Citeste prima suma partiala */
  cod = 0;
  do
  {
    if( (f1=fopen("f1o","rt")) != NULL)
      cod = (fscanf(f1,"%d",&sp1)==1);
    if(!cod)
      sleep(3);

  }while(!cod);
  fclose(f1);

  /* Citeste a doua suma partiala */
  cod = 0;
  do
  {
    if( (f2=fopen("f2o","rt")) != NULL)
      cod = (fscanf(f2,"%d",&sp2)==1);
    if(!cod)
      sleep(3);

  }while(!cod);
  fclose(f2);

  /* Afiseaza suma */
  printf("Master=%d -> suma nr. introduse este: %d\n", getpid(), sp1+sp2);
}
```

```
/*************************************************************************/

void slave_work(char* fi, char* fo)
{
  FILE *f1,*f2;
  int nr,cod, suma_partiala;

  if( (f1=fopen(fi,"rt")) == NULL)
  {
    fprintf(stderr,"Error opening file %s, err=%d\n",fi,errno); exit(3);
  }

  suma_partiala=0;
  do
  {
    cod=fscanf(f1,"%d", &nr);
    if(cod == 1)
      suma_partiala += nr;

  }while(cod != EOF);

  fclose(f1);

  if( (f2=fopen(fo,"wt")) == NULL)
  {
    fprintf(stderr,"Error opening file %s, err=%d\n",fo,errno); exit(3);
  }
  fprintf(f2,"%d",suma_partiala);
  fclose(f2);
  printf("Slave=%d -> suma partiala:%d\n", getpid(), suma_partiala);
}

/*************************************************************************/
```

Exerciţiul 20^* **din capitolul 4**.

Rezolvare. Efectul acestui program constă în următoarele: se redirectează ieşirea de eroare standard către fişierul "a", apoi apelul `fprintf(stderr,"Hello!")` va scrie de fapt în fişierul "a", iar apoi urmează `exec`-ul, care, dacă reuşeşte, afişează pe ecran fişierul "a", iar dacă eşuează, atunci urmează `fork`-ul. Ca urmare, răspunsurile corecte sunt: şirul "Hello!" este afişat pe ecran o dată (dacă reuşeşte `exec`-ul), de două ori (dacă nu reuşeşte `exec`-ul, dar reuşeşte `fork`-ul), şi respectiv niciodată (dacă eşuează şi `exec`-ul, şi `fork`-ul).

Exerciţiul 35^* **din capitolul 4**.

Rezolvare. Răspunsul corect: se va afişa de 4 ori. *Justificare*: din procesul curent se creează, în total, 7 noi procese, din care 3 sunt fii direcţi ai procesului iniţial. Părintele din vîrful ierarhiei (*i.e.*, procesul iniţial) nu va executa niciodată funcţia `my_handler`, deoarece el este singurul care nu execută primitiva de corupere `signal`. Prin urmare, doar

cele 4 procese care nu sunt fii direcţi ai procesului iniţial vor cauza, la terminarea lor, afişarea şirului "death of child!", de către părinţii acestora (în corpul *handler*-ului).

Exerciţiul 37* din capitolul 4.

Rezolvare. Iată sursa programului `hi-ho.c` care rezolvă problema Hi-ho utilizînd semnalul `SIGUSR1` pentru sincronizarea celor două procese (sunt folosite două procese, unul creat de celălalt; tatăl este responsabil cu afişarea "hi"-urilor, iar fiul este responsabil cu afişarea "ho"-urilor).

```
/*
  File: hi-ho.c  (versiunea cu semnale pentru sincronizare)
*/
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

#define NR_AFISARI 100

void my_handler(int semnal)
{
}

int main()
{
  int pid,i,ppid;

  signal(SIGUSR1,my_handler);

  if(-1 == (pid=fork()) )
  {
     perror("eroare la fork");
     exit(1);
  }

  if(0 == pid)
  { /* fiul : responsabil cu ho-urile */
     ppid = getppid();

     for(i=0; i<NR_AFISARI; i++)
     {
       pause();
       printf("ho, ");  fflush(stdout);
       kill(ppid, SIGUSR1);
     }
  }
  else
  { /* tatal : responsabil cu hi-urile */

     for(i=0; i<NR_AFISARI; i++)
```

```
    {
      printf("hi-");  fflush(stdout);
      kill(pid, SIGUSR1);
      pause();
    }
  }

  printf("\n");
  /*
    printf("Sfirsit %s.\n", 0==pid?"fiu":"parinte");
  */
  return 0;
}
```

Exerciţiul 10^* **din capitolul 5**.

Rezolvare. Răspunsul corect este că programul dat afişează pe ecran mesajul "test message root". *Justificare*: din procesul curent se creează, în total, 2 noi procese, ultimul executînd comanda `echo` ce scrie mesajul pe ieşirea standard, redirectată în capătul de scriere al canalului intern, din care procesul iniţial va citi acel mesaj "test message root" şi-l va afişa pe ecran.

Exerciţiul 11^* **din capitolul 5**.

Rezolvare. Iată sursa programului `dimens_pipe.c` care determină capacitatea unui canal intern:

```
/*
  File: dimens_pipe.c
*/
#include<stdio.h>
#include<fcntl.h>

int main(void)
{
   int p[2],dimens;
   char c=0;

   /* creare pipe */
   if( pipe(p) == -1)
   {
      perror("Error creare pipe"); exit(1);
   }

   /* setare non-blocking pentru capatul de scriere */
   fcntl(p[1],F_SETFL,O_NONBLOCK);

   dimens=0;
```

```
   while(1)
   {
      if( write(p[1],&c,1) == -1)
      {
         perror("Eroare - umplere pipe");
         break;
      }
      else
      {
         ++dimens;
         if(dimens%1024==0) printf(" %d Ko ...\n", dimens/1024);
      }
   }

   printf("Capacitatea pipe-ului este de %d octeti.\n",dimens);
   return 0;
}
```

Exerciţiul 12^* **din capitolul 5**.

Rezolvare. Iată sursa programului `suma_pipe.c` care rezolvă problema sumei distribuite utilizînd canale interne pentru comunicaţie:

```
/*
  File: suma_pipe.c
  Problema suma distribuita de la lectia fork, rezolvata cu canale interne.
*/
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <errno.h>

extern int errno;


void master_init();
void master_work();
void slave_work(int fdi, int fdo);

/* date pentru cele 3 pipe-uri */
int pipe1i[2], pipe2i[2], pipe3o[2];

int main(void)
{
  int pid1,pid2;

  printf("\n\n\n\n\n");

  /* Crearea celor 3 pipe-uri p1i, p2i si p3o */
```

```
master_init();


/* Creare primul proces slave */
if( (pid1=fork()) == -1)
{
  fprintf(stderr,"Error fork la fiul 1 !\n");
  exit(2);
}

if(pid1 == 0)
{ /* sunt in procesul fiu 1 */

  /* inchid capetele de care nu am nevoie */
  close(pipe1i[1]);
  close(pipe3o[0]);
  close(pipe2i[0]); close(pipe2i[1]);

  slave_work(pipe1i[0],pipe3o[1]);

  return 0;
  /* sfirsit executie fiu 1 */
}
/* else sunt in procesul master, executat in paralel cu if-ul de mai sus */

/* Creare al doilea proces slave */
if( (pid2=fork()) == -1)
{
  fprintf(stderr,"Error fork la fiul 2 !\n");
  exit(2);
}

if(pid2 == 0)
{ /* sunt in procesul fiu 2 */

  /* inchid capetele de care nu am nevoie */
  close(pipe2i[1]);
  close(pipe3o[0]);
  close(pipe1i[0]); close(pipe1i[1]);

  slave_work(pipe2i[0],pipe3o[1]);

  return 0;
  /* sfirsit executie fiu 2 */
}
/* else sunt in procesul master, executat in paralel cu cei doi fii */

  /* inchid capetele de care nu am nevoie */
  close(pipe1i[0]);
  close(pipe2i[0]);
  close(pipe3o[1]);
```

```
  /* Citeste secventa de la tastatura si o transmite fiiilor, apoi
     primeste de la ei cele 2 sume partiale si afiseaza suma lor. */
  master_work();
  return 0;
}

/**************************************************************************/

void master_init()
{
  /* Crearea celor 3 pipe-uri fifo1i, fifo2i si fifo3o */

  if( pipe(pipe1i) == -1 )
  {
     perror("Eroare la crearea pipe-ului pipe1i");
     exit(1);
  }

  if( pipe(pipe2i) == -1 )
  {
     perror("Eroare la crearea pipe-ului pipe2i");
     exit(1);
  }

  if( pipe(pipe3o) == -1 )
  {
     perror("Eroare la crearea pipe-ului pipe3o");
     exit(1);
  }
}

/**************************************************************************/

void master_work()
{
  int nr, flag;
  int sump1,sump2;

  /* Putem citi numerele de la tastatura dupa crearea fiilor,
     desi citirea de la tastatura poate dura oricit de mult (!),
     deoarece sincronizarea necesara aici (fiul trebuie sa-si astepte
     tatal sa scrie in canal pentru a avea ce citi) este realizata prin
     faptul ca citirea din pipe se face, implicit, in mod blocant. */

  /* citirea secventei de numere si impartirea ei intre cele doua canale */
  printf("Introduceti numerele (0 pentru terminare):\n");
  flag=1;
  while(1)
  {
    scanf("%d", &nr);
    if(nr == 0) break;
```

```
    write( (flag==1 ? pipe1i[1] : pipe2i[1]), &nr, sizeof(int));
    flag=3-flag;
  }

  close(pipe1i[1]);
  close(pipe2i[1]);

  /* Aici mai apare o sincronizare: master-ul trebuie sa citeasca
     sumele partiale abia dupa ce acestea au fost calculate si scrise
     in canalul pipe3o de catre procesele slave.
     Rezolvare: citirea din pipe este, implicit, blocanta. */

  /* Citeste prima suma partiala (sosita de la oricare din cei doi slaves) */
  if( read(pipe3o[0], &sump1, sizeof(int)) != sizeof(int))
  {
     fprintf(stderr,"Eroare la prima citire din pipe-ul pipe3o\n");
     exit(7);
  }

  /* Citeste a doua suma partiala */
  if( read(pipe3o[0], &sump2, sizeof(int)) != sizeof(int))
  {
     fprintf(stderr,"Eroare la a doua citire din pipe-ul pipe3o\n");
     exit(8);
  }

  close(pipe3o[0]);

  /* Afiseaza suma */
  printf("Master[PID:%d]> Suma secventei de numere introduse este: %d\n",
         getpid(), sump1+sump2);

}

/*****************************************************************************/

void slave_work(int fdi, int fdo)
{
  int nr, cod_r, suma_partiala;

  suma_partiala=0;

  /* citirea numerelor din pipe, pina intilneste EOF */
  do
  {
    cod_r = read(fdi, &nr, sizeof(int));
    switch(cod_r)
    {
       case sizeof(int) : suma_partiala += nr ;  break;
       case 0  : break;  /* 0 inseamna EOF */
       default : fprintf(stderr,"Eroare la citirea din canalul pipe%ci\n",
                         (fdi==pipe1i[0] ? '1' : '2') );
```

```
                exit(3);
    }
  }while(cod_r != 0);

  close(fdi);

  /* scrierea sumei in pipe */
  if( write(fdo, &suma_partiala, sizeof(int)) == -1)
  {
    perror("Eroare la scrierea in canalul pipe3o");
    exit(4);
  }

  close(fdo);

  /* mesaj informativ pe ecran */
  printf("Slave%c[PID:%d]> Suma partiala: %d\n",
    (fdi==pipe1i[0] ? '1' : '2'), getpid(), suma_partiala);
}

/*************************************************************************/
```

Exerciţiul 21^* **din capitolul 5**.

Rezolvare. Răspunsul corect este că programul dat se blochează la `open` până apare un consumator din fişierul *fifo* "a.txt" creat, şi abia apoi copie intrarea standard în fişierul *fifo*. *Justificare*: deoarece programul încearcă să deschidă doar capătul de scriere al *fifo*-ului, deci trebuie să se sincronizeze cu un "cititor".

Exerciţiul 22^* **din capitolul 5**.

Rezolvare. Iată sursa programului `dimens_fifo.c` care determină capacitatea unui canal extern:

```
/*
  File: dimens_fifo.c
*/
#include<stdio.h>
#include<fcntl.h>
#include<errno.h>

extern int errno;

int main(void)
{
   int p,dimens;
   char c=0;

   /* creare fifo */
```

```
   if( mkfifo("canal", 0666) == -1 )
   {
      if(errno == 17)   // 17 = errno for "File exists"
      {  fprintf(stdout,"Note: fifo 'canal' exista deja !\n");   }
      else
      {  perror("Error creare fifo");  exit(1);   }
   }

   /* setare non-blocking necesara pentru capatul de scriere */
   p=open("canal",O_RDWR | O_NONBLOCK);

   dimens=0;

   while(1)
   {
      if( write(p,&c,1) == -1)
      {
         perror("Eroare - umplere fifo");
         break;
      }
      else
      {
         ++dimens;
         if(dimens%1024 == 0) printf(" %d Ko ...\n", dimens/1024);
      }
   }

   printf("Capacitatea fifo-ului este de %d octeti.\n", dimens);
   return 0;
}
```

Exerciţiul 24^* **din capitolul 5**.

Rezolvare. Iată sursa programului `suma_fifo.c` care rezolvă problema sumei distribuite utilizînd canale externe pentru comunicaţie:

```
/*
  File: suma_fifo.c
  Problema suma distribuita de la lectia fork, rezolvata cu canale externe.
*/
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

extern int errno;
```

```
void master_init();
void master_work();
void slave_work(char* fi, char* fo);

int main()
{
  int pid1,pid2;

  printf("\n\n\n\n\n");

  /* Crearea celor 3 fifo-uri fifo1i, fifo2i si fifo3o */
  master_init();


  /* Creare primul proces slave */
  if( (pid1=fork()) == -1)
  {
    fprintf(stderr,"Error fork la fiul 1 !\n");
    exit(2);
  }

  if(pid1 == 0)
  { /* sunt in procesul fiu 1 */

    slave_work("fifo1i","fifo3o");
    return 0;
    /* sfirsit executie fiu 1 */
  }
  /* else sunt in procesul master, executat in paralel cu if-ul de mai sus */

  /* Creare al doilea proces slave */
  if( (pid2=fork()) == -1)
  {
    fprintf(stderr,"Error fork la fiul 2 !\n");
    exit(2);
  }

  if(pid2 == 0)
  { /* sunt in procesul fiu 2 */

    slave_work("fifo2i","fifo3o");
    return 0;
    /* sfirsit executie fiu 2 */
  }
  /* else sunt in procesul master, executat in paralel cu cei doi fii */

  /* Citeste secventa de la tastatura si o transmite fiiilor, apoi
     primeste de la ei cele 2 sume partiale si afiseaza suma lor. */
  master_work();
  return 0;
}
```

```
/*************************************************************************/

void master_init()
{
  /* Crearea celor 3 fifo-uri fifo1i, fifo2i si fifo3o */

  if( mkfifo("fifo1i", 0600) == -1 )
  {
    if(errno != 17)   /* 17 == errno for "File exists" */
    {
       perror("Eroare la crearea fifo-ului fifo1i");
       exit(1);
    }
  }

  if( mkfifo("fifo2i", 0600) == -1 )
  {
    if(errno != 17)
    {
       perror("Eroare la crearea fifo-ului fifo2i");
       exit(1);
    }
  }

  if( mkfifo("fifo3o", 0600) == -1 )
  {
    if(errno != 17)
    {
       perror("Eroare la crearea fifo-ului fifo3o");
       exit(1);
    }
  }
}

/*************************************************************************/

void master_work()
{
  int fd1, fd2, fd3;
  int nr, flag;
  int sump1,sump2;

  /* Putem citi numerele de la tastatura dupa crearea fiilor,
     desi citirea de la tastatura poate dura oricit de mult (!),
     deoarece sincronizarea necesara aici (fiul trebuie sa-si astepte
     tatal sa scrie in fifo pentru a avea ce citi) este realizata prin
     faptul ca citirea din fifo se face, implicit, in mod blocant. */

  /* deschiderea celor doua fifo-uri de intrare
    (prin care trimite numere la slaves) */
  if( (fd1=open("fifo1i",O_WRONLY)) == -1)
```

```
{
  perror("Eroare la deschiderea fifo-ului fifo1i\n");
  exit(5);
}
if( (fd2=open("fifo2i",O_WRONLY)) == -1)
{
  perror("Eroare la deschiderea fifo-ului fifo2i\n");
  exit(5);
}

/* citirea secventei de numere si impartirea ei intre cele 2 fifo-uri */
printf("Introduceti numerele (0 pentru terminare):\n");
flag=1;
while(1)
{
  scanf("%d", &nr);
  if(nr == 0) break;

  write( (flag==1?fd1:fd2), &nr, sizeof(int));
  flag=3-flag;
}

close(fd1);
close(fd2);

/* Aici mai apare o sincronizare: master-ul trebuie sa citeasca
sumele partiale abia dupa ce acestea au fost calculate si scrise
in fifo-ul fifo3o de catre procesele slave.
Rezolvare: deschiderea in master (implicit, blocanta!) a fifo-ului
se va sincroniza cu deschiderea sa in primul din cei doi slaves.
Dar astfel poate fi pierduta suma de la celalalt slave
(scenariu posibil: primul slave inchide fifo-ul si masterul citeste
astfel EOF inainte ca al doilea slave sa apuce sa deschida fifo-ul).
De aceea vom folosi wait in master pentru a fi siguri ca s-a terminat
si al doilea slave.
*/

/* deschiderea fifo-ului de iesire (prin care primeste sumele de la slaves) */
if( (fd3=open("fifo3o",O_RDWR)) == -1)
{
  perror("Eroare la deschiderea fifo-ului fifo3o\n");
  exit(6);
}

/* Citeste prima suma partiala (sosita de la oricare din cei doi slaves) */
if( read(fd3, &sump1, sizeof(int)) != sizeof(int))
{
   fprintf(stderr,"Eroare la prima citire din fifo-ul fifo3o\n");
   exit(7);
}

/* Citeste a doua suma partiala.
```

```
     Aici trebuie sa fim siguri ca si al doilea slave a apucat sa
     deschida fifo-ul, caci altfel read-ul urmator va citi EOF.
     Deci il vom astepta (cu 2 wait-uri):

  wait(NULL);
  wait(NULL);

     Totusi, aceasta solutie nu este cea mai optima - ea functioneaza
     numai in acest caz: trimiterea sumei de catre slave se face chiar
     inainte de terminarea sa.

     In cazul general (daca slave-ul ar mai fi avut apoi si altceva de facut),
     solutia este deschiderea in master a fifo-ului fifo3o la ambele capete,
     nu doar la cel de citire (cel de care avea nevoie masterul), ceea ce am
     si facut in apelul open de mai sus (am folosit O_RDWR in loc de O_RDONLY).
  */
  if( read(fd3, &sump2, sizeof(int)) != sizeof(int))
  {
     fprintf(stderr,"Eroare la a doua citire din fifo-ul fifo3o\n");
     exit(8);
  }

  close(fd3);

  /* Afiseaza suma */
  printf("Master[PID:%d]> Suma secventei de numere introduse este: %d\n",
         getpid(), sump1+sump2);

}

/*************************************************************************/

void slave_work(char* fi, char* fo)
{
  int fd1,fd2;
  int nr, cod_r, suma_partiala;

  /* deschiderea fifo-ului de intrare (prin care primeste numere de la master) */
  if( (fd1=open(fi,O_RDONLY)) == -1)
  {
    fprintf(stderr,"Eroare la deschiderea fifo-ului %s\n",fi);
    perror("");
    exit(2);
  }

  suma_partiala=0;

  /* citirea numerelor din fifo, pina intilneste EOF */
  do
  {
    cod_r = read(fd1, &nr, sizeof(int));
    switch(cod_r)
```

```
    {
       case sizeof(int) : suma_partiala += nr ;  break;
       case 0     : break;  /* 0 inseamna EOF */
       default    : fprintf(stderr,"Eroare la citirea din fifo-ul %s\n",fi);
                    exit(3);
    }
  }while(cod_r != 0);

  close(fd1);

  /* deschiderea fifo-ului de iesire (prin care trimite suma la master) */
  if( (fd2=open(fo,O_WRONLY)) == -1)
  {
    fprintf(stderr,"Eroare la deschiderea fifo-ului %s\n",fo);
    perror("");
    exit(4);
  }

  /* scrierea sumei in fifo */
  write(fd2, &suma_partiala, sizeof(int));

  close(fd2);

  /* mesaj informativ pe ecran */
  printf("Slave%c[PID:%d]> Suma partiala: %d\n",
    (!strcmp(fi,"fifo1i") ? '1' : '2'), getpid(), suma_partiala);
}

/*************************************************************************/
```