Funcţionarea sistemului de fişiere din Unix

Mihai Budiu — mihaib+@cs.cmu.edu
http://www.cs.cmu.edu/~mihaib/

18 octombrie 1997

Subiect: Funcţiuni speciale ale sistemului de fişiere în sistemul de operare Unix

Cunoştinţe necesare: oarecare familiaritate cu sistemul de operare Unix

Cuvinte cheie: atribute, protecţie, interfaţă, periferic, apel de sistem

Cuprins

Sistemul de operare Unix s-a impus în lumea calculatoarelor datorită multor calităţi remarcabile. Acest articol va încerca să ilustreze una dintre ele, şi anume *uniformitatea* remarcabilă a operaţiilor. Vom ilustra cum o singură parte a sistemului de operare, şi anume sistemul de fişiere integrează într-un mod uniform o multitudine de funcţii independente cum ar fi: stocarea informaţiei, numirea ei, protejarea informaţiei prin controlul accesului, numirea perifericelor şi controlul accesului la periferice, iar în sistemele moderne Unix chiar şi numirea proceselor, operaţiile cu procese şi controlul accesului la procese. Partea frumoasă este că toate aceste lucruri diferite se fac folosind un număr mic de mecanisme: protecţia fişierelor, perifericelor şi proceselor se face de către nucleu în exact acelaşi fel. Această uniformitate face sistemul mai uşor de folosit şi de implementat şi mai flexibil.

Într-un articol anterior (PC Report din []) am prezentat structurile de date pe care le foloseşte sistemul de operare Unix pentru a implementa operaţiile cu fişiere (structuri a căror totalitate poartă denumirea de "sistem de fişiere"). O oarecare familiaritate cu conţinutul acelui articol se poate dovedi utilă, dar (sper că) nu este absolut necesară. Articolul acela este disponibil din pagina de web a autorului în postscript. Articolul de faţă este o continuare independentă a celuilalt. Sperăm ca peste cîtăva vreme să augumentăm această suită cu un al treilea articol despre implementarea sistemului de fişiere din Unix, care este extrem de interesantă.

Am folosit acest articol şi ca un pretext pentru a trece în revistă *totalitatea* apelurilor de sistem din Unix care operează cu fişiere. E un lucru foarte instructiv să înţelegem cum orice operaţie ne putem imagina se poate sintetiza numai din aceste operaţii primitive puse la dispoziţie de nucleu.

1 Funcţiunile sistemului de fişiere

Un răspuns superficial la întrebarea "la ce foloseşte un sistem de fişiere (în Unix — dar şi în alte multe sisteme de operare)?" ar fi "la stocarea informaţiei." Răspunsul este cu siguranţă corect, dar incomplet. Sistemul de fişiere mai are şi multe alte funcţionalităţi esenţiale, cîteodată foarte departe de misiunea aceasta. Le vom enumera pe scurt, le vom discuta fugar, şi vom vedea apoi cum unele dintre aceste funcţiuni se manifestă în sistemul de operare Unix.

Funcţiuni ale sistemelor de fişiere:

- Stocarea persistentă a informaţiei;
- Numirea informaţiei (servicii de directoare);
- Protecţia informaţiei;
- Mecanism de comunicaţie inter-proces (folosirea în comun a informaţiei);
- Interfaţă spre structurile de date ale nucleului (ex: periferice (devices) în Unix; sistemul de fişiere `/proc`);
- Interfaţă universală la resurse ale reţelei (ex: URL-uri);
- Aparat de măsură a propriei sale performanţe.

Ultimele două aspecte nu vor fi discutate în acest articol.

Celorlalte le vom consacra grosso modo cîte o secţiune.

Să aruncăm o scurtă privire pe lista de apeluri de sistem care operează cu fişiere (un apel de sistem este o funcţiune oferită de sistemul de operare). Dăm în tabela 1 o listă completă (sortată alfabetic) a apelurilor de sistem care operează cu fişiere, din sistemul de operare SunOS 4.1.3, o versiune de Unix extrem de populară de la firma Sun Microsystems.

Apel	Funcţiune	Categorie
`access`	verifică dreptul de acces	A
`chdir`	schimbă directorul curent	D
`chmod`	schimbă drepturile de acces	A
`chown`	schimbă posesorul	A
`chroot`	schimbă directorul rădăcină	D
`close`	închide un fişier	F
`creat`	crează un fişier	FD
`execve`	execută un fişier	—
`fchmod`	ca şi `chmod`	A
`fchown`	ca şi `chown`	A
`fcntl`	operaţii speciale	A
`flock`	"încuie" fişier	A
`fstat`	ca şi `stat`	A
`fsync`	salvează conţinutul din cache al fişierului	F
`ftruncate`	ca şi `truncate`	F
`getdents`	citeşte conţinutul directorului	D
`ioctl`	operaţii speciale cu fişiere speciale	S
`link`	dă încă un nume unui fişier	D
`lseek`	mută cursorul la o nouă adresă în fişier	F
`lstat`	ca şi `stat`	A
`mkdir`	face un nou director	D
`mkfifo`	face un fişier special "ţeavă"	S
`mknod`	face un fişier special	S
`mmap`	transformă acest fişier într-o zonă de memorie	F
`open`	deschide un fişier pentru acces	F
`read`	citeşte din fişier date	F
`readlink`	citeşte o legătură simbolică	D
`rename`	schimbă numele unui fişier	D
`rmdir`	şterge un director	D
`stat`	citeşte atributele unui fişier	A
`symlink`	crează un nou nume pentru un fişier	D
`sync`	salvează cache-ul pe disc	—
`tell`	unde e cursorul în fişier?	F
`truncate`	reduce lungimea fişierului	F
`umask`	schimbă drepturile cu care se crează fişiere	A
`unlink`	şterge un nume al unui fişier	D
`write`	scrie date în fişier	F

Tabelul 1: Apelurile de sistem pentru fişiere din SunOS.

Am clasificat apelurile de sistem după tipul operaţiei în categoriile descrise în tabela 2.

Categorie	Semnificaţie	Apeluri	Operaţii distincte
A	operaţii cu atribute	11	7
F	operaţii cu conţinutul fişierelor	10	9
D	operaţii cu directoare	11	11
S	operaţii cu fişiere "speciale"	3	3
—	nu e o operaţie propriu-zisă pe fişiere	2	2
	Total	37	32

Tabelul 2: Tipuri de apeluri de sistem pentru fişiere

2 Fişiere

Funcţiunea cea mai evidentă a unui sistem de fişiere este de a organiza datele în fişiere. Acestea pot avea proprietăţi felurite în diferite sisteme de operare. În Unix fişierele propriu-zise (adică nu directoarele sau alte varietăţi ciudate pe care le discutăm mai jos) sunt simple array-uri de octeţi din punct de vedere al sistemului de operare; utilizatorul îşi organizează în aceste array-uri datele cum doreşte. Acest model este valabil şi în MS-DOS şi Windows. În plus fişierele mai sunt caracterizate de nişte *atribute* a căror semnificaţie este importantă pentru sistemul de operare. Atribute tipice sunt: numele fişierelor, de care utilizatorul se foloseşte pentru a indica fişierele asupra cărora vrea să facă operaţiile, posesorul fişierului, data la care fişierul a fost creat, drepturile de acces la fişier, etc. Atributele unui fişier sunt memorate de Unix într-o structură de date numită *inod*, de la "information-node".

Atributele sunt oarecum secundare funcţiei principale a fişierelor, cea de a memora date; fişierele puteau fi folosite pentru a memora date chiar şi fără a avea la-ndemînă atribute. Sistemul de operare însă menţine atributele pentru că s-a observat că funcţionalitatea pe care acestea o oferă este foarte utilă. Vom vedea că atributele sunt unul dintre mecanismele prin care sistemul de fişiere poate fi folosit şi pentru alte scopuri.

3 Operaţiile cu conţinutul fişierelor

Să trecem în revistă rapid operaţiile care se ocupă doar de manipularea datelor; operaţiile cu atribute vor fi explorate într-o secţiune ulterioară.

În tabelul 3 sunt extrase din tabelul 1 operaţiile de categoria "F", care lucrează direct cu conţinutul fişierelor.

După cum vedeţi operaţiile sunt relativ clare, poate mai puţin `mmap()`, de care ne vom ocupa altă dată, într-un articol referitor la sistemul de memorie virtuală (această funcţie, de altfel foarte importantă, este o aplicaţie imediată a sistemului de memorie virtuală).

Putem face mai multe lucruri cu un fişier: îl putem crea, îl putem scurta, ne putem plimba prin el şi putem citi/scrie în el "petice". Funcţia `fsync()` ne asigură că fişierul ajunge pe disc; sistemele de operare încearcă să ţină cît mai mult timp informaţiile în memorie pentru că discul este foarte lent comparativ cu memoria şi orice acces la el este foarte costisitor. Este demn de observat că doar `open()`, `creat()` şi `truncate()` lucrează cu fişiere date prin *nume*; celelalte funcţii lucrează cu fişiere deja deschise cu `open()` anterior, şi care sunt indicate cu rezultatul dat de funcţia `open()`. Cînd `open()` deschide un fişier returnează un număr întreg care se numeşte *descriptor de fişier*:

Apel	Sintaxa	Descriere
open close	fd = open(path, mod) close(fd)	deschide un fişier specificat prin nume (path) pentru acces în modul indicat; întoarce un număr întreg (descriptor) pentru a manipula ulterior fişierul închide un fişier deschis cu `open`
creat	creat(path, drepturi)	crează un fişier cu un nume specificat
truncate ftruncate	truncate(path, lungime) ftruncate(fd, lungime)	redu lungimea fişierului dat prin nume la cea indicată ca şi `truncate`, doar că fişierul e indicat prin descriptor
lseek tell	lseek(fd, deplasare, relativ) tell(fd)	mută cursorul de scriere/citire la o nouă adresă în fişier află unde e cursorul în fişier
read write	read(fd, memorie, lungime) write(fd, memorie, lungime)	citeşte din fişier date în memorie (de la poziţia unde e cursorul curent) scrie date din memorie în fişier (la poziţia dată de cursorul curent)
fsync	fsync(fd)	salvează conţinutul fişierului care era păstrat în cache
mmap	...	transformă acest fişier într-o zonă de memorie; scrierile în această zonă de memorie se vor reflecta în fişier, iar citirile din memorie se vor face din acel fişier. Argumentele sunt prea complicate ca să le discutăm acum.

Tabelul 3: Apeluri de sistem pentru conţinutul fişierelor.

"file descriptor", sau "file handle" (în traducere "mîner de fişier"), folosit de alte funcţii pentru a indica fişierul. Verificarea dreptului de acces se face o singură dată, la deschiderea fişierului. Toate celelalte operaţii sunt apoi mult mai rapide.[1]

Funcţia `open()` poate crea noi fişiere, ca şi `creat()`.

4 Directoare

Sistemul de directoare are ca misiune de a ordona fişierele într-o ierarhie şi de a le da nume care să fie convenabile utilizatorilor. Un director este un obiect al cărui "conţinut" este format din nume de fişiere şi din nume de alte directoare. Directoarele mai conţin pentru fiecare nume şi numărul inodului care-i corespunde (articolul citat în introducere vorbeşte mai mult despre asta). Presupunem că cititorului îi este cunoscută noţiunea de "cărare" (path) care descrie un lanţ de directoare care duce la un fişier (de exemplu `/usr/home/mihaib/data/articles/2fs.tex` este cărarea care duce la fişierul în care tocmai scriu acest articol în această clipă.

[1] Dezavantajul este că o schimbare a drepturilor de acces nu are nici un efect pentru procesele care au deschis anterior fişierul!

Să trecem rapid în revistă în tabelul 4 operaţiile (luate din tabelul 1) care lucrează cu directoare.

Apel	Sintaxa	Descriere
`chdir`	`chdir(path)`	schimbă directorul curent pentru acest proces
`chroot`	`chroot(path)`	schimbă directorul rădăcină pentru acest proces
`getdents`	`getdents(fd, memorie, lungime)`	citeşte conţinutul directorului în memorie
`creat`	`creat(path, drepturi)`	crează un fişier cu numele indicat
`mkdir`	`mkdir(path, drepturi)`	fă un nou director cu numele indicat
`rmdir`	`rmdir(path)`	şterge un director care nu conţine fişiere
`link`	`link(path1, path2)`	dă încă un nume (path2) fişierului cu numele path1
`unlink`	`unlink(path)`	şterge un nume al unui fişier
`rename`	`rename(path1, path2)`	schimbă numele unui fişier din path1 în path2
`symlink`	`symlink(path1, path2)`	crează un nou nume pentru un fişier
`readlink`	`readlink(path, memorie, lungime)`	citeşte la cine merge o legătură simbolică

Tabelul 4: Apeluri de sistem pentru directoare

Funcţia `creat()` apare din nou, pentru că schimbă conţinutul unui director atunci cînd crează un fişier. `mkdir()` este echivalentul lui `creat()`, doar că face un nou director.

Un fişier în Unix poate avea mai multe nume. Primul nume apare cînd fişierul este creat, următoarele se pot adăuga cu funcţia `link()`. Un nume al unui fişier poate fi schimbat cu `rename()`, sau poate fi şters cu `unlink()`. `rmdir()` este echivalentul lui `unlink()` pentru directoare.

În fine, în Unix există o clasă specială de fişiere numite *legături simbolice* (symbolic links), al căror conţinut este numele altui fişier (o cărare). Aceste fişiere se crează cu apelul `symlink()`.

Dacă fişierul A este o legătură simbolică la B, atunci orice acces la A este transformat de nucleu într-un acces la B. Pentru că un `read()` din fişierul A ne arată automat conţinutul fişierului B, avem la dispoziţie funcţia specială, `readlink()`, cu care putem vedea la ce fişier punctează o legătură simbolică.

Legăturile simbolice sunt foarte asemănătoare cu numele fişierelor, dar există o serie de diferenţe subtile pe care le vom discuta altă dată. Ele au fost create pentru că, deşi un fişier poate avea mai multe nume create cu `link()`, ele toate trebuie să se afle pe acelaşi disc. Pentru a putea avea nume alternative pentru un fişier pe alte discuri decît cel pe care se află fişierul, se folosesc legăturile simbolice.

URL-urile (Universal Resource Locators), cu care în Internet se indică paginile de web, sunt o extensie a noţiunii de legătură simbolică: pe lîngă numele unui fişier ele indică şi numele unui calculator plus eventual numele unui protocol, astfel:

`http://`	`www.cs.cmu.edu`	`/afs/cs/user/mihaib/`
protocolul de comunicaţii	calculatorul	cărarea

Facem acum o observaţie banală, dar *crucială*: **pentru a putea face orice operaţie cu un fişier trebuie să cunoaştem o cărare la el** (pot exista mai multe, pentru că un fişier poate avea mai multe nume); **singura metodă pentru a specifica un fişier este printr-o cărare!**

De aici rezultă imediat o serie de consecinţe foarte importante. Una dintre ele este că un fişier care nu are nume nu este accesibil nicicum. Din cauza asta în Unix fişierele nu pot fi şterse (nu există nici un apel de sistem pentru asta! verificaţi); ele dispar atunci cînd ultimul lor nume este şters cu `unlink()`.

Dar cele mai importante consecinţe vor apărea abia atunci cînd vom vorbi despre protecţie. Înainte de asta trebuie însă să vedem cum se manipulează atributele.

5 Operaţiile cu atribute

Toate atributele unui fişier (mai puţin numele) sunt ţinute de Unix în inod. (Numele sunt ţinute în directoare.) În tabela 5 sunt trecute operaţiile puse la dispoziţie de către nucleu pentru a opera cu atribute.

Apel	Sintaxa	Descriere
`access`	`access(path, drepturi)`	verifică dacă fişierul path poate fi accesat în modul indicat (de către procesul curent)
`umask`	`umask(masca)`	indică drepturile de acces pe care le vor avea fişierele create de acum încolo (cu `open`, `creat`, `mkdir`)
`chmod`	`chmod(path, drepturi)`	schimbă drepturile de acces la fişierul indicat
`fchmod`	`fchmod(fd, drepturi)`	ca şi `chmod`, pe un fişier deschis
`chown`	`chown(path, utilizator, grup)`	schimbă posesorul unui fişier
`fchown`	`fchown(fd, utilizator, grup)`	ca şi `chown`
`stat`	`stat(path, memorie)`	citeşte toate atributele unui fişier
`fstat`	`fstat(fd, memorie)`	ca şi `stat`
`lstat`	`lstat(path, memorie)`	ca şi `stat`, doar că citeşte atributele unei legături simbolice
`flock`	`flock(fd, permisiuni)`	"încuie" sau descuie un fişier (interzice — pentru sincronizare — accesele altor procese)
`fcntl`	`fcntl(fd, comanda, argumente)`	operaţii speciale...

Tabelul 5: Apeluri de sistem pentru manipularea atributelor.

Funcţia `fcntl()` (file control: controlul fişierelor) face o sumedenie de lucruri diferite în funcţie de valoarea celui de-al doilea argument: află dacă fişierul e încuiat, modifică modul de lucru cu fişierul (de exemplu poate aranja ca fiecare octet scris să meargă imediat spre disc, fără să stea în cache), poate pune alte încuietori, etc. O funcţie complicată care deocamdată nu ne interesează prea tare.

`stat()` citeşte într-o structură de date atributele fişierului.

Pentru secţiunea următoare ne interesează mai ales să ştim că următoarele sunt atribute ale unui fişier:

Funcţiile `access()`, `umask()`, `[f]chmod()` operează cu drepturile, funcţiile `[f]chown()` cu posesorul.

Posesorul
Drepturi de acces
Tipul (director, etc.)

Tabelul 6: Atribute importante ale fişierelor.

6 Protecţie

Cum implementează sistemul de operare Unix protecţia accesului la fişiere?

În primul rînd trebuie să ne fie clar *cine* poate accesa fişierele. Fişierele sunt accesate numai de procese (programele care sunt în curs de execuţie). Nu utilizatorii lucrează cu fişiere[2], ci programele lor. Fiecare program are o identitate moştenită de la programul care a identificat utilizatorul cînd acesta a tastat parola. Pe toate programele care s-au născut din programul căruia i-am tastat parola mea, scrie undeva "ăsta e al lui mihaib" (numele meu de utilizator).

Atributul "posesor" al unui fişier este de aceeaşi natură. Cînd un proces ştampilat "mihaib" crează un fişier (cu `open()`, `mknod()`, `mkdir()` sau `creat()`), acel fişier capătă aceeaşi ştampilă, care devine atributul "posesor". Comanda `chown()` poate fi folosită pentru a schimba posesorul unui fişier.

În cele ce urmează simplificăm puţin discuţia restrîngînd atenţia la drepturile "utilizatorilor". După cum ştiţi Unix poate agrega utilizatori în grupuri, dar funcţionarea sistemului de protecţie este asemănătoare pentru grupuri ca pentru indivizi.

Cum observam mai sus, orice comandă care operează asupra unui fişier trebuie să manipuleze întîi o cărare la acel fişier (pentru a putea scrie într-un fişier, el trebuie întîi deschis, şi atunci se indică prin cărare). Unix-ul clasic distinge doar trei tipuri de operaţii pe fişiere: citire, scriere şi executare.

Regula de protecţie pentru accesul la un fişier în Unix este atunci foarte simplă:

> *Un proces poate opera asupra unui fişier indicat printr-o cărare dacă are drept de inspecţie în toate directoarele din cărare, şi dacă are dreptul de a face operaţia dorită asupra ultimei componente din cărare.*

Observaţi un lucru foarte interesant: decizia dacă un proces are dreptul să facă ceva cu un fişier depinde numai de drepturile procesului pe fiecare componentă a cărării. Decizia nu depinde nicidecum de conţinutul fişierului, de pildă. Decizia nu depinde (aproape) nicidecum de *tipul* fişierului.

Chiar dacă fişierul este un director, un fişier de date, un fişier special (vom vedea ce sunt astea mai jos), sau o legătură simbolică, dreptul de acces se verifică în acelaşi fel (pentru toate fişierele care compun cărarea):

- Se compară posesorul din inod cu posesorul procesului care vrea să facă operaţia;
- Se compară drepturile din inod-ul fişierului cu tipul operaţiei cerute.

Observaţi că datorită faptului că un fişier poate avea mai multe cărări, accesul pe diferite cărări poate avea restricţii diferite pentru un acelaşi proces!

[2] Un utilizator poate "accesa" discul doar cu un ciocan.

Lăsăm ca exerciţiu cititorului ilustrarea aplicaţiilor acestui mecanism de protecţie. Exemplele abundă: directoarele fiecărui utilizator au drepturi mai mari pentru posesor, fişierele executabile de către toţi utilizatorii nu pot fi modificate (decît de administrator), (deci viruşii nu se pot propaga!), se pot folosi directoare publice pentru fişiere temporare (`/tmp`), poşta electronică poate fi stocată în fişiere ordinare fără frică de violare a secretului corespondenţei, serverele de comunicaţie în reţea (ex.: ftp, www) lucrează pe o ierarhie de directoare izolată de cea principală a sistemului (cu `chroot()`[3]), etc.

7 Abstracţia de fişier; operaţiile pe fişiere

Sa ne uităm încă odată la tabelul 3.

Observaţi că ceea ce un proces ştie despre conţinutul unui fişier este accesibil numai prin funcţiile acestea: `open(), close(), read(), write(), lseek()` (celelalte funcţii sunt mai puţin importante şi nu neapărat necesare; operaţiile lor principale ar putea fi sintetizate din cele înşirate aici).

Această interfaţă este foarte generală, pentru că nu presupune nimic despre organizarea internă a fişierelor. La o adică ne putem imagina aceste operaţii mergînd pe orice "container" cu date. *Aceste operaţii sunt o interfaţă abstractă pentru un depozit de date.*

Această putere de expresie a interfeţei a fost sesizată de proiectanţii Unix-ului, la începutul anilor '70, şi este premeditată. În Unix o mulţime de alte obiecte pe lîngă fişiere sunt accesate folosind aceste operaţii!

8 Ţevile cu nume

Unix oferă noţiunea de "pipe": o ţeavă între două procese. Există două feluri de ţevi, dar acum ne vom ocupa doar de una dintre ele, de "ţeava numită" (named pipe). Aceasta are un nume ca un fişier, dar se comportă oarecum diferit: datele pot fi citite o singură dată; citirea "consumă" datele. Partea frumoasă este că o astfel de ţeavă poate fi manipulată tot cu apelurile indicate mai sus... nimic din interfaţă nu obligă un fişier să păstreze datele scrise!

O astfel de ţeavă se poate crea cu apelul de sistem `mkfifo(path, drepturi);`. Dacă am creat un astfel de "fişier", toate celelalte operaţii pe el se efectuează cu aceleaşi funcţii: `open()` deschide ţeava şi verifică drepturile de acces, `close()` o închide, `write()` scrie în ţeavă (şi blochează scriitorul pînă apare un cititor), iar `read()` citeşte din ţeavă (sau blochează cititorul pînă apare un scriitor). Apelul de sistem `lseek()` nu se poate folosi cu astfel de fişiere.

Natura diferită a ţevii nu este vizibilă pentru procese, pentru că interfaţa este aceeaşi. Această interfaţă este extrem de puternică.

Ca un divertisment, care nu are prea mare legătură cu subiectul articolului, vom arăta cum se poate folosi o "ţeavă numită" pentru a depista pe un calculator legat în Internet cine face `finger` ca să afle informaţii despre noi.

Ideea este că atunci cînd cineva face `finger mihaib@cs.cmu.edu` demonul local de finger citeşte din directorul meu fişierul cu numele `.plan`. Dacă fac acest fişier o ţeavă numită şi pun un proces să scrie în ţeavă, acest proces va fi blocat pînă cînd cineva face un `finger` spre mine. Acest proces va fi deblocat de demonul de `finger`. Un script shell ca cel următor face acest lucru:

[3] În Unix nu numai noţiunea de "director curent" este alta pentru fiecare proces, dar fiecare proces are propriul lui "director rădăcină".

```
cd
rm -f .plan
mknod .plan p       # creaza teava
while /bin/true; do
        echo "No plan" > .plan
        netstat | grep finger >>ma-cauta
done
```

Comanda `echo` va fi blocată pînă apare un cititor. După ce apare un cititor (demonul `fingerd`), se va executa imediat comanda `netstat`, care va arăta toate conexiunile de reţea deschise în clipa aceea, inclusiv cea a demonului `fingerd` de pe maşina locală cu programul `finger` de pe maşina de la distanţă. În fişierul `ma-cauta` se va depune adresa maşinii de unde se face `finger`. (Mai mult de atît este ceva mai greu de aflat.)

9 Perifericele în Unix: fişierele "speciale"

Dar să ne întoarcem la drepturile noastre.

Am văzut că abstracţia de fişier poate îmbrăca mai multe obiecte: nu numai fişiere, ci şi ţevi. Oare nu există şi alte creaturi care să poată fi manipulate prin această interfaţă?

Ba da, ba da. O mulţime. De exemplu: toate perifericele.

În Unix toate perifericele conectate la calculator au un nume de fişier şi sunt operate ca fişiere! Aceste fişiere sunt numite "fişiere speciale", şi există două mari categorii: fişiere speciale de tip "caracter" şi fişiere de tip "bloc". Diferenţa între ele nu ne interesează aici[4].

Tot ceea ce e periferic în Unix: mouse-ul, tastatura şi ecranul (numite împreună "consolă"), terminalele conectate şi modemurile, memoria video, discurile şi unităţile de bandă, cdrom-ul şi placa audio, toate acestea sunt ascunse de Unix sub aceeaşi abstracţie, asupra căreia se operează cu `open()`, `close()`, `read()`, `write()`, `lseek()` şi `ioctl()`.

Singura funcţie nou-venită este `ioctl()`, care seamănă cu `fcntl()`, pe care am prezentat-o mai sus. Diferenţa este că `ioctl()`, de la "I/O control" (controlul intrării şi ieşirii) se foloseşte numai pentru fişiere speciale, iar `fcntl()` pentru fişiere ordinare. Funcţia aceasta este o şmecherie care permite utilizatorilor să facă toate operaţiile pe care nu le pot face cu celelalte funcţii. Cu alte cuvinte, nu toate perifericele se "ascund" bine sub abstracţia prezentată, şi atunci avem nevoie de o funcţie specială.

Deocamdată să lăsăm pe `ioctl()` la o parte şi să vedem mai multe despre fişierele speciale.

În sistemele Unix tradiţionale toate fişierele speciale sunt strînse în directorul `/dev`, de la "devices" (aparate). În sistemele moderne există o întreagă ierarhie de directoare în directorul acesta: directoare pentru discuri, directoare pentru benzi, etc.

Iată o porţiune din directorul `/dev` de pe o staţie de lucru Sun cu SunOS 4.1.3 (fac o selecţie; sunt peste 550 de fişiere!):

```
crw-rw-rw-  1 root      69,   0 May  6 17:55 audio
crw-rw-rw-  1 root      26,   0 May  6 17:55 bwone0
crw--w--w-  1 mihaib     0,   0 Oct 19 14:06 console
```

[4] Diferenţa este că cele bloc se pot accesa fizic numai în blocuri de informaţie de mărime fixă, iar cu cele caracter se pot transfera cantităţi arbitrare de informaţie. Altă diferenţă este că se pot plasa sisteme de fişiere decît pe periferice de tip bloc.

```
    lrwxrwxrwx  1 root            9 May  6 17:37 modem -> /dev/ttya
    crw-rw-rw-  1 root      11,   0 May  6 17:55 des
    brw-rw-rw-  1 root      16,   0 May  6 17:51 fd0a
    crw-rw-rw-  1 root     104,   0 May  6 17:55 lightpen
    crw-rw-rw-  1 root      13,   0 May  6 17:51 mouse
    crw-------  1 root      37,  40 May  6 17:51 nit
    crw-rw-rw-  1 root      30,   0 May  6 17:55 rmt0
    crw-rw-rw-  1 root      30,   1 May  6 17:55 rmt1
    crw-r-----  1 root      17,   0 May  6 17:51 rsd0a
    crw-r-----  1 root      17,   1 May  6 17:51 rsd0b
    crw-rw-rw-  1 root       2,   0 Sep 25 11:25 tty
    crw-rw-rw-  1 root      12,   0 Oct 19 14:44 ttya
    crw-rw-rw-  1 root      12,   1 May  6 17:51 ttyb
    ^                       ^^    ^
tipul                    major    minor
```

Numele fişierelor speciale nu sunt prea bine standardizate; fiecare sistem va numi un anumit periferic în alt fel. Tipul mai sus este "c" pentru fişiere de tip "caracter" şi "b" pentru fişiere de tip "bloc". Pe sistemul meu fişierele de mai sus corespund la (tabela 7):

audio	placa de sunet
bwone0	placa grafică alb-negru (Black-and-White ONE)
console	consola (tastatură, ecran)
des	integratul care criptează DES (Data Encription Standard)
fd0a	primul floppy disk
lightpen	creionul optic
mouse	şoarecul
nit	placa de reţea (Network Interface Tap)
rmt0	banda nr. 0 (Remote Mag Tape)
rsd0a	primul disc SCSI, partiţia A (Raw Scsi Disc 0)
ttya	primul terminal (mai precis portul serial; TeleTYpe)

Tabelul 7: Fişiere speciale.

Faptul că fiecare periferic oferă o interfaţă de fişier are următoarele consecinţe:

- Programele lucrează cu perifericele exact ca şi cu fişierele (aceleaşi operaţii); anumite programe nici nu trebuie să ştie cu ce anume lucrează;
- Protecţia accesului la periferice se face prin aceleaşi mecanisme (drepturi, posesor).

Fireşte, ca şi ţevile, fişierele speciale *nu există fizic pe disc*; pe disc există doar *numele acestor fişiere precum şi atributele*. "Conţinutul" acestor fişiere este "calculat" de nucleu. Vom reveni la acest lucru.

Să vedem în acţiune abstracţia de fişier în cazul perifericelor.

Dacă vă uitaţi cu atenţie mai sus veţi observa că "fişierul" `console` îi aparţine utilizatorului `mihaib` (eu), şi nu lui `root`, ca toate celelalte. Asta pentru că atunci cînd eu am făcut "login" la consolă, fişierul special care este consola mi-a fost trecut în posesie mie de programul "login". În felul acesta, toate procesele lansate de mine, care au "ştampila" `mihaib` pe ele pot scrie şi citi de la consolă (un lucru foarte util)!

De asemenea veţi observa că drepturile pentru alţi utilizatori permit scrierea la consolă! Acest lucru permite altor utilizatori să-mi trimită mesaje folosind comanda `write` a shell-ului, a cărei implementare nu face altceva decît să deschidă fişierul `/dev/console` şi să scrie mesajele.

Adesea se fac bancuri proaste pe un sistem pe care lucrează mai mulţi utilizatori: unul găseşte un astfel de terminal al unui coleg permisiv şi aruncă o tonă de date spre el: `cat /usr/include/*.h >/dev/console`. Pentru a nu permite aşa ceva, pot face `chmod 600 /dev/console`. (Pot face acelaşi lucru cu comanda `mesg n`.)

Iată deci cum sistemul de fişiere constrînge accesul la periferice şi comunicaţia între procese. Iar acest lucru este "pe gratis": acelaşi mecanism care era folosit pentru fişiere ordinare merge şi aici.

În general accesul la periferice este restrîns, pentru că altfel utilizatori răutăcioşi ar putea cauza stricăciuni. Dacă aveţi drepturi de administrator pe o maşină puteţi citi toate discurile direct, bloc cu bloc, direct "din" fişierele speciale. Pe maşina asta aş putea cu `od /dev/sd0a`. Dacă aveţi un Linux şi vreţi să încercaţi, discurile se numesc de obicei `/dev/hd0_`.

Şi aşa mai departe. Programele care arhivează pe bandă pur şi simplu deschid fişierul `/dev/rmt0` şi scriu apoi ce au de scris acolo. Serverul de ferestre X Windows deschide (cu `open()`) fişierele speciale `/dev/bwone0` şi `/dev/mouse` şi scrie, respectiv citeşte de acolo. Aceste programe nu au nici o instrucţiune magică pentru a opera cu benzi sau cu ecranul: ştiu doar să scrie şi să citească din "fişiere", şi în plus ştiu ce semnificaţie are "conţinutul" fişierelor (adică valorile scrise) pentru hardware.

Cum funcţionează aceste "fişiere"? Pentru implementarea fiecăruia din ele nucleul conţine o cantitate impresionantă de "soft", numită "driver" (şofer?). Pe un sistem Unix driverele pot lua mai mult de jumătate din codul întregului sistem de operare!

Fiecare fişier special are printre atribute două valori, numite "major device number" şi "minor device number". Acestea sunt cele două coloane de numere care apar în listingul de mai sus.

Cînd nucleul este compilat, înauntru are un array mare de structuri care corespund fiecărui driver. De exemplu căsuţa 0 din array corespunde driver-ului de consolă, căsuţa 17 discurilor iar căsuţa 13 mouse-ului. (Am citit numărul "major" din listing).

Aceasta este şi asocierea dintre numele de fişier şi programele care implementează abstracţia de fişier: cînd un proces deschide fişierul `/dev/console`, nucleul observă din atributele fişierului stocate în inod, că este vorba de un fişier special cu major-ul 0, deci roagă driver-ul 0 să manipuleze toate operaţiile pe acel fişier. Acel driver va face anumite acţiuni cînd se va scrie în fişier, care acţiuni vor duce la afişarea de informaţii pe ecran. Cum se face acest lucru, depinde foarte tare de hardware-ul în chestiune.

La ce foloseşte numărul minor? De exemplu sistemul de mai sus are două benzi: `rmt0` şi `rmt1`. Atunci, pentru că în hardware sunt identice, pentru ele există un singur driver. Driver-ul va opera însă cu una sau alta din unităţile de bandă, în funcţie de numărul minor al fişierului: scrierile la `rmt0` vor merge pe o banda, iar cele la `rmt1` la cealaltă.

```
               /-------------------------->banda 1
  /dev/rmt1    |
   inod        |     | driver  |         | driver    |      | Array
|---------|    |     | consola |         | banda     |      | de
| posesor |    |     |_________|__.....__|___^_______|_...__| drivere
| drepturi|    |      0         1         17/         18
| .....   |    |                           /
| major 17|----|--------------------------/
| minor  1|----/
|         |
| tip=c   |
|---------|   |
              |
 In sistemul  |  In nucleu
 de fisiere   |
     <-------|------>
```

Pentru ca schema să funcţioneze trebuie ca atributele (major-minor) ale fişierelor speciale să fie corect alocate pentru a corespunde ordinii în care sunt driverele compilate în nucleu. Administratorul de sistem este cel care crează noi fişiere speciale, cu comanda `mknod`, şi trebuie să fie grijuliu să menţină o asociere corectă cînd crează noi fişiere speciale (şi adaugă noi drivere în nucleu).

10 Pseudo-perifericele (pseudo-devices)

Observaţi că deja un periferic "seamănă" foarte puţin cu un fişier: nu poţi depune date pe care apoi să le extragi în mouse, de pildă. Totuşi perifericul este un dispozitiv care manipulează informaţie, deci interfaţa generică este potrivită.

Se poate merge şi mai departe cu ideile. La ce altceva se poate "aplica" această interfaţă? Ce fel de funcţionalitate îmi mai poate da sistemul fără a introduce abstracţii noi?

Foarte multe alte obiecte cu aceeaşi interfaţă pot fi construite. Pe lîngă fişierele speciale care corespund unor periferice concrete, în directorul `/dev` mai există o mulţime de alte fişiere speciale care corespund unor *periferice virtuale*, sau unor "pseudo-periferice". Acestea sunt nişte aparate imaginare, sintetizate în întregime în software.

Iată cîteva:

```
crw-------  1 root      16,   0 May  6 17:51 klog
crw-r--r--  1 root       3,   1 May  6 17:51 kmem
crw-r--r--  1 root       3,   0 May  6 17:51 mem
crw-rw-rw-  1 root       3,   2 Oct 19 20:50 null
crw-rw-rw-  1 root      21,   0 Oct 19 20:51 ptyp0
crw-rw-rw-  1 root      21,   1 Oct 19 20:36 ptyp1
crw-rw-rw-  1 root      20,   0 Oct 19 20:51 ttyp0
crw-rw-rw-  1 root      20,   1 Oct 19 20:36 ttyp1
crw-rw-rw-  1 root      15,   0 Apr  7  1997 win0
crw-rw-rw-  1 root      15,   1 Apr  7  1997 win1
crw-rw-rw-  1 root       3,  12 May  6 17:51 zero
```

Ele corespund următoarelor funcţiuni:

klog	depozit pentru mesajele de eroare ale nucleului
kmem	memoria virtuală vizibilă nucleului
mem	memoria fizică a calculatorului
null	"canalul fără fund" (sink)
ttyp0	terminal virtual 0; master
ptyp0	terminal virtual 0; sclav
win0	fereastra 0
zero	şirul infinit de zerouri

Tabelul 8: Fişiere speciale.

De exemplu `win0` reprezintă o fereastră de pe ecran; aceasta se comportă ca un terminal în miniatură, dar nu există fizic; este o noţiune emulată software. Un pseudo-periferic, deci.

`/dev/mem` este chiar o imagine directă a memoriei calculatorului. Citind astfel octetul 1000 din `/dev/mem`:

```
{
   char c;
   int i = open("/dev/mem", O_RDONLY);
   lseek(i, 1000);
   read(i, &c, 1);
   close(i);
}
```

vom afla ce se află în memoria fizică a maşinii la adresa 1000! (Fireşte, presupunînd că drepturile de acces ne permit acest lucru.)

Alte fişiere bizare: `/dev/null` este un fişier de lungime 0, în care orice se scrie se pierde (nu are nici un efect). Este extrem de util într-o sumedenie de circumstanţe. De pildă dacă nu vrem să vedem erorile care pot fi generate de o comandă, putem să rugăm comanda să scrie aceste erori "în" fişierul `/dev/null`: `ls -R /var/spool/ 2>/dev/null`.

Încă odată: protecţia accesului la pseudo-periferice se face în acelaşi fel, de către codul din sistemul de fişiere care caută în directoare.

11 Sistemul de fişiere `/proc`

Prezentăm aici o altă spectaculoasă extensie a noţiunii de fişier (sau mai precis o altă construcţie care foloseşte interfaţa cu care se accesează fişierele): un sistem de fişiere în care se află... procese!

Această interfaţă a fost propusă în 1991 pentru sistemul Unix de la AT&T, System V, iniţial pentru a facilita depanarea, dar a evoluat într-o metodă generală de interacţiune cu procesele care se execută pe un calculator.

Există o oarecare variabilitate în implementarea acestei construcţii, aşa că exemplul pe care îl prezentăm ar putea diferi oarecum pe sisteme specifice.

Lucrurile stau cam aşa: există un director `/proc`, iar în acest director există cîte un sub-director pentru fiecare proces care se execută pe sistem, numit ca identificatorul numeric al procesului (adică procesul “1” are un director numit “1”). În directorul fiecărui proces se află mai multe “fişiere” care descriu procesul respectiv. Pentru un proces putem avea următoarele fişiere:

`status` informaţie despre starea procesului (posesori, memorie alocată, etc);

`psinfo` informaţie folosită de comanda `ps`, pentru a descrie acest proces;

`ctl` scriind în acest fişier anumite valori se poate influenţa comportarea procesului;

`map` descrie harta memoriei virtuale a procesului; cum se traduce fiecare adresă virtuală într-una fizică;

`as` memoria virtuală alocată procesului.

etc.

Acest sistem de fişiere este folosit de debuggere: un debugger care vrea să depaneze procesul 100 va deschide fişierul `/proc/100/as` şi va scrie la adresa dorită (cu `lseek(), write()`) codul unui breakpoint!

Multe alte operaţii pot fi făcute pe fişiere în acest fel. Linux are în plus în `/proc` pentru fiecare proces o listă a tuturor fişierelor pe care procesul le are deschise.

Protecţiile pe directoarele şi fişierele din `/proc` sunt puse în aşa fel încît interacţiunile între procese independente sunt strict controlate. De exemplu directorul care corespunde fiecărui proces aparţine utilizatorului care a lansat procesul.

12 Concluzie: programarea orientată pe obiecte

Faptul că putem folosi interfaţa read-write-open-close-lseek-ioctl pentru creaturi atît de variate în constituţie, de la fişiere pînă la pseudo-periferice ne face să ne gîndim dacă nu poate fi folosită ca un mijloc universal de acces la informaţie.

Această interfaţă derivă simultan o fenomenală putere din generalitatea ei, pentru că orice tip de date poate fi văzut pînă la urmă ca o succesiune de octeţi, dar şi o fenomenală slăbiciune, pentru că semnificaţia acestor octeţi trebuie stabilită prin alte mijloace decît cele ale interfeţei. Ce înseamnă să scriu un x “în” fişierul `/dev/bwone0`, asta numai cel care a scris driver-ul pentru placa grafică poate şti (şi cel care a citit documentaţia, desigur).

Funcţia `ioctl()`, despre care n-am spus mare lucru în acest articol, este folosită pentru a simula toate funcţiunile care nu se pot obţine cu uşurinţă doar cu funcţiile citate. Un exemplu de acţiune care se obţine cu un `ioctl()` şi care este greu de implementat doar cu celelalte funcţii: pentru un driver al unităţii de bandă există o comandă `ioctl()` care îi spune să deruleze banda înapoi.

Varianta diametral opusă de proiectare este prezentă în interfaţa driver-elor de “terminal” din sistemele de operare Windows: o sumedenie de apeluri de sistem diferite pentru fiecare operaţie grafică, de la desenatul unui cerc, la şters fereastra. Este greu de arbitrat între cele două alternative. Lăsăm cititorul să gîndească la meritele fiecărei scheme.

Revenind la interfaţa uniformă din Unix: în tehnologia orientată pe obiecte această situaţie se rezumă astfel: avem o clasă de bază abstractă, care are ca metode virtuale `open()`, `close()`, `read()`, `write()`, `ioctl()` şi `lseek()`. Fişierele, perifericele, pseudo-perifericele şi descrierile proceselor sunt toate clase derivate din această clasă de bază.

Deşi sistemul de operare Unix tradiţional (şi chiar şi variantele moderne) este scris în C, astfel de “reminiscenţe” ale programării orientate pe obiecte abundă. Sperăm să putem vorbi în alte articole despre: variatele tipuri de sisteme de fişiere pe care Unix le poate folosi (în Linux există cel puţin 10!), despre sistemul de memorie virtuală şi despre STREAMS. Toate aceste construcţii folosesc aceeaşi tehnologie, de a manipula a mulţime de obiecte diferite ca natură dar folosind o unică interfaţă.