Sisteme de operare

Laborator 1

Semestrul I 2023-2024

Vlad Olaru

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 => 2 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 si pot identifica astfel kernelul ca pe un fisier oarecare (folosind pathname)
- 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 executa:
  - secventa de intializare a componentelor HW ("instaleaza driverele echipamentelor")
  - instantiaza principalele componente care ofera serviciile de sistem: 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 (inspirat de sistemele Unix), *init* cauta in */etc/inittab* asocierea terminal – program de login (*getty* sau *xdm/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
- ca raspuns, *getty* apeleaza programul */bin/login* care stabileste o noua sesiune de lucru
- *login* afiseaza promptul de parola; dupa introducerea parolei
  - 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 si ii asociaza acestuia variabile de mediu (environment) intializate cu valorile din campurile citite din intrarea corespunzatoare din */etc/passwd*, in mod notabil USER, SHELL, HOME
- shell-ul afiseaza un prompt specific 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

- 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.

Fisiere si directoare

- sistemele Linux (Unix in general) folosesc o structura ierarhica de directoare care 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 "/"
  - atribute 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 /
- cale (path): secventa de nume de 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* (dupa cum se observa, un program executabil este si el reprezentat printr-o cale in sistemul de fisiere)

Descriptori de fisiere

- intregi nenegativi folositi pentru identificarea fisierelor deschise in system
- cand kernelul deschide un fisier existent sau creaza unul nou ca urmare a solicitarii unui proces intoarce un descriptor de fisier care va putea fi folosit de proces pentru citirea/scrierea fisierului
- descriptori speciali
  - la pornirea oricarui program, shell-ul deschide pentru acesta trei descriptori de fisiere speciali:
    - 0 standard input
    - 1 standard output
    - 2 standard error
  - uzual, cei trei descriptori de fisier sunt asociati cu terminalul de login (sau terminalul de lucru, intr-un mediu grafic cu multiple X terminale)
  - pentru a afla care este terminalul asociat unui shell (in general, nu doar terminalul de login) se foloseste 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 stdout” > fisier_destinatie

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

Identificarea utilizatorului

- la login, utilizatorul primeste un ID propriu care este o valoare intreaga nenegativa prin care utilizatorul este indentificat in sistemul de operare
- user ID-ul (UID) este obtinut din intrarea corespunzatoare utilizatorului din /etc/passwd
- acest ID, in general unic, este asignat de catre administratorul sistemului, singurul care are permisiunea de a scrie in /etc/passwd, si nu poate fi schimbat de catre utilizator
- UID-ul este folosit de kernel pentru a verifica daca utilizatorul (mai exact procesele sale) are dreptul sa execute anumite operatii
- utilizatorul cu UID = 0 s.n. *root* sau *superuser* (intrarea corespunzatoare lui din /etc/passwd foloseste de regula numele de *root*)
- procesele *root* au privilegii de superuser si de regula circumventeaza verificarile pe care kernelul le face pentru o serie de operatii
- unele dintre functiile kernelului pot fi executate doar de procese root
- root-ul are control total asupra sistemului de calcul
  - Obs: din acest motiv, este puternic descurajata initiativa utilizatorilor sistemului care stiu parola de root sa ruleze programe obisnuite in calitate de root (UID = 0)

Identificarea utilizatorului (cont.)

- la login utilizatorul nu primeste doar un UID ci si un GID (Group ID), setat tot de administrator in intrarea corespunzatoare din /etc/passwd
- GID-ul permite poartajarea de resurse intre membri ai aceluiasi grup, chiar daca au UID-uri diferite
  - De ex, intrarile de director pentru fiecare fisier din system contin perechea (UID,GID) a proprietarului fisierului respective
  - comanda shell *ls –l* permite afisarea ID-urilor proprietarului fisierului
- in schimb, utilizatorii cu GID diferit nu pot accesa aceste resurse partajate ale grupului
- fisierul */etc/group* asigneaza nume lizibile GID-urilor utilizator; este de asemenea modificabil doar de catre administratorul sistemului
  - /etc/group listeaza si *supplementary GIDs*, i.e. acelasi utilizator poate avea mai multe GID-uri (poate apartine mai multor grupuri)
- comanda shell cu care se pot afla UID/GID este */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; daca procesul prinde SIGINT, la apasarea ctrl-C se va executa signal handlerul asociat SIGINT de care proces
  - cu ajutorul comenzii shell *kill* (comanda interna): kill -<signo> <pid>

    ctrl-C este echivalent cu kill –SIGINT <pid> sau kill -2 <pid>
  - 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*)
- acestea sunt un numar limitat de intrari in codul kernel, 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-uri care executa o instructiune privilegiata de tip trap (*syscall*) sau intrerupere software (*int 21h* in Linux)
- dpdv al programatorului, apelurile sistem pot fi asimilate unor simple apeluri de biblioteca, dpdv al dezvotatorului de cod kernel diferenta e fundamentala
- ex: *printf* apeleaza *write*, *malloc*/*new* apeleaza *brk*/*sbrk*, samd

Tratarea erorilor

- diferenta notabila intre apelurile sistem si apelurile obisnuite de biblioteca dpdv al utilizatorului: tratarea erorilor
- cand un apel sistem se termina cu o eroare, functia intoarce o valoare negativa si seteaza variabila *errno* pe o anumita valoare
  - o valoarea de return zero semnifica in general succes
- programatorul are acces la aceasta variabila in C in mod uzual:

```
#include <errno.h>

extern int errno;
```

- preferabila este insa 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

  cmd1 |& cmd 2 |& … |& cmdn

  "|&" e totuna cu "2>&1 |"

- liste de comenzi

  cmd1; cmd2; …; cmdn

  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 pereche name = value
- poate fi afisata cu comanda /usr/bin/env
- in cadrul unui program C variabilele de mediu sunt accesibile intr-un array de string-uri pasat de shell in cel de-al treilea parametru al functiei *main* a programului lansat in executie de catre shell

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

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 is se poate suspenda executia cu ^Z (ctrl-Z)
  - executia ei 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)
- cateva exemple utile

cd <dir> + pwd — sau altenativ pushd/popd + dirs

alias — ex: — alias l='ls –l'

fg/bg/kill <job#>

exit <status>

echo string

ex de escape chars:

```
echo –e \a
echo –e "c\tb"
echo –e "c\vb"
echo –e "c\rb"
echo –e "c\nb"
```

exec cmd — inlocuieste imaginea bash cu imaginea noului proces

ex: — exec firefox