Eficienţa în sistemele de fişiere

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

24 iunie 1997

Subiect: Tehnici pentru a face sistemele de fişiere mai rapide;

Cunoştinţe necesare: Funcţionarea unui sistem de fişiere;

Cuvinte cheie: disc, cache, fişier.

Contents

Sistemul de fişiere (file system) este o parte a unui sistem de operare care oferă utilizatorului accesul la dispozitive de memorare permanentă (discuri mai ales) sub formă de *fişiere*. Aceasta nu este o sarcină facilă, pentru că transformarea unui spaţiu de memorie cvadridimensional (cum este discul, cu coordonate cap/cilindru/sector/octet) într-o mulţime de fişiere cu nume ierarhice (directoare) şi care cresc independent şi simultan nu este o treabă tocmai banală.

De fapt tehnici pentru a face acest lucru sunt studiate în orice curs de structuri de date la capitolul "alocarea dinamică a memoriei". Problema esenţială este însă că metoda cea mai simplă — teoretic vorbind — este adesea departe de a fi cea mai rapidă atunci cînd este implementată. Am scris aiurea un articol despre structurile de date ale sistemului de fişiere din sistemul de operare Unix tradiţional (vezi PC Report Decembrie 1996, sau pagina mea de Web pentru o copie). Consacrăm acest articol expunerii la "grămadă" a unei serii întregi de trucuri făcute de proiectanţii de sisteme de fişiere comerciale în lupta cu milisecundele.

Concluzia care se impune este că cele mai multe sisteme de fişiere sunt derivate mai mult sau mai puţin din sistemul clasic Unix (UFS: Unix File System), dar că efortul depus pentru a le face mai rapide este adesea reflectat dominant în cantitatea de cod. Astfel se ajunge de la un cod de 3-4000 de linii în UFS la 60000 de linii în sistemul de fişiere din IRIX 6.x, Unix-ul de la Silicon Graphics. Reţineţi: pentru utilizator funcţionalitatea este pînă la urmă aceeaşi!

Metodele care urmează arată de ce aria sistemelor de operare este atît de mult o îndeletnicire de inginerie (sau, dacă preferaţi, de meşteşug). Fireşte că nu toate tehnicile sunt aplicabile simultan, fiecare rezolvînd cîteodată o anumită clasă de probleme, dar creînd altele, socotite mai puţin importante de arhitect. Oricum, după părerea mea, chiar înşirarea tehnicilor este instructivă.

Desigur, este posibil să fi sărit din enumerare alte tehnici importante. Cei care le cunosc sunt rugaţi să-mi atragă atenţia asupra lor, ca un viitor articol să fie mai comprehensiv.

Să mai observăm că majoritatea aserţiunilor făcute aici se potrivesc pentru sisteme de fişiere tradiţionale, de gen Unix. Noi domenii de activitate, ca multimedia, tind să ceară propriile structuri de date, specializate, pentru a oferi proprietăţi necunoscute de sistemele de fişiere tradiţionale (de exemplu pentru reproducerea unui film video de pe disc trebuie oferită o rată de transfer constantă a informaţiei, independentă de numărul de alte cereri la disc). Acest gen de fişiere probabil că nu este foarte bine reprezentat de curentul articol.

1 Hard-disc-ul

Să vedem cu ce material lucrăm.

Piaţa de discuri dure (hard-disk) la ora actuală are o creştere anuală mai mare decît cea de sisteme de calcul, de circa 60% pe an (volum de vînzări); discurile dure sunt dispozitivul dominant de memorie permanentă. Din această cauză vom discuta în acest articol despre hard-discuri în mod special (şi nu despre CD-ROM-uri de pildă). Anumite dintre consideraţiile pe care le facem sunt valabile numai pentru acest mediu de stocare a informaţiei. De aici înainte cuvîntul "disc" înseamnă disc dur.

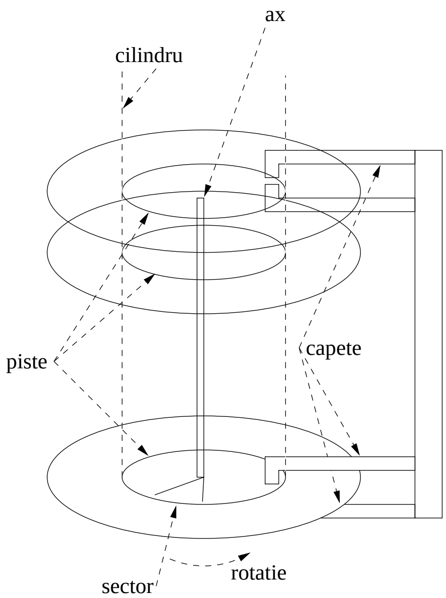

Figura 1: Geometria unui hard-disc

Figura 1 încearcă să ilustreze geometria discurilor[1]; în cuvinte lucrurile stau cam aşa: un disc este o colecţie de platane circulare pe o axă comună, fiecare avînd două capete de citire/scriere, cîte unul pe fiecare din feţe. Pe fiecare platan informaţia este aşezată pe o serie de cercuri concentrice, numite piste (tracks) (în cercuri şi nu în spirală ca la pick-up). Totalitatea pistelor de rază egală se numeşte cilindru (cylinder). Pentru că toate capetele unui disc sunt solidare, se află toate pe un acelaşi cilindru la un moment dat. Asta implică faptul că datele plasate pe un acelaşi cilindru se pot accesa fără a deplasa capetele, selectînd doar dintre capete pe cel activ (unul singur la un moment dat). Fiecare pistă este împărţită la rîndul ei în sectoare (sectors), care la rîndul lor sunt o colecţie de octeţi. Indicînd capul, pista (sau cilindrul), numărul sectorului din pistă şi numărul octetului din sector identificăm unic un octet pe disc.

[1]Un articol foarte interesant despre discuri cu capete magneto-rezistive, care intră mai adînc în alte detalii a apărut în PC Report din iulie 1997.

Discurile sunt toate sudate pe aceeaşi axă, şi se află în mişcare permanentă de rotaţie (la laptop-uri ele sunt oprite după perioade prelungite de inactivitate pentru a economisi bateriile). Viteza de rotaţie este de aproximativ 5000-8000 de ture pe minut, depinzînd de disc. Viteza este constantă, spre deosebire de CD-uri, care se rotesc mai repede cînd se citeşte de la centru.

Timpul de mişcare a capetelor de citire între cilindri vecini (track-to-track seek time) este foarte mare (la scara de viteze a microprocesoarelor), şi este dat de inerţia mecanică a ansamblului de capete. O valoare tipică este de 3-4 milisecunde. Mişcarea capetelor între doi cilindri oarecare are o valoare medie (average seek time) de 8 milisecunde pentru discuri de vîrf. (To seek = a căuta.) Accelerarea şi decelerarea sunt dominante ca durată, aşa că o mişcare de 10 piste nu durează cît 10 mişcări de la pistă la pistă.

Cantitatea de informaţie de pe o pistă este constantă, deşi pistele de la exterior sunt mult mai lungi decît cele de la interior. Asta înseamnă că densitatea informaţiei la margine este mai mică. O tendinţă nouă este de a plasa informaţia cu densitate constantă peste tot, dar asta complică un pic calculul locului pe disc. De obicei această tehnologie se foloseşte pentru a ţine nişte blocuri "ascunse", de rezervă, pe pistele exterioare. Atunci cînd blocuri ale discului se strică (lucru foarte comun, de altfel), ele sunt automat "mutate" în blocurile de la periferie, ascunse, şi cele originale sunt marcate ca defecte. Adesea această mutare este invizibilă calculatorului gazdă, fiind făcută de controlerul ataşat la disc în mod transparent. (Acest lucru are consecinţe nefaste dacă sistemul de operare optimizează accesul la disc bazat pe poziţie.) O pistă tipică are între 32 şi 64 de kilooctteţi.

Iată de pildă specificaţiile pentru un disc Western Digital Caviar de 3,5 inci, de ultimă oră:

Capacitate	2000 Mb
Timp mediu de căutare (seek)	11ms
Viteză de rotaţie	5200 rot/min
Cilindri	3876
Capete (2 * platane)	16
Sectoare/pistă	64
Cache intern	256Kb
Rata de transfer	16.6 Mb/s
Timp mediu între defecţiuni	350 000 ore

1.1 Accesul la disc

În momentul cînd utilizatorul indică ce octet de pe disc vrea să acceseze sunt implicate trei durate importante:

1. Capul trebuie mişcat pe cilindrul corect; această mişcare depinde de cilindrul curent pe care se află capul şi de distanţa pînă la destinaţie. Această întîrziere este dată de "seek time";

2. Capul trebuie să aştepte ca sectorul şi octetul corect să ajungă sub cap datorită rotaţiei; această întîrziere depinde de viteza de rotaţie şi se numeşte "rotational delay": întîrziere de rotaţie; durata ei medie este durata unei jumătăţi de revoluţie a discului;

3. După ce octetul corect a sosit sub cap începe transferul; durata transferului depinde de densitatea de informaţie, de viteza de rotaţie şi, evident, de numărul de octeţi consecutivi transferaţi.

Pentru ilustraţie să considerăm un transfer de 1 kilooctet, octeţii transferaţi fiind plasaţi consecutiv. Cei trei timpi ar putea fi pentru discul de mai sus: căutare a pistei 11ms, întîrziere medie de rotaţie 1/(5200/60) * 1/2s = 5ms şi timp de transfer 5/16 = 0.3ms (o pistă are 32k[2], deci transferul a 1k este 1/16 din timpul mediu de rotaţie, sau 1/32 din timpul de rotaţie).

Spre deosebire de memorii şi microprocesoare, creşterea performanţelor discurilor este mult mai lentă. Cele mai notabile progrese s-au înregistrat în creşterea densităţii de informaţie şi viteza de rotaţie; în ultimii 10 ani însă timpul de mişcare a capetelor a înregistrat o scădere practic neglijabilă, şi nici nu sunt semne că una s-ar produce în viitorul apropiat. Limitările mecanice sunt foarte greu de depăşit.

Iată acum tehnicile folosite pentru a mări eficienţa utilizării discurilor, fiecare discutată într-o secţiune separată.

2 Reducerea timpului de acces prin aranjarea grijulie

Datorită faptului că accesarea locului unde informaţia se află pe disc este foarte costisitoare în milisecunde, un plasament bine-gîndit al informaţiei pe disc poate aduce beneficii substanţiale.

2.1 Accesul la nivel de bloc

Calculele anterioare arată că timpul de acces la informaţie este complet dominat de timpii de căutare (seek) şi întîrziere de rotaţie. În acest caz este la fel de rapid să transferi o mulţime de octeţi consecutivi sau unul singur; timpul plătit pentru mişcarea capului este cel dominant. Diferenţa de mărime este atît de importantă încît acest principiu este folosit în *toate* sistemele de fişiere existente (chiar şi de cele care folosesc memorii flash[3] în loc de discuri.

Din cauza asta toate implementările sistemelor de fişiere tratează discul ca pe o colecţie de blocuri de mărime fixată, şi nu ca pe o colecţie de octeţi. Alocarea şi dealocarea, scrierea şi citirea se fac toate la nivel de bloc. Valori tipice pentru mărimea unui bloc? Între 512 octeţi şi 8 kiloocteţi.

Singurul dezavantaj al metodei este cunoscut din studiul algoritmilor de alocare dinamică a memoriei, şi se numeşte "fragmentare internă". Asta înseamnă că uneori blocurile nu sunt folosite în întregime, ca de exemplu în cazul fişierelor care nu ocupă spaţiu un multiplu întreg de mărimea blocului. Datorită faptului că la discuri capacitatea a crescut foarte repede în ultimii ani, iar mărimea medie a fişierelor de asemenea, spaţiul pierdut prin fragmentare nu mai este considerat o problemă importantă.

2.2 "Localitate" spaţială

În 1984 cercetătorii de la Universitatea Berkeley din California (K.M. McKusick, William Joy[4],

[2]Pentru a calcula mărimea unei piste am aflat numărul de piste = cilindri * capete = 3876 * 16 = 62016, şi am împărţit capacitatea discului la numărul de piste: 2000000k/62016 = 32k. În plus un sector are 32k/(64 sectoare/pistă) = 512 octeţi.

[3]Memoriile flash sunt memorii RAM care nu-şi pierd informaţia după întreruperea curentului. Nu confundaţi memoriile flash cu memoriile RAM ne-volatile (NVRAM), de care se deosebesc prin: preţ mai scăzut, prezenţa operaţiei de ştergere, care trebuie neapărat să survină între două scrieri ale aceleiaşi adrese, etc. Memoriile NVRAM sunt practic nişte memorii RAM cu alimentare de la baterii. Sisteme de fişiere bazate pe memorii flash sunt în mod frecvent folosite în ruterele din reţelele de calculatoare.

[4]Întemeietorul lui Sun Microsystems, una dintre cele mai mari şi agresive companii de calculatoare din lume de

S. Laffler şi L. Fabry) au publicat un articol epocal numit "A Fast File-system for Unix" (FFS). Acest articol arăta cum în sistemul de operare BSD (Berkeley Software Development, o variantă de Unix[5]) a fost implementat un sistem de fişiere mult mai rapid decît cel tradiţional (UFS), care funcţiona din 1974. Principala tehnică exploatată de McKusick şi colegii lui pentru o dublare a vitezei a fost exploatarea "localităţii".

Intenţionez să consacru un articol special managementului spaţiului liber la discuri, dar pînă atunci iată esenţa tehnicii: sistemul de fişiere tradiţional din Unix aloca blocuri libere practic la întîmplare, cum se găseau pe disc. Echipa de la Berkeley a "spart" discul în mulţimi de cilindri (cylinder groups) contigui, şi a încercat să aloce blocuri consecutive ale aceluiaşi fişier (sau inod-urile dintr-un acelaşi director) în acelaşi grup de cilindri. Între cilindri din acelaşi grup timpul de căutare (seek) este apropiat de cel minim (3ms), spre deosebire de timpul mediu (11ms) între doi cilindri mai îndepărtaţi. De aici rezultă un beneficiu imediat. În plus, ei încercau pe baza unui algoritm complex să aloce blocul "cel mai apropiat" în sens temporal: blocul liber la care se ajunge prin cea mai rapidă mişcare, luînd în considerare şi faptul că pe măsură ce se face "seek" discul se roteşte.

Algoritmul lor a avut un succes instantaneu, încît schimbările propuse de ei au devenit standard, şi au fost adoptate de întreaga lume Unix.

Şi alte cîteva detalii secundare au contribuit la succesul FFS: blocuri de mărime foarte mare (8k faţă de 0.5k din UFS), ceea ce ducea la mult mai puţine "seek"-uri pentru un fişier (încăpea în mai puţine blocuri), deci din nou la o eficienţă sporită. Pentru a contracara efectele fragmentării interne, ei au folosit şi o metodă prin care blocurile incomplet folosite sunt sparte în fragmente mai mici (1k), care pot fi folosite de fişiere diferite. Inutil de spus, algoritmii de management se complică puţin, dar viteza nu suferă scăderi considerabile.

Tehnica "localităţii" se rezumă astfel: *alocă în blocuri "apropiate" (care se pot citi repede unul după altul; asta nu înseamnă neapărat consecutive![6]) informaţiile care probabil vor fi accesate consecutiv.*

Un alt proiect de la aceeaşi universitate care a făcut vîlvă la vremea lui este cel al "Sistemului de fişiere cu log-uri" (Log-structured File-system, LFS pentru intimi) propus în 1991 de M. Rosenblum şi J.K. Ousterhout. Vom discuta pe scurt despre acest sistem de fişiere mai jos, cînd vorbim de scrierile asincrone şi folosirea cache-urilor. Performanţa este obţinută în LFS datorită faptului că toate scrierile se fac în felii de cîte 64 de kilooctèţi (2 piste de pildă), care se scriu în blocuri consecutive, fără "seek" între ele (şi dacă se poate, fără întîrziere rotaţională).

3 Cache-uri agresive (mai ales la clienţi)

Metoda cea mai bună de a face accese la disc este de a le evita. Pentru asta, datele citite de pe disc sunt ţinute într-o memorie RAM, în speranţa că vor fi refolosite. De asemenea, datele sunt citite de pe disc în blocuri, dar accesate de programe în bucăţi arbitrare (poate chiar octet cu octet). O

---

la ora actuală.

[5]Pentru o scurtă istorie a sistemului de operare Unix vedeţi şi BYTE din august 1996.

[6]Foarte adesea controlerul de disc nu poate citi imediat două blocuri consecutive pe pistă: după citirea primului îi trebuie un pic de timp pentru prelucrare, timp în care discul se roteşte şi următorul bloc fuge de sub cap.

citire secvenţială[7] va aduce un bloc întreg la primul octet, după care va lucra direct în memorie[8] Zona din RAM destinată memorării datelor de acest gen (al căror loc permanent este pe disc) se numeşte cache (sau memorie ascunsă)[9].

Singura problemă reală cu cache-urile este că în general nu avem destulă memorie pentru a ţine toate datele de pe disc, aşa încît trebuie algoritmi care să arunce din date din RAM cînd nu mai e loc. Tema iese din domeniul acestui articol.

Să observăm că atunci cînd se foloseşte un server de fişiere (de pildă NFS — Network File System — într-o reţea de calculatoare, sau un server Novell) cache-ul poate fi plasat în mod logic în 3 locuri: la server în RAM, la client pe disc sau la client în RAM. Pentru reţelele locale rapide e mai eficient RAM-ul de la server decît discul de la client: o reţea este mai rapidă decît un disc!

Un cache este util deci în cazul în care informaţia este accesată în mod repetat. Iată în secţiunile următoare cum cache-ul se poate dovedi util în feluri diferite pentru scrieri şi citiri.

4 Scrieri asincrone

Un cache funcţionează în cazul *scrierii* pe disc astfel: un proces vrea să scrie, dar datele sunt copiate (de sistemul de operare de obicei) într-un cache, unde sunt ţinute o vreme. Procesul care a scris poate însă să-şi continue execuţia imediat. Atunci cînd cache-ul crede de cuviinţă, scrie datele pe disc. Din cauză că între scrierea efectuată de proces şi accesul la disc nu există nici o sincronizare (sunt practic independente), aceste scrieri ale procesului se numesc *asincrone*.

Un dezavantaj al scrierii asincrone este pierderea definitivă a datelor în cazul unei catastrofe survenite înainte de golirea cache-urilor. Sistemul de operare MS-DOS folosea scrieri sincrone pentru a evita astfel de situaţii, dar aceasta a dus şi la o eficienţă extrem de scăzută, care pînă la urmă i-a semnat sentinţa MS-DOS-ului: nu se mai produce; cînd totuşi se mai utilizează, se folosesc programe (ca smartdrive) care implementează cache-uri.

Putem găsi patru beneficii ale scrierii asincrone:

4.1 Gruparea scrierilor (write batching)

O primă metodă prin care care cache-ul se dovedeşte benefic este că permite strîngerea mai multor scrieri laolaltă şi efectuarea lor într-un singur acces. Vom vedea că sistemul de fişiere cu "log"-uri face chiar acest lucru: toate blocurile modificate sunt scrise periodic în grup, în accese de 64k. Se economisesc astfel o grămadă de timpi de poziţionare (seek) şi întîrzieri rotaţionale, care costă, nu glumă.

4.2 Re-ordonarea acceselor

O tehnică folosită de toate driverele de disc normale este de a scrie blocurile pe disc în ordinea în care se minimizează timpul de deplasare. Astfel, cache-ul pasează permanent cereri driver-ului: vreau blocul ăsta, ia blocul ăsta. Driver-ul se întoarce asincron zicînd: "ok, mă ocup eu, tu vezi-ţi de treabă". Astfel driver-ul poate avea (şi în momentele de activitate intensă chiar are) o mulţime

[7] "Secvenţial" înseamnă că octeţii sunt accesaţi în ordine crescătoare, unul după altul.

[8] Un timp de acces la un RAM normal, fără cache de microprocesor, este de 10ns. Comparaţi această valoare cu 20ms pentru un disc: raportul este de 500000 (jumătate de milion); poate 50000 dacă citim un bloc întreg de pe disc!

[9] Un articol mai amplu despre cache-uri poate fi găsit în PC Report din martie 1997.

de cereri încă nesatisfăcute de blocuri. Atunci el aranjează aceste cereri în aşa fel încît să fie cît mai puţină mişcare.

Un algoritm faimos este cel al "liftului" (elevator algorithm): aranjează blocurile în ordine crescătoare a cilindrilor, mişcă din capete de la cilindrul 0 la cel maxim, şi scrie pe măsură ce avansezi. Algoritmul are tot felul de variante, este relativ eficient, şi nici nu face vreun un bloc să aştepte prea mult pînă îi vine rîndul.

4.3 Evitarea scrierilor

Foarte multe fişiere trăiesc un timp scurt; fişiere temporare, fişiere încuietoare (lock) există cîteodată pentru cîteva secunde. Dacă cache-ul evită scrierea acestor fişiere destul de mult timp, ar putea avea surpriza ca fişierele să dispară înainte de a le veni rîndul să fie scrise, şi ca blocurile lor să fie eliberate.

4.4 Scrierea finală

Foarte adesea modificări succesive se aplică aceluiaşi bloc, sau chiar aceluiaşi octet. În cazul acesta cache-ul cumulează rezultatele tuturor modificărilor, şi scrie numai rezultatul final, economisind nişte accese la disc.

5 Citirea în avans (read-ahead)

Al doilea beneficiu al cache-urilor se manifestă la citire. Este limpede de ce un cache este util dacă aceleaşi date sunt citite de mai multe ori. Paradoxal însă, un cache se poate dovedi util chiar dacă datele sunt citite o singură dată!

Un sistem de fişiere poate încerca să ghicească dinainte ce blocuri vor fi cerute în viitor de utilizatori, şi să ceară driver-ului să aducă blocurile din timp. Atunci cînd aplicaţia face cererea de citire, datele sunt servite din cache imediat! Utilizatorul nu observă nici o întîrziere (sau o întîrziere mai mică, dacă datele sunt pe drum). Această tehnică se numeşte "citire-înainte" (read-ahead; pre-fetching).

Inutil de spus, riscul este, ca în cazul oricărui profet, de a face o prezicere eronată, care se soldează cu o citire inutilă. Vestea cea bună este că un studiu empiric asupra programelor obişnuite a observat că se pot prezice cu destul de mare acurateţe accesele secvenţiale la un fişier: dacă un program a cerut primele 3 blocuri în ordine probabil va continua să le ceară şi pe următoarele.

Toate sistemele de fişiere moderne folosesc această optimizare, cu destul de mult succes.

5.1 Indicaţii de citire în avans (read-ahead hints)

Fireşte, există un caz în care citirea în avans nu este cu nimic dăunătoare: cînd suntem siguri că ea va fi utilă. Sisteme experimentale pun la dispoziţia programatorului apeluri de sistem speciale prin care poate indica porţiunile de fişiere pe care le va accesa în viitor.

Se fac de asemenea încercări de a crea compilatoare care să genereze automat "indicaţii" (hints) pentru sistemul de fişiere privitor la accesele viitoare, fără ca programatorul să trebuiască să abandoneze stilul obişnuit de programare.

6 Gruparea cererilor (batching)

O rudă a metodei cache-urilor folosită în cazul sistemelor de fişiere în reţea, această tehnică reduce traficul în reţea şi numărul de accese la disc: mai multe cereri de scriere/citire şi răspunsurile lor sunt grupate laolaltă (fie la client, fie la server) şi transmise într-o bucată.

7 Tehnici RAID (Redundant Arrays Of Independent Disks)

Viteza de transfer a datelor pe magistrale este mult mai mare decît cea între disc şi magistrală. Limitarea ratei de transfer a discului este dată de viteza de rotaţie: discul Caviar de mai sus face 5200 rpm, iar la fiecare rotaţie poate transfera capacitatea unei piste, de 32k. Asta înseamnă o limită superioară (în absenţa oricăror poziţionări) de 5200/60 * 32 k/sec = 2,77M/sec. (Viteza de transfer mult mai mare din specificaţiile discului de mai sus poate fi atinsă cînd acesta are cache-ul propriu deja încărcat cu date.)

O idee interesantă este atunci de a folosi mai multe discuri conectate simultan la aceeaşi magistrală, şi de a scrie informaţiile în felii (stripes) care acoperă toate discurile. Astfel, folosind 3 discuri, primul bloc o să fie pe discul 1, al doilea pe discul 2, al treilea pe discul 3, al patrulea din nou pe discul 1, etc (puteţi pune orice valoare în locul lui 3). Atunci fiecare disc poate transfera date din/spre memorie la viteza maximă a magistralei folosind propriul cache, de pe unitatea de disc, după care poate scrie cache-ul pe disc la o viteză mult mai mică, în timp ce vecinii săi se ocupă de celelalte blocuri.

Pentru că acolo unde ai trei aparate probabilitatea de defecţiune creşte de mai mult de trei ori, în general tehnica de striping se combină cu una de redundanţă (ideea este ca atunci cînd un disc se strică să nu devină toate inutilizabile). Cea mai folosită metodă este de a avea un al patrulea disc, pe care se află calculată paritatea ("sau exclusiv") a celorlalte trei. Astfel, dacă un disc se strică, informaţia de pe el se poate reconstitui integral folosind celelalte 3 discuri (folosind proprietatea funcţiei paritate: dacă A + B + C = P (notînd cu "+" sau exclusiv), atunci B = A + C + P, deci discul B se poate reconstrui folosind A, C şi discul de paritate P). De aici vine şi numele dispozitivelor, RAID: mulţime redundantă de discuri independente.

Două cuvinte în plus despre această fascinantă metodă, despre care poate vom reveni într-un articol separat:

- Cea mai simplă schemă RAID este cea cu două discuri, care menţin informaţii identice. Acestea se numesc "discuri în oglindă" (mirrored disks).
- Atunci cînd un disc dintr-un RAID se strică, operatorul uman îl înlocuieşte şi se procedează la reconstituirea informaţiei de pe discul nou plasat; acesta este modul de funcţionare "degradat" al dispozitivului RAID, pentru că se consumă o grămadă de timp cu reconstituirea, şi pentru că o a doua defecţiune în acest moment ar face toate datele inutilizabile;
- Metoda de lucru cu paritate face scrierile "mici" foarte ineficiente, pentru că atunci cînd se modifică un singur disc trebuie citite toate celelalte şi recalculată paritatea, deci se generează o grămadă de trafic suplimentar.

Cu toate aceste slăbiciuni, calităţile schemelor RAID depăşesc defectele, aşa încît este de aşteptat ca utilizarea lor să devină din ce în ce mai răspîndită. Dacă controlerul de RAID are şi o cantitate suficientă de NVRAM pentru a implementa un cache mare, atunci şocul scrierilor mici poate fi oarecum absorbit de acesta, utilizatorii ne-trebuind să aştepte calculul parităţii, percepînd deci o creştere substanţială de viteză (datorită paralelismului accesului la discuri).

8 Folosirea "log"-urilor

În această secţiune vom arunca o privire rapidă asupra sistemului de fişiere bazat pe "log"-uri LFS, citat puţin mai sus. Arhitectura sa este oarecum revoluţionară, diferind substanţial de sistemele de fişiere clasice (UFS, FFS).

Tehnica "log"-ului este împrumutată din bazele de date. Log-ul (în română o traducere aproximativă ar fi "înregistrare") descrie *modificările* făcute datelor. Astfel, în sistemul de fişiere bazat pe log, în loc să modifici fişierul, scrii un nou bloc (în log) în care zici: "modific fişierul cutare aici cu atît". Fişierul va subzista în forma originală, alături de înregistrările din log. În acest fel putem vorbi de *versiuni* ale datelor: fiecare nouă scriere în log produce o nouă versiune, care coexistă cu cele vechi, ocupînd un loc diferit pe disc. Dacă vrei să afli ultima valoare, citeşti ultima modificare.

Care e şpilul? De ce, adică, ar fi metoda asta eficientă? Pentru că avînd libertatea de a face scrierile oriunde (şi nu în locurile unde fuseseră plasate blocurile iniţial), poţi alege să scrii toate blocurile consecutiv, făcînd foarte puţine deplasări ale capului! În felul acesta se cîştigă o grămadă de timp.

Înainte de a vă entuziasma de ideea aceasta atît de simplă, să observăm şi reversul medaliei: citirile trebuie să ia de pe disc blocurile din locurile în care se află, deci nu vor beneficia probabil de alocarea contiguă a blocurilor la scriere (în sensul că vor cauza la fel de multe deplasări de capete ca în schema UFS). LFS presupune deci că traficul dominant spre disc este de scrieri; pentru ca acest lucru să fie adevărat cache-ul de disc trebuie să fie foarte mare şi cererile de citire de date trebuie să ia adesea datele din cache.

Un al doilea dezavantaj al acestui sistem de fişiere este generarea continuă de noi blocuri. Ce se întîmplă cu versiunile vechi? După ce discul a fost ocupat în întregime (ceea ce nu durează prea mult), de unde se mai obţine spaţiu suplimentar pentru scrieri, care trebuie să fie şi *contiguu* (dacă spaţiul liber este fragmentat am pierdut toate avantajele scrierii mari, pentru că trebuie să facem din nou toate deplasările de cap)?

Pentru acest scop sistemele de fişiere bazate pe log (LFS) au nevoie de un *compactor* care elimină versiunile vechi ale datelor (garbage collection) şi strînge blocurile utilizate laolaltă. În general acest compactor funcţionează ca un proces separat care traversează structurile de date ale discului atunci cînd spaţiul liber a scăzut sub o anumită cantitate, strîngînd laolaltă blocurile utilizate şi rescriindu-le. Activitatea compactorului este o sursă importantă de trafic spre disc, scăzînd dramatic din eficienţa sistemului de fişiere.

Studii detaliate au fost făcute pentru a determina dacă schema LFS este mai bună decît cea a FFS, dar un rezultat clar în favoarea unuia dintre ele nu a fost obţinut. Fiecare dintre ele se comportă mai bine în anumite condiţii şi pentru un anumit tip de trafic (de exemplu LFS cîştigă la scriere). Tehnici de gen LFS sunt folosite în sisteme comerciale, cum ar fi cel citat mai sus, Xfs, din sistemul IRIX, în care se folosesc log-uri numai pentru directoare, dar nu şi pentru fişierele propriu-zise.

Un mare avantaj al log-urilor este reconstrucţia extrem de rapidă a discurilor după o cădere a calculatorului. Pentru că sistemele de fişiere tradiţionale (UFS, FFS) fac tot timpul modificări în toate zonele, o oprire subită a calculatorului poate lăsa sistemele de fişiere într-o stare incorectă, fiindcă informaţiile din cache nu au fost salvate. Depistarea şi corectarea acestui gen de defecte cere programe foarte sofisticate (`fsck`). Sistemele bazate pe log-uri în schimb au toate modificările recente la sfîrşitul log-ului, deci sunt foarte uşor de reparat. (Atenţie: pierderi de informaţie se petrec în ambele cazuri, pentru că nimeni nu poate reconstitui informaţia ne-salvată din cache; problema care se pune este de a face din nou funcţionale sistemele de fişiere.)

9 Discuri separate

Aceste metode folosesc mai multe discuri beneficiind, ca în cazul RAID, de transferuri paralele spre controlere, care pot apoi face scrierile simultan.

9.1 LFS cu un disc pentru log

Mai curînd o variaţie a schemei LFS, această îmbunătăţire foloseşte un disc pentru scrierea log-ului, compactorul mutînd informaţiile "vii" pe un alt disc, unde informaţia care trăieşte mult timp (de pildă fişierele care se schimbă rar, cu executabile) se va strînge după o vreme.

9.2 Discuri diferite pentru fişiere diferite

Autorul acestui articol a transformat sistemul de fişiere din Linux numit `ext2` (care este de tip FFS) pentru a folosi un disc pentru directoare şi un altul pentru fişiere[10]. În felul acesta cache-ul poate programa scrieri sau citiri simultan pe ambele discuri. Au fost măsurate cu teste simple creşteri de performanţă de 30% faţă de folosirea unui singur disc.

Un alt potenţial avantaj al unei astfel de scheme este acela că se pot implementa politici de alocare a blocurilor speciale pentru directoare şi fişiere obişnuite, care să exploateze caracteristicile speciale ale accesului (de exemplu directoarele tind să fie mici şi nu "scad" niciodată).

10 Replicare

O altă idee interesantă: prea multe cereri la un server? Atunci foloseşte mai multe servere, fiecare răspunzînd la o fracţiune din cereri. Sunt, fireşte, două posibilităţi: ori îi dai fiecărui server nişte date de care e singur responsabil, ori pui aceleaşi date pe toate serverele. A doua metodă se numeşte *replicare*, pentru că are mai multe copii (replici) ale datelor. Clienţii vor face cererile la servere diferite, şi fiecare server va avea o coadă de cereri de satisfăcut mai scurtă, deci un timp de răspuns mai bun.

Cititorul ager a observat imediat că schema are iarăşi dezavantaje: cu citirea totul e bine, dar ce faci dacă cineva vrea să modifice ceva? Trebuie să faci scrierea de mai multe ori! Există o serie întreagă de tehnici pentru a modifica fişiere replicate, dar ne vom mulţumi să le enumerăm; problemele ridicate sunt mult mai mari decît par la prima vedere.

- Se pot folosi tehnici cu voturi (quorum techniques; quorum = majoritate): pentru a modifica un fişier trebuie obţinut acordul unei majorităţi de servere. Citirea la rîndul ei va contacta o

[10] O implementare este disponibilă din pagina de web a autorului.

majoritate de servere, pentru a fi sigură că accesează cea mai recentă copie. Metoda aceasta se bucură mai curînd de fiabilitate decît de eficienţă.

- Se pot folosi fişiere imutabile (care nu se pot modifica odată scrise). Cum? Nu poţi scrie? Păi ce să faci cu ele? Simplu: dacă vrei să le modifici faci o nouă copie. (Limbajul Pascal standard oferă acest tip de fişiere: odată scrise nu pot fi modificate). Sistemul de operare Amoeba implementează astfel de fişiere, care sunt foarte rapide (din această cauză serverul de fişiere se numeşte în Amoeba "the bullet server": serverul glonte).
- Relaţii master/slave: folosite cîteodată şi în schemele mirror (de care am vorbit un pic la capitolul RAID); asta înseamnă că toate modificările se fac pe un server desemnat, care le propagă apoi tuturor serverelor care au copii.

Replicarea fişierelor de fapt este un fenomen foarte întîlnit, dacă stăm să ne gîndim bine: de fiecare dată cînd avem un sistem de fişiere în reţea, care poate fi accesat de mai mulţi clienţi, fiecare client va tinde să aibă în cache-ul lui propria lui copie a (unora) din fişiere. Este tot un caz de replicare, care pune aceleaşi probleme ale modificării fişierelor. Sistemele comerciale (NFS de la Sun, AFS — Andrew File System — de la compania Transarc şi universitatea Carnegie Mellon) rezolvă fiecare problema replicării datelor în felul lui (mai mult sau mai puţin satisfăcător). Fără a intra în detalii, să reţinem şi replicarea ca o tehnică posibilă pentru creşterea eficienţei.

11 Indexare

Structurile de date de pe disc sunt foarte complexe: un arbore de directoare, blocurile unui fişier sunt grupate într-un arbore dezechilibrat, blocurile libere într-un fel de listă stufoasă sau un "bitmap", etc. Pentru a deschide un fişier trebuie traversate o mulţime din aceste structuri, accesînd poate zeci de blocuri. Construirea unor indexuri eficiente (idee împrumutată din tehnologia bazelor de date) poate mări foarte mult viteza de acces la disc.

Arbori, tabele de dispersie (hash tables), arbori B+ sunt folosiţi pentru a creşte performanţa căutării, mai ales în cazul fişierelor şi discurilor foarte mari. De asemenea, structurile de date din memorie, care descriu fişierele deschise şi cache-urile (cache-ul blocurilor din memorie, cache-ul numelor de directoare, cache-ul inodurilor din memorie, etc.) sunt organizate după astfel de structuri de date, cu căutare rapidă (de exemplu pentru a verifica dacă un bloc se găseşte în cache se foloseşte de obicei o tabelă de dispersie şi nu căutare binară).

12 Folosirea timpului de inactivitate

Activitatea unui server de fişiere are perioade febrile şi zone de repaos. Acestea din urmă pot fi folosite pentru a programa activităţi auxiliare, care sunt utile, dar care nu trebuie să consume din timpul util. Candidaţi sunt: compactarea log-urilor, calculul parităţii în sisteme RAID, reconstrucţia discurilor defecte în sisteme RAID, compresia fişierelor, compactarea spaţiului liber, defragmentarea fişierelor (pentru citire rapidă secvenţială).

12.1 Folosirea mai multor fire de execuţie (threads)

În general în sistemele de fişiere tradiţionale toate operaţiile se desfăşoară pentru că sunt cerute de cineva. În alte arhitecturi, gen LFS, există o serie de activităţi pentru care momentul execuţiei nu este precis determinat (compactarea log-ului la LFS). Toate activităţile citate mai sus, care se pot face în timpul liber, sunt de acest gen, iar cea mai eficientă implementare a lor este de a construi un thread (de preferinţă chiar în interiorul nucleului, ca să nu trebuiască să facă apeluri de sistem costisitoare) pentru fiecare din ele.

13 Alocarea întîrziată

Ajungem la o tehnică foarte interesantă, care se bazează pe o observaţie destul de subtilă. Pentru aceasta trebuie să aruncăm o privire la momentul în care se alocă blocuri fişierelor. Un nou bloc se alocă atunci cînd se scrie în el pentru prima oară. Să presupunem că un utilizator cheamă un apel de sistem `write()` pentru a scrie ceva la într-un fişier, începînd de la octetul cu numărul `offset_curent` (valoarea acestuia a fost fixată de un apel `lseek()` anterior). Codul acestui apel de sistem arată cam aşa într-un pseudo-cod C:

```
write(int fisier, char * sursa, int marime)
{
    while (marime) {
        bh = aloca_bloc(fisier, offset_curent);
        copiaza_date(sursa, bh, MIN(marime, marime_bloc));
        marcheaza_modificat(bh);

        offset_curent += marime_bloc;
        marime -= marime_bloc;
        sursa += marime_bloc;
        if (marime < 0) marime = 0;
    }
}
```

(Codul este foarte simplificat: în mod normal în `while` se cheamă o funcţie `cauta_bloc()`, care va aduce blocul care conţine `offset_curent`, sau va chema ea însăşi `aloca_bloc()` dacă acest bloc nu a fost încă alocat; în plus am asumat că datele sunt aliniate şi scrierea se face de la un început de bloc; codul real este ceva mai complicat, însă acesta ilustrează perfect ce ne interesează.) Funcţia `aloca_bloc()` caută un bloc liber pe disc şi alocă simultan un bloc în cache pentru acel bloc; rezultatul întors este un bloc de cache numit `bh`, pe care cache-ul îl va salva pe disc asincron.

Observaţia interesantă este: chiar dacă avem o scriere de un megaoctet, blocurile sunt alocate *unul cîte unul*. Asta înseamnă că s-ar putea ca blocurile să nu fie unul lîngă altul, căci rutina de alocare `aloca_bloc()` nu are idee despre blocurile alocate anterior.

În plus, rutina de alocare trebuie să parcurgă structurile de date aflate pe disc, ori această operaţie este blocantă: pînă se citeşte structura de date de pe disc procesul curent este oprit şi un altul se execută. Asta poate cauza interclasarea alocărilor blocurilor pentru două apeluri simultane `write()`: primul proces alocă un bloc, se blochează pînă structurile sunt citite, al doilea proces începe să se execute şi alocă alt bloc, se blochează, dar primul termină şi cere un al treilea, şi tot aşa.

Dacă implementarea lui `aloca_bloc()` extrage pur şi simplu blocuri dintr-o listă de blocuri libere rezultatul va fi că blocuri succesive dintr-un fişier vor fi plasate pe disc cum vine la mînă, fără nici o relaţie între ele. Paradoxul este aceste: deşi este clar de la început de cîte blocuri este nevoie (`marime/marime_bloc`), rutina de alocare obţine blocurile unul cîte unul, şi atunci ele nu sunt neapărat consecutive! (Plasarea în blocuri consecutive face accesul secvenţial la fişier mult mai eficient, mai ales în combinaţie cu citirea anticipată.)

Se pot folosi patru scheme pentru a ocoli această deficienţă a alocatorului:

13.1 Alocarea în "extents"

O metodă interesantă este de a schimba complet unitatea de alocare a discului din blocuri cu mărime fixă în cantităţi de mărime variabilă (dar nu oricît de mici), numite *extents* (întinderi?). Astfel la apelul de sistem `write()` se alocă întreaga cantitate de spaţiu pe disc de la început, după care este scrisă, dacă se poate dintr-o bucată. Aceasta complică codul alocatorului, dar nu este nicidecum o problemă nefezabilă.

13.2 Prealocare

De fiecare dată cînd trebuie să aloci un bloc, alocă şi cîteva blocuri imediat consecutive pe disc, în eventualitatea creşterii fişierului. Această schemă este folosită cu succes în sistemul de fişiere din Linux, `ext2`.

Dacă fişierul nu mai creşte, sau nu creşte secvenţial, blocurile pre-alocate sunt eliberate.

13.3 Alocare la scriere fizică

Nu se alocă pe disc blocurile pînă cînd nu vine momentul golirii cache-ului. Scrierea alocă blocuri doar în cache, iar algoritmul golirii cache-ului studiază pentru fiecare fişier cîte blocuri trebuie alocate pe disc şi face această alocare. Această tehnică este folosită în sistemul de operare IRIX.

13.4 Realocare la scriere fizică

Această tehnică este o optimizare întîlnită în anumite implementări ale FFS: alocarea merge ca în codul de mai sus, însă procedura de golire a cache-ului încearcă să "mute" blocurile dacă ar fi util: blocuri consecutive din acelaşi fişier sunt puse contiguu dacă se poate.

14 Plasamentul sistemului de fişiere; NASD

Sistemul de fişiere este un serviciu oferit aplicaţiilor, şi poate fi plasat în mai multe locuri: într-o bibliotecă de funcţii (foarte rar), în nucleul sistemului de operare sau într-un proces separat (server). Fiecare metodă are avantajele şi dezavantajele ei, iar în anumite configuraţii speciale fiecare poate fi mai eficientă.

O încercare notabilă este cea de a distribui serviciul cît mai mult, în aşa fel încît să nu existe o singură autoritate responsabilă cu majoritatea operaţiilor. Un proiect foarte interesant în curs de desfăşurare la universitatea Carnegie Mellon se numeşte "Discuri sigure legate la reţea": Network Attached Secure Disks (NASD). Ideea este de a avea o reţea locală rapidă (ATM, Fiber Channel, etc.) care leagă multe calculatoare client cu şi mai multe unităţi de disc inteligente. Clienţii folosesc serverul de fişiere numai pentru a afla care sunt discurile care conţin fişierele şi pentru a obţine o autorizaţie de acces, după care transferul de date se face direct între clienţii interesaţi şi discurile cu informaţie, fără a mai implica serverul nicicum.

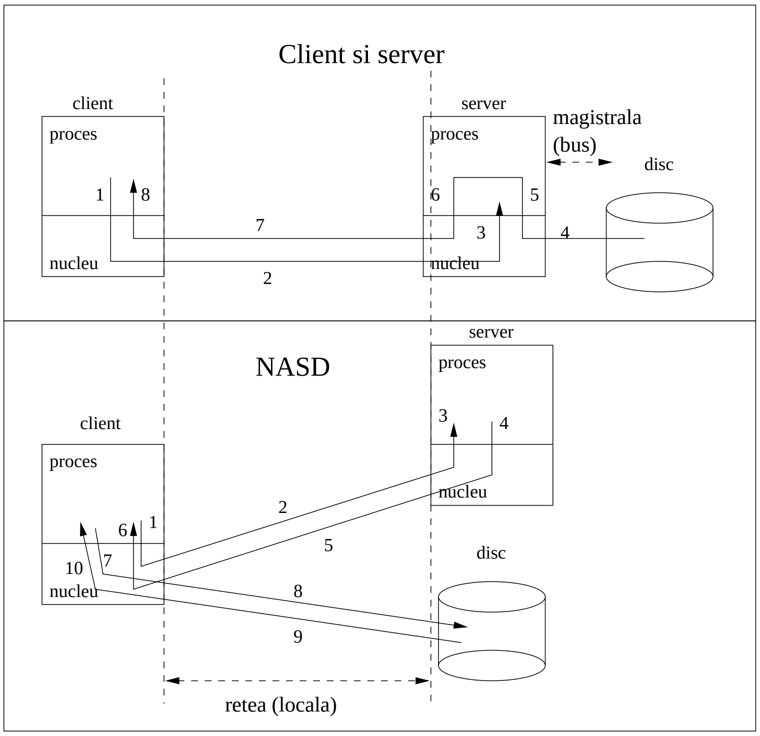

Figura 2: Traseul informaţiei în formula tradiţională şi în NASD

Comparaţi această tehnologie cu cea tradiţională, în care serverul, un proces obişnuit, trebuie să citească discul asociat în memorie şi să trimită apoi imediat rezultatul pe reţea: pe lîngă faptul că o singură autoritate face toate operaţiile, deci este foarte încărcată, se petrec o mulţime de copieri inutile; sistemele moderne în general nu oferă facilităţi de genul: "trimite de pe disc direct pe reţea", ci doar de genul "citeşte de pe disc în memorie" şi "trimite în reţea din memorie." Figura 2 arată diferenţele între mersul informaţiei în cele doua arhitecturi. Pentru citirea de date în formula client-server paşii sunt (scrierea este asemănătoare):

1. Clientul transmite un mesaj pentru server de cerere de acces la disc;
2. Mesajul este transmis spre server;
3. Serverul primeşte mesajul şi îl decodifică;
4. Nucleul de la Server transferă date dinspre discul local;
5. Serverul preia datele din nucleu;

6. Serverul roagă nucleul să trimită datele la client;
7. Nucleul clientului primeşte datele;
8. Clientul copiază datele din nucleu.

Paşii 4, 5, 6, 7, 8 copiază o cantitate mare de date dintr-un loc într-altul. Orice citire va trece prin această secvenţă de operaţii.

Pe cînd în NASD o citire trece prin următorii paşi:

1. Cleintul trimite un mesaj la server pentru a obţine autorizaţia de transfer;
2. Mesajul este transmis prin reţea;
3. Serverul primeşte mesajul şi îl decodifică;
4. Serverul trimite autorizaţia şi descrierea discului;
5. Mesajul este transmis prin reţea spre client;
6. Clientul primeşte mesajul şi descoperă autorizaţia;
7. Clientul trimite un mesaj spre discul inteligent legat la reţea;
8. Mesajul este transmis spre disc;
9. Discul pune datele direct în reţea;
10. Clientul primeşte datele din nucleu.

Transfer de date se face numai în paşii 9 şi 10. Paşii 1, 3, 4, şi 6 pot conţine criptare/decriptare, care este o operaţie relativ costisitoare. Odată obţinută autorizaţia clientul poate repeta de oricîte ori doreşte paşii 7-10 fără a mai interveni pe lîngă server.

Discul trebuie să fie "inteligent" pentru că trebuie să:

- execute un protocol de comunicaţie în reţea;
- aplice metode criptografice pentru verificarea identităţii.

15 Compresie de date transparentă

Programe care dublează capacitatea discului sunt oricui familiare din MS-DOS. Un avantaj neevident al acestor programe este că reducînd cantitatea de date scrise pe disc reduc şi timpul necesar transferului.

Dintre dezavantajele metodei: este greu de estimat spaţiul ocupat după compresie, deci alocarea este mai dificilă, compresia este cu atît mai eficientă cu cît manipulează o cantitate mai mare de date, dar decomprimarea unui întreg fişier pentru a citi ultimele două linii este o irosire, compresia însăşi consumă mult timp de procesor.