Sisteme de operare

dr. Dan Cosma, 2013

Preliminarii

Audiență și impact

Cui i se adresează cursul:

- *Studenților din anul III și tuturor persoanelor interesate să înțeleagă mecanismele esențiale ale sistemelor software moderne*

Obiective și abilități câștigate

- utilizarea *la nivel de profesionist în domeniul ingineriei software* a sistemelor de operare moderne

- înțelegerea conceptelor fundamentale asociate sistemelor de operare, cu impact direct și esențial asupra dezvoltării de aplicații software moderne

- folosirea în programe, la nivel avansat, a serviciilor oferite de sistemul de operare și de bibliotecile de funcții asociate
  - ➥ programare de sistem (system programming)

sau, altfel spus, poate ajuta la construirea unei cariere de durată:

→ → →

➡

Structura disciplinei *Sisteme de operare*

Curs

- 14 săptămâni a câte 2 ore

  → prezentări ale temelor fundamentale, explicații, exemple

- interactivitate

  → discuții, analize, probleme, răspunsuri -- nu ezitați să puneți întrebări !

- feedback

  → sugestii, observații, reclamații -- toate sunt binevenite, pot duce la îmbunătățirea cursului și a comunicării

Laborator

- lucrări de laborator individuale

  → surprind aspecte practice esențiale,
  → introduc concepte fundamentale pentru dezvoltarea de sisteme software,
  → ajută în mod direct formarea dvs. ca ingineri software profesioniști

- interactivitate

  → discuții, analize, probleme, răspunsuri -- nu ezitați să puneți întrebări !

- feedback

  → sugestii, observații, reclamații -- toate sunt binevenite, pot duce la îmbunătățirea lucrărilor de laborator și a comunicării

Evaluare

- 3 teste la laborator

  → programe complete, de mici dimensiuni, abordabile in aproximativ o oră
  → testează abilitățile de programare sistem sau utilizare avansată a SO acumulate la data programată a testului
  → definesc nota la laborator (medie ponderată a notelor la teste)
  → pondere în nota finală: 35%

- examen în sesiune

  → o suită de întrebări care evaluează gradul de înțelegere a conceptelor studiate și a tehnicilor învățate
  → o componentă practică, menită a evalua măsura în care ați asimilat cunoștințele de la laborator
  → pondere în nota finală: 65%

- feedback

  → sugestii, observații, reclamații -- toate sunt binevenite, pot duce la îmbunătățirea evaluării și depistarea eventualelor erori sau probleme

Feedback

- e-mail: dan punct cosma at cs punct upt punct ro

- în direct, la curs sau laborator

Resurse

- Site-ul cursului
  - → pe portalul “LOOSE” al cursurilor de software engineering
  - → http://loose.upt.ro/~oose/pmwiki.php/SO/OperatingSystems
- Site-ul laboratorului
  - → conține îndrumătorul de laborator complet
  - → http://labs.cs.upt.ro/labs/so/html/index.html

Bibliografie

1. W.R.Stevens, S.A.Rago, *Advanced Programming in the UNIX Environment, Third Edition;* Addison Wesley, 2013

2. W. Stallings, *Operating Systems: Internals and Design Principles, 7th edition*, Prentice Hall, 2011

3. Eric S. Raymond: *The Art of UNIX Programming*, Addison-Wesley, 2003

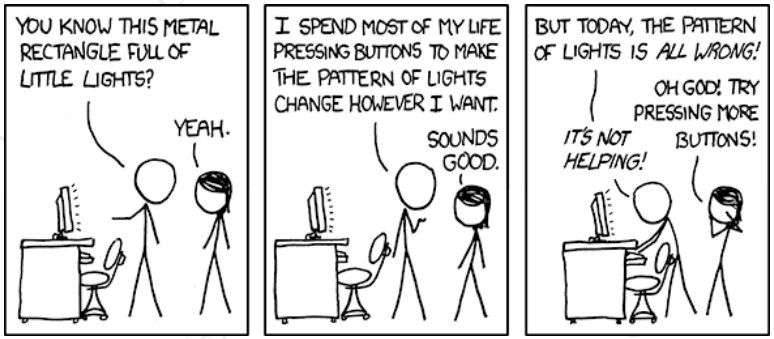
4. A. Robbins: *UNIX in a Nutshell, Fourth Edition;* O'Reilly, 20055. Ioan Jurca: *Programarea de sistem in UNIX*, Editura de Vest, Timisoara. 2005

6. A. S. Tannenbaum: *Modern Operating Systems, 2nd Edition*, Prentice Hall, 2001

1. Introducere

Ce este un sistem de operare ?

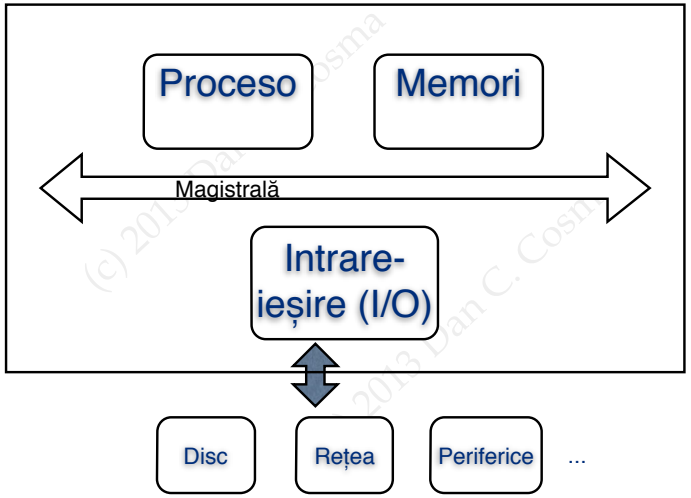
Calculatoarele sunt mașini complexe...



http://xkcd.com/722/

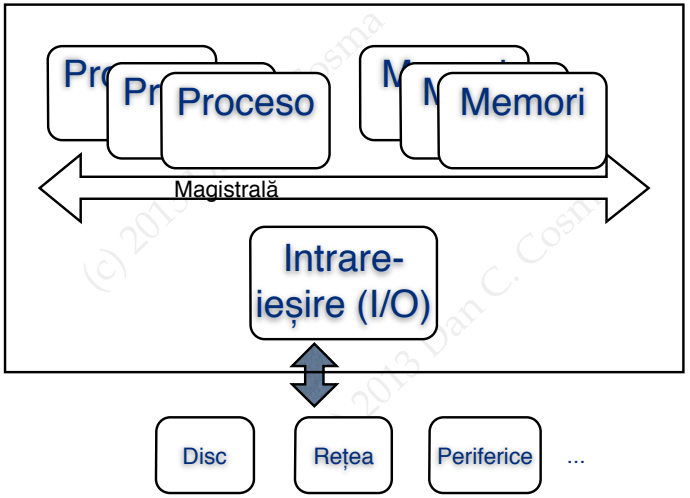
... care trebuie folosite eficient

Calculatorul



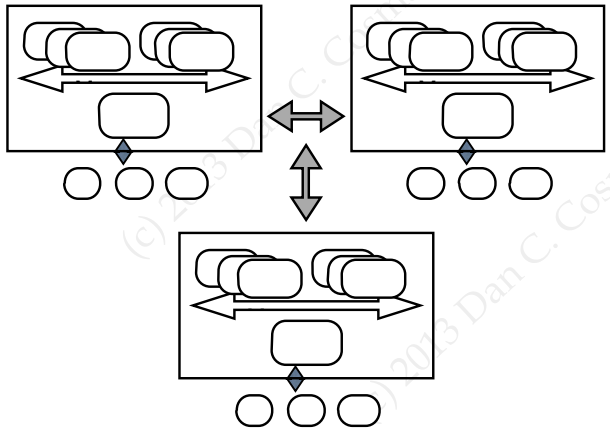
(arhitectura „clasică”)

Calculatorul



(arhitectură mai complexă)

Calculatorul



(arhitectură și mai complexă)

Calculatorul...

(... altceva)

... e nevoie de mecanisme care să le facă mai abordabile

→ în utilizare

→ în dezvoltarea de aplicații

Sistemul de operare

O colecție de programe care:

- gestionează resursele hardware
- creează abstracțiuni de nivel înalt pentru resurse
- controlează execuția aplicațiilor
- oferă o interfață spre aplicații
- oferă o interfață cu utilizatorul

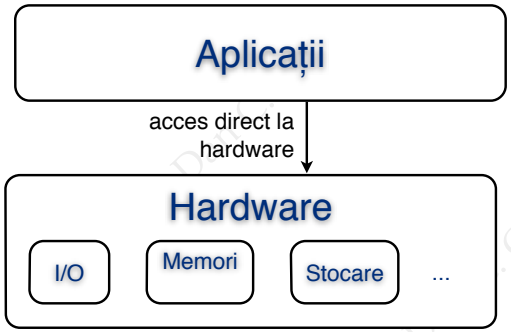
în funcție de variantele de sistem, diverse roluri pot fi preluate de aplicații (exemplu: interfața cu utilizatorul)

Obiectivele sistemului de operare [2]

- **Ușurință în utilizare**
  → pentru a facilita accesul la resurse
- **Eficiență**
  → în modul de utilizare și gestionare a resurselor
- **Abilitatea de a evolua**
  → capacitatea de a i se adăuga în timp noi funcționalități, fără ca acest lucru să afecteze oferirea serviciilor sistem

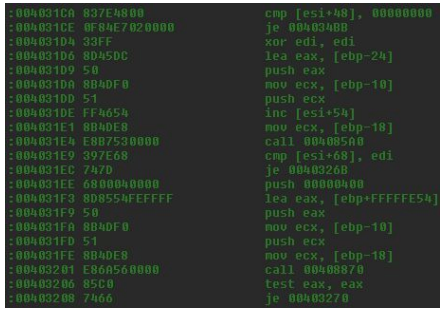
[2] W. Stallings, *Operating Systems: Internals and Design Principles, 7th edition*, Prentice Hall, 2011

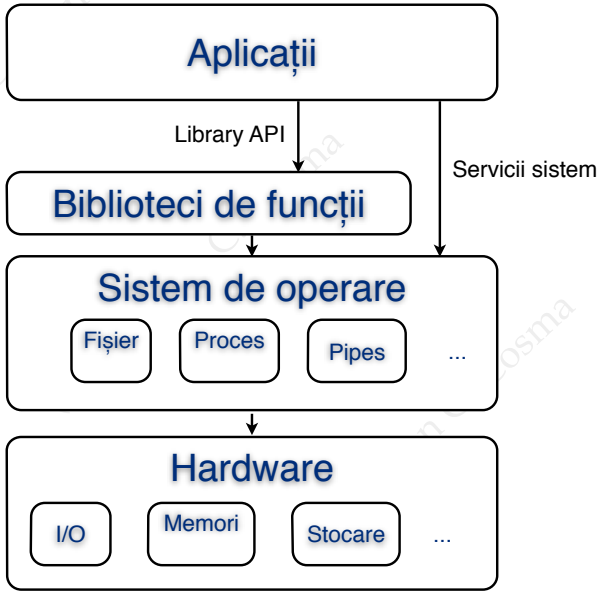
Ce-ar fi dacă...?



Acces direct la hardware

- programe extrem de complexe
- duplicări de cod în funcționalitățile aplicațiilor diferite (ex: prelucrarea formatelor de stocare pe disc)
- dependență mare de versiunile de dispozitive hardware
- lipsă de portabilitate
- vulnerabilitate la evoluția în timp a sistemului de calcul
- ...





Servicii ale sistemului de operare

- Execuția și controlul programelor
- Gestiunea memoriei
- Acces la dispozitivele I/O
- Acces simplificat și controlat la date (fișiere etc.)
- Detecția și tratarea erorilor
- Unelte de dezvoltare software
- Securitate, monitorizare, sincronizare etc.

Exploatarea resurselor

- gestiunea eficientă a timpului procesor

  → prin algoritmi potriviți de planificare la execuție a entităților/programelor care rulează în paralel sau secvențial în sistem
- gestiunea eficientă a intrărilor și ieșirilor

  → caching, reutilizarea resurselor care abstractizează dispozitive etc.
- gestiunea eficientă a memoriei

  → eliberarea memoriei nefolosite, swapping, scheme de memorie virtuală etc.
- gestiunea eficientă a comunicării între programe

  → mecanisme rapide de comunicare sincronă și asincronă
- ...

Sistemul de operare are nevoie să evolueze când

- se modifică părți din hardware

  → ex: adăugarea unui dispozitiv (disc, stick de memorie, imprimantă, ...) nou
- apare nevoia de servicii noi

  → ex: noi protocoale de conectare la rețea, facilități îmbunătățite pentru utilizator, formate noi de organizare a datelor pe disc etc.
- trebuie corectate erori

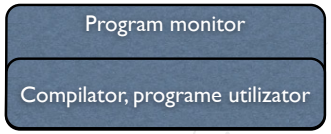
  → ex: rezolvarea unor probleme de securitate, rezolvarea unor bug-uri
- optimizări

  → ex: algoritmi mai rapizi de acces la disc, răspuns mai rapid al interfeței cu utilizatorul etc.

Sistemele de operare moderne includ nativ mecanisme avansate de actualizare (update, upgrade) și gestiune a pachetelor software

Scurt istoric al sistemelor de operare

- <1955: Mainframe, fără sistem de operare
  → utilizate pe rând, programele introduse manual, cu cartele sau benzi magnetice
- Anii ’50-’60: Mainframe, cu operare in loturi (“batch systems”)
  → monitorul, un “SO” rezident permanent in memorie, care permitea lansarea de “job”-uri



- Cca. 1955: Mainframe cu sisteme de operare dedicate

- 1969-1971: Apariția sistemului UNIX
  → Ken Thompson (Bell Labs) incepe să lucreze la un nou SO, după ce Bell Laboratories se retrage din proiectul Multics
  → Co-autor: Dennis Ritchie, care va crea și limbajul C (1971-1973), pentru a-l folosi la scrierea UNIX (prima versiune de UNIX era scrisă în asamblare, iar aplicațiile într-un limbaj interpretat numit “B”)
  → Prima versiune UNIX: pe calculatorul PDP 7 (DEC), 1969-1970 (“UNICS” - “UNiplexed Information and Computing Service”)
  → Începe să se răspândească gratuit, la nivel de surse
- 1971: UNIX pe PDP 11
  → folosit pentru procesare de text la departamentul de patente al Bell Labs
- 1972: UNIX raportează 10 instalări
  → distribuirea sa gratuită îl face extrem de popular în industrie și universități
  → multitasking nativ, multiuser

- ~1971 - anii ’80: “Revoluția” Home Computer
  → apar primele calculatoare personale, datorită producției de masă de microprocesoare
  →”sistemul de operare”: interpretor BASIC stocat in ROM, capabil să ruleze programe, oferind o simplă interfață de comandă pentru utilizator
  →aplicații: jocuri, limbaje de programare, interpretoare, compilatoare
  → exemple: Apple II, ZX Spectrum, Commodore 64, HC-85 (RO), Tim-S (RO, TM)
- ~1974: CP/M
  → “Control Program/Monitor” -> “Control Program for Microcomputers”
  → folosit in medii business, educație, pe microcalculatoare
  → aprox. 5 comenzi, unifică serviciile oferite programelor (pt. portabilitate)
  → exemple de calculatoare: Altair 8080, Amstrad PCW, CUB-Z (RO)
- 1977: BSD
  → “Berkeley Software Distribution” / ”Berkeley UNIX”
  → dezvoltat la University of California, Berkeley, derivat din sursele UNIX de la Bell Labs
  → în prezent, unul din cele mai importante sisteme de operare open source

- 1980: MS-DOS
  → IBM contractează Microsoft pentru scrierea unui SO destinat noilor calculatoare personale (PC) dezvoltate de companie, după ce o discuție similară cu creatorul CP/M eșuează; Microsoft reține dreptul de a vinde sistemul MS-DOS separat de hardware
  → bazat pe QDOS (“Quick and Dirty OS”), o clonă de CP/M, dezvoltată de Tim Paterson, in 6 săptămâni, pentru compania la care lucra
  → QDOS cumpărat de Bill Gates (Microsoft) cu 50 000 $; înțelegerea cu IBM a fost ținută secretă de Microsoft la cumpărare
  → după un an, Tim Paterson se angajează la Microsoft
  → monotasking, interfață în linie de comandă inspirată din CP/M
  → primele generații de PC-uri nu aveau capacitatea hardware de a rula UNIX
- 1980-1990: “The UNIX Wars”[3]
  → perioadă în care UNIX e exploatat comercial
  → diferitele versiuni de UNIX concurează cu succes pe piață
  → apare TCP/IP și este adoptat de UNIX, prima dată la Berkeley
  → comercializarea UNIX elimină libera circulație a codului sursă și are un efect secundar: reduce vitalitatea dezvoltării sistemului
  → diferite încercări de a porta UNIX pe i386 eșuează

[3] Eric S. Raymond: The Art of UNIX Programming, Addison-Wesley, 2003

- 1983: Richard Stallman pornește proiectul GNU
  → cu scopul de a crea un UNIX gratuit
  → introduce licența GNU General Public License
  → deși nucleul UNIX rezultat (Hurd) nu are succes, proiectul GNU devine unul din principalii promotori ai mișcării open source
- 1984: Apple Macintosh “System Software”
  → rula pe Apple Macintosh 128K (primul calculator Apple)
  → va fi numit mai târziu Mac OS
  → este sistemul de operare care a popularizat ideea de interfață grafică
- 1985: Windows 1.0
  → interfață grafică cu utilizatorul pentru sistemul MS-DOS
  → anunțată în 1983, semăna foarte mult cu interfața Apple Macintosh; la lansare a apărut într-o formă modificată
- 1990: Windows 3.0
  → primul succes semnificativ al Windows
  → multitasking parțial (cooperativ), memorie virtuala (i386)
  → versiuni importante: Windows 3.1, Windows 3.11 for Workgroups

- August 1991: Prima versiune de Linux
  → dezvoltat de Linus Torvalds, atunci student la Universitatea din Helsinki (Finlanda)
  → urmează complet standardele specifice UNIX
  → open source, devine extrem de popular, ajunge rapid un SO matur
- 1992: BSD e portat pe i386
- 1993: Windows NT
  → versiune nouă de sistem, diferită de celelalte sisteme Windows
  → multitasking, multiuser nativ
  → primul sistem complet pe 32 biți; în prezent, are și versiuni de 64 biți
  → este sistemul care va evolua devenind, pe rând, toate versiunile moderne de Windows (XP, 2000, Vista, 7, 8)
- 1999: Apple OS X
  → bazat pe UNIX
  → nucleul său (Darwin) va deveni bază inclusiv pentru versiunile mobile (iOS)
- ...

Tipuri, variante, versiuni de sisteme de operare

Tipuri de sisteme de operare

Clasificare după destinație

- sisteme de operare server
  → UNIX (e.g. Solaris), Linux, BSD, Windows Server, OS X Server
- sisteme de operare desktop
  → Windows, Linux, BSD, OS X, Chrome OS
- sisteme de operare mobile
  → Android, iOS, Windows 8, Symbian, Bada, BlackBerry OS, Palm OS
- sisteme de operare embedded
  → OpenWRT (Linux), Windows CE, LynxOS
- sisteme de operare în rețea
  → Novell NetWare, JunOS (Juniper), Cisco IOS

Tipuri de sisteme de operare

Clasificare după originea nucleului sistemului

- UNIX / Linux
  → Solaris, HP-UX, BSD, OpenBSD, FreeBSD, Linux (toate variantele), Android, OS X, OS X Server, iOS, webOS, Chrome OS, Tizen, openWRT, Firefox OS etc.
- Windows NT
  → Windows NT, 2000, XP, Vista, 7, 8 (inclusiv variantele Server acolo unde ele există), Windows Phone 8
- Windows
  → Windows 95, Windows 98, Windows Millenium
- alte nuclee proprietare
  → Symbian, Palm OS, ...

Tipuri de sisteme de operare

Clasificare după modelul de licențiere

- sisteme de operare proprietare
  → diverse variante de UNIX (e.g. Solaris), Windows, OS X, BlackBerry OS
- sisteme de operare open source
  → Linux, BSD
- sisteme de operare open source cu componente proprietare
  → Android, Tizen (Samsung, Intel, Linux Foundation), webOS (Palm→HP→LG)
- sisteme de operare proprietare cu componente open source
  → OS X, iOS (nucleu open source “Darwin”, derivat din BSD)

Modele de licențiere software

Licență software

➥ *Un instrument legal (contract) care descrie și impune condițiile în care un produs software poate fi utilizat, modificat și/sau distribuit*

Modele de licențiere software

- Free/Open source

  ➥ permite ca sursele, concepția și proiectul codului să fie utilizate, modificate și raspândite/partajate în mod liber (în anumite condiții)

  - open source “copyleft”

    → ex: GNU General Public License (GPL)

    → libertate nelimitată de utilizare, studiu, modificare și redistribuire, atâta vreme cât redistribuirea nu adaugă restricții în plus față de GPL (de exemplu nu-l transformă in software proprietar, “închis”)

  - open source permisivă

    → ex: BSD License

    → libertate nelimitată de utilizare și studiu, libertate de modificare a codului. Condițiile de redistribuire sunt mai relaxate, nu impun păstrarea caracterului complet deschis al software-ului.

- Proprietary / closed source

  ➥ utilizarea unui număr limitat de cópii, conform unei EULA (End-User Licence Agreement

  ➥ proprietatea asupra surselor rămâne la producător; nu permite libertăți de modificare și distribuție, decât în condiții foarte restrictive

Versiuni de sisteme de operare

- Windows

  → 32 bit: Windows NT, 95, 98, Millenium, XP

  → 32/64 bit: Windows XP, Vista, 7, 8

- Distribuții Linux

  → sunt sisteme de operare sub formă de soluții complete, care includ un nucleu Linux și o suită vastă de aplicații

  → majoritatea sunt gratuite / open source, unele sunt comerciale, altele oferă suport tehnic contra cost

  → includ sisteme complexe de gestionare a pachetelor software, sunt ușor extensibile și upgradabile

  → exemple: Slackware, openSUSE, Debian, Fedora, Ubuntu, Mandriva, Mint Linux, CentOS, RedHat, Arch Linux, ...

- Apple Mac OS X

  → certificat UNIX, cu nucleu open source (Darwin) derivat din BSD

  → 10.4: "Tiger", 10.5: "Leopard", 10.6: "Snow Leopard", 10.7: "Lion", 10.8: "Mountain Lion", 10.9: "Mavericks"

Elemente de arhitectură a sistemelor de operare

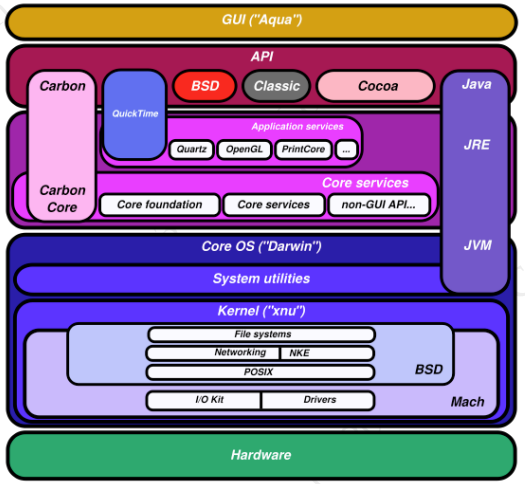
Componentele sistemului de operare

- **Nucleu (kernel)**
  → conține funcționalitatea de bază a sistemului de operare
  → dimensiunea sa depinde mult de arhitectura propriu-zisă

- **Subsisteme funcționale**
  → alte componente software din sistemul de operare, cu diverse roluri
  → pot include utilitare sistem, interfețe de programare, biblioteci de funcții specializate, servicii sistem implementate în afara nucleului etc.

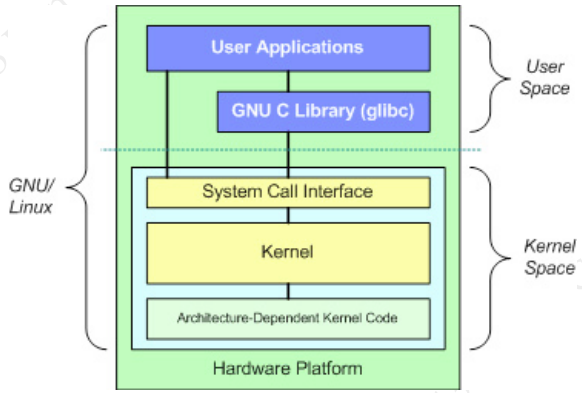
Diversele arhitecturi definesc raporturi funcționale diferite între nucleu și celelalte subsisteme

Exemplu: arhitectura Mac OS X



(c) Wikimedia Commons, http://commons.wikimedia.org/wiki/File:MacOSXArchitecture.svg

Exemplu: arhitectura Linux



sursă și (c): http://www.ibm.com/developerworks/library/l-linux-kernel/

Spații de memorie

Sistemele de operare moderne împart memoria virtuală în două categorii de spații distincte

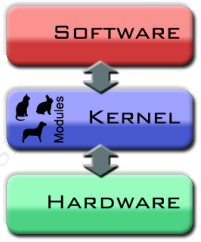
- **Spațiu nucleu (kernel space)**
  → spațiul de memorie în care rulează componentele nucleu și majoritatea driverelor

- **Spațiu utilizator (user space)**
  → în care rulează aplicațiile utilizator, utilitarele, unele servicii sau drivere din sistem

Separația duce la un bun control bazat pe privilegii al componentelor software, protecție, securitate

Tipuri de nucleu

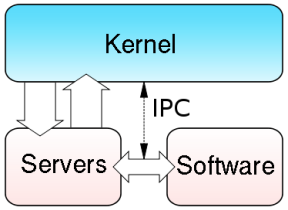
- Nucleu monolitic
  → toate serviciile rulează împreună cu firul principal de execuție, în același spațiu de memorie
  → poate conține module încărcabile dinamic (Linux)
  → avantaje: acces direct la hardware, comunicare rapidă în interiorul nucleului, ușor de implementat
  → dezavantaje: dependențe mari între componentele nucleului, mentenanță dificilă
  → exemplu: Linux



sursa: Wikipedia

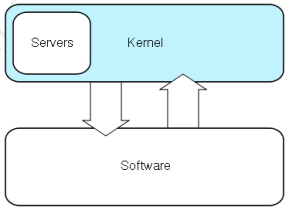
Tipuri de nucleu

- Microkernel
  → un set de componente “server”, construite în jurul unui nucleu minimal
  → nucleul oferă doar servicii de bază: comunicare (IPC), gestiunea memoriei, gestiunea proceselor
  → “serverele” implementează celelalte servicii și se află în spații de memorie diferite față de nucleu
  → avantaje: flexibilitate, mentenanță ușoară, dependențe minime
  → dezavantaje: performanță redusă (comunicare intensă între servere) memorie folosită mai mare, mai greu de depanat
  → exemplu: QNX



sursa: Wikipedia

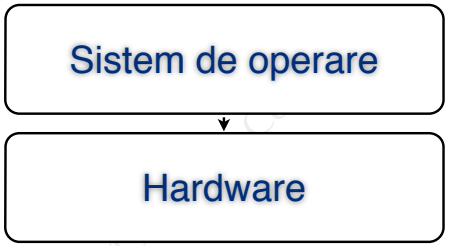
- Nuclee hibride
  → similare cu microkernel, dar includ mai multe servicii în nucleu pentru mărirea performanței
  → Exemple: OS X (Darwin), Windows NT



sursa: Wikipedia

Mașini virtuale

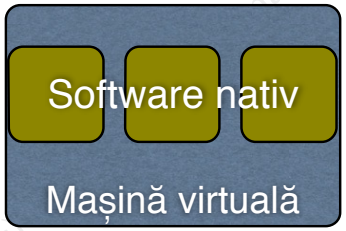
- Arhitectura tradițională



- Dezavantaje
  → un singur SO la un moment dat
  → aplicațiile trebuie portate pe mai multe sisteme de operare

- Soluție: virtualizare

- Mașină virtuală
  un software care simulează complet un sistem hardware (computer), oferind programelor un mediu virtual de execuție

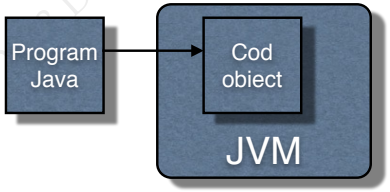


- Caracteristici:
  → pe mașina virtuală poate fi rulat software nativ, care altfel ar rula doar pe sistemul hardware original
  → un același calculator poate rula mai multe mașini virtuale simultan, cu arhitecturi originale diferite
  → programele instalate în mașina virtuală rulează ca și cum ar funcționa pe o mașină reală și sunt complet izolate de sistemul gazdă și de celelalte mașini virtuale

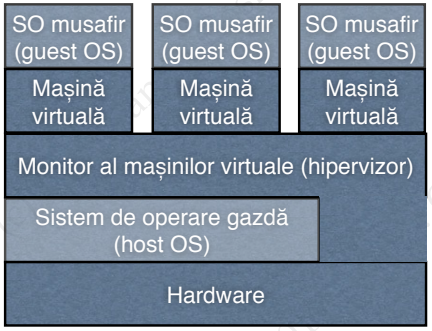
Tipuri de mașini virtuale

- Mașină virtuală tip sistem
  → simulează un hardware complet, de cele mai multe ori un sistem de calcul real

- Mașină virtuală tip proces
  → oferă un mediu virtual de execuție pentru a rula programe scrise într-un limbaj de programare specific
  → mașina e dezvoltată doar pentru a rula codul obiect al acestor programe, nu simulează o mașină reală
  → oferă portabilitate programelor scrise în acel limbaj
  → exemplu: Java Virtual Machine (JVM)



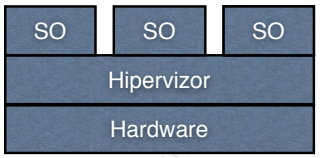
Arhitectura generală



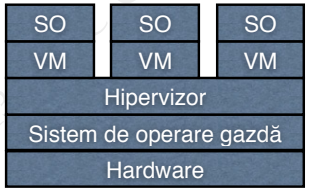
Tehnici de virtualizare

A. Nativă
→ virtualizează arhitectura hardware pe care rulează

- Hipervizor de tip 1 (nativ)*
  → accesează direct hardware-ul



- Hipervizor de tip 2*
  →rulează peste un sistem de operare convențional



*După Gerald J. Popek and Robert P. Goldberg: "Formal Requirements for Virtualizable Third Generation Architectures", 1974

Tehnici de virtualizare

B. Emulare arhitecturală
→ virtualizează o arhitectură hardware străină
→ arhitectura virtuală nu e neapărat una cu corespondent real în hardware

C. Virtualizare la nivelul sistemului de operare
→ tehnologie care virtualizează *servere* în cadrul sistemului de operare
→ nucleul sistemului de operare oferă mai multe spații utilizator distincte, care apar utilizatorului ca servere de sine stătătoare
→ spațiile distincte sunt izolate unele de altele și nu se influențează reciproc
→nu toate serviciile sunt oferite severelor virtuale
→nu pot găzdui sisteme de operare diferite de cel real

2. Utilizarea avansată a sistemelor de operare

“Filozofia” de proiectare a sistemelor de operare

Deziderate urmărite la proiectare

- Publicul-țintă
  → uz general, domenii specifice (desktop, server, mobile, embedded), tipul de utilizator (avansat, novice, consumer)
- Modalitatea de expresie
  → tipul prelucrărilor (job-uri, multiuser), tipul de interfață cu utilizatorul (linie de comandă, interfață grafică), mecanismul principal de interacțiune cu utilizatorul (comenzi directe, gesturi touch, voce, butoane si ferestre,...) etc.
- „Filozofia” urmărită
  → principii care stau la baza construcției sistemului, aplicabile întregului sistem, indiferent de celelalte deziderate menționate

„Filozofia” UNIX, pe scurt

- Modularitate și interconectivitate
  → scrierea de componente software ușor conectabile: ieșirea oricărui program să poată fi intrarea altuia
- Claritate și simplitate
  → sunt de preferat aplicații clare și simple față de programe complicate inutil
- Scop precis
  → un program să facă un singur lucru, dar bine

Prin combinarea liberă a unor componente simple, cu scop precis și ușor interconectabile se pot obține comportamente complexe și diverse, fără a avea dezavantajul dat de o complexitate exagerată a software-ului propriu-zis.

De ce UNIX ?

Comenzi UNIX

Interfața în linie de comandă UNIX

Expresivă
→ permite exprimarea directă și nemijlocită a necesităților utilizatorului
→ face apel la concepte simple, bine definite și cu utilitate precisă

Puternică
→ varietate mare de comenzi disponibile
→ configurabilitate extinsă a comenzilor (opțiuni, fișiere de configurare, variabile de mediu)
→ flexibilitate maximă prin combinarea liberă a comenzilor pentru a realiza acțiuni mai complexe

Adaptabilă
→ configurabilă, permite utilizarea de către persoane cu diferite niveluri de experiență: nu complică prea mult interacțiunea, dar nici nu o limitează

Independentă
→ nu depinde de facilități hardware speciale: poate fi folosită ușor pe orice sistem, local sau de la distanță, ramânând la fel de puternică

Interpretorul de comenzi

= Program interactiv prezentat utilizatorului

→ oferă o interfață în linie de comandă
→ oferă metode de lansare în execuție a comenzilor, de înlănțuire a acestora, de control
→ în UNIX există mai multe interpretoare de comenzi, selectabile de utilizator
→ utilizatorii pot porni mai multe interpretoare, după dorință (atât în mediu grafic, în ferestre separate, cât și în mod text)
→ oferă facilități de programare (scripting)
→ exemple: sh, csh, tcsh, bash

Terminologie:

- *shell* (în sens larg) - orice interfață cu utilizatorul
- *shell* (în sens restrâns) - interpretorul de comenzi
- *shell script* - un program scris folosind sintaxa si semantica recunoscute de interpretorul de comenzi, care folosește comenzi ale sistemului de operare pentru a efectua diverse operații

Tipuri de comenzi

- Comenzi interne
  → recunoscute direct de interpretorul de comenzi
  → exemple: cd, break, fg, bg, source, eval, exec, exit
- Comenzi externe
  → programe executabile din afara interpretorului, existente ca programe separate pe disc, incluzând atât comenzile specifice sistemului de operare, cât și (o parte din) aplicațiile instalate
  → exemple de comenzi UNIX externe: ls, man, cat, cut, ps, top

Comenzi, parametri, intrări și ieșiri

- Parametri
  → specificați de utilizator în linia de comandă
  → sunt cuvinte separate prin spațiu
- Intrare
  → implicit tastatura
  → poate fi redirectată din fișiere sau poate veni de la alte comenzi
  → descriptor standard: 0 (“stdin”)
- Ieșiri
  → implicit ecranul sau fereastra interpretorului curent
  → pot fi redirectate spre fișiere sau poate fi trimisa către alte comenzi
  → descriptori standard: 1 (“stdout”) si 2 (“stderr”)

Comenzi, parametri, intrări și ieșiri

- Parametri

```
ls -al . | grep ".gmail-"
```

- Intrare

```
$ grep "abc"
abc 123
abc 123
asdf 234
123 abc 5678
123 abc 5678
```

- Ieșiri

```
$ ls -al .
 drwx------ 28 root root  4096 Oct  4 15:35 .
 drwxr-xr-x 25 root root  4096 Jul 10 13:35 ..
 -rw-------  1 root root  8902 Oct  9 16:29 .bash_history
 -rw-r--r--  1 root root   570 Jan 31  2010 .bashrc
 drwxr-xr-x  2 root root 4096 Jul  5 18:55 Desktop
```

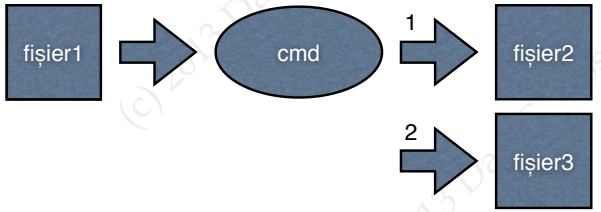
Comenzi, parametri, intrări și ieșiri

- Valoare returnată
  → orice comandă, program returnează către sistem, la terminare, o valoare întreagă
  → C: `exit(valoare);` sau `return valoare;` în *main()*
- Convenție:
  → 0: programul s-a terminat corect
  → ≠0: programul s-a terminat cu eroare (iar valoarea e codul erorii)
- În linia de comandă UNIX:

```
$ test 1 -eq 2
$ echo $?
1
```


Redirectarea intrărilor și ieșirilor

- Redirectare
  → *cmd < fișier1*
  → *cmd > fișier2*
  → *cmd >> fișier2*
  → *cmd 2> fișier3*



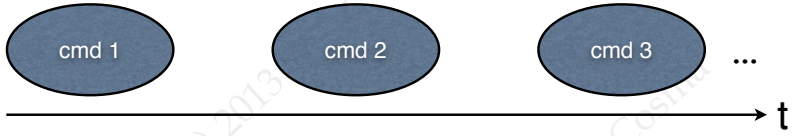
Înlănțuirea comenzilor

- Comenzile UNIX sunt interconectabile
  → intrarea și ieșirile: în mod normal text
  → ieșirea unui program poate deveni intrare pentru altul
- Înlănțuire
  → se folosește operatorul "pipe": |
  → *cmd1* | *cmd2* | *cmd3 ...*
  → comenzile sunt lansate în paralel

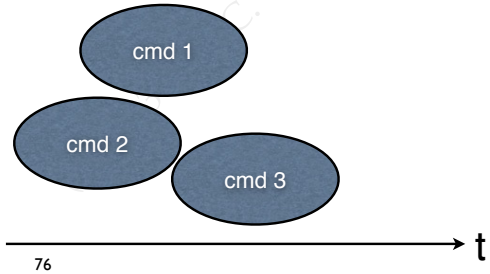

```
ls -al . | grep ".gmail-"
```


Înlănțuirea comenzilor

- Execuție secvențială
  → `cmd1 ; cmd2 ; cmd3` ▹*Valoare returnată: valoarea întoarsă de ultima comandă*
  → `cmd1 || cmd2 || cmd3` ▹*Valoare returnată: SAU logic*
  → `cmd1 && cmd2 && cmd3` ▹*Valoare returnată: ȘI logic*



- Execuție în paralel
  → *cmd1 & cmd2 & cmd3 &*



Exemple de comenzi UNIX

```
man [opţiuni] [secţiune] comandă
pwd
cd director
ls [-adgilrst] fişier ...
mv fişier1 fişier2
cp fişier1 fişier2
sort
du
df
who
ps
```


Exemple de redirectări și înlănțuiri

```
sort < fis1 > fis2

ls -l | grep student | wc -l > fis3

ls -a | grep ".qmail-" | grep -v ".qmail-ac:" | grep ":" | less

tar czf - /etc/ /var/named/ /service /var/lib/qmail /var/qmail
| ssh subspace.cs.upt.ro "dd of=/disk2/bf2-bak-20130710.tar.gz"

/opt/bin/mediaclient --cat /dev/dvb/adapter0/dvr0
| /Applications/VLC.app/Contents/MacOS/VLC file:///dev/stdin

JARS=`find ${ANT_LIB} -name '*jar' | while read JAR_FILE; do
    echo -n ":$JAR_FILE"; done`
```


Fișiere de comenzi UNIX

- *shell script* - un program scris folosind sintaxa si semantica recunoscute de interpretorul de comenzi, care folosește comenzi ale sistemului de operare pentru a efectua diverse operații

interpretorul de comenzi

bash

bash

→ “Bourne-again Shell” (joc de cuvinte inspirat de denumirea Bourne Shell)
→ înlocuitorul modern și gratuit al unuia din interpretoarele tradiționale din UNIX (Bourne Shell -- *sh*)
→ facilități diverse de procesare a comenzilor și programare: structuri de control, wildcarding, pipe, substituție de comenzi, bucle de iterare, evaluări de condiții, istoric al comenzilor, autocompletare în linia de comandă etc.
→ sintaxa e un superset al celei din *sh*, aduce îmbunătățiri, extensii
→ cvasiprezent în toate versiunile/distribuțiile de UNIX moderne

bash

→ “Bourne-again Shell” (joc de cuvinte inspirat de denumirea Bourne Shell)
→ înlocuitorul modern și gratuit al unuia din interpretoarele tradiționale din UNIX (Bourne Shell -- *sh*)
→ facilități diverse de procesare a comenzilor și programare: structuri de control, wildcarding, pipe, substituție de comenzi, bucle de iterare, evaluări de condiții, istoric al comenzilor, autocompletare in linia de comandă etc.
→ sintaxa e un superset al celei din *sh*, aduce îmbunătățiri, extensii
→ cvasiprezent în toate versiunile/distribuțiile de UNIX moderne

alte interpretoare: csh, ksh, tcsh, ...

Variabile de mediu

→ sunt variabile utilizabile în linie de comandă și fișiere de comenzi
→ tipul lor este întotdeauna șir de caractere

• Atribuire
VARIABILA=*valoare*

• Preluarea valorii
$VARIABILA

```
$ VAR1=abcd
$ echo $VAR1
abcd
```


• Concatenare

```
$ VAR1=abcd
$ VAR2=x
$ echo $VAR1$VAR2
abcdx
$ echo 123$VAR1
123abcd
$ echo ${VAR1}123
abcd123
$ VAR3=y${VAR1}123; echo $VAR3
yabcd123
```

• Citare

```
$ VAR1="ab cd"
$ VAR2=x
$ echo $VAR1
ab cd
$ echo "123 $VAR2"
123 x
$ echo '123 $VAR2'
123 $VAR2

$ echo \$VAR1
$VAR1
$ echo \$$VAR1
$ab cd
```

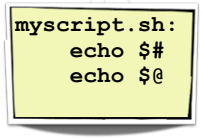

Variabile de mediu predefinite

PWD – directorul curent
HOME – directorul gazdă al utilizatorului curent
PATH – căile de căutare a comenzilor
exemplu: /bin:/usr/bin:usr/local/bin
PS1 – șirul prompter principal (afișat înaintea liniei de comandă)
PS2 – șirul prompter secundar
UID – identificatorul utilizatorului curent
HOSTNAME – numele calculatorului curent
...

```
mycomputer:~ janedoe$ echo $PS1
\h:\W \u\$
mycomputer:~ janedoe$ echo $PS2
>
mycomputer:~ janedoe$ echo #HOME
/home/janedoe
```


Variabile speciale

***, @** – parametrii din linia de comandă
– numărul de parametri din linia de comandă
? – starea ultimei comenzi efectuate
0 – numele fișierului de comenzi curent sau al interpretorului curent
1, 2, ... – parametrul n din linia de comandă
...




```
$ sh myscript.sh 1a 2 b 3
4
1a 2 b 3
$ sh myscript.sh 1a "2 b" 3
3
1a 2 b 3
```


Expandarea numelor de cale (de fișiere)

→ după împărțirea liniei în cuvinte, bash caută în fiecare cuvânt caracterele *, ? și [.
→ dacă le găsește, consideră că în acele puncte e specificat un tipar care va fi înlocuit cu o listă de nume de fișiere care îi corespunde

* - orice șir de caractere, inclusiv șirul vid
? - un singur caracter
[...] - oricare din caracterele specificate între parantezele drepte

```
$ ls *txt
a.txt abctxt txt
$ ls abc?1
abcx1 abcd1 abc11
$ ls ab[12]x
ab1x ab2x
```

Expandarea folosind acolade

→ seamănă cu expandarea căilor, dar numele rezultate nu trebuie neapărat să existe ca fișiere
→ sintaxă:
prefix{expresie}postfix
→ expresia poate fi formată din cuvinte separate prin virgulă sau din specificatori de interval (..)

```
$ echo a{x,y)
ax ay
$ echo a{x..z}
ax ay az
$ mkdir a{1,2,3}x
$  (va crea directoarele a1x, a2x, a3x)
```


Expandarea caracterului “~”

→ *~nume* e interpretat ca directorul gazdă al utilizatorului *nume*
→ ~ e interpretat ca directorul gazdă al utilizatorului curent

```
$ echo ~
/home/janedoe
$ echo ~gregory
/home/gregory
```


Substituția comenzilor

→ în orice construcție sintactică de forma `` `șir` `` sau **$(*șir*)** , șirul e interpretat ca fiind o comandă
→ comanda e executată, iar textul pe care ea îl *generează la ieșirea standard* va fi preluat și va înlocui întreaga construcție sintactică
→ atenție: ` este “accent grav”, nu “apostrof” (de obicei pe tasta de lângă “1”)

```
$ echo `pwd`
/home/janedoe/a1x
$ A=`ls -a`
$ echo $A
. .. abc myscript.sh xyz
```


Redirectarea ieșirii și intrării

→ formatul general pentru redirectarea ieșirii:
`[n]>cuvânt`
`[n]>>cuvânt`
→ redirectează descriptorul de fișier **n** spre fișierul **cuvânt**; dacă n lipsește, redirectarea se face pentru ieșirea standard
→ dacă e folosit **>>**, ieșirea va fi adăugată la sfârșitul fișierului (nu va fi suprascris)

→ formatul general pentru redirectarea intrării:
`[n]<cuvânt`
→ redirectează descriptorul de fișier **n** spre fișierul **cuvânt**; dacă n lipsește, redirectarea se face pentru ieșirea standard

```
$ cat x 1>>fisier
```

“Here Documents”

```
<<word
    here-document
delimiter
```

→ interpretorul va considera **word** ca fiind un delimitator
→ textul dat de liniile here-document e preluat și trimis comenzii la intrare
→ delimitatorul arată sfârșitul intrării
→ dacă în **word** apar ghilimele simple sau duble, ele sunt ignorate, dar în interiorul here-document nu se va face expandare de variabile sau substituție de comenzi

```
cat <<SFARSIT
 abc
 123
 x
 SFARSIT
abc
123
x
```


Alte facilități bash

→ aliasuri (comenzile interne alias, noalias)
→ funcții
→ variabile tablou
→ autocompletarea comenzilor (tasta TAB)
→ istoricul comenzilor (fișierul .bash_history)
→ pornirea automată de comenzi la login, la logout și la pornirea interpretorului (.bash_profile, .bash_logout, .bashrc)
...

Expresii condiționale

→ comenzile interne **test** și **[**
→ folosite pentru a compune expresii logice
→ valoarea returnată de comanda condițională e 0 (true) sau 1 (false), astfel încât poate fi ușor integrată în specificatorii de condiție (de exemplu la if)

• exemple de parametri pentru **test** și **[**:
-e fișier - true dacă fișierul există
-d fișier - true dacă fișierul există și este un director
-f fișier - true dacă fișierul există și este un fișier obișnuit
șir1 == șir2 - true dacă șir1 e identic cu șir2
șir1 != șir2 - true dacă șir1 e diferit de șir2
șir1 < șir2 - true dacă șir1 e, alfabetic, înainte de șir2
șir1 > șir2 - true dacă șir1 e, alfabetic, după șir2
arg1 operator arg2 - true dacă arg1 și arg 2 sunt în relația dată de operator
⇒ în care, operator: -eq (egal), -ne (diferit), -lt (mai mic), -gt (mai mare), -ge (mai mare sau egal), -le (mai mic sau egal)

```
$ [ 1 -eq 2 ]
$ echo $?
1
```


Structuri de control

```
for name [ in word ] ; do list ; done
```

```
for (( expr1 ; expr2 ; expr3 )) ; do list ; done
```

→ expr*i* - expresii aritmetice
→ se evaluează **expr1**; **expr2** se evaluează la fiecare iterație până ajunge la zero; dacă **expr2**≠0 se execută **list** și se evaluează **expr3**

```
case word in [ [(] pattern [ | pattern ] ... )
list ;; ] ... esac
```

```
if list; then list; [ elif list; then list; ] ...
[ else list; ] fi
```

```
while list; do list; done
```

```
until list; do list; done
```

Exemple

```
for (( i=1; i<=4; i++ ))
do
   echo $i
done
```

```
$ sh script
1
2
3
4
```

```
for i
do
   echo $i
done
```

```
$ sh script a b c
a
b
c
```

```
i=0
while [ -f .dotask ]
do
   (( i++ ))
   echo Starting task: $i
   /usr/local/bin/myprogram --start
done
```

```
$ sh script
Starting task: 1
Starting task: 2
Starting task: 3
Starting task: 4
Starting task: 5
...
```


Exemple

```
case "$1" in
  start)
      /usr/local/myservice -d
      ;;
  stop)
      /usr/local/myservice -x
   ;;
  status)
      if /usr/local/myservice --isrunning 2>/dev/null
     then
          echo Service is running
     else
          echo Service not started
     fi
      ;;
  restart)
      stop
      start
      ;;
  *)
      echo $"Usage: $0 {start|stop|restart}"
      exit 1
esac
```

```
$ sh service.sh status
Service is running
$ sh service.sh reboot
Usage: service.sh {start|
stop|restart}
```


Numele interpretorului de comenzi care va rula acest script

Primul parametru al funcției

Funcție recursivă

```
#!/bin/sh

fact()
{
    if [ $1 -gt 1 ]
    then
        i=`expr $1 - 1`
        j=`fact $i`
        k=`expr $1 \* $j`
        echo $k
    else
        echo 1
    fi
}

read -p "Numar:" x
fact $x
```


Exemple

```
for i in episode-S02E*avi
do
  nr=`ls "$i" | cut -c 13-14`
  fn=${i%\.*}
  mv episode-subtitle-en-2x${nr}*srt "$fn".srt
  echo $nr $fn
done
```

Expresii regulate în UNIX

Expresie regulată

O expresie formată dintr-o secvență de caractere care descrie un tipar de căutare, folosit pentru a desemna reguli de potrivire pentru porțiunile de text căutate.

UNIX

Multe comenzi recunosc expresii regulate, de obicei în următoarele forme:

→ POSIX basic (BRE)
→ POSIX extended (ERE)

Expresii regulate POSIX

. - se potrivește cu un singur caracter
Exemplu: a.x se potrivește cu abx, aax, acx etc.

[ ] - se potrivește cu un singur caracter din cele dintre paranteze
Exemple: [abc] se potrivește cu a, b sau c
[a..x] - orice caracter între a și x

[^] - se potrivește cu un singur caracter în afara celor dintre paranteze
Exemple: [^abc] se potrivește cu orice caracter, mai puțin a, b sau c
[^a..x] - orice caracter, mai puțin cele dintre a și x

^ - se potrivește cu poziția de început (de obicei în linie)
Exemplu: ^a înseamnă “caracterul a la început”

$ - se potrivește cu poziția de sfârșit (de obicei în linie)
Exemplu: a$ înseamnă “caracterul a la sfârșit”

Expresii regulate POSIX

( ) - definește o subexpresie
Valoarea care se potrivește tiparului dintre paranteze poate fi preluată mai târziu folosind operatorul \n . În modul “basic” (BRE), parantezele trebuie citate: \(și \) .

***n* - referință la subexpresie**
Se potrivește (preia, referă) a *n*-a subexpresie desemnată între paranteze, unde $n \in [1, 9]$.

*****
Descrie o expresie formată din caracterul care o precede, de zero sau mai multe ori.
Exemple:
ab*x se potrivește pe ax, abx, abbbbx etc.
[abc]* se potrivește pe șirul vid, a, aa, aaaa, b, bbb, ab, ba, abcc, abc, aabbcc, aabbcca etc.

{*m*,*n*}
Descrie o expresie formată din caracterul care o precede, de minim *m* și maxim *n* ori. În modul “basic” (BRE), acoladele trebuie citate: \{ și \} .

Expresii regulate POSIX

+
Doar în modul extins (ERE). Descrie o expresie formată din caracterul care o precede, de una sau mai multe ori.

?
Doar în modul extins (ERE). Descrie o expresie formată din caracterul care o precede, de zero ori sau o dată.

Notă: în modul extins nu sunt valabile referirile către subexpresii (***n***), iar o citare cu **** va introduce caracterul următor, așa cum e **(\(** înseamnă chiar caracterul **(**)

Exemple

```
[cm]asa             casa masa

^c?are              care are (strict la început)

.are                sare mare tare xare fare ...

[ab]*re             re are aabre abbre aaarea aabbbre baare ...

a+re                are aare aaare ... (dar nu “re”)

episode\ [123]x.*   episode 1x01 - The Super Hero
                    episode 1x02 - The Hero Cries
                    episode 3x22 - Hero No More
                    ...
```


Comenzi care recunosc expresii regulate

grep, sed, awk

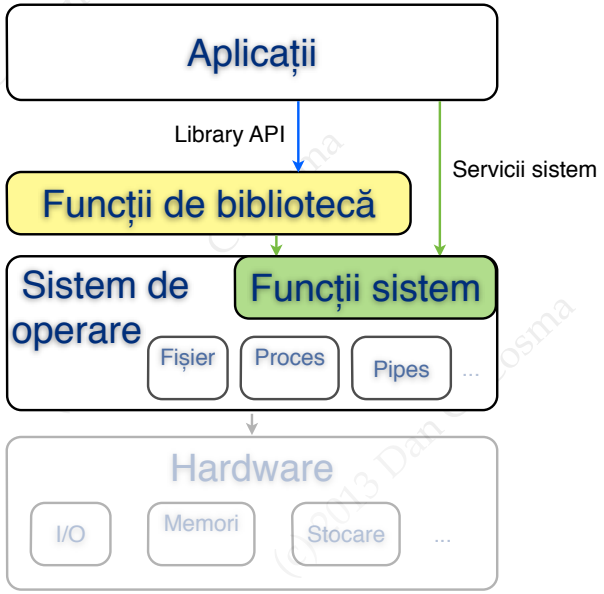
Notă: de obicei, pentru a recunoaște modul extins (ERE), comenzii trebuie sa i se dea un parametru special (de regulă -E)

```
grep -E tipar
egrep tipar

      ls -l | grep -E ^Fisierul\ meu.*txt$

      Fisierul meu cu scrisori.txt
      Fisierul meu preferat.txt
      Fisierul meu cu txt
      ...
```

3. Sisteme de fișiere


```
  if((fd1=open(argv[1], O_RDONLY))<0)
    {
      printf("Error opening input file\n");
      exit(2);
    }
  if((fd2=open(argv[2], O_WRONLY | O_CREAT |
O_EXCL, S_IRWXU)) < 0)
    {
      printf("Error creating destination file
\n");
      exit(3);
    }

  while((n = read(fd1, &c, sizeof(char))) > 0)
  {
      if(write(fd2, &c, n) < 0)
      {
       printf("Error writing to file\n");
       exit(4);
      }
  }

  if(n < 0)
  {
      printf("Error reading from file\n");
      exit(5);
  }

  close(fd1);
```


Programarea cu fișiere



Lecția de engleză

Lecția de engleză. Și de română

Lecția de engleză. Și de română

Library - *noun,*
: a place in which literary, musical, artistic, or reference materials (as books, manuscripts, recordings, or films) are kept for use but not for sale
: a collection resembling or suggesting a library

The Free Merriam-Webster dictionary, www.m-w.com

Librărie - *substantiv,*
: Magazin în care se vând cărți.

Dicționarul explicativ al limbii române, ediția 1998

Lecția de engleză. Și de română

Library - *noun,*
: a place in which literary, musical, artistic, or reference materials (as books, manuscripts, recordings, or films) are kept for use but not for sale
: a collection resembling or suggesting a library

The Free Merriam-Webster dictionary, www.m-w.com

~~***Librărie*** - *substantiv,*~~
~~: Magazin în care se vând cărți.~~
Bibliotecă - *substantiv,*
: Instituție care colecționează cărți, periodice etc. spre a le pune în mod organizat la dispoziția cititorilor
: Colecție de cărți, periodice, foi volante, imprimate etc.

Dicționarul explicativ al limbii române, ediția 1998

Lecția de engleză. Și de română

Library - *noun,*
: a place in which literary, musical, artistic, or reference materials (as books, manuscripts, recordings, or films) are kept for use but not for sale
: a collection resembling or suggesting a library

The Free Merriam-Webster dictionary, www.m-w.com

Bibliotecă - *substantiv,*
: Instituție care colecționează cărți, periodice etc. spre a le pune în mod organizat la dispoziția cititorilor
: Colecție de cărți, periodice, foi volante, imprimate etc.

Dicționarul explicativ al limbii române, ediția 1998

Lecția de engleză. Și de română

Bookstore - *noun,*
: a place of business where books are the main item offered for sale —called also bookshop

The Free Merriam-Webster dictionary, www.m-w.com

Librărie - *substantiv,*
: Magazin în care se vând cărți.

Dicționarul explicativ al limbii române, ediția 1998

Library ≠ Librărie

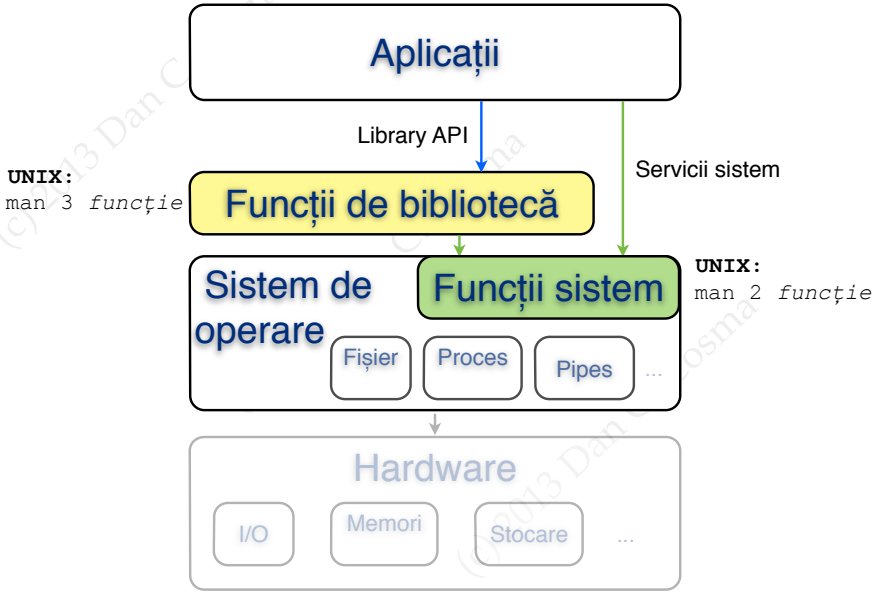
Library = Bibliotecă

Library ≠ Librărie

"False Friend"

Library = Bibliotecă

Concluzie: nu vă ajutați singuri să deveniți ridicoli... ;)



Funcții sistem pentru lucrul cu fișiere

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

- **pathname** - numele fișierului
- **oflag** - opțiunile de deschidere a fisierului. Este un șir de biți. Există constante în fcntl.h care se pot combina folosind operatorul '|' , de exemplu::
  - O_RDONLY - deschidere numai pentru citire
  - O_WRONLY - deschidere numai pentru scriere
  - O_RDWR - deschidere pentru citire si scriere
  - O_APPEND - deschidere pentru adaugare la sfarsitul fisierului
  - O_CREAT - crearea fisierului, daca el nu exista deja; daca e folosita cu aceasta optiune, functia **open** trebuie sa primeasca si parametrul *mode*.
  - O_EXCL - creare "exclusiva" a fisierului: daca s-a folosit O_CREAT si fisierul exista deja, functia **open** va returna eroare
  - O_TRUNC - daca fisierul exista, continutul lui este sters
- **mode** - doar la creare - drepturile de acces asociate fisierului. Constante:
  - S_IRUSR - drept de citire pentru proprietarul fisierului (*user*)
  - S_IWUSR - drept de scriere pentru proprietarul fisierului (*user*)
  - S_IXUSR - drept de executie pentru proprietarul fisierului (*user*)
  - S_IRGRP - drept de citire pentru grupul proprietar al fisierului
  - S_IWGRP - drept de scriere pentru grupul proprietar al fisierului
  - S_IXGRP - drept de executie pentru grupul proprietar al fisierului
  - S_IROTH - drept de citire pentru ceilalti utilizatori
  - S_IWOTH - drept de scriere pentru ceilalti utilizatori
  - S_IROTH - drept de executie pentru ceilalti utilizatori

Returnează un descriptor de fișier

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

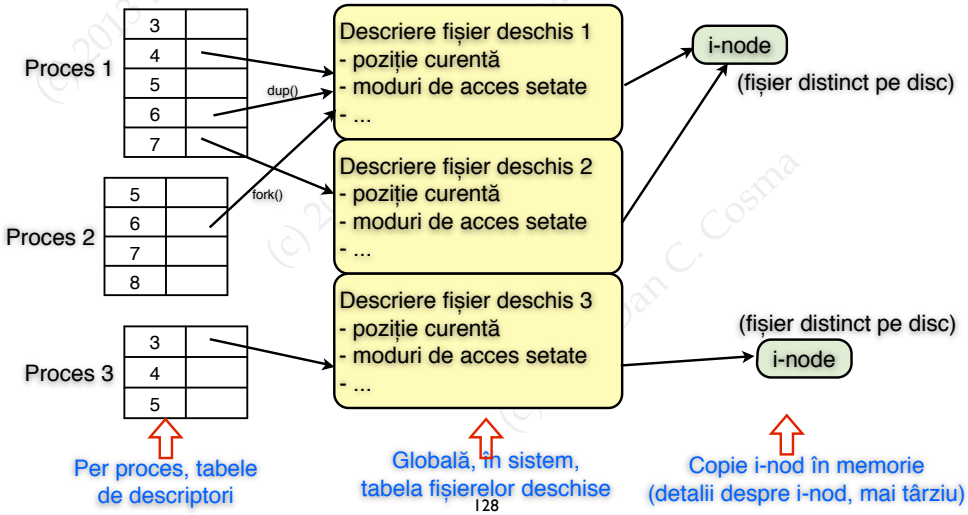

Funcții sistem pentru lucrul cu fișiere

Descriptori de fișier standard:

```
STDIN_FILENO
STDOUT_FILENO
STDERR_FILENO
```


Gestiunea fișierelor deschise

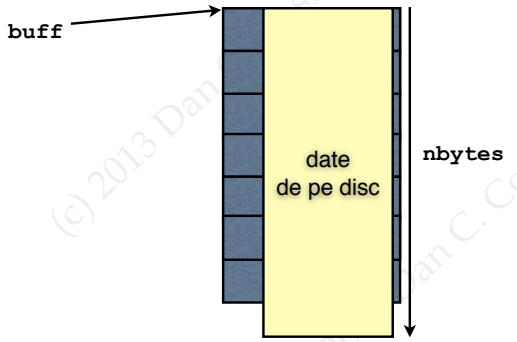
→ descriptorii de fișier în sistem - numere naturale 1..n, n depinde de sistem
→ un descriptor inițializat desemnează un fișier deschis; după închidere, numărul (descriptorul) poate fi refolosit
→ fiecare proces deține câte o tabelă care conține descriptorii deschiși



→ un apel open() creează
- un nou descriptor
- o nouă descriere de fișier deschis


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

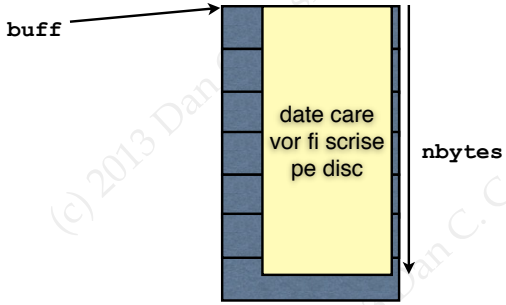
→citește un număr de exact *nbytes* octeți de la poziția curentă
→îi pune în zona de memorie *indicată* de *buff*
→returnează numărul de octeți citiți de fapt (0 la sfârșit) sau -1 la eroare



Ce se întâmplă dacă buffer-ul nu e alocat corect ?


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

→scrie pe disc primii exact *nbytes* octeți din buffer
→îi pune în zona de memorie *indicată* de *buff*



Ce se întâmplă dacă buffer-ul nu e alocat corect ?


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

Poziționează indicatorul la deplasamentul *offset* in fișier, astfel:
- dacă pos = SEEK_SET, poziționarea se face relativ la începutul fișierului
- dacă pos = SEEK_CUR, poziționarea se face relativ la poziția curentă
- dacă pos = SEEK_END, poziționarea se face relativ la sfârșitul fisierului

```
int mkdir(const char *pathname, mode_t mode)

int rmdir(const char *pathname)
```

Funcții de bibliotecă pentru lucrul cu fișiere

```
FILE *fopen(const char *filename, const char *mode);

int fclose(FILE *stream);

int fprintf(FILE *stream, const char *format, ...);

int fscanf(FILE *stream, const char *format, ...);

size_t fread(void *ptr, size_t size, size_t nmemb, FILE
*stream);
```

citește din fișierul indicat de *stream* un număr de *nmemb* elemente, fiecare de dimensiunea *size*, și le pune în zona de memorie indicată de *ptr*.

```
size_t fwrite( void *ptr, size_t size, size_t nmemb,  FILE
*stream);
```

scrie în fișierul indicat de *stream* un număr de *nmemb* elemente, fiecare de dimensiunea *size*, pe care le ia din zona de memorie indicată de *ptr*

Funcții de bibliotecă pentru lucrul cu fișiere

Descriptori de fișier standard:

```
stdin
stdout
stderr
```


Aflarea informațiilor despre fișiere (apeluri sistem)

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

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

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

```
struct stat
  {
      dev_t         st_dev;      /* device */
      ino_t         st_ino;      /* inode */
      umode_t       st_mode;     /* protection */
      nlink_t       st_nlink;    /* number of hard links */
      uid_t         st_uid;      /* user ID of owner */
      gid_t         st_gid;      /* group ID of owner */
      dev_t         st_rdev;     /* device type (if inode device) */
      off_t         st_size;     /* total size, in bytes */
      unsigned long st_blksize;  /* blocksize for filesystem I/O */
      unsigned long st_blocks;   /* number of blocks allocated */
      time_t        st_atime;    /* time of last access */
      time_t        st_mtime;    /* time of last modification */
      time_t        st_ctime;    /* time of last change */
  };
```


Funcții de bibliotecă pentru lucrul cu directoare

```
DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
void rewinddir(DIR *dir);
int closedir(DIR *dir);
```

```
/*Linux*/
struct dirent {
        ino_t          d_ino;       /* inode number */
        off_t          d_off;       /* offset to the next dirent */
        unsigned short d_reclen;    /* length of this record */
        unsigned char  d_type;      /* type of file; not supported
                                       by all file system types */
        char           d_name[256]; /* filename */
};
```

- **int link(const char *oldpath, const char *newpath);** - creeaza legaturi fixe spre fisiere
- **int symlink(const char *oldpath, const char *newpath);** - creeaza legaturi simbolice spre fisiere sau directoare
- **int unlink(const char *pathname);** - sterge o intrare in director (legatura, fisier sau director)
- **int rename(const char *oldpath, const char *newpath);** - redenumire / mutare de fisiere
- **int rmdir(const char *pathname);** - stergere de directoare
- **int chdir(const char *path);** - schimbarea directorului curent
- **char *getcwd(char *buf, size_t size);** - determinarea directorului curent

Un exemplu

→ program care copiază un fișier cu numele dat ca argument în linia de comandă într-o destinație (fișier) specificată de asemenea ca argument
→ sunt afișate mesaje de eroare când e nevoie


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

void usage(char *name)
{
  printf("Usage: %s <source> <destination>\n", name);
}

int main(int argc, char *argv[])
{
  int fd1, fd2;
  int n;
  char c;

  /*** Verificarea argumentelor din linia de comanda */
  if(argc!=3)
    {
      usage(argv[0]);
      exit(1);
    }
```



```c
  /*** Deschiderea fisierelor */
  if((fd1=open(argv[1], O_RDONLY))<0)
    {
      printf("Error opening input file\n");
      exit(2);
    }
  if((fd2=open(argv[2], O_WRONLY | O_CREAT | O_EXCL, S_IRWXU)) < 0)
    {
      printf("Error creating destination file\n");
      exit(3);
    }

  while((n = read(fd1, &c, sizeof(char))) > 0)
  {
      if(write(fd2, &c, n) < 0)
      {
       printf("Error writing to file\n");
       exit(4);
      }
  }

  if(n < 0)
  {
      printf("Error reading from file\n");
      exit(5);
  }

  close(fd1);
  close(fd2);

  return 0;
}
```

```
  /*** Deschiderea fisierelor */
  if((fd1=open(argv[1], O_RDONLY))<0)
    {
      printf("Error opening input file\n");
      exit(2);
    }
  if((fd2=open(argv[2], O_WRONLY | O_CREAT | O_EXCL, S_IRWXU)) < 0)
    {
      printf("Error creating destination file\n");
      exit(3);
    }

  while((n = read(fd1, &c, sizeof(char))) > 0)
  {
      if(write(fd2, &c, n) < 0)
       {
        printf("Error writing to file\n");
        exit(4);
       }
  }

  if(n < 0)
  {
      printf("Error reading from file\n");
      exit(5);
  }

  close(fd1);
  close(fd2);

  return 0;
}
```


O mică modificare în program...


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

  #define BUFSIZE 4096

  void usage(char *name)
  {
    printf("Usage: %s <source> <destination>\n", name);
  }

  int main(int argc, char *argv[])
  {
    int fd1, fd2;
    int n;
    char buf[BUFSIZE];

    /*** Verificarea argumentelor din linia de comanda */
    if(argc!=3)
      {
        usage(argv[0]);
        exit(1);
      }
```



```
  /*** Deschiderea fisierelor */
  if((fd1=open(argv[1], O_RDONLY))<0)
    {
      printf("Error opening input file\n");
      exit(2);
    }
  if((fd2=open(argv[2], O_WRONLY | O_CREAT | O_EXCL, S_IRWXU)) < 0)
    {
      printf("Error creating destination file\n");
      exit(3);
    }

  while((n = read(fd1, buf, BUFSIZE)) > 0)
  {
      if(write(fd2, buf, n) < 0)
       {
        printf("Error writing to file\n");
        exit(4);
       }
  }

  if(n < 0)
  {
      printf("Error reading from file\n");
      exit(5);
  }

  close(fd1);
  close(fd2);

  return 0;
}
```

```
date; ./copyfile2 beethoven-symph-5-1.wav b.wav; date
```

Dimensiunea fișierului	Dimensiunea buffer-ului (octeți)	Timp de copiere
74 MB	1	6 minute 30 secunde
74 MB	100	3 secunde
74 MB	4096	1 secundă

Notă: în plus, chiar și simpla apelare a unei funcții sistem costă timp. Nu abuzați.

Mici chestiuni de C și de... bun simț

```
read(fd1, buf, BUFSIZE);

read(fd1, buf, sizeof(buf));
read(fd1, buf, strlen(buf));


int v;
...
read(fd1, &v, sizeof(int));
```

```
char buf[BUFSIZE];

char *buf;
```


```
...

/*citesc fișierul de pe disc*/
read(fd1, buf, dimensiune_fisier);

...
```

Sisteme de fișiere

Organizarea spațiului pe disc

→ Un disc este format, la nivel logic, dintr-o succesiune de sectoare
→ Dimensiunea sectorului e fixă și depinde de tipul discului. Exemplu: hard disc "obișnuit" 512 bytes, hard disc nou: 4096 bytes
→ Discul poate fi împărțit în partiții

Exemplu de disc partiționat

Scheme de partiționare

- Diferă în funcție de tipul discului, al calculatorului, al sistemelor de operare instalate etc.
- Cele mai populare scheme de partiționare:
  - MBR (Master Boot Record)
    → este schema clasică de partiționare, întâlnită pe, practic, toate PC-urile în uz
  - GPT (GUID Partition Table)
    → noua schemă de partiționare pentru PC-uri, mai flexibilă

MBR

Boot sector

→ o regiune de pe un disc, de obicei la începutul acestuia, care conține, printre altele, un cod executabil pe care firmware-ul unui sistem de calcul îl poate încărca și lansa în execuție, ca punct de start al procedurii de inițializare a sistemului respectiv
→ codul executabil va încărca, la rândul lui un alt program de pe acel disc, de obicei o componentă de inițializare și încărcare a sistemului de operare

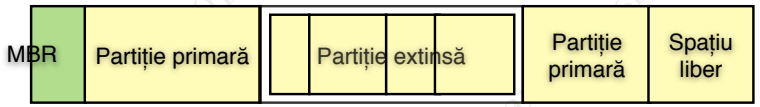
Master Boot Record

→ un tip special de sector de boot, specific calculatoarelor compatibile IBM PC (cu o evoluție până în zilele noastre)
→ conține, pe lângă programul încărcător, informații despre partițiile de pe disc

MBR

Schema de partiționare

→ Informațiile despre structura partițiilor se găsesc într-o tabelă de partiții, aflată în MBR
→ Tabela de partiții are maxim 4 intrări, deci permite definirea a maxim 4 partiții, numite partiții primare
→ O partiție din cele 4 poate fi desemnată ca fiind partiție extinsă, caz în care scopul acesteia e să conțină alte partiții în interiorul ei

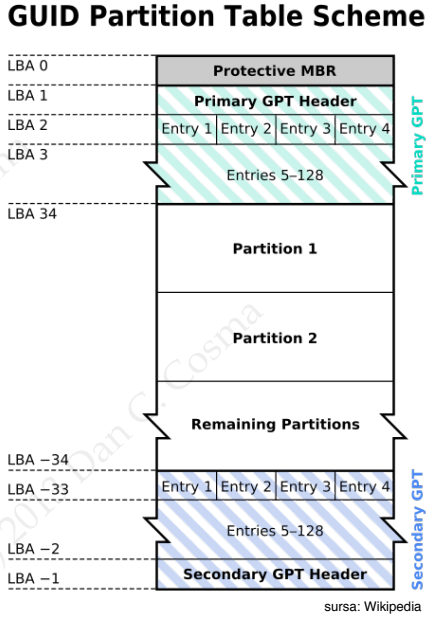


Exemplu de disc partiționat MBR

GPT

Descriere

→ Face parte din standardul UEFI (United Extensible Firmware Interface), propus pentru înlocuirea PC BIOS
→ Utilizează GUIDs (Globally unique identifier) pentru identificarea discului și a tipurilor partițiilor, pentru a evita duplicate
→ Permite crearea unui număr arbitrar de partiții (dependent doar de dimensiunea spațiului alocat pentru tabela de partiții)



sursa: Wikipedia

Sistem de fișiere

Modul de organizare logică a datelor pe un suport fizic sau virtual, în vederea stocării, citirii și regăsirii acestora

→ Pe un calculator pot fi instalate mai multe sisteme de fișiere, de exemplu pe discuri sau pe partiții diferite
→ Sistemele de operare definesc și recunosc tipuri de sisteme de fișiere care le sunt specifice
→ Sistemele de operare pot recunoaște mai multe sisteme de fișiere, chiar și dacă nu le sunt caracteristice
→ Un sistem de fișiere definește atât structurile de date care stochează informații, cât și modul logic de organizare a acestora (de exemplu ca ierarhie de fișiere și directoare)

Exemple de sisteme de fișiere

FAT16 (File Allocation Table, 16 bit)
→ Specific MS-DOS, Windows. Fișierele au nume de maxim 8+3 caractere. Dimensiunea maximă a volumului (partiției): 2 GB / 4GB (Win NT)

FAT32
→ Windows. Nume de fișiere mai lungi. Dimensiunea maximă a volumului: 8 GB. Dimensiunea maximă a unui fișier: 4 GB.

NTFS
→ Windows NT și succesorii. Dimensiune maximă a fișierului: 16 TB (<=Win7), 256 TB (Win8). Sistem de fișiere cu jurnalizare.

HFS Plus
→ Specific OS X. Dimensiune maximă a volumului: 8 EB*. Dimensiune maximă a fișierului: 8 EB*. Sistem cu jurnalizare.

** 1 exabyte = 10^{18} bytes = 10^{9} gigabytes*

Exemple de sisteme de fișiere

Ext2 - Second Extended File System
→ Specific Linux. Dimensiune maximă a volumului: 2-32 TB. Dimensiune maximă a fișierului: 16 Gb - 2 TB

Ext3 - Third Extended File System
→ Specific Linux. Dimensiune maximă a volumului: 2-32 TB. Dimensiune maximă a fișierului: 16 Gb - 2 TB. Este un sistem de fișiere cu jurnalizare.

Ext4 - Fourth Extended File System
→ Specific Linux. Dimensiune maximă a volumului: 1 EB*. Dimensiune maximă a fișierului: 16 TB. Este un sistem de fișiere cu jurnalizare.

** 1 exabyte = 10^{18} bytes = 10^{9} gigabytes*

Sisteme de fișiere suportate de sistemele de operare

Windows
→ FAT, NTFS, exFAT

Linux
→ Zeci de sisteme de fișiere. Exemple: ext2, ext3, ext4, XFS, FAT, NTFS, HFS+, JFFS, JFFS2 (Journaling Flash File System)

OS X
→ HFS+, UFS, FAT, NTFS (read only)

→ Pe un calculator pot fi instalate mai multe sisteme de fișiere, de exemplu pe discuri sau pe partiții diferite

```
# fdisk /dev/sda

The number of cylinders for this disk is set to 30401.
There is nothing wrong with that, but this is larger than 1024,
and could in certain setups cause problems with:
1) software that runs at boot time (e.g., old versions of LILO)
2) booting and partitioning software from other OSs
   (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help): p

Disk /dev/sda: 250.0 GB, 250059350016 bytes
255 heads, 63 sectors/track, 30401 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Disk identifier: 0xd42ad42a

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1        4476    35953438+   7  HPFS/NTFS
/dev/sda2           29095       30400    10490445    7  HPFS/NTFS
/dev/sda3            4477       29094   197744085    5  Extended
/dev/sda5            4477        6428    15679408+  83  Linux
/dev/sda6            6429        6695     2144646   82  Linux swap / Solaris
/dev/sda7            6696       29094   179919936   83  Linux
```


Organizarea ierarhică a unui sistem de fișiere

- Arbore / arbori de directoare și fișiere
  → UNIX: un singur arbore de directoare (o singură rădăcină: /)
  → DOS/Windows: mai mulți arbori (câte o rădăcină pentru fiecare disc/ partiție: A:\, B:\, C:\, D:\, ... Construcția \ se referă la rădăcina discului curent)
- Fișierele pot fi referite
  → **prin nume de căi absolute:**
  UNIX: /usr/bin/ls, /home/jane/myscript, /jome/jane/my\ files/file1
  Windows: C:\Windows\wordpad.exe, “D:\games\My Super Game”
  → **prin nume de căi relative la directorul curent:**
  UNIX: myscript, “my files/file1”, .ssh/known_hosts
  Windows: wordpad.exe, “My Super Game\startgame.exe”

Directoare “consacrate” în UNIX

/ - directorul rădăcină
/bin - comenzi esențiale, apelabile și când e montat doar sistemul de fișiere rădăcină
/dev - dispozitive
/etc - configurări sistem
/home - directoarele gazdă ale utilizatorilor
/lib - biblioteci sistem esențiale și module nucleu
/opt - directoare pentru aplicații suplimentare
/sbin - executabile sistem (exclusiv pt. administrare)
/tmp - fișiere și directoare temporare
/usr - baza unei ierarhii importante de organizare a fișierelor accesibile în sistem
/usr/X11 - sistemul de ferestre X11
/usr/X11R6 - sistemul de ferestre X11R6
/usr/bin - utilitare, comenzi apelabile de utilizatori
/usr/lib - biblioteci de programare
/usr/local - aplicații locale
/usr/local/bin - executabile locale
/usr/share - date independente de arhitectură
/var - fișiere de date variabile: poștă electronică, jurnale (log-uri), cache-uri etc.

Directoare “consacrate” în Windows (NT, ..., 8):

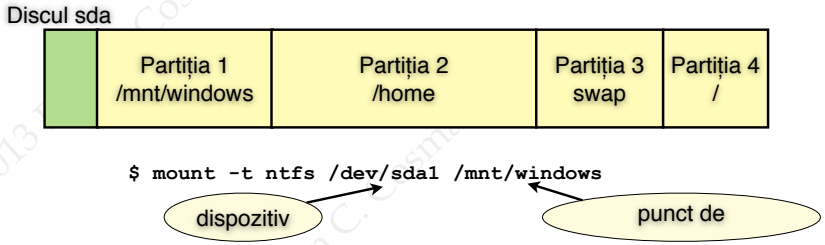
C:\, D: - directoare rădăcină
C:\Windows - fișierele sistemului de operare
C:\Windows\System - fișiere sistem și biblioteci sistem pe 16 biți
C:\Windows\System32 - fișiere sistem și biblioteci sistem pe 32 sau 64 biți
C:\Windows\SysWOW64 - fișiere sistem și biblioteci sistem pe 32 biți pentru aplicații pe 32 biti când sistemul e pe 64 biți (WOW = Windows on Windows)
C:\Documents and Settings - directoarele gazdă ale utilizatorilor (NT, 2000, XP)
C:\Users - directoarele gazdă ale utilizatorilor (Vista, 7, 8)
C:\Temp, C:\Windows\Temp - fișiere și directoare temporare

Montarea sistemelor de fișiere (UNIX)

- Pe același calculator pot coexista mai multe sisteme de fișiere
  → pe diferite discuri, partiții, dispozitive de stocare, în memorie etc.
- Există o singură rădăcină de arbore de directoare

➠ *Un sistem de fișiere poate fi montat în orice director din arborele de fișiere existent*
➠ *Primul sistem de fișiere montat e sistemul de fișiere rădăcină, montat automat la inițializare, în directorul /*
➠ *Sistemul de fișiere rădăcină trebuie să conțină toate fișierele și directoarele esențiale funcționării sistemului de operare*

Exemplu de montare și demontare




```
$ umount /dev/sda1
        sau
$ umount /mnt/windows
```

Fișierul de configurare /etc/fstab

→ specifică sisteme de fișiere cu puncte prestabilite de montare
→ dacă nu se specifică altfel, ele vor fi montate automat la inițializare

```
/dev/sda3    swap          swap    defaults              0 0
/dev/sda4    /             ext3    acl,user_xattr        1 1
/dev/sda2    /home         ext3    acl,user_xattr        1 2
/dev/sda1    /mnt/windows  ntfs    user,noauto           0 0
```


Utilizatori și drepturi de acces

- Utilizatori
  → pe același sistem UNIX pot să existe mai mulți utilizatori
  → fiecare utilizator are un nume și un identificator numeric (uid)
  → fiecare utilizator are un director gazdă, asupra căruia e proprietar
  → configurarea utilizatorilor se face în fișierele /etc/passwd și /etc/shadow
- Grupuri
  → utilizatorii sunt grupați în grupuri
  → un utilizator poate aparține mai multor grupuri
  → grupurile au fiecare câte un nume și un identificator (gid)
  → configurarea grupurilor se face în fișierul /etc/group

- Pentru fișiere sunt setate drepturi de acces pentru trei categorii de utilizatori
  → **proprietarul fișierului: user, u** -- utilizatorul care deține fișierul; de regulă e cel care l-a creat, dar poate fi și altul
  → **grupul proprietar al fișierului: group, g** -- fiecare fișier poate fi deținut de un grup; implicit e grupul proprietarului, dar poate fi schimbat
  → **ceilalți utilizatori: others, o**
- Există trei tipuri de drepturi pentru fișiere:
  → **citire: read, r** -- conținutul fișierului poate fi citit
  → **scriere: write, w** -- conținutul fișierului poate fi modificat
  → **execuție: execute, x** -- fișierul poate fi lansat în execuție; pentru directoare arată că se poate intra în directorul respectiv
- Combinația acestor elemente se concretizează în 9 drepturi, codificate în 9 biți de acces (*file mode bits*):

r w x r w x r w x
user group others

Exemple. Comenzile chmod, chown și chgrp

```
$ ls -l
total 4
-rw-r--r-- 1 danc users 0 2013-10-23 23:54 file1
-rw-r--r-- 1 danc users 5 2013-10-23 23:54 file2.txt
$ chmod a+x file1 ; ls -l
total 4
-rwxr-xr-x 1 danc users 0 2013-10-23 23:54 file1
-rw-r--r-- 1 danc users 5 2013-10-23 23:54 file2.txt
$ chmod g-rw file1 ; ls -l
total 4
-rwx--xr-x 1 danc users 0 2013-10-23 23:54 file1
-rw-r--r-- 1 danc users 5 2013-10-23 23:54 file2.txt
$ chmod 766 file2.txt ; ls -l
total 4
-rwx--xr-x 1 danc users 0 2013-10-23 23:54 file1
-rwxrw-rw- 1 danc users 5 2013-10-23 23:54 file2.txt
$ chown jane.users file1 ; ls -l
total 4
-rwx--xr-x 1 jane users 0 2013-10-23 23:54 file1
-rwxrw-rw- 1 danc users 5 2013-10-23 23:54 file2.txt
$ chgrp staff file2.txt ; ls -l
total 4
-rwx--xr-x 1 jane users 0 2013-10-23 23:54 file1
-rwxrw-rw- 1 danc staff 5 2013-10-23 23:54 file2.txt
```

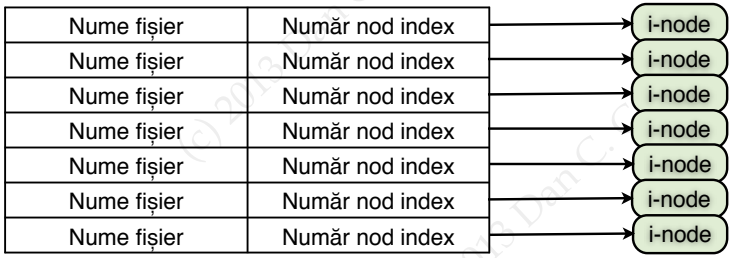

Structura unui sistem de fișiere UNIX

➠ O partiție UNIX poate conține:

- **Blocul de boot (încărcare)** - programele care realizează încărcarea sistemului de operare Unix.
- **Superblocul -** contine informații generale despre sistemul de fișiere de pe disc: începutul zonelor următoare, începutul zonelor libere de pe disc.
- **Zona de noduri index -** conține câte o intrare pentru fiecare *fișier* (*în sens larg*) din partiție
- Ultima zonă conține blocurile care memorează fișierele propriu-zise.

➠ Pe disc sunt memorate fișiere și directoare, într-o organizare ierarhică (de arbore)

➠ Directorul este un fișier special - un tabel în care fiecare intrare descrie un fișier conținut de directorul respectiv:

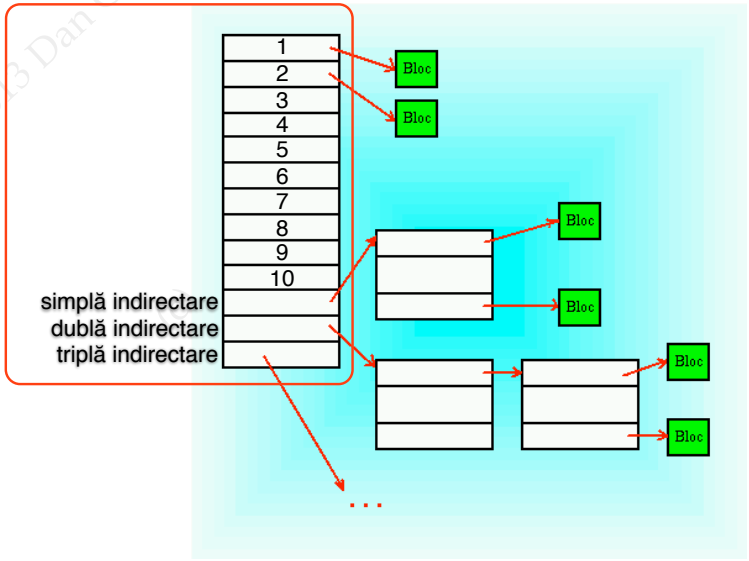


➠ Nod index (i-node)

→ conține informații despre un anumit fișier de pe disc

- **identificatorul utilizatorului**: *uid (user-id.).* Identifica proprietarul fisierului
- **identificatorul de grup al utilizatorului**
- **drepturile de acces la fisier.** Drepturile sunt de trei tipuri (***r****-read,* ***w****-write,* ***x****-execute)* si sunt grupate pe trei categorii:
  - *user* - drepturile proprietarului fisierului
  - *group* - drepturile utilizatorilor din grupul proprietarului
  - *others* - drepturile tuturor celorlalti utilizatori
- **timpul ultimului acces la fisier**
- **timpul ultimei actualizari a fisierului**
- **timpul ultimului acces pentru actualizarea nodului index**
- **tipul fisierului.** Fisierele pot fi: fisiere obisnuite (-), directoare (d), periferice (c) etc.
- **lungimea fisierului (in octeti)**
- **contorul de legaturi al fisierului.** Reprezinta numarul de legaturi existente spre acest nod index. Este utilizat la operatia de stergere a nodului index.
- **lista de blocuri** care conțin fișierul

- **lista de blocuri** care conțin fișierul



➠ Nod index (i-node)

→ conține informații despre un anumit fișier de pe disc

- **identificatorul utilizatorului**: *uid (user-id.).* Identifica proprietarul fisierului
- **identificatorul de grup al utilizatorului**
- **drepturile de acces la fisier.** Drepturile sunt de trei tipuri (***r****-read,* ***w****-write,* ***x****-execute)* si sunt grupate pe trei categorii:
  - *user* - drepturile proprietarului fisierului
  - *group* - drepturile utilizatorilor din grupul proprietarului
  - *others* - drepturile tuturor celorlalti utilizatori
- **timpul ultimului acces la fisier**
- **timpul ultimei actualizari a fisierului**
- **timpul ultimului acces pentru actualizarea nodului index**
- **tipul fisierului.** Fisierele pot fi: fisiere obisnuite (-), directoare (d), periferice (c) etc.
- **lungimea fisierului (in octeti)**
- **contorul de legaturi al fisierului.** Reprezinta numarul de legaturi existente spre acest nod index. Este utilizat la operatia de stergere a nodului index.
- **lista de blocuri** care conțin fișierul

Tipuri de fișiere în UNIX

- Fișier
  → în UNIX e folosit ca un concept general, unificator, pentru a reprezenta diferite resurse logice sau fizice
- Tipuri de fișiere
  → fișier obișnuit
  → director
  → fișier legătură simbolică
  → FIFO (named pipe)
  → socket
  → dispozitiv periferic orientat pe caracter
  → dispozitiv periferic orientat pe blocuri
  → ... (în funcție de varianta de UNIX pot exista mai multe tipuri de fișiere)

Dispozitive periferice

➠Reprezentate de fișiere în directorul /dev

→ /dev/sda, /dev/sdb ...

→ /dev/sda1, /dev/sda2, /dev/disk/by-id/scsi-SATA_ST3250820AS_9QE499JB-part5

→ /dev/cdrom

→ /dev/dvdrw

```
dd if=/dev/sdb2 of=backup-partition2.img bs=1024
```

```
strings /dev/sda3 > strings_on_attacked_rootpartition.txt
```

Dispozitive “periferice” virtuale

→ /dev/random /dev/urandom

→ /dev/null

→ /dev/zero

→ /dev/full

```
sh myscript >/dev/null 2>/dev/null
```

```
dd if=/dev/zero of=foobar count=1024 bs=1024
```


Legături spre fișiere

➠ Sistemul de fișiere UNIX permite crearea de legături spre fișiere existente

≈ nume alternative pentru un același fișier

➠ Două tipuri:

→ Legături fixe
→ Legături simbolice

Legături fixe (hard links)

➠ mai multe intrări în director care referă un același i-nod

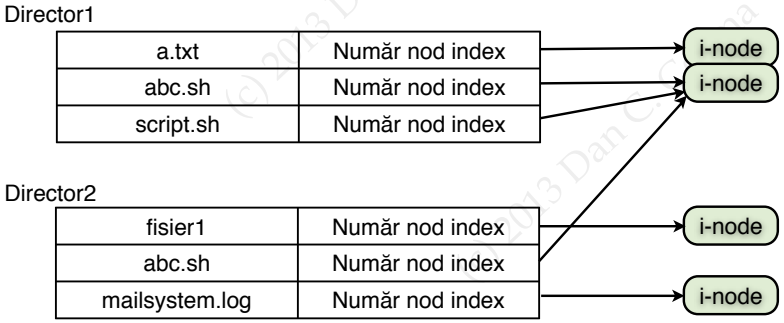
➠ referiri doar în cadrul aceluiași sistem de fișiere (partiții)

➠ nu pot fi referite directoare

→ pentru a nu crea dependențe circulare; excepție: versiuni mai noi de HFS+ (OS X), dar numai pentru sistemul de backup automatizat

➠ i-nodul memorează un contor de legături

→ folosit la crearea și ștergerea de legături; fișierul e șters doar când contorul era 1



Legături fixe (hard links)

```
$ touch abc
$ ls -l
total 0
-rw-r--r--  1 danc  staff  0 23 Oct 17:24 abc
$ ln abc abc1
$ ls -l
total 0
-rw-r--r--  2 danc  staff  0 23 Oct 17:24 abc
-rw-r--r--  2 danc  staff  0 23 Oct 17:24 abc1
$ ln abc xyz
$ ls -l
total 0
-rw-r--r--  3 danc  staff  0 23 Oct 17:24 abc
-rw-r--r--  3 danc  staff  0 23 Oct 17:24 abc1
-rw-r--r--  3 danc  staff  0 23 Oct 17:24 xyz
$ rm abc
$ ls -l
total 0
-rw-r--r--  2 danc  staff  0 23 Oct 17:24 abc1
-rw-r--r--  2 danc  staff  0 23 Oct 17:24 xyz
```

Legături simbolice (symbolic links)

➟ fișier special, de sine stătător, care referă un fișier existent
➟ se pot face referințe și către fișiere din sisteme de fișiere (partiții) distincte
➟ pot fi referite directoare
➟ fișierul legătură are i-nod propriu și ocupă loc pe disc
➟ dacă fișierul destinație e șters, legăturile ramân, însă vor indica locații inexistente

Legături simbolice (symbolic links)

```
$ echo "text" > abc
$ cat abc
text
$ ln -s abc link1
$ cat link1
text
$ ls -l
total 8
-rw-r--r--  1 danc  staff  0 23 Oct 17:34 abc
lrwxr-xr-x  1 danc  staff  3 23 Oct 17:34 link1 -> abc
$ ln -s abc link2
$ ls -l
total 16
-rw-r--r--  1 danc  staff  0 23 Oct 17:34 abc
lrwxr-xr-x  1 danc  staff  3 23 Oct 17:34 link1 -> abc
lrwxr-xr-x  1 danc  staff  3 23 Oct 17:34 link2 -> abc
$ rm abc
$ ls -l
total 16
lrwxr-xr-x  1 danc  staff  3 23 Oct 17:34 link1 -> abc
lrwxr-xr-x  1 danc  staff  3 23 Oct 17:34 link2 -> abc
```


Din nou puțină programare…

Un program

➟ parcurge recursiv un director dat ca argument în linia de comandă

➟ afișează
→ pentru legături simbolice: numele și calea indicată
→ pentru alte fișiere: numele
- dacă fișierul e executabil, afișează * la sfârșitul numelui acestuia
→ indentează afișarea în funcție de adâncimea în arbore

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <string.h>
#include <errno.h>
#include <dirent.h>
#include <unistd.h>
#include <sys/stat.h>
#include <limits.h>

void parcurge(char *nume_dir, int nivel)
{
    DIR *dir;
    struct dirent *in;
    char *nume;
    struct stat info;
    char cale[PATH_MAX], cale_link[PATH_MAX + 1], spatii[PATH_MAX];
    int n;

    memset(spatii, ' ', 2*nivel);
    spatii[2*nivel]='\0';

    if(!(dir = opendir(nume_dir)))
    {
        printf("%s: ", nume_dir); fflush(stdout);
        perror("opendir");
        exit(1);
    }
```



```c
        {
          printf("%s  %s", spatii, cale);
          if(info.st_mode & S_IXUSR || info.st_mode & S_IXGRP ||
 info.st_mode & S_IXOTH)
            printf("*");
          printf("\n");
        }

    }
    closedir(dir);
}

int main(int argc, char *argv[])
{
    if(argc != 2)
    {
      printf("Mod de utilizare: %s director\n", argv[0]);
      exit(1);
    }

    parcurge(argv[1], 0);

    return 0;
}
```



```c
printf("%sDIR %s:\n", spatii, nume_dir);

while((in = readdir(dir))>0)
{
    nume = in->d_name;
    if(strcmp(nume, ".") == 0 || strcmp(nume, "..")==0)
        continue;

    sprintf(cale, "%s/%s", nume_dir, nume);
    snprintf(cale, sizeof(cale), "%s/%s", nume_dir, nume);

    if(lstat(cale, &info)<0)
    {
      printf("%s: ", cale);  fflush(stdout);
      perror("eroare la lstat");
      exit(1);
    }

    if(S_ISDIR(info.st_mode))
      parcurge(cale, nivel + 1);
    else
    if(S_ISLNK(info.st_mode))
    {
      n = readlink(cale, cale_link, sizeof(cale_link));
      cale_link[n]='\0';
      printf("%s  %s -> %s\n", spatii, cale, cale_link);
    }
    else
```

de ce nu e bună această variantă ?


```
$ ./rd /etc
DIR /etc:
  /etc/AFP.conf
  /etc/afpovertcp.cfg
  /etc/aliases -> postfix/aliases
  /etc/aliases.db
  DIR /etc/apache2:
    DIR /etc/apache2/extra:
      /etc/apache2/extra/httpd-autoindex.conf
      /etc/apache2/extra/httpd-dav.conf
      /etc/apache2/extra/httpd-default.conf
      /etc/apache2/extra/httpd-info.conf
      /etc/apache2/extra/httpd-languages.conf
      /etc/apache2/extra/httpd-manual.conf
      /etc/apache2/extra/httpd-mpm.conf
      /etc/apache2/extra/httpd-multilang-errordoc.conf
      /etc/apache2/extra/httpd-ssl.conf
      /etc/apache2/extra/httpd-userdir.conf
      /etc/apache2/extra/httpd-vhosts.conf
    /etc/apache2/httpd.conf
    /etc/apache2/httpd.conf~previous
    /etc/apache2/magic
```

Cum s-ar face ?

→ aflarea dimensiunii fișierului
→ evidențierea fișierelor spre care există legături fixe
→ aflarea drepturilor utilizatorului proprietar
→ aflarea utilizatorului proprietar
→ modificarea drepturilor pentru un fișier
→ ștergerea fișierului
→ aflarea informațiilor despre un fișier indicat de link

Cum aflăm…

- ce fișiere header să includem (#include) ?
- ce returnează funcțiile sistem ?
- care sunt macro-urile pentru determinarea tipului unui fișier ?

4. Procese

Concepte de bază

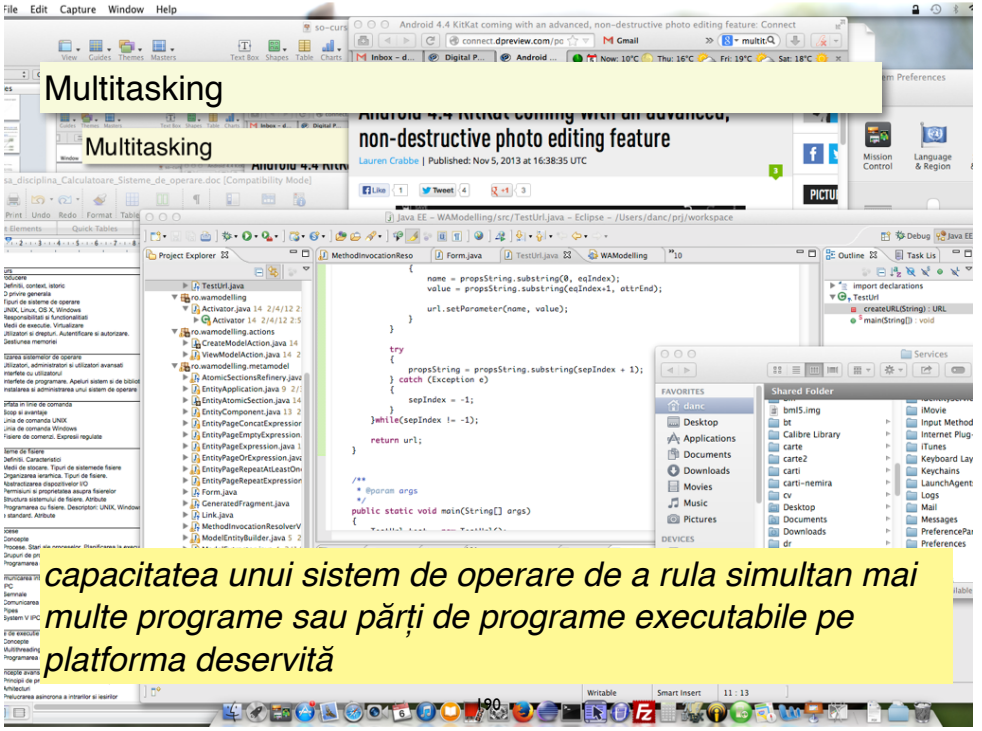
Multitasking

= abilitatea de a efectua mai multe activități în același timp

Multitasking

= abilitatea de a efectua mai multe activități în același timp

capacitatea unui sistem de operare de a rula simultan mai multe programe sau părți de programe executabile pe platforma deservită



capacitatea unui sistem de operare de a rula simultan mai multe programe sau părți de programe executabile pe platforma deservită

capacitatea unui sistem de operare de a rula simultan mai multe programe sau părți de programe executabile pe platforma deservită

programe *părți de programe*

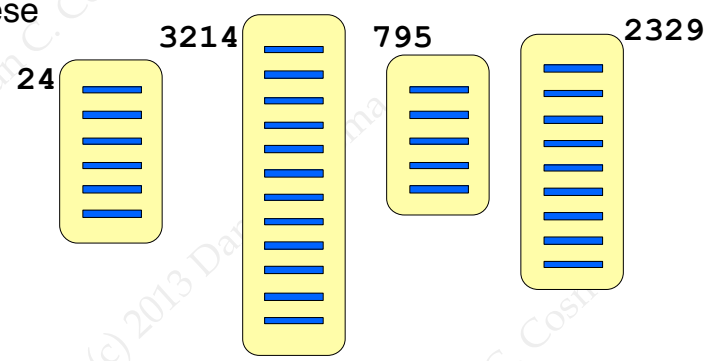
Procese

Procese

Proces = conceptul de bază utilizat de sistemul de operare pentru a modela entitățile software care rulează în paralel

Proces = un program sau o parte a unui program aflat în execuție sub controlul sistemului de operare

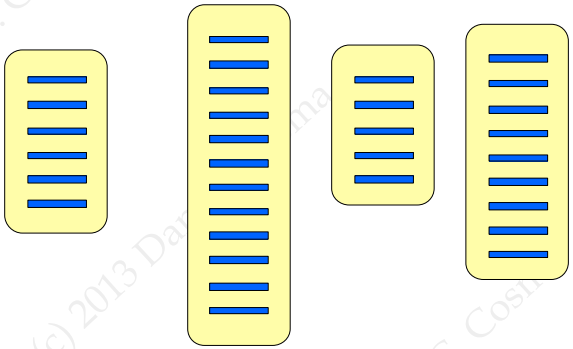
Procese



Procesele sunt identificate prin numere (process identifier - PID)

La un același moment dat, nu există două procese cu același identificator, dar identificatorii pot fi refolosiți după ce procesele s-au terminat

Procese

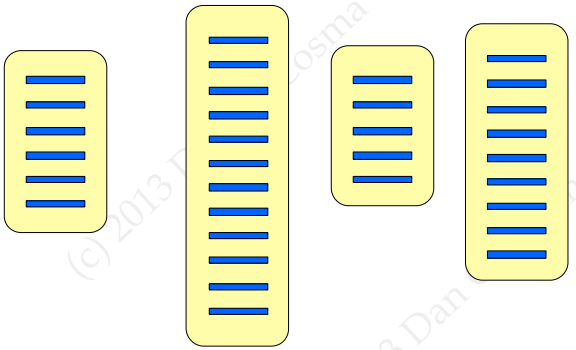


La un moment dat, într-un sistem de operare pot să ruleze simultan mai multe procese

- procese sistem
- procese utilizator

mai multe procese

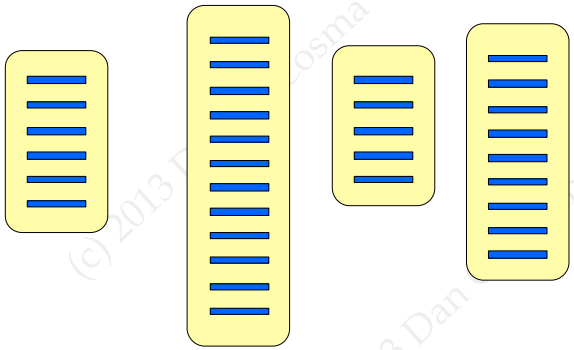
Un singur procesor ?
N procesoare ?



mai multe procese

Un singur procesor ?
N procesoare ?

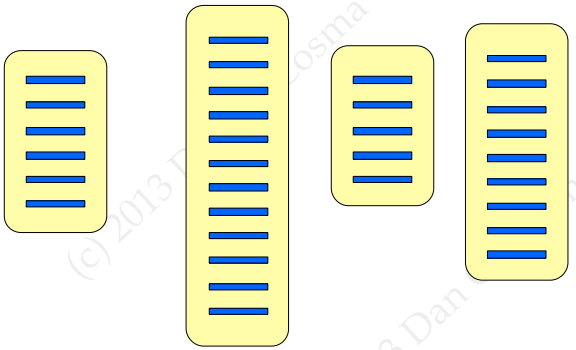
Cum ?!



mai multe procese

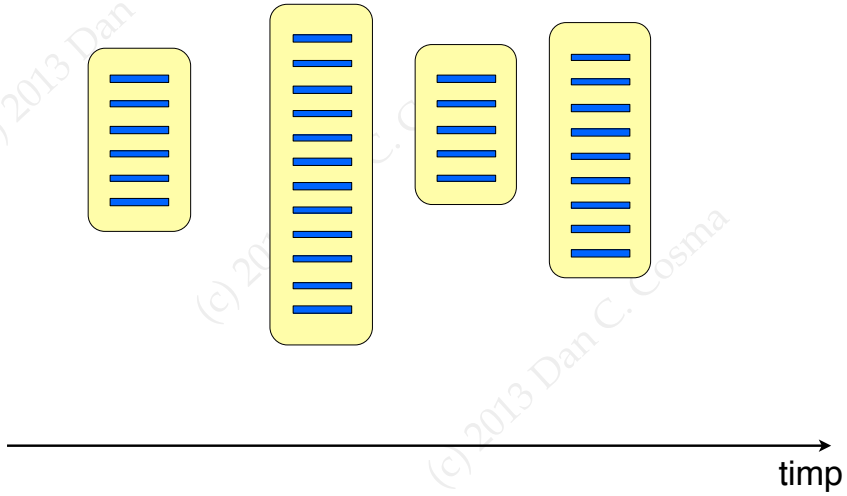
Un singur procesor ?
N procesoare ?

Cum ?!

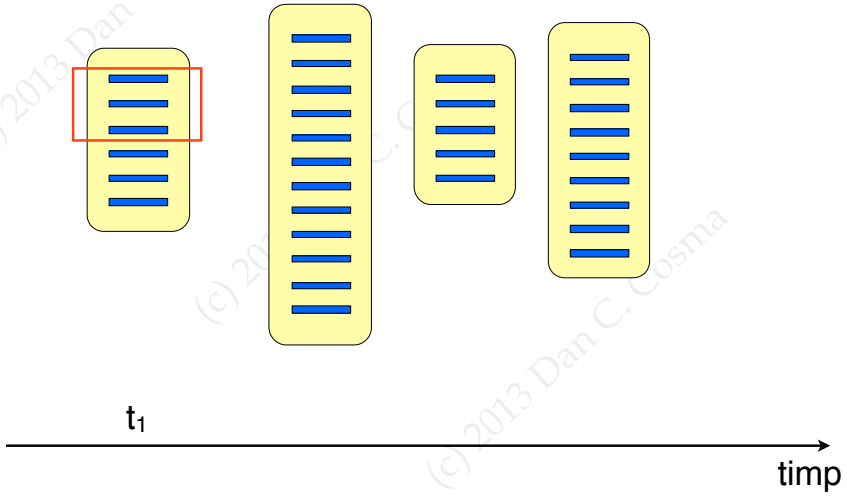


Planificarea la execuție a proceselor (Process Scheduling)

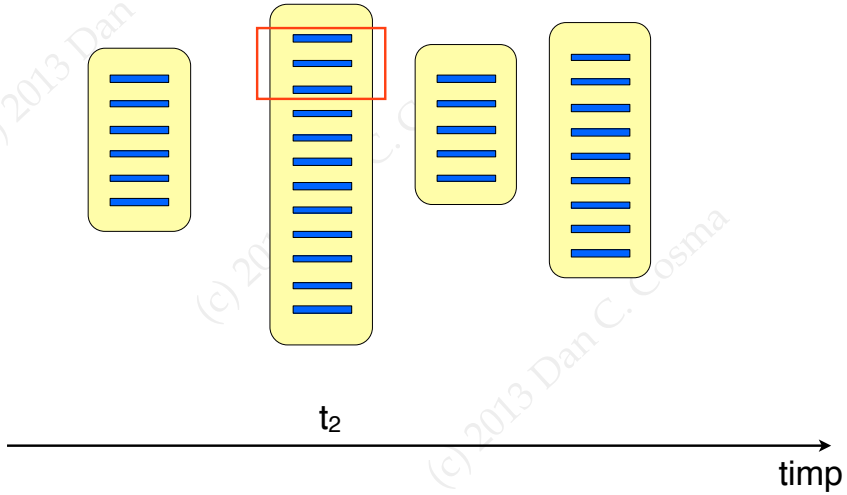
Planificarea la execuție a proceselor (Process Scheduling)



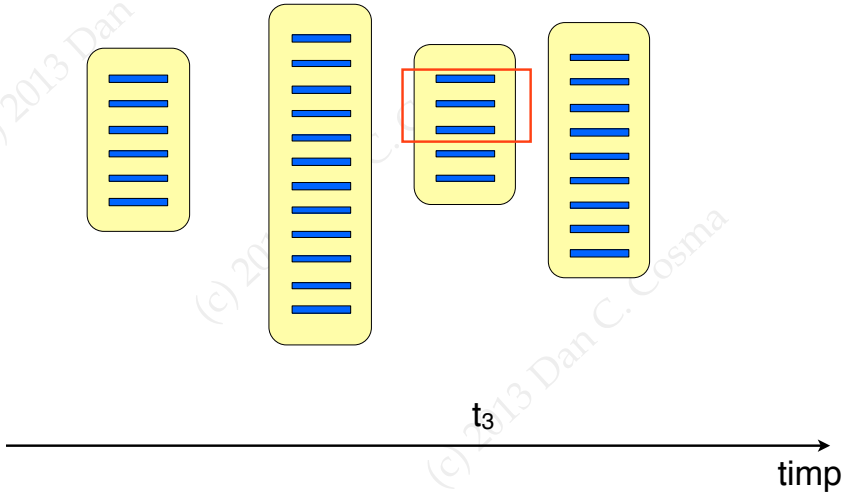
Planificarea la execuție a proceselor (Process Scheduling)



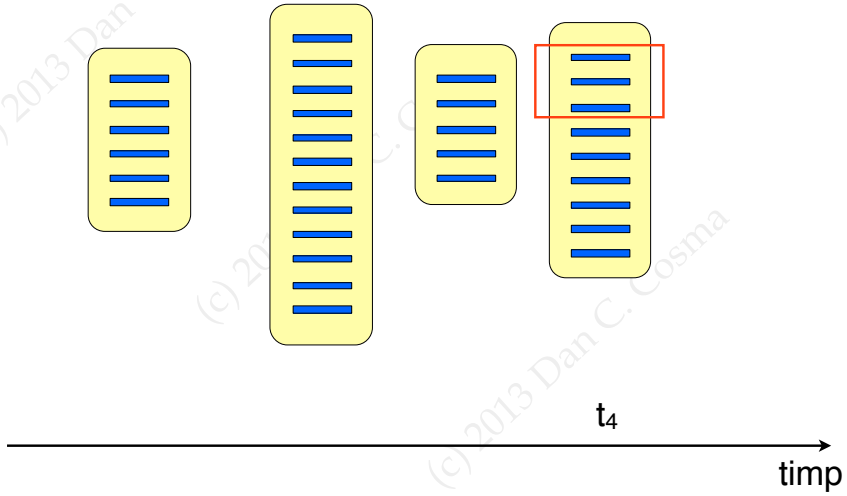
Planificarea la execuție a proceselor (Process Scheduling)



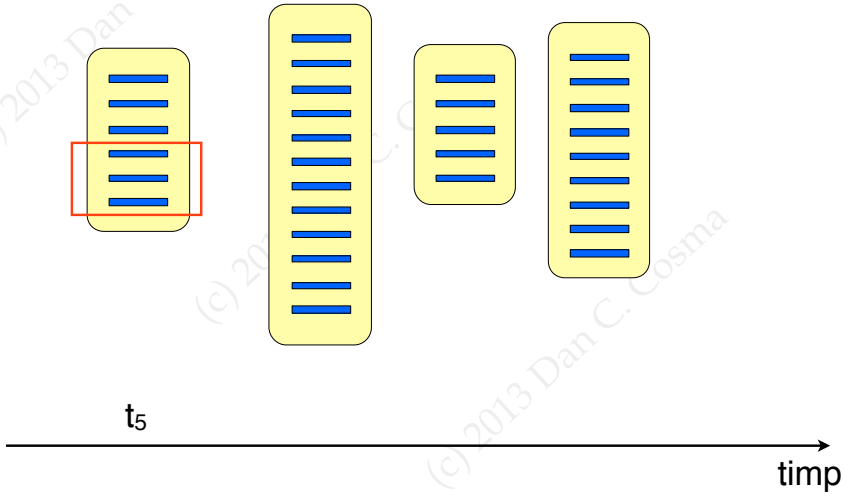
Planificarea la execuție a proceselor (Process Scheduling)



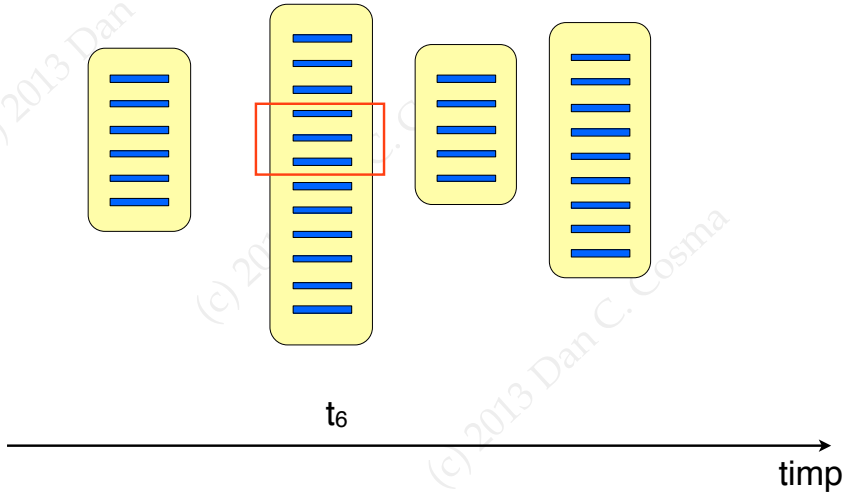
Planificarea la execuție a proceselor (Process Scheduling)



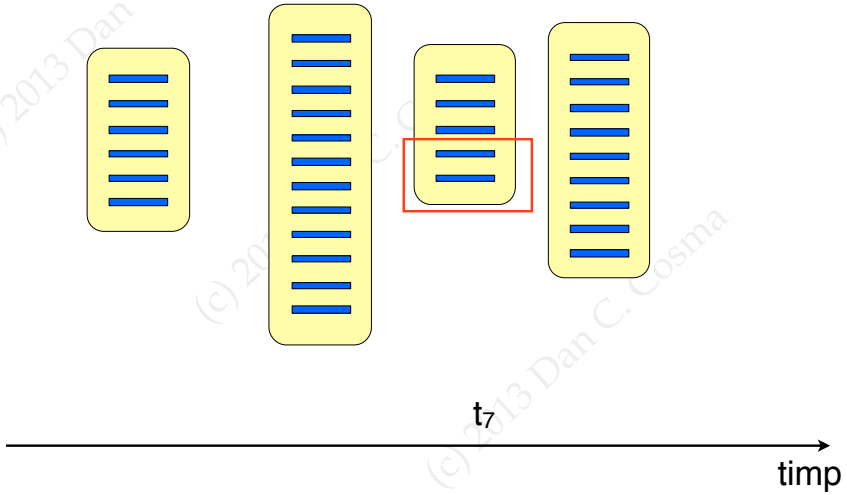
Planificarea la execuție a proceselor (Process Scheduling)



Planificarea la execuție a proceselor (Process Scheduling)



Planificarea la execuție a proceselor (Process Scheduling)



...

→ algoritmi de planificare la execuție

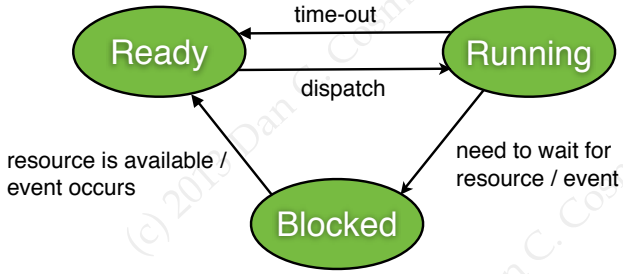
- implementați în nucleul sistemului de operare, care are, astfel, rol de *dispecer* de procese
- pot urma diverse strategii: "round-robin", bazate pe priorități etc.

Execuția proceselor este coordonată de către sistemul de operare, care este responsabil de gestionarea întregului ciclu de viață al proceselor

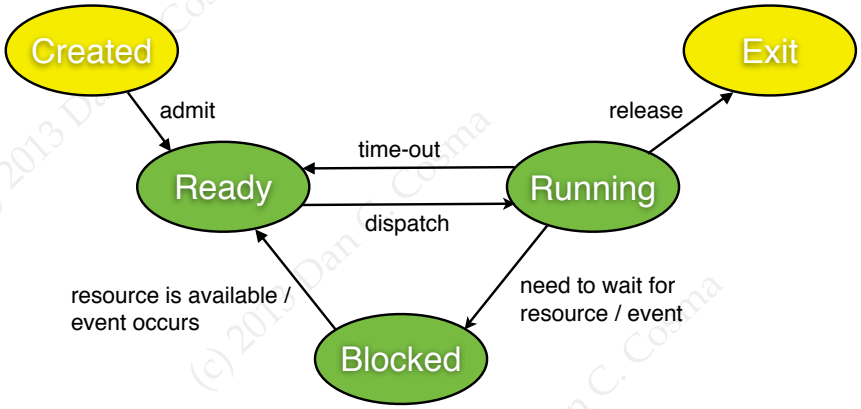
Stările unui proces

De-a lungul „existenței” sale, un proces poate să se afle în una din următoarele stări principale:

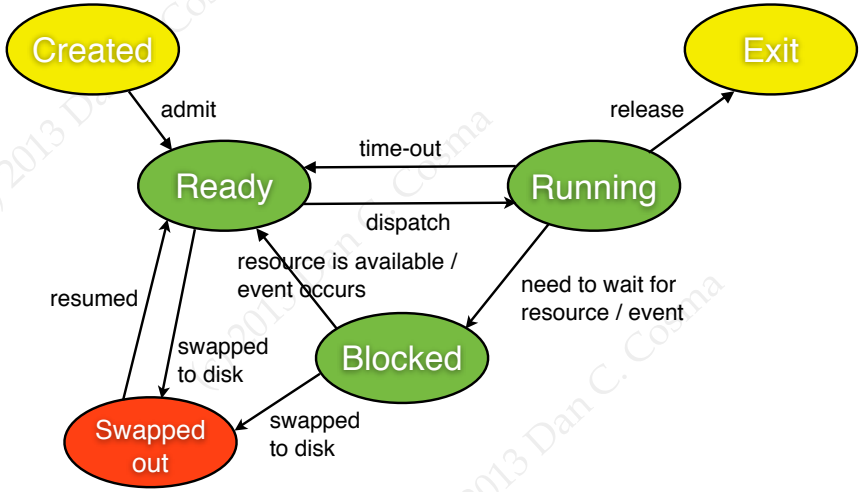
- Pregătit de execuție (Ready, Runnable)
  → procesul poate fi rulat, dar nu i-a venit încă rândul
- În execuție (Running)
  → procesul rulează
- Blocat (Blocked / Waiting)
  → procesul e blocat în așteptarea unor resurse sau evenimente (exemplu: date de la intrare, dintr-un fișier etc.)



După: W. Stallings, *Operating Systems: Internals and Design Principles, 7th edition*, Prentice Hall, 2011



După: W. Stallings, *Operating Systems: Internals and Design Principles, 7th edition*, Prentice Hall, 2011



După: W. Stallings, *Operating Systems: Internals and Design Principles, 7th edition*, Prentice Hall, 2011

Stările unui proces (altă perspectivă)

- extras din manualul comenzii **`ps`** din Linux -

```
PROCESS STATE CODES
Here are the different values that the s, stat and state
output specifiers(header "STAT" or "S") will display to
describe the state of a process.
D    Uninterruptible sleep (usually IO)
R    Running or runnable (on run queue)
S    Interruptible sleep (waiting for an event to complete)
T    Stopped, either by a job control signal or because it
is being traced.
W    paging (not valid since the 2.6.xx kernel)
X    dead (should never be seen)
Z    Defunct ("zombie") process, terminated but not reaped
by its parent.
```

Comanda **ps** (Process Status)

→afișează lista proceselor din sistem și informații despre ele

```
$ ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.0  0.1  39348  4460 ?        Ss   Nov06   0:02 /sbin/init
root          2  0.0  0.0      0     0 ?        S    Nov06   0:00 [kthreadd]
root          3  0.0  0.0      0     0 ?        S    Nov06   0:00 [ksoftirqd/0]
root          6  0.0  0.0      0     0 ?        S    Nov06   0:00 [migration/0]
root          7  0.0  0.0      0     0 ?        SN   Nov06   0:00 [rcuc0]
root         45  0.0  0.0      0     0 ?        S    Nov06   0:00 [scsi_eh_4]
root       1433  0.0  0.0  21052   612 ?        Ss   Nov06   0:00 /sbin/rpcbind
root       1435  0.0  0.0  62216  2192 ?        Ss   Nov06   0:02 /usr/sbin/nmbd -D -s /etc/samba/smb.conf
danc       2185  0.0  0.0  23256   704 ?        Ss   Nov06   0:00 /usr/bin/gpg-agent --sh --daemon --write-env-file /home/d
danc       2186  0.0  0.0  18568   528 ?        Ss   Nov06   0:00 /usr/bin/ssh-agent /etc/X11/xinit/xinitrc
danc       2198  0.0  0.0  20092   876 ?        S    Nov06   0:00 dbus-launch --sh-syntax --exit-with-session
danc       2199  0.0  0.0  22944  1820 ?        Ss   Nov06   0:00 /bin/dbus-daemon --fork --print-pid 5 --print-address 7 -
root       2318  0.0  0.1 164652  4440 ?        Sl   Nov06   0:00 /usr/lib/upower/upowerd
danc       2353  0.0  0.2 134304  7048 ?        S    Nov06   0:00 /usr/lib/xfce4/panel/wrapper /usr/lib64/xfce4/panel/plugi
danc       2361  0.1  0.3 112224  8904 ?        S    Nov06   0:07 /usr/lib/xfce4/panel-plugins/xfce4-orageclock-plugin  1 2
danc       2363  0.0  0.1 146904  3012 ?        Sl   Nov06   0:00 /usr/lib/gvfs/gvfs-afc-volume-monitor
danc       2367  0.0  0.0  58124  2624 ?        S    Nov06   0:00 /usr/lib/gvfs/gvfs-gphoto2-volume-monitor
danc       2380  0.0  0.1  48512  4092 ?        S    Nov06   0:00 /usr/lib/GConf/2/gconfd-2
danc       2387  0.0  0.1  58796  3880 ?        S    Nov06   0:00 /usr/lib/gvfs/gvfsd-trash --spawner :1.10 /org/gtk/gvfs/e
danc       2392  0.0  0.0  42512  2520 ?        S    Nov06   0:00 /usr/lib/gvfs/gvfsd-burn --spawner :1.10 /org/gtk/gvfs/ex
danc       2396  0.5  1.3 524352 37188 ?        Sl   Nov06   0:35 /usr/bin/python -OO /usr/bin/gmixer -d
root       3092  0.0  0.1  90004  3800 ?        Ss   Nov06   0:00 sshd: danc [priv]
danc       3100  0.0  0.0  90004  2092 ?        S    Nov06   0:00 sshd: danc@pts/0
danc       3101  0.0  0.1  20772  3508 pts/0    Ss   Nov06   0:00 -bash
root       3186  0.0  0.0      0     0 ?        S    Nov06   0:00 [flush-8:16]
danc       5611  1.2  0.0  35708  1936 ?        SN   00:01   0:06 grav -root
danc       6242  0.0  0.0  13252  1148 pts/0    R+   00:09   0:00 ps aux
```

*acesta e doar un extras din lista afișată de **ps***

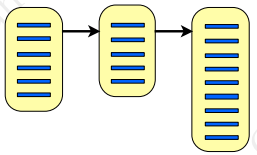
Structuri de date pentru procese

Pentru a gestiona procese, sistemul de operare menține structuri de date dedicate.

Operația de „trecere” de la un proces la altul (context switching, process switching) presupune costuri semnificative în timp/resurse

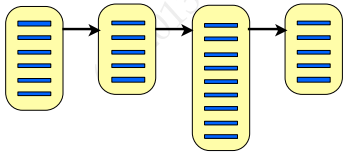
Structuri de date (exemple)

Lista proceselor pregătite de execuție



din această listă se selectează, pe rând, procesele care rulează efectiv (li se dă timp procesor)

Lista proceselor blocate



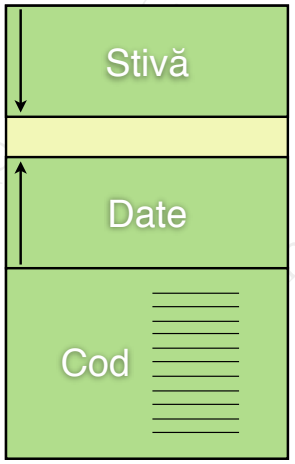
în această listă sunt memorate informații despre procesele blocate. Există mai multe astfel de liste, pentru diversele resurse sau evenimente cauză a blocării

Când un proces se deblochează, e trecut înapoi în lista proceselor pregătite de execuție

...

Fiecărui proces i se alocă zone de memorie și structuri de control/gestiune separate

La schimbarea stării unui proces, informațiile despre acesta sunt reținute; zonele de memorie pot fi salvate/restaurate de pe disc, dacă e cazul



Identificare proces — PID, PID proces părinte

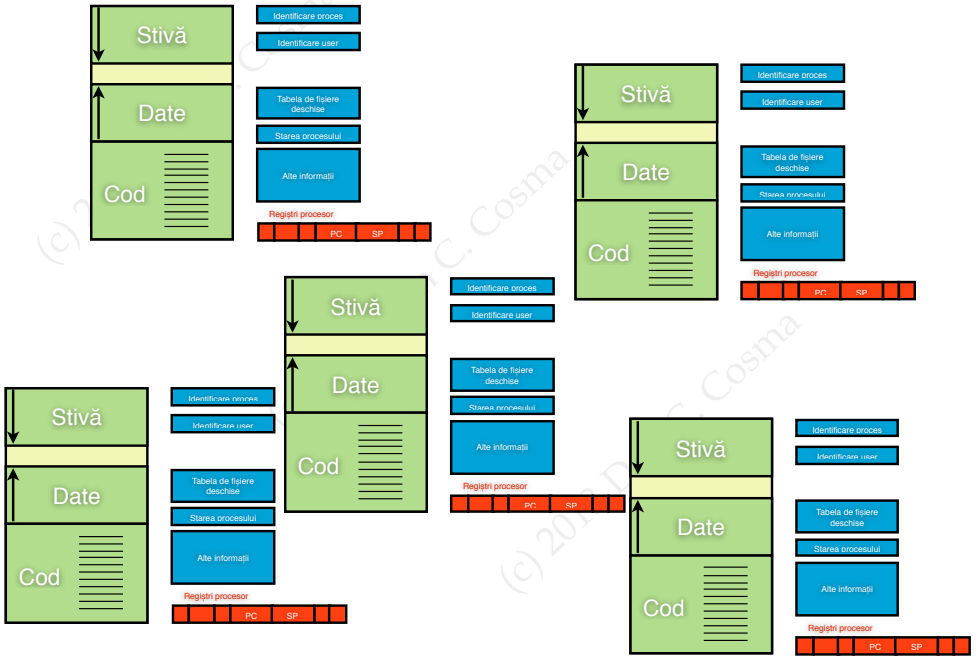
Identificare user — - *real UID* (proprietarul fișierului), - *effective UID* (procesul poate rula temporar cu privilegiile unui alt user decât proprietarul)

Tabela de fișiere deschise

Starea procesului — Pregătit de execuție, blocat etc.

Alte informații — Prioritate, director curent, terminalul de control, semnale primite, locația curentă (în memorie sau pe disc) etc.

Regiștri procesor



Crearea proceselor

Crearea proceselor

Orice proces poate să creeze un nou proces
→ procesul creat se numește proces fiu (child process)
→ procesul creator se numește proces părinte (parent process)

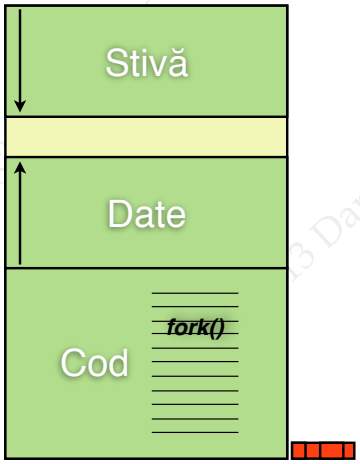
➥Este singura modalitate de a genera noi procese
→ astfel, fiecare proces din sistem va avea un proces părinte
→ se creează, astfel, un arbore de procese care descrie relația părinte-fiu

La inițializarea sistemului se creează automat un proces inițial, numit procesul init
→ init are PID = 1
→ este baza întregului „arbore genealogic" de procese

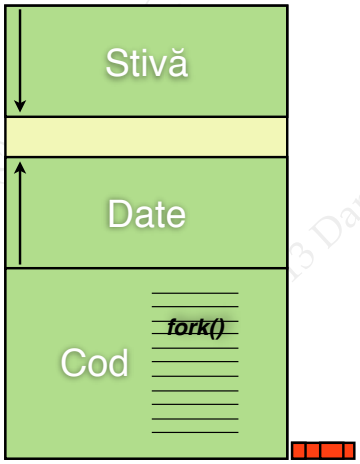
Funcția sistem *fork()*

→creează un proces fiu

```
#include <unistd.h>

pid_t fork(void);
```

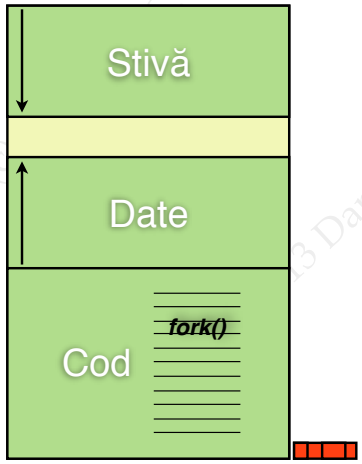



proces părinte

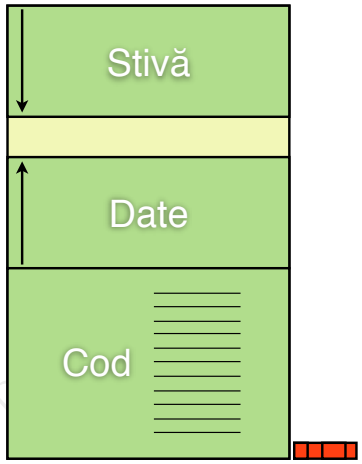


fork()

proces părinte

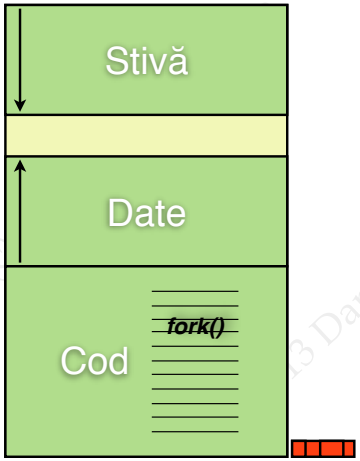


fork()

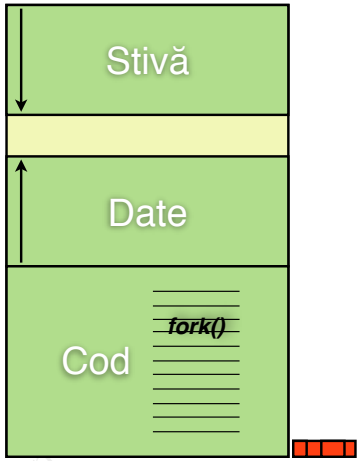


proces părinte

proces fiu



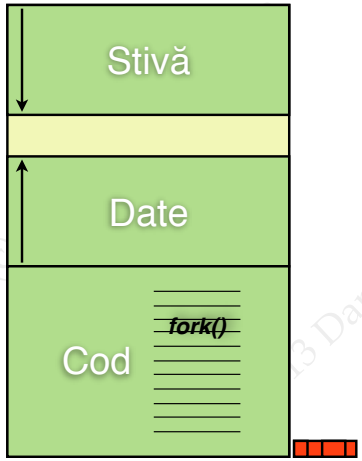
fork()



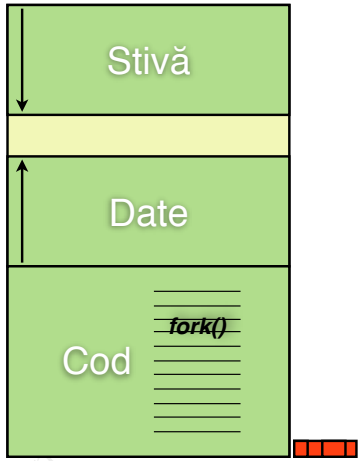
proces părinte

proces fiu

Procesul fiu este o copie (aproape) fidelă, inclusiv la nivelul datelor, a procesului părinte



fork()



proces părinte

proces fiu

După creare, **ambele procese** își vor **continua execuția**, **în paralel**, de la instrucțiunea imediat următoare apelului fork()

După creare, **ambele procese** își vor **continua execuția**, **în paralel**, de la instrucțiunea imediat următoare apelului fork()

După crearea procesului

După crearea procesului

Există două procese identice dar independente
→ au zone separate de date, stivă, regiștri etc.

Procesul fiu moștenește din părinte
→ toate datele (variabile globale), cu valorile existente în părinte imediat înainte de *fork()*
→ poziția curentă în program, stiva de apeluri, variabile locale
→ tabela de fișiere deschise: toate fișierele deschise de părinte până imediat înainte de *fork()* vor fi accesibile și utilizabile de procesul fiu

După crearea procesului

Există două procese identice dar independente
→ au zone separate de date, stivă, regiștri etc.

Procesul fiu moștenește din părinte
→ toate datele (variabile globale), cu valorile existente în părinte imediat înainte de *fork()*
→ poziția curentă în program, stiva de apeluri, variabile locale
→ tabela de fișiere deschise: toate fișierele deschise de părinte până imediat înainte de *fork()* vor fi accesibile și utilizabile de procesul fiu

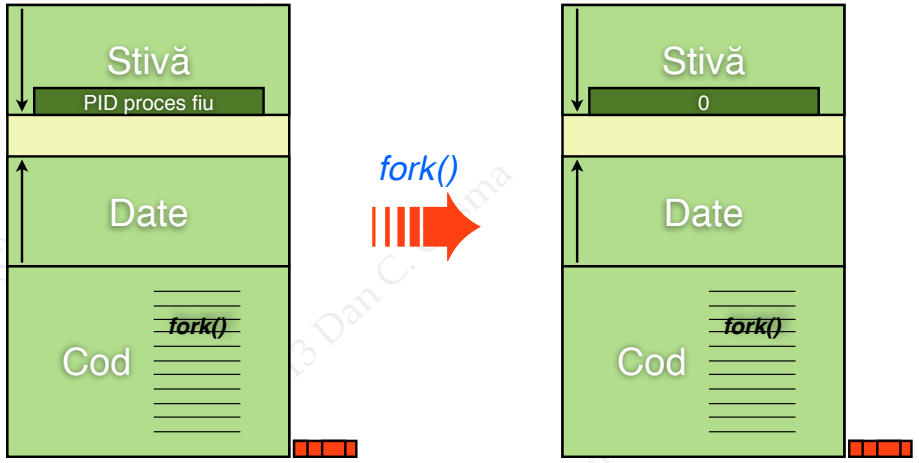
Ce diferă ?

După crearea procesului

Ce diferă ?

Funcția fork() returnează valori diferite în fiu față de părinte:
→ în fiu returnează 0
→ în părinte returnează PID-ul procesului fiu nou creat

Dacă a apărut o eroare, fork() returnează -1 și nu mai creează un nou proces



proces părinte

proces fiu

Astfel, un program poate face:

```
...
if( ( pid=fork() ) < 0)
  {
    perror("Eroare");
    exit(1);
  }
if(pid==0)
  {
    /* codul fiului */
    ...
    exit(0);
  }
/* codul parintelui */
...
```

⇒ codul părintelui și fiului se vor comporta diferit

Ce efect are următoarea secvență de cod ?

```
...
fork();
printf("a");
...
```

Ce efect are următoarea secvență de cod ?

```
...
int i;
for(i=0; i<=10; i++)
   fork();
...
```

Familia de funcții *exec...()*

→ folosite pentru a putea avea procese complet diferite în sistem
→ apelate de obicei imediat după *fork()*, în procesul fiu

→ încarcă de pe disc codul unui program executabil și îl suprascrie pe procesul curent, ștergând complet și inițializând zonele de memorie ale acestuia

Familia de funcții *exec...()*

Observații

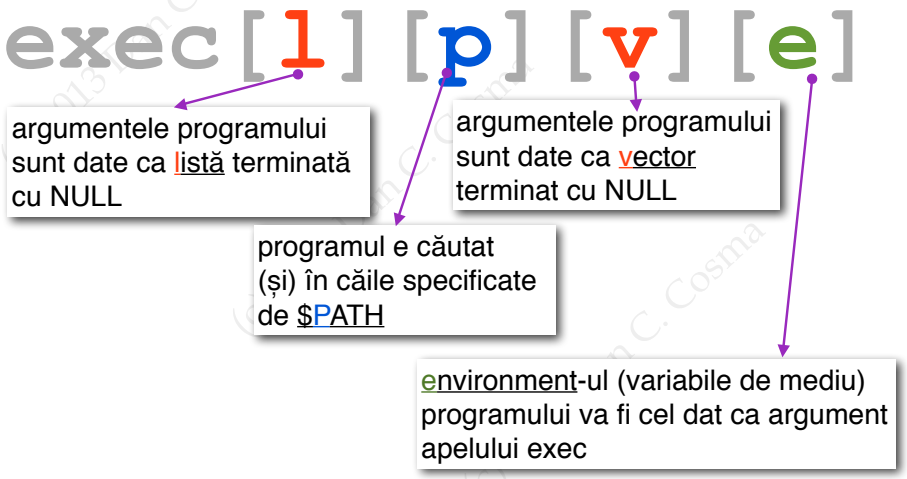
→ programul încărcat va rula de la prima sa instrucțiune (adică începând cu funcția main())
→ majoritatea atributelor procesului sunt păstrate
- identificatorul de proces (PID)
- relația părinte-fiu (Parent PID - PPID),
- semnalele în curs, timpul rămas până la alarmă
- fișierele deschise și redirectările existente (cu excepția cazurilor în care la deschidere se specifică flag-ul FD_CLOEXEC)
- real UID, terminalul de control, directorul curent, directorul rădăcină, prioritatea etc.

→ dacă se execută cu succes, funcția *exec* nu se întoarce (nu are cum, a fost suprascrisă)
→ în caz de eroare funcția se întoarce (cu valoarea -1)
→ funcțiile fork() și exec...() combinate oferă flexibilitate în crearea de procese

Familia de funcții *exec...()*

```
#include <unistd.h>

extern char **environ;

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

dintre toate, doar *execve()* e apel sistem, restul sunt funcții de bibliotecă



Exemplu: afișarea variabilelor de mediu

```
#include <stdio.h>
extern char **environ;

int main(int argc, char *argv[])
{
   char **p;
   p = environ;
   while(*p)
   {
     printf("%s\n", *p);
     p++;
   }
   printf("\n\n----------------\n\n");

/**
    * acelasi cod se poate scrie, echivalent, astfel:
    */
   for(p=environ; *p; p++)
     printf("%s\n", *p);
}
```


Exemplu

```
int main(int argc, char *argv[])
{
 pid_t pid;

 if((pid=fork())<0)
       {
           printf("Eroare la fork\n");
           exit(1);
       }
 if(pid==0) /* procesul fiu */
       {
           execlp("ls","ls","-l",NULL); /* procesul va rula
                                           comanda ls */
           printf("Eroare la exec\n");
           /* Daca execlp s-a intors, inseamna ca programul
            nu a putut fi lansat in executie */
       }
 else /* procesul parinte */
       {
           printf("Proces parinte\n");
           exit(0);

       }
}
```


Preluarea valorii de return a proceselor

Valoare returnată

- Valoare returnată
  → orice ~~program~~ proces returnează către sistem, la terminare, o valoare întreagă
  → C: `exit(valoare);` sau `return valoare;` în *main()*
- Convenție:
  → 0: procesul s-a terminat corect
  → ≠0: procesul s-a terminat cu eroare (iar valoarea e codul erorii)

- Valoarea returnată trebuie preluată de procesul părinte
  → astfel se verifică modul în care procesul fiu a rulat
  → procesele nepreluate (încă) de părinte rămân în sistem chiar după terminare, sub forma de "procese zombie"
  → procesele pentru care părintele se termină fără să fie preluate sunt preluate de procesul *init*

- Preluarea valorii returnate
  → orice proces poate apela funcțiile wait() și waitpid() pentru a prelua valorile returnate de procesele fiu


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

pid_t wait(int *status);

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

- wait()
  → se blochează până când unul (oricare) din procesele fiu se termină
  → completează în status starea de terminare a procesului fiu terminat, inclusiv valoarea returnată de acesta

  → pentru utilizarea informațiilor din status se pot folosi macro-uri specifice:

```
WIFEXITED(status)
   returnează true dacă fiul s-a terminat normal, adică
   apelând exit() sau prin returnarea unei valori in main()
WEXITSTATUS(status)
   returnează starea returnată de procesul fiu terminat
```

- waitpid()
  → analog cu wait() dar așteaptă după un proces fiu anume, specificat ca argument

Funcția de bibliotecă exit()

```
#include <stdlib.h>

void exit(int status);
```

→ termină procesul curent și returnează valoarea dată ca argument
→ înainte de terminare sunt închise toate fișierele deschise în proces, inclusiv stream-urile specifice bibliotecii stdio (FILE *)
→ la terminare apelează funcțiile care au fost instalate prin apeluri anterioare la atexit() sau on_exit()

Funcția sistem _exit()

```
#include <unistd.h>

void _exit(int status);
```

→ termină procesul curent și returnează valoarea dată ca argument
→ închide toți descriptorii de fișier deschiși în proces, dar **nu** închide stream-urile specifice bibliotecii stdio (FILE *)
→ la terminare **nu** apelează funcții gen atexit() sau on_exit()


```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
void process(char chr, int n){
  int i;
  for(i=0; i<=n-1; i++)
      printf("%c", chr);
}
int main(){
  pid_t pid; int status;

  if((pid=fork())<0){
      printf("Error creating child process\n"); exit(1);
  }
  if(pid==0) /* procesul fiu */ {
      process('c', 2000);
      exit(0);
  }
  /* procesul părinte*/
  process('p', 3000);
  wait(&status);
  if(WIFEXITED(status))
    printf("\nChild ended with code %d\n", WEXITSTATUS(status));
  else
    printf("\nChild ended abnormally\n");
}
```

Întrebare

Program care lansează două comenzi:
a) în paralel
b) secvențial

a)

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

int main(int argc, char *argv[])
{
  pid_t pid1, pid2, wpid;
  char *arg1[]={ "echo", "a", "b", "c", NULL };
  char *arg2[]={ "ls", "-l", ".", NULL };
  int i, status;

  if((pid1=fork())<0)
  {
    printf("Eroare la fork\n");
    exit(1);
  }
  if(pid1==0) /* procesul fiu 1 */
  {
    execvp("echo", arg1);
    printf("Eroare la exec\n");
    exit(2);
  }

  /* procesul parinte */
  if((pid2=fork())<0)
  {
    printf("Eroare la fork\n");
    exit(1);
  }
  if(pid2==0) /* procesul fiu 2 */
  {
    execvp("ls", arg2);
    printf("Eroare la exec\n");
    exit(2);
  }

  /* din nou procesul parinte */
  for (i=1; i<=2; i++)
  {
    wpid = wait(&status);
    if(WIFEXITED(status))
      printf("\nChild %d ended with code %d\n", wpid, WEXITSTATUS(status));
    else
      printf("\nChild %d ended abnormally\n", wpid);
  }
}
```

Important !

Important !

duplicare de cod ?!

a)

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

int main(int argc, char *argv[])
{
  pid_t pid1, pid2, wpid;
  char *arg1[]={ "echo", "a", "b", "c", NULL };
  char *arg2[]={ "ls", "-l", ".", NULL };
  int i, status;

  if((pid1=fork())<0)
  {
    printf("Eroare la fork\n");
    exit(1);
  }
  if(pid1==0) /* procesul fiu 1 */
  {
    execvp("echo", arg1);
    printf("Eroare la exec\n");
    exit(2);
  }

  /* procesul parinte */
  if((pid2=fork())<0)
  {
    printf("Eroare la fork\n");
    exit(1);
  }
  if(pid2==0) /* procesul fiu 2 */
  {
    execvp("ls", arg2);
    printf("Eroare la exec\n");
    exit(2);
  }

  /* din nou procesul parinte */
  for (i=1; i<=2; i++)
  {
    wpid = wait(&status);
    if(WIFEXITED(status))
      printf("\nChild %d ended with code %d\n", wpid, WEXITSTATUS(status));
    else
      printf("\nChild %d ended abnormally\n", wpid);
  }
}
```

Important !

Important !


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

int main(int argc, char *argv[])
{
  pid_t pid[2], wpid;

  char *arg1[]={ "echo", "a", "b", "c", NULL };
  char *arg2[]={ "ls", "-l", ".", NULL };
  int i, status;

  char ** param[2];

  param[0] = arg1;
  param[1] = arg2;

  for(i=0; i<2; i++)
  {
     if((pid[i]=fork())<0)
     {
       printf("Eroare la fork\n");
       exit(1);
     }
     if(pid[i]==0) /* procesul fiu i */
     {
       execvp(param[i][0], param[i]);
       printf("Eroare la exec\n");
       exit(2);
     }
  }

  /* procesul parinte */
  printf("Processes started:\n");
  for(i=0; i<2; i++)
    printf("%d ", pid[i]);
  printf("\n");
  for (i=1; i<=2; i++)
  {
    wpid = wait(&status);
    if(WIFEXITED(status))
      printf("\nChild %d ended with code %d\n", wpid, WEXITSTATUS(status));
    else
      printf("\nChild %d ended abnormally\n", wpid);
  }
}
```

Important !

b)

temă...

Aflarea ID-urilor procesului curent și al procesului părinte

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

pid_t getpid(void);
pid_t getppid(void);
```

Lansarea unei comenzi

```
#include <stdlib.h>

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

- folosește fork și exec pentru a lansa comanda într-un proces separat care execută `/bin/sh -c command`
- așteaptă terminarea comenzii și returnează starea acesteia

5. Semnale

Concepte de bază

Semnal

= o întrerupere de nivel software, utilizată pentru a modela apariția de evenimente asincrone
→ semnalele sunt trimise către procese
→ sursele de semnal sunt procese sau sistemul de operare (și pot avea și cauze hardware)

Semnalele au identificatori și nume derivate din evenimentul pe care îl modelează. Exemple (`man 7 signal`, Linux):

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard (CTRL C)
SIGQUIT	3	Core	Quit from keyboard (CTRL \)
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

Un proces poate specifica ce dorește să facă la primirea unui semnal:

• să ignore semnalul
→ există semnale care nu pot fi ignorate: SIGKILL, SIGSTOP
→ ignorarea semnalelor cu cauze hardware pot duce la comportamente nedefinite

• să trateze semnalul
→ programul trebuie să definească pentru acel proces o funcție de tratare a semnalului (signal handler)
→ nucleului sistemului de operare trebuie să i se specifice care e funcția de tratare, prin apeluri de genul signal() sau sigaction()
→ la apariția semnalului, nucleul va întrerupe procesul, va apela funcția instalată ca handler, iar după execuția ei, procesul va continua din punctul în care a rămas
→ apariția unui semnal poate determina deblocarea unor apeluri sistem cu blocare (exemplu: read). În acest caz, apelul respectiv va returna cod de eroare (-1), iar variabila errno va fi setată pe EINTR

• să accepte comportamentul implicit pentru acel semnal
→ pentru majoritatea semnalelor, acesta este terminarea procesului

Comenzile kill, killall

`kill -SIGNAL PID`
→Trimite un semnal către un proces

`killall -SIGNAL command`
→Trimite un semnal către toate procesele care rulează comanda dată

→ Dacă nu este specificat un semnal, este trimis semnalul SIGTERM

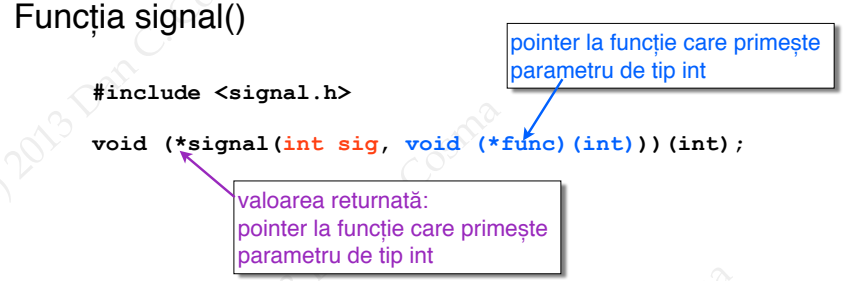
Exemple:

```
kill -SIGUSR1 2346
killall -9 java
```

Funcția sistem signal()

```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```


Funcția signal()




```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```

→ specifică, pentru procesul curent, modul de reacție la apariția unui semnal sau instalează în sistem o funcție de tratare a semnalului
→ parametrul sig este numărul semnalului pentru care se specifică modul de reacție
→ parametrul func
- este pointer la funcția de tratare a semnalului;
- poate avea și valorile:
  SIG_IGN : semnalul va fi ignorat
  SIG_DFL : se setează comportamentul implicit pentru acel semnal

→ funcția returnează vechea valoare a handler-ului pentru acel semnal (poate fi și una din valorile SIG_IGN, SIG_DFL) sau SIG_ERR în caz de eroare.

După instalarea funcției de tratare a semnalului, orice apariție a acelui semnal către procesul care a instalat-o ca handler va duce la apelarea asincronă a funcției.

Funcția handler primește ca argument numărul semnalului care a apărut. O aceeași funcție poate fi instalată pentru a trata mai multe semnale, astfel că acest parametru e util pentru a defini comportamente diferite pentru diferite semnale.

Exemplu

Un program format din două procese, părinte și fiu. Părintele (procesul a) numără continuu începând de la zero, până în momentul în care este întrerupt de către utilizator. Întreruperea se face generând către proces semnalul SIGINT, în mod explicit (folosind comanda kill) sau implicit (apăsând Ctrl-C în consola în care programul rulează lansat în foreground). Pentru vizualizarea optimă a rezultatelor, la fiecare pas procesul apelează funcția usleep() generând astfel o întârziere de câte 1000 microsecunde.

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

pid_t child_pid = 0;
int n = 0;

void process_a_ends(int sig)
{
  int status;

  if (kill(child_pid, SIGUSR2) < 0)
    {
      printf("Error sending SIGUSR2 to child\n");
      exit(2);
    }

  /* waiting for the child to end */

  wait(&status);
  printf("Child ended with code %d\n", WEXITSTATUS(status));

  printf("Process a ends.\n");
  exit(0);
}
```



```c
void process_a()
{
  int i;

  if (signal(SIGINT, process_a_ends) == SIG_ERR)
    {
      printf("Error setting handler for SIGTERM\n");
      exit(1);
    }
  for (i = 0;;i++)
    {
      usleep(1000);
      if (i%10 == 0)
   if (kill(child_pid, SIGUSR1) < 0)
     {
      printf("Error sending SIGUSR1 to child\n");
      exit(2);
     }
    }
}

void process_b_writes(int sig)
{
  printf("Process b received SIGUSR1: %d\n", ++n);
}
```



```c
void process_b_ends(int sig)
{
  printf("Process b ends.\n");
  exit(0);
}

void process_b()
{
  /* Ignoring SIGINT. Process b will end only when receives SIGUSR2 */
  if (signal(SIGINT, SIG_IGN) == SIG_ERR){
      printf("Error ignoring SIGINT in process b\n");
      exit(3);
    }
  /* Setting the signal handlers */
  if (signal(SIGUSR1, process_b_writes) == SIG_ERR){
      printf("Error setting handler for SIGUSR1\n");
      exit(4);
    }
  if (signal(SIGUSR2, process_b_ends) == SIG_ERR){
      printf("Error setting handler for SIGUSR2\n");
      exit(5);
    }

  /* Infinite loop; process b only responds to signals */
  while(1)
   ;
}
```

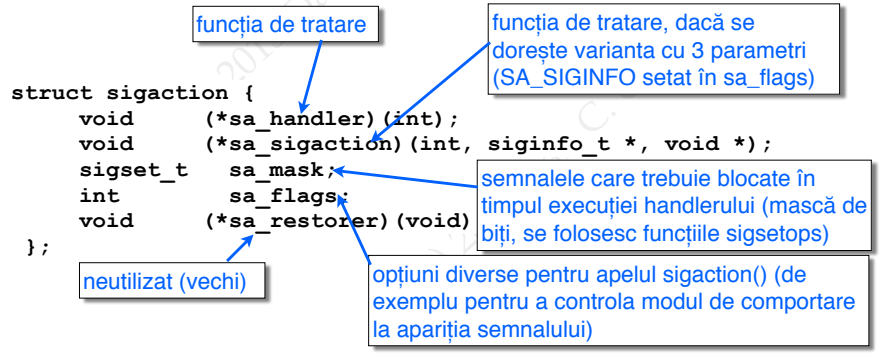


```c
int main()
{
  /* First, ignore the user signals, to prevent interrupting the
child process before setting the appropriate handlers */
  signal(SIGUSR1, SIG_IGN);
  signal(SIGUSR2, SIG_IGN);

  /* Creating the child process.A global variable is used to store
the child process ID in order to be able to use it from the signal
handlers */
  if ((child_pid = fork()) < 0){
      printf("Error creating child process\n");
      exit(1);
    }
  if (child_pid == 0){ /* child process */
      process_b();
      exit(0);
    }
  else /* parent process */
    {
      process_a();
    }
  /* this is still the parent code */
  return 0;
}
```

Funcția sistem sigaction()

→ *recomandată a fi folosită în locul funcției signal()*

```
#include <signal.h>

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



- câmpul **sa_flags** (exemple de opțiuni):

```
 SA_NOCLDSTOP - dacă signum e SIGCHLD, procesul nu va primi un
semnal SIGCHLD atunci când procesul fiu este suspendat (de
exemplu cu SIGSTOP), ci numai când acesta își termină execuția;

SA_NOMASK sau SA_NODEFER - semnalul in discutie nu va fi inclus
in mod automat in sa_mask (comportamentul implicit este acela
de a impiedica aparitia unui semnal in timpul executiei rutinei
de tratare a semnalului respectiv);

SA_SIGINFO - se specifica atunci cand se doreste utilizarea lui
sa_siginfo in loc de sa_handler.
```


Funcția sistem sigprocmask()

```
#include <signal.h>

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

→ citește sau schimbă masca de biți care specifică semnalele blocate ale firului de execuție apelant

```
how:
  SIG_BLOCK - setul de semnale specificat in argumentul set e
    adăugat la setul curent de semnale blocate
  SIG_UNBLOCK - setul de semnale specificat in argumentul set e
    șters din setul curent de semnale blocate
  SIG_SETMASK - setul curent de semnale blocate e inlocuit
    cu setul din argumentul set

set: setul de semnale folosit de apel conform opțiunii how

oldset: dacă nu e NULL, aici va fi memorat vechiul set de
    semnale blocate
```


Alte funcții

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

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

→Trimite semnalul sig către procesul pid

```
#include <signal.h>

int raise(int sig);
```

→Trimite semnalul sig către procesul curent

```
#include <unistd.h>

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

→ Instalează o alarmă; după ce trec *seconds* secunde, un semnal SIGALRM va fi generat către procesul curent

6. Pipes

Crearea și utilizarea pipe-urilor

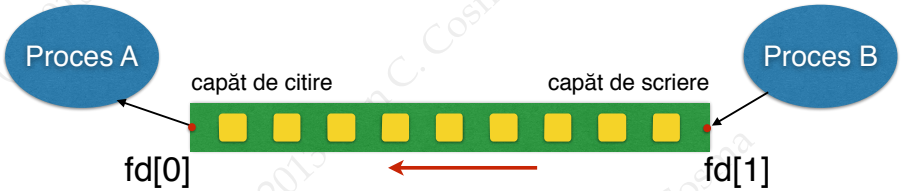
Pipe

→ o primitivă de comunicare între procese UNIX

→ se constituie într-o “conductă” de date prin care două procese își pot trimite date

Pipe

Pipe



- Capetele de citire și scriere sunt modelate ca *descriptori de fișier*

- Pipe-ul este *unidirecțional*

Apelul sistem pipe()

```
#include <unistd.h>

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

→ Creează un pipe
→ completează tabloul dat ca argument cu capetele pipe-ului:
- pipefd[0]: capătul de citire
- pipefd[1]: capătul de scriere

→ din momentul creării, procesul curent va putea scrie și citi din pipe, folosind apelurile sistem read() și write()

→ din momentul creării, procesul curent va putea scrie și citi din pipe, folosind apelurile sistem read() și write()

?

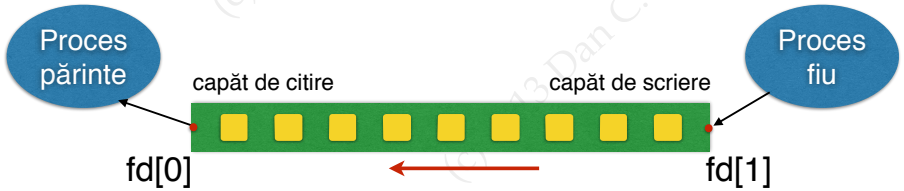
→ din momentul creării, procesul curent va putea scrie și citi din pipe, folosind apelurile sistem read() și write()

→ Capetele pipe-ului sunt tratate ca descriptori de fișier
→ Descriptorii de fișier sunt *moșteniți* de procesul fiu după fork()

Și pipe-ul este o structură de date care este moștenită de un proces de la părinte, în momentul creării procesului

Cum se folosește pipe()

→ pipe-ul e creat de un proces
→ procesul inițial apelează fork(): fiul va moșteni pipe-ul, deci va avea acces complet la capetele lui (pentru scriere și citire)
→ cele două procese (părinte, fiu) își stabilesc direcția de utilizare a pipe-ului: un proces va scrie, celălalt doar va citi
→ stabilirea direcției de utilizare: fiecare proces închide capătul pe care nu îl folosește




```
...
int pfd[2];
int pid;
...
if(pipe(pfd)<0)
  { printf("Eroare la crearea pipe-ului\n"); exit(1); }
...
if((pid=fork())<0)
  { printf("Eroare la fork\n"); exit(1); }

if(pid==0) /* procesul fiu */
  {
    close(pfd[0]); /*inchide capatul de citire => va scrie in pipe*/
    ...
    write(pfd[1],buff,len); /* operatie de scriere in pipe */
    ...
    close(pfd[1]); /* la sfarsit inchide si capatul utilizat */
    exit(0);
  }
else /* procesul parinte */
  {
    close(pfd[1]);/*inchide capatul de scriere => va citi din pipe*/
    ...
    read(pfd[0],buff,len); /* operatie de citire din pipe */
    ...
    close(pfd[0]); /* la sfarsit inchide si capatul utilizat */
    exit(0);
  }
```


Observații

→ Dacă un proces citește dintr-un pipe pentru care capătul de scriere e închis, operația read() va returna 0

→ Dacă un proces scrie într-un pipe al cărui capăt de citire este închis, operația write() va eșua, astfel:
- procesul respectiv va primi semnalul SIGPIPE
- dacă procesul nu tratează, blochează sau ignoră semnalul SIGPIPE, procesul va fi terminat; în caz contrar, valoarea returnată de write() va fi -1, iar *errno* va avea valoarea EPIPE

Observații

→ Pipe-urile pot fi moștenite de mai multe procese (tot subarborele de procese al procesului care a creat pipe-ul)
→ Un proces trebuie să închidă întotdeauna capetele de pipe pe care nu le folosește. Dacă nu folosește deloc pipe-ul, trebuie să le închidă pe amândouă
→ *Recomandare* importantă: procesul/procesele care scriu în pipe și procesul/procesele care citesc trebuie să își stabilească în mod clar, la orice moment, cantitatea și semnificația datelor transmise

Exemplu: procesul A transmite n octeți; procesul B care citește trebuie să citească (aștepte) exact n octeti: nici mai mult, nici mai puțin

Exemplu: procesul A transmite n octeți; procesul B care citește trebuie să citească (aștepte) exact n octeti: nici mai mult, nici mai puțin

Explicație:
- dacă procesul B citește (așteaptă) mai mulți octeți,funcția read() s-ar putea bloca în așteptarea unor date care nu vor mai veni niciodată (dacă A nu mai trimite) sau va citi date dintr-o "transmisie" ulterioară ca aparținând celei curente

- dacă B citește mai puțini octeți, atunci datele citite la etapa curentă vor fi incomplete, iar în pipe vor rămâne date necitite care *probabil* vor fi citite în viitor ca aparținând, în mod eronat, etapei respective a comunicării

- există *posibilitatea* ca astfel de erori de protocol să ducă la blocarea unuia sau mai multor procese (așteaptă la nesfârșit date care nu vor mai veni niciodată)

- sigur, sunt cazuri în care se poate ignora această recomandare, dar trebuie să știți exact ce faceți în acel caz

De ce trebuie închise întotdeauna capetele de pipe nefolosite ?

Explicație:
- Fie procesul A părinte pentru B, A a creat pipe-ul înainte de fork()
- A citește din pipe, B scrie în pipe
- A „uită" să închidă capătul de scriere

- A citește datele din pipe într-un ciclu

```
while((n=read(pfd[0], buff, no_of_bytes))>0) {
  ...
}
```

- La un moment dat, B își termină transmisia de date

⇓

A se blochează la nesfârșit în read(), deoarece read() nu va returna zero ("sfârșit de fișier") câtă vreme mai există un proces care, teoretic, ar putea transmite date în pipe (are capătul de scriere deschis): acest proces e chiar procesul A

Nu uitați !

→ creați pipe-ul înainte de fork()
→ închideți toate capetele nefolosite, ale tuturor pipe-urilor vizibile în procesul respectiv
→ închideți și capetele folosite, imediat după ce nu mai e nevoie de ele (de ce ?)
→ stabiliți un protocol precis de comunicare între procesele care scriu și citesc din pipe (câți octeți se scriu la un un moment dat, exact atâția trebuie citiți la momentul respectiv de către procesul cititor)

Duplicarea și redirectarea descriptorilor de fișier

Duplicarea descriptorilor de fișier

```
#include <unistd.h>

int dup(int oldfd);
```

→ duplică descriptorul oldfd, creând un nou descriptor care va pointa spre același fișier; descriptorul nou e returnat de funcție
→ ambii descriptori vor partaja indicatorul poziției curente în fișier, flag-urile de deschidere a fișierului etc.
→ descriptorul nou alocat va fi întotdeauna cel mai mic descriptor disponibil (nedeschis)

Exemplu

```
...
fd=open("Fisier.txt", O_WRONLY);
...
fd1=dup(fd);
...
write(fd1, "Un text", 8);
...
```


Duplicarea descriptorilor de fișier cu dup2()

```
#include <unistd.h>

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

→ duplică descriptorul oldfd, creând un nou descriptor care va pointa spre același fișier;
→ descriptorul nou va avea valoarea dată de argumentul newfd
→ dacă newfd era un descriptor alocat deja unui fișier deschis, acesta va fi mai întâi închis, acordându-i-se apoi descriptorului noua semnificație
→ returnează noul descriptor alocat (chiar newfd)

Redirectarea descriptorilor de fișier

→ acordarea unei noi semnificații unui descriptor existent, pentru a pointa spre un alt fișier decât cel pe care îl indica inițial
→ e un caz particular de duplicare
→ se face duplicând descriptorul ce indică noul fișier, asigurându-ne că valoarea obținută din duplicare e chiar cea a descriptorului care se dorește a fi redirectat

De exemplu, o duplicare care face ca descriptorului cu numărul 1 să îi corespundă un fișier de pe disc reprezintă practic redirectarea ieșirii standard
→ *funcțiile care scriu "la ieșirea standard" (exemplu: printf) scriu de fapt în descriptorul 1 (STDOUT_FILENO); redirectând acest descriptor, efectul acestor funcții se va vedea în fișierul spre care s-a redirectat*

Exemplu de redirectare a ieșirii standard

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


Din nou despre pipe…

→ Capetele pipe-ului sunt tratate ca descriptori de fișier

⇒ pot fi utilizate în duplicări sau redirectări

→ de exemplu, pot fi redirectate
- ieșirea standard: spre capătul de scriere al unui pipe
- intrarea standard: dinspre capătul de citire al unui pipe

Exemplu

→ conectarea prin pipe a două procese, unul din ele rulând (exec) un program citit de pe disc

Exemplu

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

  ...
  if(pipe(pfd)<0)
    {
      printf("Eroare la crearea pipe-ului\n");
      exit(1);
    }
  ...
  if((pid=fork())<0)
    {
      printf("Eroare la fork\n");
      exit(1);
    }
```

```
  if(pid==0) /* procesul fiu */
  {
    close(pfd[0]); /* inchide capatul de citire; */
                   /* procesul va scrie in pipe  */
    ...
    dup2(pfd[1],1); /* redirecteaza iesirea standard spre pipe*/
    ...
    execlp("ls","ls","-l",NULL); /* procesul va rula comanda ls*/
    printf("Eroare la exec\n);
  }
  else /* procesul parinte */
  {
    close(pfd[1]); /* inchide capatul de scriere; */
                   /* procesul va citi din pipe  */
    ...
    stream=fdopen(pfd[0],"r");
    /* deschide un stream (FILE *) pentru capatul de citire */
    while(...)
     { ...
       fscanf(stream,"%s",string);
       /* citire din pipe, folosind stream-ul asociat */
       ...
     }
    ...
    close(pfd[0]); /* la sfarsit inchide si capatul utilizat */
    exit(0);
  }
}
```


Observație: exemplul folosește funcția fdopen()

```
#include <stdio.h>

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

→ asociază un stream de tip FILE * (gestionat de funcțiile de bibliotecă stdio) unui fișier deschis spre care pointează descriptorul (specific funcțiilor sistem) *fd*
→ în final, fișierul respectiv va trebui închis cu fclose(), nu cu close() (pentru a se face corect toate de-alocările de resurse, de exemplu pentru a se goli buffer-ele gestionate de biblioteca stdio). Funcția fclose() va apela ea close() în final.
→ opțiunile din *mode* trebuie să fie compatibile cu modul în care a fost deschis descriptorul *fd*

Observație

→ pipe-ul este primitiva de comunicare între procese pe care o folosește interpretorul de comenzi atunci când înlănțuie comenzi folosind operatorul ‘ | ’

Exercițiu: program care realizează (în mod simplificat) efectul liniei de comandă:

```
prog1 | prog2
```



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


int pfd[2];


int main(int argc, char *argv[])
{
  int pid_a, pid_b;

  if(argc != 3)
  {
    printf("Utilizare: %s prog1 prog2\n",
argv[0]);
    exit(1);
  }

  if(pipe(pfd)<0)
  {
    printf("Eroare la crearea pipe-ului
\n");
    exit(1);
  }

  if((pid_a=fork())<0)
  {
    printf("Eroare la fork\n");
    exit(1);
  }
  if(pid_a==0) /* a */
  {
    close(pfd[0]);

    dup2(pfd[1],1);

    execlp(argv[1], argv[1], NULL);
    printf("Eroare la exec\n");
    exit(1);
  }
```



```
  /* parinte */

  if((pid_b=fork())<0)
  {
    printf("Eroare la fork\n");
    exit(1);
  }

  if(pid_b==0) /* b */
  {
    close(pfd[1]);

    dup2(pfd[0],0);

    execlp(argv[2], argv[2], NULL);
    printf("Eroare la exec\n");
    exit(1);
  }

  /* b */
  close(pfd[0]);
  close(pfd[1]);

  /* Procesul parinte preia rezultatele */

  int status;

  waitpid(pid_a, &status, 0);

  waitpid(pid_b, &status, 0);

  /*varianta simplificata de tratare a
valorii de return*/
  if(WIFEXITED(status))
    return WEXITSTATUS(status);
  else
    return 1;

  return 0;
}
```

Named pipes

Named pipes

→ sunt pipe-uri care pot fi create explicit din linie de comandă sau programe, acordându-li-se nume
→ sunt *vizibile în sistemul de fișiere* ca fișiere speciale, asupra cărora se pot efectua operații obișnuite de scriere și citire
→ operațiile de scriere și citire se fac respectând principiul FIFO
→ pipe-urile cu nume pot fi folosite explicit, de exemplu în script-uri pentru a realiza comunicarea între procese, comenzi, pentru a înlocui fișiere temporare etc.

Observație: pipe-urile discutate anterior (create cu funcția pipe()) se mai numesc și pipe-uri anonime

Crearea pipe-urilor cu nume

mkfifo [-m mode] nume

în care mode: drepturile de acces la fișierul special creat

```
mkfifo --mode=0766 ~/tmp_pipe
```

◆

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

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


Exemplu

```
$ mkfifo pipe1
$ wc -l < pipe1 > rezultat.txt &
[1] 768
$ ls -l > pipe1
[1]+  Done                    wc -l < pipe1 > rezultat.txt
$ cat rezultat.txt
      28
```

7. Fire de execuție

Multitasking

= abilitatea de a efectua mai multe activități în același timp

Multitasking

= abilitatea de a efectua mai multe activități în același timp

→ Procese

Multitasking

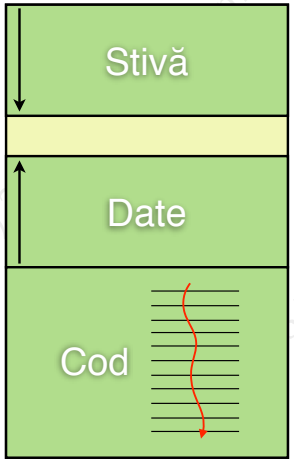
= abilitatea de a efectua mai multe activități în același timp

→ Procese

→ Fire de execuție

Fir de execuție (thread)

= o execuție secvențială în cadrul unui proces

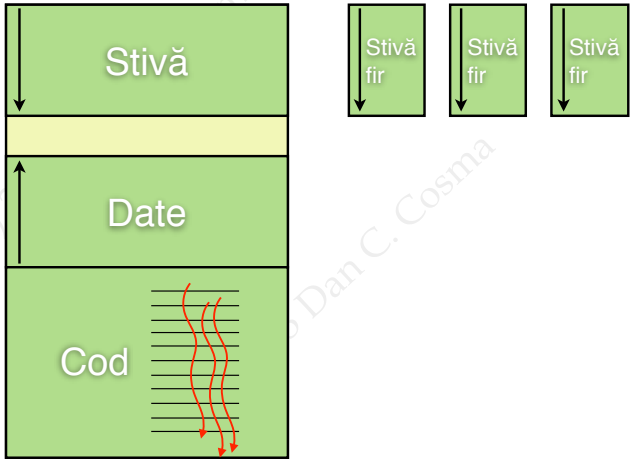


Pot exista mai multe fire de execuție în același proces

→ rulează în paralel și execută părți diferite din cod, sau chiar același cod

→ partajează memoria procesului, dar au stive separate ← Consecințe?

La creare, un proces are un singur thread (firul principal)



Avantaje față de procese

→ Este nevoie de mai puține resurse în sistem pentru gestionarea unui fir de execuție

→ Trecerea, la execuție, de la un fir la altul e mai rapidă

→ Firele de execuție pot comunica ușor între ele prin memoria comună

Observații:

→ La creare, un proces este format dintr-un singur fir de execuție.

→ Toate firele de execuție din cadrul unui proces rulează în paralel.

→ Un proces se termină:

- când firul său principal se termină
- dacă un fir apelează exit()
- când se termină funcția main() (deci firul principal)
- dacă procesul primește un semnal netratat
- …

→ Dacă un proces format din mai multe fire de execuție se termină, toate firele de execuție ale sale se vor termina.

→ Deoarece împart aceeași zonă de date, firele unui proces vor folosi în comun toate variabilele globale. Variabilele locale și parametrii de funcții nu sunt partajate, deoarece stivele firelor de execuție sunt separate.

→ Multe apeluri sistem și funcții de bibliotecă au efect la nivelul întregului proces, deci asupra tuturor firelor acestuia, indiferent de firul de execuție din care au fost apelate. Exemplu: funcția sleep().

Notă: pentru lucrul cu fire de execuție se folosește biblioteca pthreads.

→ *de obicei, aceasta nu e activată legată (linked) automat de gcc* la codul obiect al programului, drept urmare trebuie cerut acest lucru în mod explicit (opțiunea -lpthread). În unele versiuni mai noi de UNIX și gcc, suportul pentru fire de execuție e inclus direct în biblioteca glib și e activat folosind opțiunea -pthread*

→ *funcțiile din această bibliotecă returnează, de regulă, 0 în cazul în care s-au efectuat corect și un cod de eroare altfel*

*de fapt, de editorul de legături

Identificarea firelor de execuție

→ identificatori unici în cadrul procesului respectiv

```
#include <pthread.h>

pthread_t pthread_self(void);
```

→ obține identificatorul firului de execuție curent
→ definiția efectivă a tipului pthread_t depinde de implementare, poate fi o structură de date

Crearea firelor de execuție

```
#include <pthread.h>

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

locație la care funcția va completa identificatorul firului nou creat

atribute pentru creare (sau NULL pentru a seta atributele implicite)

argumentul transmis funcției

funcția principală (corpul) firului de execuție (ea poate, evident, apela alte funcții)

→ creează un fir de execuție care va fi imediat lansat în execuție apelându-se funcția start_routine cu parametrul arg.

*Notă: cuvântul **restrict** spune că, pe durata de viață a pointerului respectiv p, doar el sau un pointer exprimat direct folosindu-l pe acesta (cum ar fi p + 1) este singurul pointer spre zona respectivă de memorie. Informația e utilizată de compilator pentru optimizări și e responsabilitatea programatorului să se asigire că ea rămâne adevărată în cod.*

Alte funcții

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);
```

→ așteaptă terminarea firului de execuție cu identificatorul *thread* și îi preia valoarea de terminare la adresa *value_ptr*. Argumentul *value_ptr* poate fi NULL, dacă valoarea de terminare nu e dorită.
→ poate fi apelată de orice fir de execuție din proces
→ dacă apelantul e chiar firul așteptat sau există o dependență circulară, funcția returnează un cod de eroare

```
#include <pthread.h>

void pthread_exit(void *value_ptr);
```

→ termină firul de execuție curent setând valoarea de terminare la *value_ptr*.

Alte funcții

```
#include <pthread.h>

pthread_t pthread_self(void);
```

→ returnează identificatorul firului de execuție curent

```
#include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);
```

→ returnează non-zero dacă t1 și t2 reprezintă același fir de execuție, altfel returnează zero.

Exemplu

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

void *thread_code(void *arg)
{
   int i;

   for(i=0; i<1000; i++)
     printf("%s", (char *)arg);
   printf("\n");

   return (void *)( *((char *)arg) - 'A' + 1);
}

int main(int argc, char *argv[])
{
   pthread_t th1, th2;
   void *ret1, *ret2;

   pthread_create( &th1, NULL, thread_code, (void*) "A");
   pthread_create( &th2, NULL, thread_code, (void*) "B");

   printf("Threads created.\n");

   pthread_join(th1, &ret1);
   pthread_join(th2, &ret2);

   printf("Thread 1 ends returning: %d.\n", (int)ret1);
   printf("Thread 2 ends returning: %d.\n", (int)ret2);
   exit(0);

   return 0;
}
```


Alt exemplu. Există vreo greșeală ?

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

void *thread_code(void *arg)
{
   int i;
   for(i=0; i<1000; i++)
     printf("%c", *((char *)arg) );
   printf("\n");

   return (void *)( *((char *)arg) - 'A' + 1);
}

int main(int argc, char *argv[])
{
   pthread_t th1, th2;
   void *ret1, *ret2;
   char c;

   c='A';
   pthread_create( &th1, NULL, thread_code,  &c);
   c='B';
   pthread_create( &th2, NULL, thread_code, &c);

   printf("Threads created.\n");
   pthread_join(th1, &ret1);
   pthread_join(th2, &ret2);
   printf("Thread 1 ends returning: %d.\n", (int)ret1);
   printf("Thread 2 ends returning: %d.\n", (int)ret2);
   exit(0);

   return 0;
}
```

?

Ce efect are codul următor:

```
for (i=0; i<100; i++)
  pthread_create(&th[i], NULL, thread_code, NULL);
```

?

Tipuri de fire de execuție

- *joinable*
  → valoarea returnată la terminare poate fi preluată de alt thread
  → resursele ocupate de fir nu sunt eliberate până când un alt fir îi preia starea cu join()

- *detached*
  → nu li se poate prelua o valoare de terminare
  → resursele ocupate se eliberează imediat ce firul se încheie

În majoritatea implementărilor, pthread_create() creează în mod implicit fire de execuție *joinable*

Fire de execuție "*detached*"

→ un fir de execuție *joinable* poate fi transformat în *detached* folosind:

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

Exemplu:

```
pthread_detach(pthread_self());
```

→ un fir de execuție poate fi creat direct *detached* setând atribute în parametrul *attr* al funcției *pthread_create()*.

Gestionarea atributelor pentru *pthread_create()*

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);

int pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int detachstate);

int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                int *detachstate);

int pthread_attr_destroy(pthread_attr_t *attr);
```

Notă: există și alte atribute, dar nu fac obiectul acestei discuții

Pași:

1. Se inițializează o variabilă pentru atribute cu *pthread_attr_init()*
2. Se setează atributul dorit apelând funcția *pthread_attr_set…()* corespunzătoare
3. După ce firul a fost creat, se eliberează resursele ocupate de variabila pentru atribute, apelând *pthread_attr_destroy()*.

Exemplu

```
…
pthread_attr_t attr;
…
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
…
pthread_create(…);
…
pthread_attr_destroy(&attr);
…
```

Observații

→ Odată ce un fir de execuție a fost marcat ca *detached*, nu mai poate fi făcut din nou *joinable*
→ Setarea atributului detached se referă doar la modul în care sistemul eliberează resursele alocate thread-ului. Firele *detached* **nu** rămân în sistem după terminarea procesului
→ Pentru fiecare fir creat într-un proces (în afară de cel principal) trebuie apelat ori *pthread_join()*, ori *pthread_detach()*

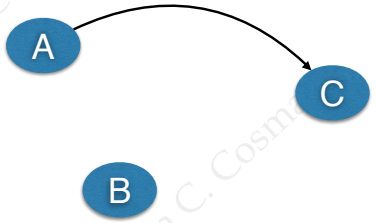
Recapitulare: pipes

→ program format din 3 procese, apelat în linia de comandă astfel:

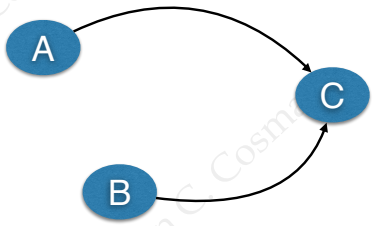
```
program n fișier
```

→ procesul A: folosește o comandă externă pentru a afișa ultimele *n* linii din fișierul dat, pe care le trimite procesului C
→ procesul B: trimite 100 numere aleatoare procesului C și apoi primește rezultatele finale de la acesta, pe care le afișează
→ procesul C preia, pe rând, de la A și B, datele și:
- numără literele mici primite de la A
- găsește numărul par maxim primit de la B
- trimite către B rezultatele, imediat ce sunt disponibile

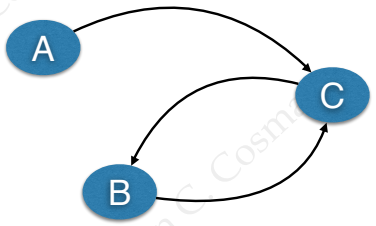
→ procesul A: folosește o comandă externă pentru a afișa ultimele *n* linii din fișierul dat, pe care le trimite procesului C
→ procesul B: trimite 100 numere aleatoare procesului C și apoi primește rezultatele finale de la acesta, pe care le afișează
→ procesul C preia, pe rând, de la A și B, datele și:
- numără literele mici primite de la A
- găsește numărul par maxim primit de la B
- trimite către B rezultatele, imediat ce sunt disponibile



→ procesul A: folosește o comandă externă pentru a afișa ultimele *n* linii din fișierul dat, pe care le trimite procesului C
→ procesul B: trimite 100 numere aleatoare procesului C și apoi primește rezultatele finale de la acesta, pe care le afișează
→ procesul C preia, pe rând, de la A și B, datele și:
- numără literele mici primite de la A
- găsește numărul par maxim primit de la B
- trimite către B rezultatele, imediat ce sunt disponibile



→ procesul A: folosește o comandă externă pentru a afișa ultimele *n* linii din fișierul dat, pe care le trimite procesului C
→ procesul B: trimite 100 numere aleatoare procesului C și apoi primește rezultatele finale de la acesta, pe care le afișează
→ procesul C preia, pe rând, de la A și B, datele și:
- numără literele mici primite de la A
- găsește numărul par maxim primit de la B
- trimite către B rezultatele, imediat ce sunt disponibile



→ procesul A: folosește o comandă externă pentru a afișa ultimele *n* linii din fișierul dat, pe care le trimite procesului C
→ procesul B: trimite 100 numere aleatoare procesului C și apoi primește rezultatele finale de la acesta, pe care le afișează
→ procesul C preia, pe rând, de la A și B, datele și:
- numără literele mici primite de la A
- găsește numărul par maxim primit de la B
- trimite către B rezultatele, imediat ce sunt disponibile


```c
#include <stdio.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <limits.h>
#include <time.h>

int pipe_ac[2], pipe_bc[2], results_pipe[2];

enum processes { COUNTER, MAXRANDOM };
struct result_data {
  enum processes type;;
  long data;
};

void process_c()
{
   int count = 0, n1, n2, index;
   int num;
   long max = LONG_MIN;
   int end1=0, end2=0;
   char c;
   struct result_data res;

   close(pipe_ac[1]);
   close(pipe_bc[1]);
   close(results_pipe[0]);


   while( !(end1 && end2) )
    {
      if(!end1)
      {
        if((n1=read(pipe_ac[0], &c, sizeof(char)))<0)
        {
          printf("Eroare la citire din pipe a-c\n");
          exit(1);
        }

        if(n1 >0)
        {
          if(islower(c))
            count++;
        }
        else
      {
         res.type = COUNTER;
         res.data = count;

         if(write(results_pipe[1], &res, sizeof(struct
result_data))<0)
           {
            printf("Eroare la scriere in pipe rezultate
\n");
           exit(1);
           }
          end1=1;
      }
      }


      if(! end2)
      {
        if((n2=read(pipe_bc[0], &num,
sizeof(long)))<0)
        {
          printf("Eroare la citire din pipe b-c\n");
          exit(1);
        }
        if(n2>0)
        {
          if(num%2 == 0)
          {
            if(num >= max)
              max = num;
          }
          else
      {
         res.type = MAXRANDOM;
         res.data=max;

         if(write(results_pipe[1], &res, sizeof(struct
result_data))<0)
           {
             printf("Eroare la scriere in pipe rezultate
\n");
             exit(1);
           }
           end2=1;
      }

        }
    } /* while */

    close(pipe_ac[0]);
    close(pipe_bc[0]);
    close(results_pipe[1]);
}

int main(int argc, char *argv[])
{
  int pid_a, pid_c;


  if(argc != 3)
  {
    printf("Utilizare: %s nr_linii fisier\n",
argv[0]);
    exit(1);
  }

  if(pipe(results_pipe)<0)
  {
    printf("Eroare la crearea pipe-ului\n");
    exit(1);
  }

  if(pipe(pipe_ac)<0)
  {
    printf("Eroare la crearea pipe-ului a-c\n");
    exit(1);
  }

  if((pid_a=fork())<0)
  {
    printf("Eroare la fork\n");
    exit(1);
  }
  if(pid_a==0) /* a */
  {
    close(pipe_ac[0]);
    close(results_pipe[0]); close(results_pipe[1]);

    dup2(pipe_ac[1],1);

    execlp("tail", "tail", "-n", argv[1], argv[2],
NULL);
    printf("Eroare la exec\n");
    exit(1);
  }

  /* b */
  if(pipe(pipe_bc)<0)
  {
    printf("Eroare la crearea pipe-ului b-c\n");
    exit(1);
  }

  if((pid_c=fork())<0)
  {
    printf("Eroare la fork\n");
    exit(1);
  }

  if(pid_c==0) /* c */
  {
    process_c();
    exit(0);
  }

  /* b */
  close(pipe_bc[0]);
  close(pipe_ac[0]); close(pipe_ac[1]);
  close(results_pipe[1]);

  unsigned int i;
  long num;
  srand((int)time(NULL));
  for(i=0; i<100; i++)
  {
    num = 3*rand();
    if(write(pipe_bc[1], &num, sizeof(long))<0)
    {
      printf("Eroare la scriere in pipe b-c\n");
      exit(1);
    }
  }

  close(pipe_bc[1]); /* important sa fie aici */

  /* Procesul parinte (b) preia rezultatele */
  struct result_data res;

  for(i=0; i<2; i++)
  {
    if(read(results_pipe[0], &res, sizeof(struct
result_data))<0)
      {
       printf("Eroare la citire din pipe rezultate\n");
       exit(1);
      }
    printf("Rezultat primit: %s = %ld\n",
res.type==COUNTER ? "Nr. litere mici" : "Numar generat
maxim", res.data);
  }


  close(results_pipe[0]);

  int s;

  wait(&s); wait(&s); /* preluarea starii fiilor - in
mod normal ar trebui facuta aici si verificarea
starilor de return */

  return 0;
}
```


Cum ați rezolva următoarea problemă ?

→ Program format din mai multe procese.

→ Un număr variabil de procese „producător” care generează, concurent, date. Producătorii sunt de mai multe tipuri (categorii) clar definite.
→ Un număr de procese „consumator” egal cu numărul de tipuri de producători existenți, fiecare consumator fiind responsabil de una din categoriile de producători.
→ Datele de la producători trebuie să ajungă numai la consumatorul care e responsabil de tipul respectiv de producători.

8. Concepte avansate

Drepturi, utilizatori, identificatori

Identificatori asociați oricărui proces

→ real User ID, real Group ID: proprietarul real al procesului
→ effective User ID, effective User ID: utilizatorul/grupul cu a cărui identitate rulează procesul (poate fi diferită de cea reală)
→ supplementary Group IDs: grupurile din care utilizatorul face parte
→ saved set user-ID, saved-set-group-ID: còpii ale ID-urilor, salvate de exec

Când un program de pe disc e lansat în execuție

→ fișierul de pe disc are un user și grup proprietar

- de obicei, effective UID/GID sunt egale cu real UID/GID ale procesului curent (care lansează programul)
- între modurile (drepturi etc.) memorate pentru un fișier de pe disc există și două fanioane (flag-uri) speciale:
  - set-user-ID (SETUID): dacă e setat (activat) pentru un program, programul va fi lansat în execuție setându-i-se effective UID la valoarea UID a proprietarului fișierului (în loc de cea a procesului lansator)
  - set-group-ID (SETGID): dacă e setat pentru un program, programul va fi lansat în execuție setându-i-se effective GID la valoarea GID a grupului proprietar al fișierului (în loc de cea a procesului lansator)

Exemplu:

→ program care rulează cu drepturi de root, deși a fost lansat de un utilizator obișnuit (exemplu din Linux):

```
> ls -l /usr/bin/passwd
-rwsr-xr-x 1 root shadow 81792 oct 29  2011 /usr/bin/passwd
```

Cum se activează SETUID și SETGID:

```
> chmod 4766 fisier
```
```
> chmod u+s fisier
```
SETUID

```
> chmod 2766 fisier
```
```
> chmod g+s fisier
```
SETGID

```
> chmod 6766 fisier
```
```
> chmod ug+s fisier
```
ambele

Schimbarea identității (UID, GID) reale/efective a procesului

E necesară atunci când
- procesul are nevoie de mai multe drepturi pentru a efectua operații privilegiate
- procesul își reduce privilegiile pentru a preveni accesul la unele resurse

E recomandată strategia adoptării privilegiilor minime, adică un proces să își păstreze întotdeauna cele mai puține drepturi necesare efectuării sarcinilor care îi revin

Schimbarea identității e supusă unor reguli stricte.

Funcțiile setuid, setgid

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

int setuid(uid_t uid);
int setgid(gid_t gid);
```

Reguli:

1. Dacă procesul are drepturi de root (superuser):
   real UID ← *uid*, effective UID ← *uid*, saved SETUID ← *uid*
2. Dacă procesul nu are privilegii de root ȘI
   (uid == effective UID SAU uid == saved SETUID):
   effective UID ← *uid*
3. Altfel, funcțiile returnează -1 și errno e setată pe EPERM

(similar și pentru grupuri, cu setgid())

Observații:

- numai un proces cu drepturi de root poate schimba real UID/GID

- când procesul cu drepturi de root folosește setuid, setgid, toate cele trei tipuri de identificatori sunt modificate, astfel încât procesul respectiv nu mai poate reveni la drepturile de root.
  → util atunci când un program privilegiat (exemplu: login) lansează un program al utilizatorului, care nu mai are voie să primească privilegii ridicate niciodată
  → de altfel, root nu are alt motiv de a apela setuid, decât de a reduce permanent privilegii
  → dacă dorește reducerea doar temporară a drepturilor, se folosește altă funcție (seteuid)

Funcțiile exec

a) dacă pentru fișierul executabil există activat flag-ul SUID:
   exec setează effective UID pe UID-ul proprietarului fișierului
b) întotdeauna, exec salvează effective UID în saved SUID;

Operația b) se face după efectuarea lui a) (dacă e cazul), deci id-ul salvat e cel preluat de la fișierul executabil

Funcțiile seteuid, setegid

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

int seteuid(uid_t euid);
int setegid(gid_t egid);
```

Setează doar effective UID/GID, chiar dacă procesul e privilegiat.

Un proces care nu are privilegii de root poate să seteze atributul doar pe valorile real UID/GID sau saved SUID/SGID deja asociate procesului.

Exemplu*

Comanda **at** care planifică executarea unor programe în viitor.

Probleme legate de securitate:
- **at** trebuie să ruleze cu privilegiile specifice utilizatorului, cât mai mult timp posibil
- are nevoie să acceseze fișiere de configurare sistem, deci la un moment dat va avea nevoie de privilegii sporite
- programul care va fi lansat va trebui să ruleze exclusiv cu drepturile utilizatorului care l-a planificat

Există două componente ale sistemului descris:
- comanda **at**, folosită pentru a specifica programe și data lansării lor
- serviciul **atd** (rulează în background), care lansează programele la momentele planificate

*După: W.R.Stevens, S.A.Rago, *Advanced Programming in the UNIX Environment, Third Edition;* Addison Wesley, 2013

În unele sisteme (ex.: Linux 3.1) fișierul /usr/bin/at are setat SETUID, iar owner-ul e root. Sunt parcurși următorii pași:

1. La lansarea **at**, deoarece e setat SETUID pt root, atributele procesului sunt:
   - real UID == UID-ul utilizatorului care a lansat **at**
   - effective UID = root
   - saved SUID = root
2. La început, **at** își reduce privilegiile pentru a rula cu drepturile userului care l-a lansat. Apelează pentru aceasta seteuid(). Avem:
   - real UID == UID-ul utilizatorului care a lansat **at**
   - effective UID = UID-ul utilizatorului care a lansat **at**
   - saved SUID = root
3. După un timp, **at** are nevoie de drepturi sporite. Apelează seteuid() pentru a primi drepturi de root. Are voie, pentru ca root a fost salvat în saved SUID (acest caz arată utilitatea existenței lui saved SUID):
   - real UID == UID-ul utilizatorului care a lansat **at**
   - effective UID = root
   - saved SUID = root

4. după ce a terminat de accesat fișierele de configurare, revine la privilegiile utilizatorului care l-a lansat, apelând iar seteuid():
   - real UID == UID-ul utilizatorului care a lansat **at**
   - effective UID = UID-ul utilizatorului care a lansat **at**
   - saved SUID = root
5. Serviciul **atd** este un program care rulează în sistem cu privilegii de root. Când urmează să lanseze programul planificat de utilizator, trebuie să o facă în așa fel încât acesta să ruleze strict cu drepturile utilizatorului.

   Pentru a lansa programul, **atd** apelează fork(). Apoi, procesul fiu apelează *setuid(uid-ul_utilizatorului_care_planificase_programul)*. Deoarece procesul e cu drepturi de root, toate cele trei tipuri de UID sunt setate pe UID-ul utilizatorului. Este apoi lansat, de procesul fiu, programul planificat.
   - real UID == UID-ul utilizatorului care a lansat **at**
   - effective UID = UID-ul utilizatorului care a lansat **at**
   - saved SUID = UID-ul utilizatorului care a lansat **at**

Drepturi asociate fișierelor

Dreptul de acces la fișiere al proceselor

Când un proces încearcă să creeze/modifice/citească/șteargă un fișier, sunt făcute următoarele verificări, în ordine*:
- dacă effective UID al procesului e root, accesul e permis;
- altfel, dacă effective UID == ID-ul proprietarului fișierului, accesul e permis numai dacă biții de permisiune corespunzători operației sunt setați (cei din categoria “user”), altfel accesul e refuzat;
- altfel, dacă effective GID sau unul din supplementary GIDs este egal cu GID-ul fișierului, accesul e permis numai dacă sunt setați biții de permisiune pentru grup corespunzători operației, altfel accesul e refuzat;
- altfel, dacă biții corespunzători din categoria “other” sunt setați, accesul e permis, altfel accesul e refuzat;

* Prima regulă care se potrivește se aplică, apoi nu se mai trece la celelalte reguli.

Proprietarul unui fișier nou creat

Când un fișier e creat:
— proprietarul (UID) fișierului e setat ca fiind effective UID al procesului care creează fișierul
— grupul proprietar (GID) al fișierului e setat în funcție de versiunea de UNIX sau de opțiunile specificate la montarea sistemului de fișiere. Poate fi unul din:
- effective GID al procesului care creează fișierul
- GID-ul directorului în care fișierul e creat (Linux: face așa dacă directorul respectiv are setat flag-ul SETGID)

Funcția fcntl

Funcția fcntl

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

int fcntl(int fd, int cmd, ... /* arg */ );
```

Aplică diverse operații pe descriptorul fd. Valoarea de return depinde de operația aplicată. La fel și parametrii. Returnează -1 în caz de eroare.

Operații (argumentul cmd):
- F_DUPFD (long) — duplică fd și returnează noul descriptor, care va fi cel mai mic număr disponibil (nedeschis) mai mare decât arg (considerat ca long)
- F_DUPFD_CLOEXEC (long) — duplică fd (ca mai sus) și setează flag-ul FD_CLOEXEC. (close on exec, adică descriptorul va fi închis la exec)
- F_GETFD (void) — citește flag-urile descriptorului; arg e ignorat; în prezent, doar fanionul FD_CLOEXEC e definit
- F_SETFD (long) — setează flag-urile descriptorului pe valoarea dată în arg
- F_GETFL (void) — obține flag-urile de stare ale fișierului; e vorba de cele care sunt inițializate și de open(): O_RDONLY, O_RDWR etc.
- F_SETFL (long) setează flaguri de stare; numai unele pot fi modificate, în Linux ele sunt O_APPEND, O_ASYNC, O_DIRECT, O_NOATIME, O_NONBLOCK)
- F_GETLK, F_SETLK, F_SETLKW: preluarea, testarea, eliberarea de lacăte (locks) pentru porțiuni din fișiere (nu fac obiectul acestui curs, detalii în bibliografie)

Non-blocking I/O

Operații de intrare-ieșire fără blocare

Este posibil ca unele operații de i/e care în mod normal se fac cu blocare (ex: read, write) să se facă fără blocare (ex: read citește, dar dacă nu sunt disponibile imediat date, se întoarce fără să le aștepte)

Două metode:
- La open() se setează și flag-ul O_NONBLOCK
- Se folosește fcntl, adăugându-se flag-ul O_NONBLOCK pentru un descriptor deja deschis

Operații de intrare-ieșire fără blocare

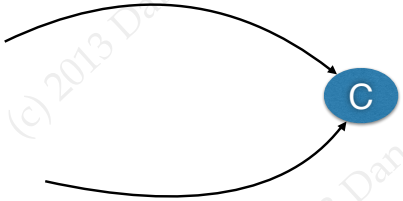
Efect:
- operațiile se vor efectua fără blocare, apelurile se vor întoarce imediat
- operațiile (exemplu: read) pot returna eroare, dar setând errno pe EAGAIN — aceasta înseamnă că operația nu a reușit imediat dar poate continua (pt. read(): nu au fost disponibile date în acel moment)

Exemplu

```
int oldflags;
...
if ((oldflags = fcntl(fd, F_GETFL, 0)) < 0)
  { printf("Error at fcntl\n"); exit(1); }
if (fcntl(fd, F_SETFL, oldflags | O_NONBLOCK) < 0)
  { printf("Error at fcntl\n"); exit(1); }
...
while(1)
{
  do_stuff();

  size = read(fd, buff, expected_size);
  if(size < 0)
  {
    if(errno == EAGAIN)
       continue; /* or do other stuff */
    else
      { printf("Error at read\n"); exit(1); }
  }
  else
  if(size == 0) /* end of file */
    break;
}
```


I/O Multiplexing

Multiplexarea operațiilor de intrare-ieșire




```
…
read(pipe_ac[0], &c, sizeof(char))
…
read(pipe_bc[0], &num, sizeof(long))
…
```

Probleme?

Funcția *select()*

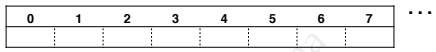
Urmărește seturi de descriptori blocându-se până când cel puțin unul din ei devine pregătit pentru operația de intrare-ieșire

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

int select(int nfds, fd_set *readfds, fd_set *writefds,
                     fd_set *exceptfds, struct timeval *timeout);
```

- readfds, writefds, exceptfds = seturile de descriptori de fișier urmăriți
  → readfds: pentru citire, writefds: pentru scriere, exceptfds: excepții*
  → pot fi NULL dacă nu se dorește urmărirea operației respective
- timeout: timpul maxim de așteptare (NULL = nelimitat)
- nfds: valoarea (numerică) maximă a descriptorilor urmăriți plus unu (trebuie calculată)

*condiții excepționale — în prezent singura condiție excepțională este prezența de date “out of band” pe un socket; nu face obiectul acestui curs

Setul poate fi văzut ca un tablou de biți, câte unul pentru fiecare descriptor posibil



Seturile de descriptori se completează folosind macro-uri speciale. Inițial se apelează obligatoriu FD_ZERO și apoi se setează (adaugă) descriptorii doriți cu FD_SET

```
void FD_CLR(int fd, fd_set *set); - șterge toți descriptorii
int  FD_ISSET(int fd, fd_set *set); - verifică dacă un descriptor e în set
void FD_SET(int fd, fd_set *set); - setează (adaugă) un descriptor în set
void FD_ZERO(fd_set *set); - inițializează setul de descriptori
```


Când funcția select() se întoarce datorită faptului că unul sau mai mulți descriptori au devenit pregătiți, ea va reinițializa cele trei seturi, introducând (setând) în ele doar descriptorii care au devenit pregătiți

Valoarea de retur a funcției select():

-1: eroare sau a apărut un semnal (caz în care errno va fi EINTR)
0: select s-a întors din cauză că a expirat timpul limită
> 0: succes, returnează numărul total de descriptori pregătiți, disponibili în cele trei seturi

Un descriptor este considerat pregătit, în funcție de setul în care se află, astfel:

- *readfds:* un imediat apel read pe acel descriptor nu se va bloca (sunt disponibile date)

- *writefds:* un apel write pe acel descriptor nu se va bloca

- *exceptfds:* a apărut o condiție excepțională pe acel descriptor

Alte observații:

- pentru fișiere obișnuite, descriptorii sunt considerați întotdeauna pregătiți
- la “sfârșit de fișier”, descriptorul e considerat pregătit

Exemplu

```
int nfds = 1 + (pipe_ac[0]>pipebc[0] ? pipe_ac[0] : pipe_bc[0]);
…
int over=0, n;
while (1) {
…
FD_ZERO(&readfds);
FD_SET(pipe_ac[0], &readfds);
FD_SET(pipe_bc[0], &readfds);

if((nready = select(nfds, &readfds, NULL, NULL, NULL))<0)
   { printf(“Eroare\n”); exit(1); }

if(FD_ISSET(pipe_ac[0], &readfds))
{
   n = read(pipe_ac[0], &c, sizeof(char));
   if(n==0) over++; /* end of data */
   …
}
if(FD_ISSET(pipe_bc[0], &readfds))
{
   n = read(pipe_bc[0], &num, sizeof(long));
   if(n==0) over++; /* end of data */
    …
}
if(over == 2)
   break;
…
}
```

Grupuri de procese, job-uri

Grupuri de procese

= o mulțime de unul sau mai multe procese, de obicei asociate cu un același *job*.

Orice proces poate aparține unui grup de procese.

Orice grup de procese are un identificator (GID). Procesul cu identificatorul egal cu cel al grupului se consideră a fi "liderul grupului".

După fork(), procesul fiu moștenește GID-ul părintelui (se află în același grup)

Aflarea identificatorului de grup:

```
#include <unistd.h>

pid_t getpgid(pid_t pid);
  dacă pid==0, returnează grupul procesului curent

pid_t getpgrp(void);
  returnează grupul procesului curent
```


Job-uri

Un *job* este un grup de procese care poate fi controlat în cadrul oferit de interpretorul de comenzi. Nu toate interpretoarele oferă controlul job-urilor.

Un grup de procese e creat de obicei de o înlănțuire cu pipe-uri, de linii de comandă lansate în background etc. Exemplu: liniile următoare creează 2 grupuri de procese și, implicit, 2 job-uri:

```
$ ls -l | grep abc &
[1] 1165
$ ls &
[2] 1166
(apăsarea tastei ENTER)
[1]   Exit 1                  ls -l | grep abc
[2]+  Done                    ls
```

Controlul job-urilor

```
$ program
```
CTRL+Z: suspendarea job-ului din foreground (SIGTSTP)
CTRL+C: terminarea (SIGINT) job-ului din foreground
CTRL+\: terminarea (SIGQUIT) job-ului din foreground

`$ bg %1` trimiterea în background și reluarea jobului suspendat 1
`$ bg` trimiterea în background și reluarea ultimului job suspendat

`$ fg %1` aducerea în foreground a jobului 1

Crearea unui grup și aderarea proceselor la grupuri

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

Setează grupul pentru un proces dat.

- dacă pid == 0, setează grupul procesului curent
- dacă pid==pgid, procesul pid devine liderul unui nou grup
- dacă pgid == 0 atunci grupul procesului pid va fi făcut egal cu pid (procesul pid devine liderul unui nou grup)
- altfel, pgid trebuie să fie un grup existent, din aceeași sesiune cu cel curent

Un proces poate apela setpgid doar pentru el însuși și pentru oricare din fiii săi care nu a apelat încă exec

Un proces poate trimite un semnal unui întreg grup de procese.

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

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

→ pid>0: Trimite semnalul sig către procesul pid
→ pid=0: Trimite semnalul sig către grupul curent de procese
→ pid<0: Trimite semnalul sig către grupul de procese pid

Sesiuni

Sesiune

= o mulțime de unul sau mai multe grupuri de procese

O sesiune are cel mult un *terminal de control*, care este un dispozitiv capabil să afișeze date și să ofere intrare (tastatură)

Un terminal de control poate fi asociat unei sesiuni la cererea procesului “lider de sesiune”, numai dacă acest terminal nu a fost asociat deja. Modul în care poate fi cerut un terminal diferă de la o variantă de UNIX la alta. La login se alocă automat un terminal.

Dacă sesiunea are terminal de control:
- Un singur grup de procese din sesiune este grupul foreground. Doar acest grup va putea citi de la terminal (tastatură). Toate procesele din acest grup vor fi afectate de CTRL-C, CTRL-\ (vor primi semnalele corespunzătoare)
- Toate celelalte grupuri de procese sunt grupuri background. Un read() de la terminal făcut de un grup background va suspenda grupul respectiv (grupul va primi semnalul SIGTSTP)

Crearea unei noi sesiuni

```
#include <unistd.h>

pid_t setsid(void);
```

Dacă procesul apelant NU e lider de grup, este creată o nouă sesiune:
- sesiunea NU va avea terminal de control
- procesul devine “lider al sesiunii”
- procesul devine lider al unui nou grup de procese, primul din sesiune
- dacă procesul avea terminal de control înainte de apel, conexiunea cu acesta este întreruptă

Dacă procesul apelant este lider de grup, funcția returnează eroare (-1).