Un proiect în Sisteme de Operare

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

ianuarie 1998

Subiect: cum se argumentează în sprijinul unei idei într-un articol

Cunoştinţe necesare: cunoştinţe destul de serioase despre funcţionarea sistemul de operare Unix

Cuvinte cheie: apel de sistem, intercepţie, performanţă, model teoretic

Articole anterioare din PC Report utile (disponibile din pagina de web a autorului):

- "Nucleul", octombrie-noiembrie 1996
- "Multithreading", ianuarie 1996
- "Eficienţa în sistemele de fişiere", octombrie 1997
- "Funcţionarea sistemului de fişiere din Unix", decembrie 1997
- "Anatomia unui apel de sistem în Linux", ianuarie 1998
- "Sistemul de fişiere virtual", februarie 1998

Contents

1 Informatica

Cea mai grosolană diviziune a "ştiinţei calculatoarelor" (cum o numesc anglo-saxonii), sau, dacă preferaţi, a "informaticii" (în accepţiunea francofonă a termenului) va segmenta domeniul în două zone relativ distincte prin natura cercetării şi metoda aplicată. Cele două zone sunt "Teoria" şi "Sistemele".

1.1 Teoria

Despre prima este destul de limpede cum stau lucrurile: este o ramură a matematicii, care uzînd de arsenalul acesteia (definiţii, leme, teoreme, demonstraţii, etc.), manipulează o serie de abstracţii ale fenomenelor petrecute în calculatoare. Multe părţi din informatică aparţin majoritar acestui domeniu; cele mai evidente sunt: teoria algoritmilor, teoria complexităţii, metodele numerice (numite şi "scientific computing"), etc.

Trebuie spus că aproape întotdeauna dezvoltarea teoretică a unei ramuri a calculatoarelor o precede pe cea practică; calculatoarele însele există pe hîrtie dinainte de anii 1930, deşi realizarea lor concretă început să prindă contur abia în timpul celui de-al doilea război mondial.

De fapt rolul teoriei în informatică este enorm; foarte puţini dintre "butonari" realizează în ce măsură sculele pe care le folosesc există numai pentru că au fost iniţial concepute în termeni matematici. Un exemplu frapant este teoria compilării, care a fost dezvoltată la începutul anilor '70, şi care a deschis era unor limbaje mult mai clare şi mai coerente, care realmente au mărit enorm sfera de accesibilitate a calculatoarelor pînă la "marele public".

Scopul acestui articol *nu* este însă să vorbească despre această ramură, aşa că mă voi opri aici cu consideraţiunile care o privesc.

1.2 Sistemele

A doua parte a informaticii este un teren de activitate "inginerească"; denumirea sa dominantă pare să fie "sisteme". Trăsătura definitorie este că manipulează "sisteme" foarte complexe (din foarte multe părţi) a căror totalitate sau comportare nu se pretează foarte bine la o manipulare analitică (adică folosind uneltele teoretice). Munca în acest domeniu se bazează pe o mare cantitate de fapte empirice (observate experimental) şi pe capacitatea de a agrega mental părţile componente într-un tot.

1.3 Totul

Dichotomia aceasta (teorie-sisteme) este exprimarea dichotomiei ştiinţă-tehnică. Ştiinţa încearcă să descrie fenomene idealizate, simplificate, tehnica încearcă să aplice aceste descrieri obiectelor reale, care sunt o întreţesere încîlcită de fenomene simple, din care foarte adesea unele încă "necunoscute" din punct de vedere ştiinţific. Distanţa este cea de la ecuaţiile cîmpului magnetic ale lui Maxwell şi fizica cuantică la construcţia unui televizor[1].

Din observaţiile mele personale aceste două arii de activitate cer însuşiri deosebite. Există astfel teoreticieni renumiţi care nu ştiu mai mult decît cîteva comenzi ale unui calculator, pentru a manipula poşta electronică. Există de asemenea "legende vii" în sisteme care nu ştiu prea multe metode de sortare. Fireşte, există şi celelalte două extreme: matematicieni de forţă care totodată scriu şi manipulează sisteme extrem de complexe (două nume faimoase ar fi John von Neumann, care este unul dintre cei mai mari matematicieni ai secolului şi care este considerat părintele arhitecturii contemporane a calculatoarelor, şi Donald Knuth, care, pe lîngă cele 3 faimoase volume de teoria algoritmilor a scris programele TeX şi Metafont pentru tehnoredactare computerizată), şi există (marea masă a utilizatorilor de calculatoare) inşi care nu ştiu nici teorie şi nici sisteme.

2 Un raport tehnic în sisteme

Acest articol îşi propune să arate ceva din natura cercetării ştiinţifice şi a argumentării în domeniul "sistemelor". Eu însumi sunt un proaspăt învăţăcel, aşa că sunt mult mai multe fapte pe care nu le ştiu decît lucruri pe care le ştiu. Cu toate acestea aş vrea să spun cîteva lucruri neevidente (cel puţin eu nu le-am ghicit de unul singur) care ar putea fi de folos celor care activează în acest domeniu.

Prezentarea se bazează pe un proiect de curs pe care l-am realizat în toamna lui 1997. Proiectul şi raportul însoţitor au fost realizate de Raluca şi Mihai Budiu. Textul care urmează este destul de "greu"; mi-ar fi plăcut să mă pot apleca mai mult asupra unora dintre detalii, dar şi aşa textul

[1] Paragraful acesta trebuie luat ca o opinie strict personală.

a ieşit cam mare. Voi folosi tacit informaţii dintr-o mulţime de articole pe care le-am publicat anterior în PC Report.

Textul care urmează va încerca să evidenţieze următoarele două aspecte care trebuie puse în valoare într-un raport tehnic:

- Detaliile sunt cele care contează;
- O propunere de nouă tehnologie trebuie să demonstreze prin măsurători de performanţă o creştere a eficienţei faţă de vechile metode.

O să vedeţi că articolul poate părea plictisitor prin amănunte, mai ales în secţiunile de performanţă. Specifică tot felul de detalii, parametrii cu care se fac măsurătorile, tipul maşinilor, încărcătura reţelei, ce se măsoară, cînd începe şi cînd se termină măsurătoarea, etc. Cu toate acestea, este relativ tipic (nu ca nivel ştiinţific, ci ca alcătuire) pentru un articol de cercetare contemporan în "sisteme".

> În absenţa acestor detalii nu există nimic; acestea trebuie să fie suficiente ca oricine să poată duplica experimentele întocmai. Valoarea unui singur parametru poate schimba complet rezultatele, deci trebuie încercată menţinerea sub control a tuturor parametrilor semnificativi.

3 O propunere: "cîini de pază"

Dorinţa noastră iniţială a fost nu prea ambiţioasă: să implementăm în cadrul sistemului de operare Linux un mecanism descris în urmă cu 10 ani într-un articol apărut în "Computing Systems". Titlul articolului era "Watchdogs — Extending the UNIX File System", de Brian Bershad si Brian Pinkerton, de la universitatea Washington din Seattle[2]. Traducerea titlului ar fi "Cîini de pază: o extensie a sistemelor de fişiere din Unix".

Articolul propunea implementarea unui mecanism în interiorul nucleului prin care anumite fişiere pot fi supravegheate de către procese. Cînd un alt proces încearcă să acceseze un fişier supravegheat, supervizorul (cel care este de fapt numit "cîine de pază") este executat şi i se pasează informaţii despre tipul accesului încercat. Supervizorul decide care trebuie să fie rezultatul accesului.

3.1 Aplicaţii

Vom face mai tîrziu un desen care arată cum funcţionează un astfel de sistem. Pentru început să observăm că numărul de aplicaţii al unui astfel de mecanism este uriaş. Iată cîteva posibilităţi:

ACL (access control lists: liste de control al accesului): Unix oferă un sistem de protecţie al fişierelor relativ rudimentar, bazat pe diviziunea utilizatorilor în 3 categorii: utilizatorul (unic) care posedă un fişier, grupul utilizatorilor care posedă fişierul şi restul lumii. Se pot imagina scheme mai flexibile, în care un fişier are drepturi pentru mai mulţi utilizatori, de genul: X are voie să scrie dar Y nu are voie să citească. Un "watchdog" ar putea implementa un astfel de control;

[2] Departamentul de calculatoare al acestei universităţi este clasat între primele 10 din Statele Unite; Brian Bershad însuşi este o somitate în lumea sistemelor de operare.

fişiere active: în Windows NT fişierele sunt obiecte “active” în sensul că procesele pot să primească automat evenimente (events) atunci cînd alte procese acţionează asupra fişierelor. În acest fel un manager de fişiere (File Manager) poate afişa permanent situaţia cea mai recentă a fişierelor. În Unix aşa ceva nu este cu putinţă. Un watchdog ar putea însă intercepta acţiunile şi trimite semnale proceselor interesate de schimbările pe fişiere;

conţinut generat dinamic: un watchdog poate genera dinamic conţinutul fişierelor în funcţie de anume parametri, cum ar fi identitatea procesului care accesează fişierul (cu alte cuvinte fiecare proces ar putea vedea altceva în acelaşi fişier). Acest mecanism ar putea fi folosit pentru a implementa un server de web simplu emulînd mecanismele cgi-bin, prin care serverul execută alte procese cînd i se cer anume fişiere. Watchdog-ul poate subsuma aceste alte procese;

managere pentru “obiecte de memorie”: sistemul de operare Mach permite utilizatorilor să construiască propriile lor sisteme de memorie virtuală; fiecare zonă de memorie are un proces responsabil care paginează zona. În Unix de acest lucru se ocupă nucleul. Un watchdog, împreună cu apelul de sistem `mmap()`, ar putea permite implementarea unor mecanisme ca cele din Mach: `mmap()` transformă un fişier într-o zonă de memorie, iar managementul fişierului este făcut de un watchdog. Schema asta este foarte flexibilă; folosind-o se poate implementa de pildă un sistem de memorie partajată distribuită (DSM: distributed shared memory), în care mai multe procese de pe calculatoare diferite scriu în “aceeaşi” zonă de memorie;

colectare de statistici: putem scrie un watchdog care numără accesele la fişiere pentru a permite eventual optimizarea plasării fişierelor des accesate sau studii despre obiceiurile programatorilor;

mecanismul de portal din BSD Unix: BSD 4.4 conţine un mecanism numit “portal” care foloseşte nume de fişiere pentru “obiecte” din reţea; de pildă o legătură `telnet` TCP/IP la calculatorul foo.bar.org are un fişier numit `/portal/tcpip/foo.bar.org/telnet`. Un proces care deschide acest fişier stabileşte de fapt o legătură telnet cu acel calculator. (Un portal este un fel de URL).

controlul versiunii: sistemul de fişiere Translucent de la Sun crează pentru fiecare fişier cîte o “versiune” nouă atunci cînd este modificat, permiţînd astfel recuperarea versiunilor mai vechi. Watchdog-ul poate face ceva asemănător;

fişiere la distanţă: vom vedea această aplicaţie în articolul de faţă;

etc. mai putem imagina şi alte aplicaţii ale mecanismului de watchdog, dar ne oprim aici.

4 Arhitectura sistemului

4.1 Terminologie

Să stabilim nişte convenţii terminologice ca să simplificăm discuţia. Voi numi “watchdog” atît procesul care supervizează accesul la un fişier cît şi structurile de date din nucleu prin care watchdog-ul este implementat.

Procesele care acţionează asupra fişierului supervizat vor fi numite "clienţi" ai watchdog-ului.

4.2 Arhitectura

Un singur lucru mai avem de lămurit: cum comunică procesul watchdog cu nucleul. În Unix nu există nici un mecanism prin care nucleul să ceară servicii unui proces. Am ales următoarea variantă, inspirată de sistemul de fişiere `/proc`: un watchdog comunică cu nucleul folosind apelurile de sistem pentru a scrie/citi fişiere. (Această soluţie este diferită de cea oferită de articolul lui Bershad.)

Introducem un nou apel de sistem prin care nucleul crează un "fişier" special, de tip "watchdog". Să ne reamintim că în interiorul nucleului fişierele se reprezintă printr-o structură de date specială numită *vnod*[3]. Din multe puncte de vedere funcţionarea unui watchdog seamănă cu cea a unui *pipe* între două procese; în interiorul nucleului se vor implementa la fel, sub forma unor "fişiere" (sau mai exact vnod-uri).

Figura 1 ne arată cum lucrează un watchdog.

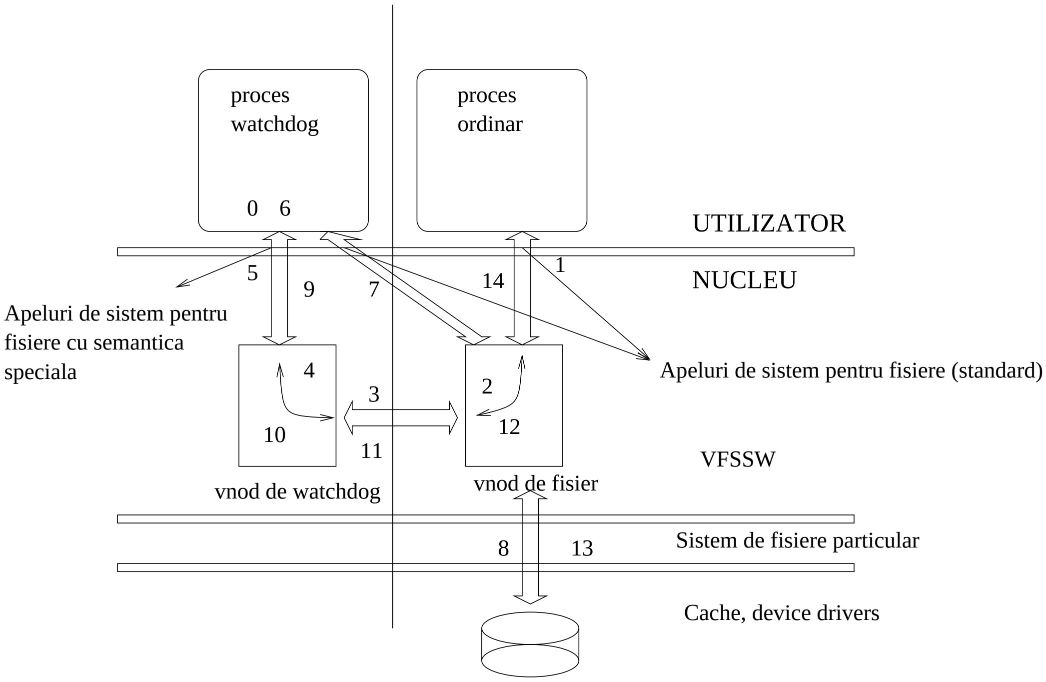

Figura 1: Arhitectura unui watchdog

Paşii execuţiei sunt următorii:

Întîi procesul supervizor execută apelul de sistem `watchdog(fisier)`, prin care anunţă nucleul că vrea să devină un watchdog pentru fişierul indicat. Nucleul construieşte în interior un fişier (vnod) care este asociat procesului watchdog şi returnează acest fişier special (să-l numim în cele ce urmează `wd`).

[3]Vedeţi şi articolul meu din PC Report din februarie 1998, a cărui copie este disponibilă din pagina mea de web.

0. Procesul watchdog încearcă să citească ceva din acest fişier special (cu `read(wd, request, sizeof(wd_request))`; nucleul blochează watchdog-ul în operaţia de citire pînă la apariţia unui client;
1. Un proces (client) încearcă să acceseze fişierul supervizat;
2. Nivelul VFSSW (Virtual File System Switch; descris în articolul citat mai sus din PC Report din februarie 1998) interceptează apelul de sistem;
3. Nucleul construieşte o structură de date (de tipul `wd_request`) care descrie apelul de sistem al clientului; pasează această structură în vnod-ul watchdog-ului.
4. Procesul client este suspendat şi watchdog-ul este trezit;
5. Procesul watchdog termină apelul de sistem început la pasul 0; rezultatul acelei citiri este structura `wd_request`;
6. Watchdog-ul descifrează structura şi acţionează în consecinţă;
7. Watchdog-ul ar putea decide să acceseze el însuşi fişierul supervizat;
8. Accesul watchdog-ului la acel fişier nu este interceptat ci merge direct la fişier;
9. Watchdog-ul pune rezultatul deciziei sale într-o structură de tipul `wd_reply` şi face un `write(wd, reply, sizeof(wd_reply))`, trimiţînd structura nucleului;
10. Nucleul decodifică răspunsul watchdog-ului;
11. Nucleul re-porneşte clientul blocat;
12. Nivelul VFSSW execută acţiunea indicată de watchdog; dacă watchdog-ul a trimis date acestea sunt pasate procesului;
13. Dacă watchdog-ul a spus că putem lăsa clientul să facă cu fişierul ce a cerut, atunci clientul accesează direct fişierul;
14. Apelul de sistem al clientului se termină.

Cu toţii sunt gata pentru un nou ciclu de operaţii de la pasul 0.

Observaţi că toată procesarea watchdog-ului este absolut independentă de tipul fişierului accesat (pentru că se petrece în nivelul VFSSW); fie că fişierul este DOS, Unix sau NFS, watchdog-ul face acelaşi lucru. Din cauza asta un watchdog poate superviza orice fel de fişier sau obiect care se reprezintă în nucleu ca un fişier, cum ar fi un periferic, un pipe sau o conexiune de reţea (socket).

4.3 Un exemplu de watchdog

Iată o implementare a unui proces watchdog foarte simplu, care permite deschiderea unui fişier, dar nu scrierea în acel fişier. Cînd un proces citeşte din fişier, rezultatul va fi întotdeauna şirul "Paseaza datele astea clientului cind citeste" (sau un prefix, dacă clientul a citit mai puţin decît lungimea şirului).

```
#include "watchdog.h"

int main()
{
   int fd, /* Fisier de supervizat. */
       wd; /* Interfata watchdog-ului cu nucleul. */
   char buf[] = "Paseaza datele astea clientului cind citeste";

   fd = open("nume_fisier", O_RDWR); /* e nevoie de permisiuni de scriere/citire. */
   wd = watchdog(fd);   /* apel de sistem: ne inregistram ca watchdog. wd=descriptor de fisier */

   while (1) {
      struct wd_request wq;   /* nucleul va descrie apelul de sistem aici */
      struct wd_reply   wr;   /* watchdog-ul raspunde nucleului aici */

      read(wd, (char*)&wq, sizeof(wq));     /* Asta blocheaza watchdog-ul pina
                                               cind nucleul are ceva de zis */
      switch (wq.operation) {
        case WD_OPEN:
                printf("Procesul %d deschide `nume_fisier', mod %d\n", wq.mode);
                wr.action = WD_ALLOW;  /* lasam procesul sa faca ce a cerut */
                break;
        case WD_READ:
                printf("Procesul %d citeste din `nume_fisier', offset %ld,"
                       "marime %ld\n", wq.pid, wq.offset, wq.size);
                wr.action = WD_FAKE;  /* Watchdog-ul o sa dea datele procesului */
                if (wq.size < sizeof(buf)) wr.size = wq.size;
                else wr.size = sizeof(buf);  /* Ii dam atitea date pentru read() */
                wr.buffer = buf;      /* Aici sunt datele de pasat procesului */
                break;
        case WD_WRITE:
                printf("Procesul %d vrea sa scrie in `nume_fisier', offset %ld,"
                       "marime %ld\n", wq.pid, wq.offset, wq.size);
                wr.action = WD_DENY;  /* Nu permitem scrierea. */
                break;
      }
      write(wd, (char *)&wr, sizeof(wr));    /* Trimite raspunsul nucleului. */
      if (wq.operation == WD_CLOSE) break;   /* Daca clientul a inchis fisierul, terminam*/
   }
   close(fd);  /* Gata cu supervizarea fisierului. */
   close(wd);  /* Inchidem fisierul ``watchdog''. */
}
```

Iată cum arată în implementarea noastră informaţiile de la nucleu şi răspunsurile:

```
enum operation {                /* apeluri de sistem interceptate */
        WD_READ, WD_WRITE, WD_OPEN, WD_CLOSE, WD_LSEEK
};

enum action {                   /* ce spune watchdog-ul nucleului */
        WD_ALLOW,               /* :da-i voie sa faca ce a cerut */
```

```
        WD_DENY,                /* :clientul nu are voie sa faca asta */
        WD_FAKE                 /* :OK, paseaza datele la/de la
                                        watchdog (nu din fisier) */
};

struct wd_request {     /* Descrierea operatiei, primita de watchdog de la nucleu */
        enum operation operation; /* operatia */
        int fd;                 /* care fisier */
        int pid;                /* care proces */
        int uid;                /* posesorul procesului */
        int gid;                /* grupul procesului */
        long offset;            /* unde in fisier se opereaza */
        int size;               /* citi octeti se transfera */
        int mode;               /* drepturile cu care se deschide fisierul */
        int whence;             /* pentru lseek: de unde se face lseek */
        int inode_count;        /* cite procese au deschis acest fisier */
};

struct wd_reply {               /* Raspunsuri de la watchdog la nucleu: */
        enum action action;     /* ce actiune sa faca nucleul */
        int fd;                 /* pe care fisier (pot fi mai multe simultan) */
        char * buf;             /* daca e FAKE aici sunt datele pentru read/write */
        int count;              /* cit de multe date pentru read/write */
};
```

Singura operaţie care merită explicaţii este `FAKE` ("simulează"). Cînd un client face o citire şi watchdog-ul răspunde cu `FAKE`, nucleul trebuie să preia datele de citit nu din fişier ci din buffer-ul indicat de watchdog.

Cînd un client face o scriere şi watchdog-ul zice `FAKE`, nucleul nu trebuie să scrie datele în fişier ci în buffer-ul indicat de watchdog.

5 O întrebare: eficacitatea

Iată deci principalul avantaj al unui astfel de mecanism: *flexibilitatea*. Un watchdog ne permite să implementăm o sumedenie de servicii noi dar fără a modifica cu nimic interfeţele existente, păstrînd în întregime compatibilitatea. Procesele care lucrează cu fişiere nu vor afla niciodată că de fapt vorbesc cu un watchdog, pentru ca folosesc exact aceleaşi apeluri de sistem ca mai-nainte.

Nimic nu e pe gratis, însă. Ce avem de plătit pentru această flexibilitate?

Răspunsul este clar: performanţă. De fiecare dată cînd un proces accesează un fişier cu watchdog avem de făcut multe alte lucruri în plus faţă de cazul "normal".

Aceasta este una din tezele acestui articol:

> *Înainte de a ne lansa în implementarea unui proiect trebuie să estimăm dacă merită. Pentru asta trebuie să avem o idee despre performanţa sistemului.*

Mai ales dacă avem de construit un sistem foarte mare, trebuie să ne facem o idee despre performanţă *înainte* de a construi sistemul. Dacă rezultatul o sa fie de 10 ori mai lent decît orice există atunci nu merită; trebuie reconsiderat întreg proiectul.

Hai deci să estimăm ce avem de pierdut. Care este deci *cărarea critică*: traseul execuţiei care costă cel mai mult.

5.1 Componentele costului

Din păcate cam toţi paşii din figura de mai sus 1 sunt pe cărarea critică, mai puţin pasul 0, care este executat în paralel de watchdog şi de client.

5.1.1 Costul fără watchdog

Dacă nu avem watchdog, o operaţie pe un fişier constă doar din paşii 1, 2, 13 şi 14. Toţi ceilalţi paşi sunt suplimentari (overhead), iar costul lor *relativ* trebuie să fie scăzut ca operaţia watchdog-ului să nu fie disruptivă pentru performanţă.

5.1.2 Costul suplimentar cu watchdog

Ia să vedem ce fel de costuri plătim pentru watchdog:

- 2 comutări de privilegiu între nucleu şi spaţiul utilizator: una în pasul 5 cînd nucleul vorbeşte cu watchdog-ul şi una în pasul 9, cînd watchdog-ul raspunde; asta înseamnă cam cît costul unui apel de sistem obişnuit;
- În pasul 7 watchdog-ul ar putea face el însuşi apeluri de sistem (depinde de funcţionalitatea pe care o oferă);
- Pasul 8 al watchdog-ului ar putea costa foarte mult: cît un acces la fişier;
- Paşii 12 sau 3 copiază date între proces şi watchdog; dacă datele sunt multe asta costă substanţial;
- Paşii 4 şi 11 comută execuţia între proces şi watchdog; avem deci cel puţin două comutări de procese.

Ceilalţi paşi sunt relativ neglijabili în cost[4].
Rezultatele sunt prezentate într-un tabel:

Operaţie suplimentară	Cantitate
Apel de sistem	2
Comutare de proces	2 — zeci
Copiere a datelor	2

Copierea datelor între procese. Întrebarea este "cum putem copia date între două procese"? Problema constă în faptul că atunci cînd un proces se execută, foloseşte propriul său spaţiu de memorie virtuală; fiecare proces are propria lui adresă 5, care nu coincide cu a nici unui alt proces. Atunci cum pot muta date de la o adresa 7 dintr-un proces la adresa 12 a altui proces? Nicicum direct, pentru că atunci cînd se execută primul, adresa celui de-al doilea nu are sens şi invers.

Soluţia este ca datele să fie copiate în *interiorul nucleului*, care aparţine ambelor spaţii de adrese. (Acesta este şi mecanismul folosit de *pipe* în Unix pentru a transfera datele între procese.) Soluţia este deci următoarea:

- Pentru fiecare vnod watchdog alocăm în nucleu un buffer temporar (nu prea mare, pentru că memoria nucleului costă, nefiind paginabilă);
- Ca să copiem date între două procese facem aşa: executăm primul proces, luăm date cît să umple buffer-ul, îl blocăm pe primul, după care comutăm la al doilea;

[4] Mai există un cost ascuns al serviciilor oferite de un watchdog: şi anume memoria watchdog-ului este paginabilă, spre deosebire de a nucleului. Dacă implementăm un serviciu în nucleu ar putea fi mai rapid decît dacă îl punem în watchdog pentru că memoria nucleului rămîne tot timpul în memoria fizică. Vom ignora acest lucru în acest articol.

- Cînd se execută al doilea golim buffer-ul, blocăm procesul şi ne întoarcem la primul;
- Repetăm operaţia pînă am transferat toate datele.

Iată de ce transferul de date implică mai multe comutări de procese; numărul lor depinde de raportul dintre cantitatea de date transferate şi mărimea buffer-ului intern. În implementarea noastră am folosit un buffer intern de 8Kb (două pagini). Asta înseamnă că 1Mb de transferat implică $1000/8 = 125$ de comutări de proces înainte şi înapoi.

5.2 Costul componentelor

Acum hai să vedem cît costă *concret* fiecare componentă. Asta o să ne dea o idee despre fezabilitate. Nişte măsurători simple[5] pentru sistemul pe care am implementat watchdog-ul (PII 266MHz cu 64M RAM, rulînd Linux 2.0.30) au dat următoarele valori:

Operaţie	Cost în μs
Apel de sistem	2
Comutare de proces	7.5
Copiere de date	20 000-40 000/Mb

(Durata copierii depinde de mai mulţi factori; dimensiunea influenţează rata de succes în cache, care influenţează viteza.)

Se vede că dominant este costul *copierii*, chiar pentru cantităţi mici de date (1Kb ar lua peste 10μs) pentru o copiere, iar noi avem de făcut două, una “în sus” şi una “în jos”.

5.3 Performanţa relativă

Toate aceste evaluări nu s-au preocupat de loc de costul accesului la disc. Ori discul este un periferic extrem de lent; numai mutarea braţului durează de ordinul a 10ms (10 000 de μs!). Costul *suplimentar* (overhead) al watchdog-ului pentru astfel de circumstanţe devine oarecum neglijabil.

Citirea dintr-un fişier de pe disc (deschis dinainte) a 1Mb de date durează cam 150ms. Costul operaţiilor watchdog-ului pentru o operaţie care implică 1Mb este cam de 40ms + 0.0075ms * 125 + 0.004 = 41ms, deci cam de 25%.

Pe scurt, iată importanţa costului suplimentar al unui watchdog, în funcţie de tipul operaţiei efectuate:

Discul accesat	Cantitate de date	Overhead
nu	puţine	1000%
nu	multe	200%
da	oricîte	25-50%

5.4 Utilitatea proiectului revizuită

Iată cum o simplă evaluare *teoretică* ne spune deja foarte multe lucruri despre mecanismul de watchdog: ne spune de pildă ce fel de servicii merită să fie oferite de watchdog şi care nu. Dacă un serviciu costă mult mai puţin decît overhead-ul, atunci watchdog-ul este prea costisitor şi trebuie să căutăm altă soluţie.

De exemplu nu rentează să implementăm ACL (controlul accesului) cu watchdog, pentru că astfel de operaţii nu transferă date şi nu accesează discul (decît prima oară cînd un fişier este deschis; următoarele accese vor merge în cache). Pentru astfel de operaţii citim din tabelul de mai sus că watchdog-ul încetineşte operaţia de 10 ori! Locul unui astfel de serviciu este în nucleul sistemului de operare.

Un watchdog nu va fi prea costisitor în următoarele cazuri:

[5] Sper să pot discuta într-un articol viitor despre metoda prin care am făcut aceste măsurători.

- Cînd operaţiile pe fişiere implică acces frecvent la disc (de exemplu datorită unei localităţi proaste în cache: accese aleatoare);
- Cînd operaţiile se fac pe cantităţi mari de date şi pot beneficia de optimizări speciale specifice aplicaţiei.

5.5 O măsurătoare a performanţei

Acum că am văzut că un watchdog nu e complet inutil ne-am apucat să implementăm mecanismele necesare. Am construit în nucleu intercepţia apelurilor de sistem şi am scris apoi un watchdog foarte foarte simplu (care seamănă cu cel din exemplul de cod anterior), cu care am măsurat performanţa reală. Acum vom vedea dacă estimarea noastră teoretică de mai sus a fost realistă.

Am făcut un set de 5 experimente, după cum urmează:

1. Am executat un apel de sistem `lseek()` (nu atinge discul);
2. Am executat un `lseek()` după care am citit din fişier un octet;
3. Am executat `lseek()` şi am citit o pagină (4K);
4. `lseek()` urmat de citirea a 40K;
5. `lseek()` urmat de citirea a 1M.

Am măsurat costul acestor operaţii atunci cînd sunt efectuate de cinci entităţi diferite:

1. Costul atunci cînd folosim pentru acelaşi scop două procese care comunică printr-un *named pipe* (am remarcat mai sus asemănarea dintre un watchdog şi un pipe); (operaţia `lseek()` nu are sens pe un pipe; se va termina mereu cu o eroare; am pus-o ca să măsurăm acelaşi lucru în toate cazurile);
2. Costul operaţiei executate de sistemul de operare (SO);
3. Costul operaţiei atunci cînd watchdog-ul interzice operaţia;
4. Costul atunci cînd watchdog-ul zice "da" (ALLOWED) iar nucleul execută operaţia;
5. Costul atunci cînd watchdog-ul execută el însuşi operaţia şi pasează datele clientului (FAKE).

Toate operaţiile se fac cu datele în cache; discul nu este folosit niciodată. Iată rezultatele, în microsecunde:

Operaţie	seek	read(1)	read(4096)	read(40K)	read(1M)
SO	2.1	5.1	8.6	220	21406
pipe	2.2	11	66	720	57812
interzis	17	33	33	33	33
permis	17	35	42	270	22968
watchdog	20	44	88	634	50300

Rezultatele par să confirme estimările noastre: interzicerea costă o valoare constantă: 17μs pentru fiecare serviciu (2 comutări de procese plus 1 apel de sistem). Darea permisiunii adaugă un cost constant la operaţia cu SO: cel al comutării la watchdog şi înapoi. Cînd watchdog-ul face operaţia el însuşi costul copierii datelor între watchdog şi client apare în ecuaţie şi domină la cantităţi mari de date.

Demn de remarcat este că datorită unor optimizări simple, watchdog-ul este mai rapid ca un *pipe*. Codul nostru foloseşte buffere în nucleu de două ori mai mari decît un pipe, (deci face de două ori mai puţine comutări), şi apoi permite unui apel de sistem `read()` să returneze mai mult decît o pagină de date (pipe nu permite aşa ceva), deci reduce numărul de apeluri de sistem executate de client.

6 O aplicaţie a mecanismului: un sistem de fişiere la distanţă

Un watchdog este un *mecanism*. Avem nevoie de o aplicaţie pentru a demonstra utilitatea lui. Am ales o aplicaţie al cărei cost intrinsec promitea să facă overhead-ul unui watchdog neglijabil: accesul la fişiere la distanţă, prin reţea. Timpul mare de transfer al unei reţele este factorul pe care ne bazăm. Să observăm că latenţa (timpul de transfer) reţelelor este relativ mare, fiind limitată de viteza de transmisiune, care nu poate fi mai mare ca viteza luminii. Cantitatea de date transmise (bandwidth sau throughput) este oarecum nelimitată; vezi reţelele giga-bit/secundă care devin populare.

6.1 Alţi candidaţi şi slăbiciunile lor

Sisteme de fişiere la distanţă există de cînd lumea; NFS (Network File System) de la firma SUN este extrem de popular. Ne-am propus să scriem serviciul nostru în aşa fel încît să batem NFS-ul. Pentru asta trebuie să profităm de lucrurile la care NFS este slab. NFS este (în implementarea pe Linux) foarte slab la scrieri: deşi ţine într-un cache la client datele citite, scrierile le trimite direct la server, ca nu cumva alţi clienţi care folosesc acelaşi fişier să nu le vadă. Mărimea blocului de cache pentru NFS este de 8K.

Un al doilea candidat faimos este sistemul de fişiere distribuit dezvoltat la Universitatea Carnegie Mellon numit AFS (Andrew File System). Andrew este mult mai inteligent şi foloseşte mult mai bine cache-ul de la client[6]. Mărimea blocului pentru cache-ul Andrew este de 64K.

6.2 Un serviciu de "caching" inteligent

Am discutat pe larg despre performanţa în sistemele de fişiere într-un articol publicat în PC Report în noiembrie 1997. Voi recapitula aici faptele care ne interesează:

Să observăm că pentru o aplicaţie scrierile şi citirile se comportă în mod deosebit:

- Durata scrierilor nu contează, pentru că odată trimise datele aplicaţia poate continua execuţia, iar datele se îndreaptă spre destinaţia lor în acelaşi timp. Aplicaţia nu trebuie să aştepte terminarea scrierii. Din cauza asta scrierile sunt *asincrone*.
- Citirile costă, pentru că aplicaţia nu poate continua execuţia fără prezenţa datelor. Dacă nu se folosesc tehnici speciale de anticipare a citirilor (read-ahead sau prefetching) citirile sunt practic *sincrone*.

Toate sistemele de fişiere de reţea încearcă să amortizeze latenţa reţelei făcînd transferuri în *blocuri mari* de date. Cam tot atîta costă să aduci un octet cît costă 10K, aşa că de ce să nu aduci 10K; poate mai ai nevoie de ei. Această caracteristică a cache-urilor, de a manipula blocuri mari, este uneori şi slăbiciunea lor.

Iată de ce: problema constă în *scrierile mici*. Cînd un proces scrie mai puţin de un bloc, pentru că cache-ul manipulează numai blocuri întregi, trebuie să aducă de la server blocul întreg şi să aplice scrierea peste bloc. Din cauza asta *scrierile mici sunt sincrone*, ca şi citirile.

Acest lucru este adevărat atît în NFS cît şi în AFS.

Am scris deci propriul nostru serviciu de cache. Am încercat să exploatăm slăbiciunile sistemelor de fişiere de reţea în două feluri:

- Cache-ul pe care l-am implementat nu mai foloseşte ca unitate de bază blocul, ci *intervalul*. Dimensiunea intervalului nu mai este fixă; dacă utilizatorul manipulează 1Mega, atunci cache-ul stochează un mega şi eventual transmite un mesaj de 1M, fiind mult mai eficace decît multe mesaje mici;
- Cache-ul nostru poate absorbi scrierile mici asincron; sunt doar nişte intervale ca toate altele.

[6] AFS este un sistem de fişiere "mondial"; sper să pot reveni într-un articol asupra lui. Practic este un singur arbore de directoare pentru cîteva sute de instituţii din cîteva zeci de ţări; cu un simplu `cd` eu pot citi directoare aflate pe servere din Elveţia. Toate calculatoarele din universitatea Carnegie Mellon sunt clienţi ai acestui sistem de fişiere.

Deşi este foarte interesant în sine, nu vom explora aici arhitectura cache-ului cu intervale. Doar intern cache-ul manipulează datele tot în blocuri, iar mărimea blocului din cache are un oarecare impact asupra performanţei.

Să vedem cum este el folosit împreună cu un watchdog pentru a oferi servicii îmbunătăţite clienţilor.

6.3 Serviciul de fişiere la distanţă

Am implementat deci un watchdog care este şi un cache manager. Am implementat apoi un server care va comunica cu watchdog-ul prin TCP-IP. Cache-ul este împărţit în două thread-uri care funcţionează asincron:

- "Jumătatea de jos" (bottom half) se ocupă de comunicarea în reţea şi goleşte asincron cache-ul cînd e plin;
- "Jumătatea de sus" (top half) este şi watchdog-ul: interceptează apelurile clienţilor şi copiază datele din/în cache.

Pachetul de thread-uri folosit a fost scris tot de autorul acestui articol, şi a fost prezentat în PC Report din ianuarie 1997.

Figura 2 prezintă arhitectura sistemului de fişiere.

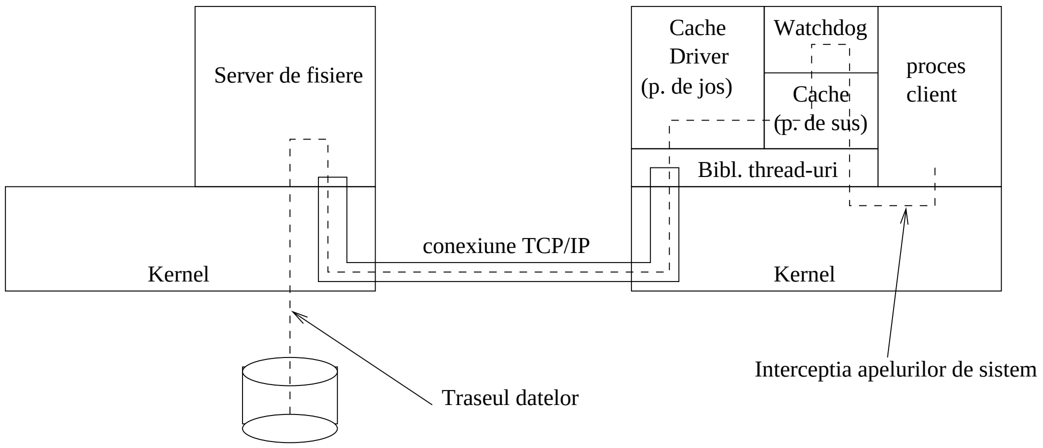

Figura 2: Arhitectura Sistemului de fişiere la distanţă.

6.4 Măsurători de performanţă

Iată secţiunea crucială, cea a măsurătorilor de performanţă comparative. Am făcut experimente numai pentru cazurile care ne interesează (unde anticipăm că sistemul nostru o să fie "mai tare"), şi anume:

- Scrieri mici răspîndite, care în alte sisteme sunt sincrone;
- Operaţii cu date foarte multe, secvenţiale (unde intervalele mari transmise sunt foarte eficace).

În testele noastre am avut sub control calculatoarele client (întotdeauna maşina Linux descrisă mai sus). Serverul pentru watchdog şi NFS a fost pe o maşină Sun SPARC4. Maşinile nu erau încărcate la măsurători. Maşinile sunt pe acelaşi cablu Ethernet, separate doar de bridge-uri.

Nu am avut sub control:

- Încărcarea reţelei;

- Serverul AFS.

Din cauza asta am repetat măsurătorile de mai multe ori şi am considerat *cele mai bune* valori ale performanţei.

6.4.1 Teste secvenţiale

Iată un test secvenţial care doar copiază un fişier de 2M între cele două maşini (cu comanda Unix `cp`). Cache-ul nostru avea un bloc intern de 64K. Cache-urile *erau goale* la începutul transferului.

Cache-ul folosit de AFS şi NFS poate creşte pînă la aprox. 60M, mărimea memoriei calculatorului. WD a avut un cache fixat la 640K capacitate totală.

Comparăm 4 sisteme:

UFS: Sistemul de fişiere din Unix de pe discul local;

NFS: Sun Network File System;

AFS: Andrew File System;

WD: sistemul nostru de fişiere bazat pe watchdog.

Ceea ce măsurăm este *timpul total real al copierii* (adică cît timp trece de cînd începe pînă se termină); asta nu e totuna cu timpul de execuţie al proceselor, pentru că procesele vor dormi mult timp aşteptînd datele, deci nu vor consuma timp de procesor. Copierea se termină cu un apel de sistem `fsync()` care ne asigură că toate datele au fost trimise la destinaţie, şi că nu au rămas în cache-ul local!

Protocol	Sursă	Destinaţie	Timp (s)
UFS	disc local	disc local	0.2
NFS	la distanţă	disc local	8.4
WD	la distanţă	disc local	6.9
AFS	la distanţă	disc local	5.3
NFS	disc local	la distatţă	61.0
WD	disc local	la distanţă	11.3
AFS	disc local	la distanţă	9.5

După cum se vede am reuşit să-l batem pe NFS. NFS face citire anticipată (read-ahead), de aceea performanţa la citire este rezonabilă. NFS trimite datele una cîte una în blocuri de 8K, deci pierde foarte mult timp la scrieri pentru a iniţia pachetele.

AFS trimite blocuri de 64K, deci este foarte eficient. AFS face şi el citire anticipată. AFS este implementat în nucleul sistemului de operare (ca şi NFS), deci nu are overhead atît de mare ca WD pentru comutarea proceselor şi copierea datelor (mai ales!). De aceea este mai performant.

6.4.2 Performanţa secvenţială variind bufferele

Datele trebuie să circule între client şi server prin două canale:

- Între watchdog şi serverul la distanţă pe reţea;
- Între watchdog şi clientul local prin memorie.

Performanţa primului transfer este influenţată de mărimea blocului intern de cache folosit (deşi noi transmitem mai multe blocuri pe reţea într-o singură operaţie, pentru că folosim apelul de sistem `writev()`, acesta impune un număr maxim de 16 blocuri scrise simultan). Performanţa celei de-a doua este influenţată de *cîte date cere clientul într-un apel de sistem*. Cît cere un client într-un apel de sistem constituie pentru cache un interval.

De exemplu, programul `cp` copiază datele în felii de cîte 4K, chiar dacă fişierele sunt uriaşe.
De aceea am scris un alt client, care face ce face şi `cp`, dar care face transferul datelor în felii mai mari. Iată performanţa măsurată, din nou în secunde:

Interval	1K	4K	16K	64K
bloc în cache	Spre discul local			
64K	8	7	7	8
32K	13	13	13	9
16K	27	27	28	8
8K	54	54	27	8
	Spre discul distant			
64K	10	10	10	4
32K	16	16	16	4
16K	30	23	4	4
8K	48	57	4	4

Performanţa depinde mai puţin de mărimea internă a blocului (cum am spus, cache-ul poate trimite multe blocuri pe reţea dintr-un foc), cît de mărimea intervalului (cerută printr-un apel de sistem). Cu cît intervalele sunt mai mari, cu atît performanţa este mai bună.

Performanţa scrierilor este excelentă pentru că scrierile sunt asincrone tot timpul (indiferent de raportul interval/bloc).

O anomalie avem în tabel pe coloana a treia, unde scăderea blocului creşte performanţa, un rezultat complet ne-intuitiv. Acesta este un artifact al modului de implementare: inspectarea codului cache-ului a revelat faptul că în cazul în care intervalul este mai mare ca blocul, totul e în regulă, dar atunci cînd blocul este mai mare decît intervalul, cache-ul iniţiază operaţii care puteau fi evitate. (Iată deci cum măsurătorile descoperă deficienţe de implementare şi sugerează îmbunătăţiri!) Asta se întîmplă pe coloana a treia: cînd blocul devine mai mic decît intervalul performanţa creşte brusc.

6.4.3 Teste de acces aleator

Programele de calcul intensiv (scientific computing) manipulează adesea cantităţi enorme de date într-un mod cvasi-aleatoriu şi în bucăţi mici (au deci localitate proastă în cache şi poate scrieri mici). Am scris două aplicaţii (oarecum artificiale, dar reprezentative la o scară redusă, sperăm) care încearcă să simuleze astfel de accese:

- Un program care înmulţeşte două matrici de 50*50 de elemente direct într-un fişier (în total 10K);
- Un program care sortează prin selecţie 200 de numere aleatoare dintr-un fişier "in situ" (800 octeţi).

Am plasat fişierul la distanţă şi am pus blocul în WD la 4K. Iată timpii măsuraţi:

Test	Protocol	Durată (s)
Matrici	UFS	1.6
	NFS	301.0
	WD	22.5
	AFS	4.6
Sortare	UFS	0.2
	NFS	13.6
	WD	1.7
	AFS	0.4

Performanţa lui NFS este îngrozitor de slabă (de 12 ori mai lent decît metoda noastră!) Măsurători care variau dimensiunea n a matricii au arătat că pentru valori mici ale lui n timpul lui NFS creşte ca n^2. Pentru că înmulţirea face n^3 citiri şi n^2 scrieri deducem că problema lui NFS sunt scrierile mici (exact cum am anticipat).

AFS este excelent pentru că toate datele încap într-un singur bloc din cache (64K) care este adus imediat la început, deci face un singur transfer pe reţea.

WD este mai slab din cauza overhead-ului comutării proceselor.

6.4.4 Un test greu pentru AFS: "strides"

Am vrut să găsim şi un test dificil pentru AFS. Pentru că AFS aduce datele în blocuri de 64K soluţia era să facem un test care exploatează foarte puţin din fiecare felie de 64K, deci care-l pune pe AFS la muncă degeaba.

Testul este simplu: scrie într-un fişier pre-existent de 2M petice de cîte 20 de octeţi (scrieri mici!) la distanţe variabile (distanţa dintre două scrieri o vom numi "stride"). Am făcut două serii de teste (blocul în cache-ul WD de 64K):

- În care cache-ul local la client era pre-încărcat cu fişierul (cache "cald");
- În care cache-ul local era gol ("rece").

Iată rezultatele:

Protocol	Cache	Stride	Intervale scrise	Timp (s)
AFS	rece	1K	64*25	9.8
AFS	cald	1K	64*25	5.5
AFS	rece	64K	25	10.0
AFS	cald	64K	25	5.6
WD	rece	1K	64*25	6.3
WD	cald	1K	64*25	6.4
WD	rece	64K	25	6.4
WD	cald	64K	25	6.4

WD este remarcabil de constant, pentru că manipulează intervalele scurte (20 de octeţi) în exact acelaşi fel indiferent unde se află (practic datele sunt permanent în cache). AFS este mai slab decît WD cînd are cache-ul gol, dar nu cu mult mai slab!

Numerele arată că performanţa AFS nu depinde de cîte *date* se scriu, ci de cîte intervale de 64K se traversează; chiar dacă scriu 64*25 intervale sau doar 25, dacă ating acelaşi număr de blocuri, tot atîta contează.

7 Concluzii

Acest proiect demonstrează un lucru foarte interesant:

> Preţul flexibilităţii obţinute prin implementarea serviciilor înafara nucleului (un watchdog, de pildă) poate fi cîteodată răscumpărat dacă serviciile oferite se potrivesc foarte bine cu nevoile clientului. Nucleul oferă un singur serviciu, care este eficace pentru o clasă largă de aplicaţii (de exemplu cele care au localitate bună în cache). Dar folosind schema noastră putem implementa *simultan* diferite scheme de cache pentru aplicaţii diferite, potrivind comportarea cache-ului pentru fiecare aplicaţie, şi optimizînd independent.

8 Bibliografie

Deşi nu am citat aceste lucrări, raportul de faţă se bazează pe ele. Rolul unei bibliografii într-un articol este enorm pentru cititori, dar despre asta altă dată[7].

References

[Bersh88] B. B. Bershad, C. B. Pinkerton. Watchdogs — Extending the UNIX File System. *Computing Systems, 1, 1988*, pp. 169.

[Saltz84] J. Saltzer, D. Reed, D Clark — The End-to-End Argument in System Design. *ACM Transactions on Computer Systems, vol 2, nr. 4, 1984*, pp 277–288.

[Faul91] R. Faulkner, R. Gomes. The Process File System and Process Model in UNIX System V. *Proceedings of the 1991 Winter USENIX Conference*, Jan. 1991, pp.243–252.

[Klei86] S. R. Kleiman, 1986. Vnodes: An Architecture for Multiple File System Types in Sun Unix. *USENIX Summer Conference Proceedings 1986*, pp. 238–247.

[Vaha96] Uresh Vahalia. Chapters 9, 10, 11 in *UNIX Internals.*, Prentice Hall, 1996.

9 Traseul parcurs

Mai interesante decît concluziile (care ştiinţific vorbind nu sunt extraordinare) este drumul parcurs. Să-l revedem:

1. Un proiect este propus;
2. Este comparat cu alte proiecte similare sau disimilare (această secţiune este mai puţin reprezentată în acest articol);
3. O evaluare teoretică a performanţei este făcută; acesta este şi un studiu de fezabilitate; eventual proiectul este simulat sau evaluat din părţile componente;
4. Designul este adaptat pentru a ataca părţile slabe indicate de evaluare;
5. O serie de ipoteze asupra comportării sistemului sunt enunţate explicit;
6. Proiectul este implementat;
7. Proiectul este măsurat cu mare grijă, într-o varietate de circumstanţe, controlînd (pe cît posibil) toţi parametri influenţi;
8. Eventualele slăbiciuni sunt remediate;
9. Concluziile verifică în ce măsură ipotezele s-au dovedit corecte şi explică rezultatele obţinute;
10. Noi direcţii de cercetare şi întrebări sunt formulate (lipsesc din acest text).

Iată de ce spun că în sisteme totul stă în detalii: dacă pentru watchdog raportul valorilor duratelor diferitelor componente ar fi fost altul (de exemplu dacă un apel de sistem ar fi fost de 10 ori mai costisitor decît copierea datelor), atunci relevanţa şi aplicabilitatea watchdog-ului ar fi fost cu totul altele. Cum spunea cineva în fizică:

> Dacă ai de-a face cu valori [pentru un parametru] care diferă cu un ordin de mărime, de fapt ai de-a face cu două fenomene diferite.

Asta am avut de zis; probabil am fost cam ocolit…

[7] E adevărat că în textele trimise la PC Report am omis în mod sistematic bibliografiile. Mea culpa.