

# Sisteme de operare

Laborator 1

Semestrul I 2024

# Prezentare generala laborator

- programare de sistem in Linux (clona Unix)
- “Advanced Programming in the Unix Environment”, W. Richard Stevens
- gestiunea fisierelor si I/O
- controlul proceselor (+threads)
- semnale
- IPC (interprocess communication)
  - semafoare System V, monitoare pt pthreads
  - shared memory System V vs POSIX
  - programare client-server (TCP/IP sockets, RPC)
- advanced topics

# Laborator 1

- notiuni introductive
- pornirea sistemului (procesul de boot)
- procesul de login utilizator
- interpretorul de comenzi
- interfata cu nucleul (kernelul) sistemului de operare

# Procesul de boot

- primul sector al discului de boot (MBR, respectiv succesorul sau GPT)
  - tabela de partitii de disc
  - cod de bootstrap (boot loader)
- loader-ul identifica partitia de boot, si incarca codul kernel (nucleul sistemului de operare)
- Obs: la acest nivel nu exista notiunea (abstractia) de fisier, ci doar sectoare de disc; solutii posibile
  - 1. loader-ul cunoaste o harta a sectoarelor de disc care contin codul kernel
    - solutie hardcoded
    - implica actualizari alei hartilor atunci cand imaginea kernelului pe disc se schimba, la defragmentarea discului, etc
  - 2. loader-ul are acces la drivere care inteleg structura sistemului de fisiere de pe disc
    - identifica kernelul ca pe un fisier oarecare (folosind calea)
- ex boot loaders Linux: Lilo, Grub

# Procesul de boot (cont.)

- odata identificat fisierul care contine imaginea kernelului (eg, */boot/vmlinuz* pt Linux), se incarca kernelul in memorie si i se da controlul
- subsecvent, kernelul:
  - initializeaza drivere pt componentele HW
  - instantiaza principalele servicii SO: controlul proceselor, gestiunea memoriei, gestiunea fisierelor, accounting, gestiunea timpului sistem, mecanismele de protectie HW si de securitate, etc
  - ramane rezident in memorie in asteptarea unor evenimente externe (“program interrupt-driven”)
  - la sfarsitul secventei de initializare executa primul proces, procesul cu ID = 1: */sbin/init*
- *init* este responsabil (printre altele) cu pornirea proceselor de *login* pt utilizator, */sbin/getty* respectiv desktop manager-ul in cazul interfetelor grafice de tip X Window
  - istoric (in sistemele Unix), *init* cauta in */etc/inittab* asocierea terminal – program de login (*getty* sau *xm/gdm*)
  - sistemele Linux moderne au un director */etc/init* care contine toate fisierele de configurare ale diverselor servicii

# Logarea utilizatorului

- *getty* afiseaza prompt-ul de login
- utilizatorul introduce numele de utilizator
- *getty* apeleaza programul */bin/login* cu numele de utilizator ca parametru
- *login*: stabileste o noua sesiune de lucru
  - afiseaza promptul pt parola
  - cauta in */etc/passwd* o intrare corespunzatoare numelui de utilizator
  - verifica parola (de regula stocata criptat in alt fisier, eg. */etc/shadow*)
  - daca parola e corecta
    - executa interpretorul de comenzi (*shell*-ul) asociat intrarii identificate
    - ii asociaza *shell*-ului variabile de mediu (environment) initializate cu valorile din campurile citite din intrarea corespunzatoare din */etc/passwd* (in mod notabil USER, SHELL, HOME)
- *shell*-ul afiseaza un prompt specific (\$) pt utilizator obisnuit) si asteapta comenzile utilizatorului
- *init* monitorizeaza sesiunea de lucru a utilizatorului si cand acesta incheie activitatea (paraseste *shell*-ul), reporneste o instanta a programului *getty* pe terminalul respectiv

# Interpretorul de comenzi, shell

- utilizabil deopotriva in mod interactiv cat si batch (folosind *shell script*-uri)
- ofera posibilitatea de a executa atat comenzi interne (executate in cadrul interpretorului) cat si externe (programe incarcate de pe disc)
- functionalitati principale
  - asigurarea unui mediu de lucru utilizatorului (v. comanda *env*)
  - comenzi de manipulare a fisierelor si directoarelor
  - comenzi de control al executiei programelor
  - controlul si monitorizarea activitatilor de I/O
  - administrarea sistemului (rezervata unui utilizator special cunoscut in mod uzual sub numele de *root*, cu `UID = 0`, v. prima intrare din */etc/passwd*)
  - *samd.*
- ex. interpretoare de comenzi: Bourne Shell (*/bin/sh*), Bourne Again Shell (*/bin/bash*), C Shell (*/bin/csh*), Korn Shell (*/bin/ksh*), etc.

*/etc/shells* contine interpretoarele de comenzi disponibile in sistem

# Fisiere si directoare

- sistemele tip Unix folosesc o structura ierarhica de directoare
  - incepe dintr-un director special numit *root* (radacina), desemnat prin caracterul “/”
- un director este un fisier care contine intrari de director
  - nume fisier: poate contine orice caracter mai putin “/”
  - attribute fisier: tip, dimensiune, proprietar, permisiuni, timpul ultimei modificari, etc
- directoare speciale create automat atunci cand se creeaza un nou director
  - . directorul current (directorul nou creat)
  - .. directorul parinte (directorul in care a fost inserata o noua intrare corespunzatoare noului director creat)
  - in cazul directorului radacina ./ si ../ reprezinta acelasi director, si anume /



# Fisiere si directoare (cont.)

- cale (path): secventa de nume de directoare si fisiere separate de caracterul /
  - cai absolute: incep intotdeauna cu /
  - cai relative: nu incep cu /, fiind interpretate relativ la directorul de lucru curent (*current working directory*)
- la login, directorul de lucru curent este setat la valoarea obtinuta din */etc/passwd* pentru utilizatorul logat si care se numeste *home directory*
- comanda de tiparire a intrarilor intr-un director: */bin/ls*
  - *Obs:* un program executabil este si el reprezentat printr-o cale in sistemul de fisiere
  - ex:

```
$ ls      # afiseaza intrarile din directorul current
```

```
$ ls -l   # afisare in format lung, include attributele fisierelor
```

```
$ ls <dir/file>      # afiseaza continutul directorului/fisierului  
                     # specificat
```

```
$ ls -l /etc
```

```
$ ls -l /etc/passwd
```

# Descriptori de fișiere

- întregi nenegativi folosiți pentru identificarea fișierelor deschise în sistem
- când kernelul deschide un fișier existent sau creează unul nou ca urmare a solicitării unui proces întoarce un descriptor de fișier care va putea fi folosit de proces pentru citirea/scrierea fișierului
- descriptori speciali
  - la pornirea oricărui program, shell-ul deschide pentru acesta trei descriptori de fișiere speciali:
    - 0 standard input
    - 1 standard output
    - 2 standard error
  - uzual, cei trei descriptori de fișier sunt asociați cu terminalul de login (sau terminalul de lucru, într-un mediu grafic cu multiple X terminale)
  - pentru a afla care este terminalul asociat unui shell (în general, nu doar terminalul de login) se folosește comanda `/usr/bin/tty`

# Redirectarea operatiilor de I/O

- redirectarea operatiilor de I/O se poate face programatic sau direct din shell
- shell-ul intelege constructii sintactice de tipul urmator ca fiind redirectari ale operatiilor de I/O

[n] < filename     redirecteaza citirile de pe descriptorul n catre  
fisierul desemnat; daca n lipseste, se foloseste *stdin*

[n] > filename     redirecteaza scrierile pe descriptorul n catre fisierul  
desemnat; daca n lipseste, se foloseste *stdout*

[n] >> filename    adauga scrierile pe descriptorul n la sfarsitul  
fisierului desemnat ("append"); daca n lipseste, se  
foloseste *stdout*

- ex:

\$ echo "redirectarea iesirii in fisierul cu numele out" > out

# Programe si procese

- program = fisier executabil stocat pe disc (mediu de stocare persistenta, in general)
- proces = imaginea in memorie a unui program, o instanta in executie a unui program
- unele sisteme de operare (in special cele de timp real) folosesc denumirea de task pentru procese
- orice proces dintr-un sistem Unix/Linux are asociat un numar intreg nenegativ unic in sistem numit PID (Process ID)
- mai multe instante in executie (procese) ale aceluiasi program au PID-uri diferite
- comanda shell cu care se poate afla asocierea dintre PID si un proces este */bin/ps*
  - Obs: */bin/ps* este un program executabil de pe disc, odata ce o instanta a sa se incarca in memorie, respectiva instanta a programului *ps* va avea propriul PID

\$ ps -auxw      # afiseaza informatii despre toate procesele active in sistem momentan

# Identificarea utilizatorului

- la login, utilizatorul primește un ID propriu (valoare întreaga nenegativă care identifică utilizatorul în sistemul de operare)
  - obținut din intrarea corespunzătoare utilizatorului din */etc/passwd*
  - în general unic
  - asignat de către administratorul sistemului, singurul care are permisiunea de a scrie în */etc/passwd*, și nu poate fi schimbat de către utilizator
  - folosit de kernel pentru a verifica dacă utilizatorul (mai exact procesele sale) are dreptul să execute anumite operații
- utilizatorul cu UID = 0 s.n. *root* sau *superuser*
  - intrarea corespunzătoare din */etc/passwd* folosește de regulă numele de *root*
- procesele *root* au privilegii de *superuser* și de regulă circumventează verificările pe care kernelul le face pentru o serie de operații
- unele dintre funcțiile kernelului pot fi executate doar de procese *root*
- *root*-ul are control total asupra sistemului de calcul
  - Obs: din acest motiv, este puternic descurajată inițiativa utilizatorilor sistemului care știu parola de *root* să ruleze programe obișnuite în calitate de *root* (UID = 0)

# Identificarea utilizatorului (cont.)

- la login utilizatorul nu primește doar un UID ci și un GID (Group ID), setat tot de administrator în intrarea corespunzătoare din */etc/passwd*
  - GID-ul permite partajarea de resurse între membri ai aceluiași grup, chiar dacă au UID-uri diferite
  - De ex, intrările de director pentru fiecare fișier din sistem conțin perechea (UID,GID) a proprietarului fișierului respective
  - comanda shell *ls -l* permite afișarea ID-urilor proprietarului fișierului
  - în schimb, utilizatorii cu GID diferit nu pot accesa aceste resurse partajate ale grupului
- fișierul */etc/group* asignează nume lizibile GID-urilor utilizator;
  - modificabil doar de către administratorul sistemului
  - listează și *supplementary GIDs*, i.e. același utilizator poate avea mai multe GID-uri (poate aparține mai multor grupuri)
- comanda shell cu care se pot afla UID/GID : */usr/bin/id*

# Semnale

- notificari asincrone ale procesului referitoare la producerea anumitor evenimente (reprezentare software a exceptiilor)
  - ex: procesul care executa o impartire la zero primeste un semnal SIGFPE
- procesul are trei optiuni de tratare a situatiei (tratarea semnalului)
  - poate ignora semnalul; politica nerecomandata pt semnale care notifica exceptii hardware (impartire la zero, accesul unei zone de memorie nealocata, etc)
  - poate lasa sa se execute actiunea implicit asociata semnalului primit (eg., impartirea la zero termina programul in mod implicit)
  - poate furniza o rutina de tratare a semnalului (signal handler) care se va apela la primirea semnalului (se spune ca procesul "prinde" semnalul)
- semnalele pot fi generate voluntar de catre utilizator
  - ex: ctrl-C genereaza SIGINT care in mod normal termina procesul rulat de shell;
  - cu ajutorul comenzii shell *kill* (comanda interna shell):
 

```
kill -<signo> <pid>
$ kill -SIGINT <pid>          # kill -2 <pid>, genereaza CTRL-C pt un anumit
                              # proces
```
  - programatic, cu ajutorul apelului sistem *kill*

# Apeluri sistem

- toate procesele din sistem (inclusive comenzile shell, interne sau externe) pot solicita serviciile kernelului prin intermediul apelurilor sistem (*system calls*)
  - intrari in codul kernel (nr limitat)
  - apelabile in mod uzual prin intermediul bibliotecii *libc* care este linkeditata automat (cel mai adesea dinamic) la orice program executabil
  - *libc* contine stub-uri cu acelasi nume ca si apelurile de sistem,
    - stub-urile executa o instructiune privilegiata de tip trap (*syscall*) sau intrerupere software (*int 21h* in Linux)
- dpdv programator, apelurile sistem pot fi asimilate unor simple apeluri de biblioteca
- dpdv dezvoltator de cod kernel diferenta e fundamentala
- ex: *printf/cout* apeleaza *write*, *malloc/new* apeleaza *brk/sbrk*, *rand*
  - *write*, *brk* sunt apelurile sistem (de fapt stub-uri in *libc*)
  - *printf/malloc* sunt apeluri de biblioteca



# Tratarea erorilor

- diferenta notabila intre apelurile sistem si apelurile obisnuite de biblioteca dpdv al utilizatorului: *tratarea erorilor*
- apel sistem terminat cu eroare => valoare de return negativa si variabila *errno* setata pe o anumita valoare
  - o valoarea de return zero semnifica in general succes
- programatorul are acces la *errno* in C in mod uzual:

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

```
extern int errno;
```

- preferabil, folosirea functiilor de biblioteca de tip *strerror* sau *perror*

# The Bourne-Again SHell

- */bin/bash*, urmasul primului shell istoric, */bin/sh* (Bourne Shell)
- fisiere de configurare
  - fisiere de start-up inspectate doar la login
    - system-wide: */etc/profile*
    - locale, in home directory: *~/.profile*, *~/.bash\_profile*, *~/.bash\_login*
    - continutul lor e executat automat la login
  - fisiere de start-up inspectate la crearea fiecarui terminal (rc file, “run commands”)
    - *~/.bashrc*
    - continutul lor poate fi executa voluntar cu comanda *source* (sau “.”)  

```
source .bashrc
```

```
..bashrc
```
  - fisiere de logout: *~/.bash\_logout*
    - continutul este executat la iesirea din shell (cu *exit* sau *^D*)
    - Obs: ctrl-D in Unix este caracterul EOF, cand utilizatorul tipareste *^D* shell-ul se termina
- istoria comenzilor este inregistrata in *~/.bash\_history*

# Structura comenzilor bash

- pipeline-uri

`cmd1 | cmd2 | ... | cmdn` # executie paralela a comenzilor

`cmd1 | & cmd 2 | & ... | & cmdn`

“| &” e totuna cu “2>&1 |”

- liste de comenzi

`cmd1; cmd2; ...; cmdn` # executie secventiala a comenzilor

`cmd1 && cmd2 && ... && cmdn`

`cmd1 || cmd2 .... || cmdn`

- variabila bash “?” contine codul de terminare (“exit status”) al ultimei comenzi executate (valoarea zero inseamna succes)

`echo $?`

# Mediul de lucru (environment)

- o lista de perechi *nume = valoare*
- stocare de shell si disponibile programelor executate din shell
- comanda */usr/bin/env* afiseaza variabilele de mediu
- intr-un program C, variabilele de mediu sunt accesibile in cel de-al treilea parametru al functiei *main*

```
int main(int argc, char* argv[], char *envp[])
```

- cand shell-ul executa programul C, ii furnizeaza mediul de lucru in cel de-al treilea argument al functiei *main*

# Job control

- doua categorii de programe
  - executate in foreground, au acces R/W la terminal
  - executate in background
- comanda care se executa cu un "&" la final se va rula in background (shell-ul returneaza imediat utilizatorului prompt-ul)

\$ cmd &

- unei comenzi executate in foreground i se poate suspenda executia cu ^Z (ctrl-Z)
  - de fapt, e semnalul SIGTSTP (kill -SIGTSTP <pid>)
  - executia comenzii poate fi reluata ulterior, fie in foreground, fie in background
- comanda *jobs* listeaza procesele (joburile) rulate la momentul curent de shell
  - joburile sunt identificate printr-un numar afisat de comanda *jobs*
  - numarul job-ului poate fi folosit impreuna cu urmatoarele comenzi

\$ kill %n	termina procesul/job-ul n
\$ fg %n	muta in foreground procesul n
\$ bg %n sau %n &	muta jobul n in background

# Controlul istoriei comenzilor

- fisierul `~/.bash_history`
- exemple

`!n` re-executa comanda cu nr `n`

`!-n` re-executa comanda curenta – `n`

`!string` re-executa cea mai recenta comanda care incepe cu `string`

`!?string?` re-executa cea mai recenta comanda care contine  
`string`

`^str1^str2` repeta comanda anterioare inlocuind `str1` cu `str2`

- interactiv, utilizatorul poate apasa `ctrl-R` si tipareste un substring al comenzii din istoric pe care o cauta

# Comenzi interne

- executate direct de catre bash (nu sunt programe de pe disc)

- Ex: `cd <dir>` `# alternativ, pushd/popd + dirs.`

`alias`

ex: `alias l='ls -l'`

`fg/bg/kill <job#>`

`exit <status>`

`exec cmd`

`# inlocuieste imaginea bash cu imaginea noului proces`

ex:

`exec firefox`

- Comenzi externe utile

`pwd`

`# impreuna cu cd`

`echo string`

ex escape chars:

`echo -e \a`

`# bell`

`echo -e "aaa\tbbb"`

`# horizontal tab`

`echo -e "aaa\v\bbbb"`

`# vertical tab + backspace`

`echo -e "aaa\t\rbbb"`

`# carriage return`

`echo -e "aaa\t\nbbb"`

`# newline`