**Laborator 1**

Linux = clona Unix

→primul sector al discului de boot(MBR, GPT)

→loader-ul identifica partitia de boot si incarca codul kernel

→la acest nivel nu exista notiunea de fisier ci doar de sectoare de disc => 2

1.loaderul cunoaste harta sectoarelor de disc care contin codul kernel

2. loaderul are acces la drivere care inteleg structura sistemului de fisiere de pe disc si pot identifica aastfel kernelul ca pe un fisier oarecare

Ex: boot loaders Linux: Lilo, Grub

→odata identificat fisierul care contine imaginea kernelului, se incarca kernelul in memorie si i se da controlul

→kernelul executa: secventa de initializare a componentelor HW/ instantiaza principalele componente care ofera serviciile de system/ramane rezident in memorie in asteptarea unor evenimente externe/ la sfarsitul secventei de initializare executa primul process, procesul cu ID=1, /sbin/init

→init este responsabil cu pornirea proceselor de login pt utilizator /sbin/getty

→getty afiseaza promptul de login→apeleza programul /bin/login care stabileste o noua sesiune de lucru

→login afiseaza promptul de parola; daca parola e corecta executa interpretorul de comenzisi asociaza acestia variabile de mediu initializate cu valorile din campurile citite din intrarea corespunzatoare din /etc/passwd, USER,SHELL,HOME

→shell ul afiseaza un prompt specific si asteapta comenzile utilizatorului

→init(cand user paraseste shell) reporneste o instanta a programului getty pe terminalul respective

Interpretorul de comenzi

→interactiv cat si batch

→comenzi interne si externe

→functionalitati: asig mediu de lucru, comenzi de manipulare file si dir, comenzi de control al exe progr, controlul activitatilor I/O, administrarea sistemului pt root (UID=0)

Fisiere si directoare

→Linux foloseste struct ierarhica de directoare incepe din root (“/”)

→un director este un fisier care contine intrari de director

→cai absolute incep intotdeauna cu /

→cai relative: nu incep cu /

→la login directorul de lucru current e setat la valoarea obt din /etc/passwds pt utilizatorul logat si care se numeste home directory

Descriptori de fisiere

→intregi >=0 fol pt identificarea fisierelor deschise in system

→cand kernelul deschide un fisier intoarce un file descriptor

Redirectarea op I/O

→se poate face programmatic sau direct din shell

Programe si procese

→program = fisier executabil stocat pe disc(persistent)

→proces = imaginea in memorie a unui program

→unele so fol denumirea de task pt procese

→orice process are asociat un nr >=0 PID

→mai multe instante in executie ale aceluiasi program au PID difer

→asocierea PID process = /bin/ps

Identificarea Utilizatorului

Login: ID propriu, val >=0

UID = intrarea coresp din /etc/passwd

ID-assignat de admin, care are permisiunea de a scrie in /etc/passwd

UID folosit de kernel pt a verifica dreptul de executare a op

UID = 0 root/superuser

Procesele root au privilegii de superuser

→unele dintre functiile kernelului pot fi executate doar de procese root

→root ul are control total asupra sistemului de calcul

→user primeste si GID – permite partajarea de resurse intre membri aceluiasi grup indif de UID

→fisierul /etc/group asigneaza nume GID urilor – modificabil doar de admin

/usr/bin/id -> UID/GID

Semnale (priveste le ca exceptions de la sgbd)

→notificari asincrone ale procesului referitoare la producerea anuitor evenimente (reprez software a excep)

3 opt de tartare

→ignorare/executie/furnizare rutina de tartare a semnalului

→semnalele pot fi generate voluntar de user (ctrl-C genereaza SIGINT)

Apeluri de system

→toate procesele din system pot solicita kernelului cv prin syscall

→nr limitat de intrari in codul kernel

→printf apeleaza write, malloc/new apeleaza brk/sbrk

Tratarea erorilor

→dif dintre syscall si apeluri usual din bibl: tratarea erorilor

→eroare = valoare negative si seteaza errno pe val

→0=success

→<errno.h> bibl pt erori

→preferabil folosirea strerror sau perror

The BASH

→fisiere de configurare

Fisere de start-up inspectate doar la login

La crearea fiecarui terminal

Fisiere de logout

→istoria comenzilor e inregistrata in ~/.bash\_history

Structura comenzilor bash

→pipeline-uri / liste de comenzi

→? = exit status pt ult cmd

Mediu de lucru

→lista de pereche name=value

→afisata cu /usr/bin/env

→in C var de mediu = array de string uri -> al treilea argument in main

Job Control

→doua categ de program (exec in foreground(au acces RW) sau background)

→& - rulare in background

→cmd in foreground se suspenda cu ctrl-Z, dar poate fi reluata atat in bckg cat si in frgr

→jobs listeaza procesele identificate printr un numar

Kill %n -termina fg %n muta in fg bg %n sau %n & muta in bg

Controlul istoriei comenzilor

→fisierul ~/.bash\_history (!n reia comanda cu nr n// !string reia cea mai recenta cmd care incepe cu string etc)

Comenzi interne

→executate direct catre bash

LABORATOR 2

File descriptors

→fisierele deschise de un process sunt referite in kernel prin FD

→intreg >=0 intors de open/creat

!de scris apelurile

→citire/scriere folosesc FDs nu nume

→open returneaza garantat cel mai mic FD nealocat

!de scris flaguri

→creat fisierul e creat doar pt scriere

!de scris apel

→close executa exit() toate files automat inchise cu close

Pozitionarea File Pointerului

→orice fisier deschis are asociat un offset crt, nr>=0 = nr de octeti

→citire/scriere folosesc implicit offset si ++ corespunzator offsetul current

→initialiat cu 0 dc fisierul nu a fost deschis cu O\_APPEND

→pozitionarea explicita a FP se face cu syscall lseek

!de scris apel

→seek\_\* set-inceput, cur=crt, end=end

→intoarcerea offsetului crt poate fi folosit la testarea file pt seek

→fcntl schimba prop unui fisier deschis

→stat/fstat/lstat -> info despre un fisier

Tipuri de fisiere

→regular files: contin date

→directoare: contin numele altor files + info (dir = doar kernelul poate scrie in ele)

→orice device din sys e fisier block sau caracter

→FIFO = named pipe, mechanism IPC

→socket= abstractie pt IPC

→link symbolic = file care refera un altul

→set-UID/set-GID copii pt UID-GID efective

Permisiuni de acces

→codificate in st\_mode al struct stat

3:

→perm utilizator: S\_IRUSR, S\_IWUSR, S\_IXUSR

→perm grup: S\_IRGRP, S\_IWGRP, S\_IXGRP

→perm pt alti users: S\_IROTH, S\_IWOTH, S\_IXOTH

→succesul accesului e conditionat de combinatia dintr UID si GID efective

→!!! Int access(const char \*pathname, int mode) (or logic intre r\_ok, w\_ok, x\_ok, f\_ok)

→cmask = or intre bitii de permisiune

→chmod si fchmod permit schimbarea permisiunilor de acces: ! de scris apelurile

→stergerea fisierelor int unlink(const char \*pathname)

→stergerea are nev de : permisiune de W si de a cauta in dir

→rmdir pt directoare(doar goale) (int rmdir(const char \*pathname))

→unlink pt fisiere

→ int mkdir(const char \*pathname, mode\_t mode) (mode trb sa include permisiunea X)

Structura citire: DIR \*openddir(const char \*pathname)

Struct dirent \*readdir(DIR \*dp)

Closedir(DIR \*dp);

Schimbare director crt: int chdir (Const char \*pathname)

Int fchdir(int fd)

Dir crt: char \*getcwd(char \*buf, size\_t size)

LABORATOR 3

Primul process init creat de kernel la sf secv boot

Init = stramosul tuturor proceselor din system

Toate procesele sunt create la fel: pid\_t fork(void); (singurul mod de creat procese in Unix)

→fork se cheama o singura data si se intoarce de doua ori

→discriminarea intre parinte/copil se face in funct de PID(intors de fork) PID a noului process = parinte, PID= 0 => copil

→copil = copie la parinte

→dupa fork nu exista garantie asupra ordinii de exec parinte/copil

→doua var dupa fork: parintele asteapta sa se term exec copilului//se executa independent

→doua moduri de a utiliza fork : un process se duplica pt ca parintele si copilul sa execute concurrent sectiuni dif de cod /// un process vrea sa execute un program diferit (tipic shell)

VFORK

→dupa vfrok copilul partajeaza spatiul de adrese al parintelui pana la exec/exit

→parintele asteapta copilul sac heme exec/exit => garantat dupa vfork copilul ruleaza primul

→dc copilul se blocheaza inainte de exec/exit si deblocarea depinde de parinte => deadlock

Terminarea proceselor

→terminare normala: return din main, sau exit/\_exit

→anormala: apel explicit abort(SIGABRT), procesul primeste anumite semnale

→codul kernel inchide fisierele, elibereaza mem alocata etc

→parintele e notificat printr un cod de stare(val return din mai/parametrul exit/generat de kernel pt anormla)

→parintele recupereaza codul cu wait/waitpid

→copil zombie – se termina inainte de procesul parinte – kernel salveaza info min despre copil si elibereaza resurse – parintele afla info cand apeleaza wait/waitpid

→copiii lu init nu ajung niciodata zombie, init apeleaza wait mereu pt copiii care termina exec

→proces termina => kernel trimite SIGCHLD

→actiunea implicita pt SIGCHLD este sa fie ignorat

Pid\_t wait(int \*status);

Pid\_t waitpid(pid\_t pid, int \*status, int options);

→wait poate bloca apelantul dc toti copii ruleaza, asteapta dupa primul process copil care se termina

→waitpid are o opt de apel neblocant, are o serie de optiuni care dicteaza terminarea carui process copil o asteapta

Pid == -1 -oricare process/// >0 procesul pid // ==0 oricarui copil cu GID = GID apelant // < -1 oricarui copil cu GID = abs(PID)

→comanda shell din program: system(“ls -l > out”); int system(const char\* cmd);

→system implementat = fork+exec\_waitpid

Scripturi shell

→fisiere txt care incep cu #! => kernelul apeleaza exec -> shell interpreteaza

LABORATOR 4

→wait obliga parintele sa astepte sincron terminarea copilului

→nu orice semnal poate fi prins(tratat) (Ex: SIGKILL orice process e terminat la intalnirea lui)

→permisiunea de trimitere semnal: real/efectiv UID sender == real/effective UID receiver

→proc root potr trimite semnale oricui

→raise permite unui proces sa s idea semnal singur int raise(int signo)

Alarm unsigned int alarm(unsigned int seconds);

→cand alarma expira -> SIGALRM (implicit termina procesul)

→la seconds se adauga timpul pt ca procesul sa trateze semnalul

→daca la apel exista o alarma anterioara, se reseteaza si se intoarce nr de secunde ramas pana la exp celei vechi

→alarm(0) anuleaza o alarma ant care nu a expirat inca si intoarce nr de sec ramas

→Pause suspenda exec proce apelant pana cand primeste orice alt semnal int pause(void)

LABORATOR 5

→pipe: pot fi folosite intre procese cu acelasi stramos (parintele creeaza pipe si apeleaza fork) int pipe (int fd[2]);

→fd[0] deschis pt citire si fd[1] pt scriere

→iesirea lui fd[1] este intrarea lui fd[0]

→pt scriere/citire din pipe se foloses read/write \_(fd[\_], buf, len);

→directia de comunicare parinte->copil, dupa fork se inchide unul din capete close(fd[0])-parinte, copil-inv

→citirea dintr un pipe gol/scrierea intr un pipe plin blocheaza tot procesul pana s a citit suf cat sa se poata termina scrierea

→I/O nonblocant cu fcntl

→capete inchise read=0 octeti (EOF)

→capete inchise write = SIGPIPE , daca se ignora write = -1 errno =EPIPE

→scrieri atomice = <PIPE\_BUF octeti

→FIFO permite IPC intre procese diferite (FIFO = named pipes)

→scrierile multiple concurente = atomice dc scriu < PIPE\_BUF octeti

→Unix creaza FIFOs cu syscall mknod

→creere fisiere in UNIX : S\_IF(TIP) S\_IFREG, S\_IFCHR, etc

Int mknod(const char\* pathname, mode\_t mode, dev\_t dev);

Int mknod(“myfifo”, S\_IFIFO | 0644, 0);

→apelurile de creare pt cozi de mesaje, semafoare, mem partajata necesita o cheie convertita de kernel in ID

IPC rendez-vous

1)un proc creeaza o struct noua IPC folosing o cheie IPC\_PRIVATE si salveaza ID-ul returna in kernel

2)serverul si clientul specifica o cheie intr un header comun

3)serverul si clientul folosesc o cale comuna catre un fisier si un ID de proiect pe care le convertesc intr o cheie cu ftok

Structure IPC: reliable, flow-control, pot fi procesate si in alta ordine decat FIFO

Cozi de msj: create cu msgget, atributele pot fi manipulate cu msgctl, se scriu citesc cu msgsnd/msgrcv, apelurile folosesc o structura definita de utilizator

Type=0 la msgrcv politica FIFO, altfel se citeste primul msj din coada de acel tip

LABORATOR 6

Semafoare

→primitive de sincronizare inter-procese

→folosite la protejarea sectiunilor critice

→ofera doua operatii asupra unui counter, down, up

→acces permis in sectiune critica daca val semaf > 0

→ acces la o resursa partajata +1 count

→eliberare resursa partajata -1 count

Semafoare binare : controleaza o sg resursa, initializat cu 1

Semafoare numaratoare: initializate cu o val poz oarecare, val initiala = nr resurse disponibile pt partajare

→daca un program termina fara sa elibereze semafoarele pe care le-a alocat trb undo

Semafor : semval – >0 // sempid – PID ul procesului care a ef ultima op asupra sem // semncnt – nr proceselor care asteapta semval > val crt // semzcnt – nr proceselor care asteapta semval = 0

Creare: int semget(key\_t key, int nsems, int flag); key obt cu IPC\_PRIVATE sau ftok

Val poz = eliberare resurse

Val negative = alocare resurse

→cea mai rapida forma de IPC ptc nu e nev de copierea datelor, dar procesele trb sa se sincronizeze

LABORATOR 7

Semafoare POSIX: cu nume sau fara

Cele cu nume pot fi folosite de procese neinrudite, celelalte nu

→sem\_t \*sem;

Sem = sem\_open(“SEM”, 0\_CREAT, 0666, 1); - cu nume

→sem\_t sem;

Sem\_init(&sem, 0, 1);