“Micro” sau “macro”?

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

12 iunie 1997

Subiect: Nucleele monolitice par să fie preferate în sistemele de operare.

Cunoştinţe necesare: Cunoştinţe de bază despre sisteme de operare.

Cuvinte cheie: monolit, micronucleu, apel de sistem, proces, domeniu protejat.

Contents

De la apariţia ideii de micro-nucleu aceasta a suscitat un enorm entuziasm printre cercetători şi industriaşi. Tehnica promitea să rezolve elegant o mulţime de probleme din proiectarea sistemelor de operare, şi să permită scrierea de sisteme distribuite cu mare uşurinţă. În mod paradoxal însă, la această dată toate sistemele de operare de uz general au mai curînd o arhitectură monolitică. Chiar şi despre Windows NT, un cal pe care multă lume serioasă pariază ca învingător în cursa sistemelor de operare, se rîde adesea: “a plecat ca un micro-nucleu, dar s-a umflat pînă a ajuns mai mare ca un macro-nucleu”.

Acest articol îşi propune să explice care este motivaţia acestei spectaculoase rezistenţe a tehnologiei tradiţionale. Pentru cei nerăbdători concluzia se poate rezuma într-un rînd: costul serviciilor este prea mare (mult prea mare) într-un sistem de operare micro-nucleu.

Am spus mai sus “sisteme de operare de uz general”. Toate consideraţiile arhitecturale pe care le prezint sunt valabile pentru majoritatea sistemelor de operare existente la zi. Consideraţiile despre eficienţă, care sunt cruciale în supravieţuirea comercială a unui sistem, sunt însă semnificative numai pentru sistemele de operare pentru calculatoare obişnuite. Prin contrast, sistemele de operare specializate (de exemplu sistemele de timp real pentru controlul proceselor, sau pentru maşini electronice de jocuri) sunt într-adevăr micro-nuclee, şi îşi fac foarte bine treaba lor. Cheia este însă aceasta: treaba lor este într-adevăr foarte specializată; o maşină SEGA de jocuri electronice nu are nici disc, nici reţea, nici periferice prea multe, aşa că sistemul de operare este special scris. Atenţia noastră se apleacă mai ales asupra sistemelor tip Unix/Windows (3.1/NT/95)/VMS, care sunt concepute să permită rularea unei varietăţi nelimitate de aplicaţii şi partajarea resurselor între programe care nu ştiu unul despre celălalt, adesea în medii “deschise” (reţele).

O să procedez pe parcursul acestui articol indirect: voi atinge tot felul de probleme care aparent nu au mare legătură cu subiectul central, după care, în final, într-o secţiune sumară voi arăta cum consecinţele faptelor pe care le-am tot înşiruit, şi pe care le socotesc nu lipsite de interes în sine, se adună întru concluzia indicată mai sus, care promite dominaţia sistemelor monolitice.

1 Serviciile oferite de nucleu: apeluri de sistem

În această secţiune voi face o scurtă recapitulare a modului de funcţionare al nucleului, pentru a înţelege de unde izvorăsc toate problemele. Pentru că am vorbit aiurea despre aceste lucruri mai pe larg, aici voi fi oarecum succint. Cititorul interesat poate găsi o descriere a funcţionării unui nucleu de sistem de operare în articolul meu publicat în serial în PC Report în septembrie/octombrie 1996[1].

Nucleul unui sistem de operare se poate asemui cu o *bibliotecă* de funcţii care sunt puse la dispoziţia proceselor utilizatorilor. Practic întreg accesul la perifericele conectate este mediat de nucleu, din motive de reutilizare a codului, eficienţă şi, mai ales, securitate[2].

Pentru utilizatorul normal acest lucru se manifestă prin prezenţa unei colecţii de funcţii gata făcute, cu care el poate manipula perifericele (terminal, disc, fişiere, reţea, etc.). Un exemplu tipic este funcţia `write()` din Unix, prin care se pot trimite date spre un periferic. Funcţiile puse la dispoziţie de către nucleu se numesc *apeluri de sistem (system calls)*.

O altă funcţie importantă a nucleului, vizibilă utilizatorului prin apeluri de sistem pentru crearea, distrugerea şi manipularea proceselor, este cea de management al proceselor. Un proces este un program care se execută. Nucleul permite mai multor programe independente să fie

[1] Pentru cei care nu au cumpărat revista, articolele la care fac referinţă sunt disponibile în postscript din pagina mea de web.

[2] Se înregistrează fireşte şi tendinţe de a permite accesul proceselor direct la periferice, cum ar fi de exemplu în tehnologia Unet, în care procesele pot scrie direct pe placa de reţea; deocamdată sistemele comerciale însă nu au mers atît de departe.

"încărcate" în memorie, puse în execuţie, oprite şi terminate. O funcţie care este mai rar sub controlul utilizatorului este cea de "planificare" (scheduling) a proceselor: oprirea proceselor care s-au executat prea mult şi pornirea celor care tînjesc după puţină activitate.

Nucleul implementează de asemenea noţiunea de *spaţiu de adrese*, folosind *sistemul de memorie virtuală*. În arhitecturile "clasice" fiecare proces are impresia că posedă în întregime memoria calculatorului. Acest truc este realizat folosind translatarea adreselor (address mapping): pentru fiecare proces nucleul menţine o listă a zonelor de memorie care-i sunt vizibile, iar orice referinţă la memorie a unui proces este re-calculată şi tradusă într-o referinţă într-una din zonele care i-au fost alocate. Astfel, adresa 5 ("adresă virtuală") va indica o locaţie diferită de memorie în RAM ("adresă fizică") pentru fiecare proces.

Vom vedea un pic mai jos că sistemul de memorie virtuală permite cîteodată vizibilitate a unei zone de memorie mai multor procese, pentru uşurarea comunicării între ele.

Nucleul însuşi este protejat folosind memoria virtuală. Pentru a putea apela serviciile nucleului, el trebuie să fie cumva vizibil proceselor. Dar zona de memorie în care se află nucleul devine accesibilă numai atunci cînd procesele invocă serviciile nucleului, fiind invizibilă sau inaccesibilă în mod normal.

1.1 Structura unui apel de sistem

Să vedem cum poate un proces ordinar beneficia de serviciile nucleului, păstrînd totuşi nucleul inaccesibil. (Nucleul posedă o grămadă de structuri de date, despre toate procesele, aşa încît citirea lor ar putea reprezenta o periculoasă scurgere de informaţii. Cu atît mai mult scrierea în zona de memorie fizică în care se află nucleul trebuie să fie prohibită în mod normal).

Atîta vreme cît un proces se execută el foloseşte Unitatea Centrală într-un mod neprivilegiat (user mode). Procesul "vede" din memoria fizică numai porţiunea care i-a fost alocată de nucleu, cam ca în figura 1.

```
memorie                    memorie
virtuală                  fizică
proces curent
-------------__________    ------------
|           |          \   | memorie  |
|  proces 1 |           \  | proces k |
|           |            \_|__________|
|___________|___________   | memorie  |
|inaccesibil|           \  | fizica   |
|           |            \ | proces 1 |
-------------             \|__________|
                           |          |
                           |__________|
                           |          |
                           |  nucleu  |
                           ------------
```

Figura 1: Traducerea adreselor în modul utilizator.

Să presupunem că procesul vrea să cheme un apel de sistem (`write()`, ca să fim concreţi). (Un scenariu asemănător este valabil pentru cazul survenirii unei întreruperi sau executării unei operaţii ilegale.) Pentru acest scop procesul cheamă o funcţie de bibliotecă oferită de fabricanţii sistemului, care *împachetează* argumentele în nişte regiştri, iar într-un registru convenit (de pildă AX) codul apelului de sistem (`write()` să zicem că are codul 3), după care execută o instrucţiune specială a microprocesorului.

Această instrucţiune are un efect dramatic: cauzează trecerea procesorului în mod privilegiat (kernel-mode), după care sare la o rutină specială. Această rutină în primul rînd transformă modul în care se face translatarea adreselor, "aducînd" nucleul în spaţiul de adrese al procesului curent. (Această "aducere" se poate face automat prin faptul că zonele de memorie ale nucleului pot fi accesibile numai în modul privilegiat; depinde de caracteristicile unităţii de management a memoriei şi procesorului prin ce detalii anume se obţine vizibilitatea.) Cert este că subit imaginea arată ca în figura 2.

```
memorie                      memorie
virtuală                    fizică
proces curent
________________________    ____________
|           |          \    | memorie  |
|  proces 1 |           \   | proces k |
|           |            \__|__________|
|___________|____________   | memorie  |
|  nucleu   |   \        \  | fizica   |
|___________|    \        \ | proces 1 |
             \    \        \_|__________|
              \    \        |          |
               \    \_______|__________|
                \           |  nucleu  |
                 \__________|__________|
```

Figura 2: Traducerea adreselor în modul nucleu.

Deodată toate structurile de date şi codul nucleului au devenit vizibile. Apoi rutina specială (care tocmai se execută) se uită în regiştri conveniţi pentru a depista apelul făcut (în exemplul nostru găseşte în AX un 3). Apoi în funcţie de acesta cheamă una sau alta din procedurile de tratare din codul nucleului (de-multiplexează apelul, de obicei folosind o tabelă care pentru fiecare apel conţine o adresă în nucleu: `call syscall[AX]`).

Mai departe, procedura de tratare a apelului de sistem, care este specifică pentru apelul nostru (`write`) caută în locurile convenite argumentele (de obicei tot în regiştri), verifică validitatea lor şi începe executarea apelului.

Un apel de sistem de genul lui `write()` roagă nucleul să transfere date spre un periferic. În cazul lui `write` datele sunt indicate prin adresa virtuală a unui buffer şi mărimea lui: `write(periferic, buffer, marime)`. În mod normal nucleul trebuie să *copieze* conţinutul întregului buffer în interiorul nucleului pentru prelucrare. De ce? Pentru că acest proces va fi suspendat acum, în aşteptarea terminării executării apelului de sistem (în general interacţiunea cu perifericele este foarte lentă şi cauzează suspendarea proceselor). Ori dacă acest proces este suspendat, un altul va fi pornit. Dar acest lucru va schimba modul în care este translatat spaţiul de adrese virtuale, deci adresa virtuală a buffer-ului indicat nu va mai avea aceeaşi semnificaţie pentru nucleu!

Făcînd tot felul de trucuri uneori nucleul reuşeşte să evite copierea datelor[3]. Pentru anumite

[3]De pildă nucleul poate reţine adresa *fizică* a buffer-ului, şi poate marca în tabelele interne acea zonă ca fiind

operaţii însă copierea datelor în interiorul nucleului nu poate fi evitată: de exemplu cînd datele trebuie să plece în reţea ele trebuie împachetate şi sparte în bucăţi mai mici, sau atunci cînd merg spre disc trebuie re-aliniate şi mutate în cache. De asemenea, cînd datele se duc spre un alt proces (de pildă printr-un `pipe` în Unix) ele trebuie din nou copiate în interiorul nucleului, pentru a lăsa procesul care face `write` să continue să folosească buffer-ul fără a modifica datele deja trimise (după scrierea într-o "ţeavă" (pipe) procesul care face scrierea de obicei îşi continuă execuţia, dar datele sunt păstrate pînă cînd un proces de la celălalt capăt al "ţevii" le citeşte. Păstrarea se face în nucleu).

1.2 Comutarea proceselor

Să ne uităm acum şi la operaţiile care însoţesc comutarea execuţiei de la un proces la altul, pentru că acest cost este foarte important în micro-nuclee.

Comutarea proceselor implică salvarea stării procesului curent şi încărcarea stării procesului care urmează pentru execuţie. Pentru că cea mai mare parte din stare este conţinută în tabele aflate în memorie, schimbarea se poate face relativ simplu încărcînd valoarea unui pointer spre noua căsuţă din tabelă care se va folosi (pentru a comuta de la procesul 3 la procesul 5 nucleul va pune în pointerul spre căsuţa din tabel cu datele procesului curent valoarea 5 în locul lui 3).

În general însă trebuie luate în calcul mai multe operaţii. Anume trebuie făcute următoarele operaţiuni, nici una foarte complicată:

Faza 1: salvarea stării:

1. Salvarea regiştrilor curenţi;

2. Salvarea stării coprocesorului matematic;

3. Salvarea regiştrilor de depanare, dacă procesul curent era depanat;

4. Salvarea regiştrilor şi stării unităţii de management a memoriei; salvarea tabelei de translatare a adreselor a procesului curent (tabela de translatare indică modul în care se interpretează adresele virtuale pentru procesul curent);

5. Modificarea contoarelor şi ceasurilor de execuţie pentru a reflecta timpul consumat de procesul care se opreşte;

Faza 2: comutarea:

6. Rularea algoritmului de planificare (scheduling), care parcurge cozile de procese gata de execuţie în ordinea priorităţilor, alegînd pe cel mai urgent;

7. Golirea cache-urilor de translatare a adreselor — Translation Lookaside Buffer[4] TLB este un cache care reţine felul în care se traduc adresele virtuale cele mai des folosite pentru procesul curent; din moment ce semnificaţia adresei virtuale 5 pentru noul proces va fi alta, vechea ei asociere trebuie ştearsă şi din TLB.

Faza 3: încărcarea noului proces:

8. Încărcarea tuturor regiştrilor salvaţi (de paşii 1–3), cu valorile lor pentru noul proces;

"tabu" (pentru a nu fi modificată de algoritmii de paginare, care refolosesc memoria fizică) pînă conţinutul ei a fost prelucrat.

[4] vedeţi şi articolul meu despre cache-uri din PC Report din martie 1997 pentru o discuţie a TLB.

9. Încărcarea tabelei de translatare a adreselor a noului proces şi a regiştrilor unităţii de management a memoriei. În acest fel noul spaţiu de adrese virtuale devine vizibil şi cel vechi invizibil;

10. Pentru că în timp ce noul proces "dormea" s-au putut întîmpla evenimente interesante pentru el (de exemplu, în Unix i-a fost trimis un semnal), acum este momentul de a lua acţiuni speciale (în cazul semnalelor Unix se construiesc cadre pe stivă pentru procedurile de tratare a semnalelor, sau procesul este omorît);

11. Schimbarea pointerilor spre a puncta spre noul proces.

După cum vedeţi sunt totuşi o sumedenie de operaţii de făcut. Nuclee foarte sofisticate pot avea operaţiile de comutare a proceselor chiar mai complicate decît cele descrise aici.

Să observăm că în comutarea proceselor mai există cel puţin un cost ascuns, implicat de operaţia de schimbarea *localităţii* de adresare: pentru că începem rularea unui nou proces, care va folosi un spaţiu de adrese complet diferit, cache-ul microprocesorului va genera foarte multe rateuri pentru început, fiind încărcat cu date din spaţiul vechiului proces. De asemenea, TLB a fost golit (în pasul 7 mai sus), deci pentru a-l umple din nou cu traducerea adreselor în noul proces, va trebui să fie consultată tabela de traducere a adreselor pentru noul proces, operaţie costisitoare, deoarece implică accese suplimentare la memorie.

Un alt posibil cost va fi plătit pînă noul proces îşi aduce de pe memoria secundară (disc) paginile de memorie din setul de lucru (working set); datorită faptului că paginile de memorie îndelung ne-folosite sunt de obicei scoase afară pe disc, s-ar putea ca procesul care tocmai porneşte să trebuiască să şi le ia de acolo. Aducerea unei pagini este o operaţie extrem de costisitoare, care implică pe lîngă accesul la disc şi oprirea procesului care cere pagina pînă la venirea acesteia, ceea ce înseamnă încă o comutare de procese!

2 Plasarea serviciilor

Există în mod logic trei locuri unde poate fi implementat un serviciu:

1. În spaţiul procesului care îl foloseşte, ca o bibliotecă de funcţii;

2. În interiorul nucleului, accesat printr-un apel de sistem;

3. În gestiunea unui proces separat, numit "server".

(Un al patrulea loc, mai puţin uzual, va fi de asemenea discutat.)

Figura 3 arată cele trei variante. Le vom analiza pe fiecare pe scurt. Pentru un serviciu dat, foarte adesea proiectantul sistemului are la dispoziţie toate cele 3 posibilităţii.

Modul dominant în care sunt plasate serviciile (adică locul majorităţii serviciilor) dă şi clasificarea unui sistem în taxonomia sistemelor de operare. Practic fiecare sistem de operare va avea servicii în toate cele trei părţi, astfel încît diferenţa este mai curînd una de grad decît de natură. Astfel, sistemele care aleg varianta 2 pentru majoritatea serviciilor se numesc monolitice, pentru că tind să aibă un nucleu foarte mare, cu o grămadă de cod.

Sistemele care optează pentru varianta 3 se numesc prin contrast "micro-nuclee", pentru că nucleul avînd puţine servicii devine foarte mic.

În fine, sisteme în care majoritatea serviciilor sunt plasate în spaţiul proceselor însele, în funcţii de bibliotecă, sunt relativ puţin răspîndite. Vom vedea însă nişte candidaţi puţin mai jos.

```
------------      ------------      -----------------------
|  proces  |      |          |      |  proces  |  proces  |
|          |      |          |      | (client) | (server) |
|          |      |          |      |          |          |
|          |      |  proces  |      |          |          |
----|-------      |          |      |          |          |
|          |      |          |      |          |          |
|biblioteca|      |          |      |          |          |
============      ====|=======      =====|===========|=====
|          |      |          |      |     \__________/    |
| nucleu   |      | nucleu   |      | nucleu              |
------------      ------------      -----------------------
 varianta 1        varianta 2             varianta 3
```

Figura 3: Plasamentul serviciilor.

2.1 Biblioteci

Cu servicii plasate în biblioteci este obişnuit orice programator care a folosit un limbaj de genul C sau Pascal. O bibliotecă este o colecţie de funcţii gata scrise, la care programele utilizatorilor se pot "lega", şi pe care le pot folosi. Legarea (linking) la funcţiile din biblioteci se poate face fie atunci cînd programul este creat (la sfîrşitul compilării), şi atunci se numeşte "legare statică" (static linking), fie abia după ce programul a fost pornit în execuţie, fiind atunci numită legare dinamică (dynamic linking). Cert este că se face doar odată, aşa încît costul legării se "amortizează" cînd funcţiile din bibliotecă sunt folosite intens.

"Costul" unui astfel de serviciu este extrem de scăzut; cel mai scăzut posibil probabil, pentru că implementarea unui apel de funcţie în termenii microprocesorului este foarte ieftină. Trebuie însă să observăm că natura codului din bibliotecile partajate care se încarcă dinamic[5] îl face cîteodată mai ineficient decît codul obişnuit, cu factori cuprinşi între 1% şi 30%.

Să vedem nişte exemple concrete faimoase de servicii plasate în biblioteci.

2.1.1 Cazul MS-DOS

Într-un anumit sens sistemul de operare MS-DOS este o mare bibliotecă de funcţii pe care procesele le pot chema; este adevărat că chemarea funcţiilor se face nu plasînd pe stivă argumente, ci în regiştri, şi apelînd apoi o "întrerupere software" (de exemplu, dacă îmi aduc aminte bine, toate funcţiile pentru operaţiuni grafice sunt chemate punînd în registrul AX codul funcţiei şi executînd INT 10h). Această întrerupere software este un rudimentar apel de sistem, care de fapt este un apel indirect de funcţie din "nucleu".

Datorită faptului că MS-DOS nu oferă memorie virtuală, apelul de sistem este mult mai simplu decît cel descris mai sus în prima secţiune a articolului, şi este practic la fel de eficient ca un apel de procedură.

Viteza MS-DOS este explicaţia popularităţii sale enorme, pe care s-a clădit averea Microsoft şi a fabricanţilor de jocuri. Din păcate (sau din fericire), viteza de execuţie a apelurilor de sistem nu este singurul criteriu de merit; faptul că nu oferă multitasking (mai multe procese simultan, ceea ce implică şi memorie virtuală pentru izolarea lor) şi că lucrul cu perifericele este foarte ineficient, au dus la moartea acestui sistem de operare.

[5] Este vorba de faptul că acest cod este "independent de poziţie" (*position independent code*; PIC).

2.1.2 Cazul C

Limbajele de programare de nivel înalt posedă adesea funcţii de bibliotecă pentru a fi independente de arhitectura calculatorului. În Pascal astfel de funcţii sunt `write` şi `new`. O să ne aruncăm privirea asupra unei instanţe în limbajul C.

Limbajul ANSI C pune la dispoziţia utilizatorilor o serie de funcţii de bibliotecă pentru manipularea de "stream"-uri (nu îmi vine în minte nici o traducere rezonabilă). Un "stream" este un cîrnat de octeţi; tipul stream este în C notat cu `FILE`. Operaţiile pe stream-uri au numele prefixate cu litera `f`: `fopen()`, `fclose()`, `fprintf()`, `fscanf()`, `fflush()`, `fputc()`, `fgetc()`, `fseek()`, `fputs()`, etc. C transformă toate perifericele în stream-uri; astfel se pot folosi aceleaşi funcţii pentru terminale, fişiere, şi alte minuni, depinzînd de sistemul de operare.

După ce un stream este creat (cu `fopen()`) sau moştenit, (ca `stdio`), pot fi trimise date spre el cu `fputc()`, `fputs()` sau `fprintf()`. `fputc(caracter, stream)` trimite un caracter spre stream-ul indicat, oricărui periferic i-ar fi acesta asociat. "Trimiterea" se realizează de obicei prin invocarea unui apel de sistem pentru transmis date; la sistemul de operare Unix folosind apelul `write()`.

`puts(sir, stream)` trimite un şir de mai multe caractere. Funcţia `fprintf()`, (a cărei binecunoscută variantă `printf()` este o abreviere) face două operaţii: *formatează* şi transmite datele (de exemplu `fprintf(stdio, "%d", x);` transformă întîi valoarea lui `x` într-un şir de caractere, după care trimite acest şir spre stream).

Pe lîngă acest gen de servicii de conversie, biblioteca de operaţii cu stream-uri mai face un serviciu de "buffer"-ing: strînge caracterele trimise spre stream-uri laolaltă şi le trimite în grămezi. De ce? Am văzut că un apel de sistem este o operaţie relativ costisitoare. Din cauza asta, în loc să invoce nucleul pentru fiecare caracter, mai multe caractere sunt strînse laolaltă şi pasate cu un singur apel de sistem. Asta implică imediat o creştere de eficienţă.

Ca să verific această aserţiune am compilat următoarele două programe:

```
/* streams */
int main(void)
{
  unsigned long i;

  for (i=0l; i < 1000000; i++)
    putchar('0');
  return 0;
}
```

```
/* system call */
int main(void)
{
  unsigned long i;

  for (i=0l; i < 1000000; i++)
    write(1, "0", 1);
  return 0;
}
```

Varianta din stînga scrie de un milion de ori caracterul 0 folosind stream-uri. Stream-ul strînge (în implementarea de bibliotecă pe care o am eu) cîte 1024 de caractere pe care le trimite[6]; asta înseamnă că face 1 000 000/1024 = 977 de apeluri de sistem `write()`.

Varianta din dreapta pur şi simplu scrie un milion de caractere făcînd un milion de apeluri de sistem, unul pentru fiecare caracter. Iată timpii cronometraţi[7] pe o maşină Linux 486/33Mhz:

streams	apel de sistem	raport
4.66 sec	87.08 sec [sic!]	1:18

Pot, fireşte, estima aproximativ durata unui apel de sistem. Astfel am durata a 1 000 000 − 977 apeluri de sistem `write()` spre un periferic nul, de aproximativ 82.5 secunde (scăzînd dispare

[6]Am verificat acest lucru folosind comanda `strace`.

[7]Timpii au fost cronometraţi cu comanda `time program >/dev/null`, cu ieşirea trimisă la perifericul null, care face operaţiile instantaneu, deci nu intervine în măsurătoare.

diferenţa dintre timpii de legare). Asta înseamnă un timp de 82 microsecunde pentru un apel de sistem `write`. Prin contrast, apelul unei funcţii durează sub 1 microsecundă pe aceeaşi maşină, dar este mai greu de estimat precis.

2.1.3 Cazul NT

Sistemul Windows NT de la Microsoft este aparent un sistem micro-nucleu, dar vom mai avea un cuvînt de spus asupra acestei categorisiri. Oricum, proiectanţii lui Windows NT au sesizat şi ei importanţa plasării serviciilor frecvente în biblioteci legate direct la codul proceselor, şi au încercat din răsputeri să folosească această tehnică pentru eficienţă. Iată o ilustraţie:

La NT 3.5 gestiunea ecranului era făcută de un proces separat, un "server"; la NT 4.0 gestiunea ecranului a fost mutată în interiorul nucleului. Important este acum pentru noi că există o entitate exterioară proceselor utilizatorilor care gestionează în întregime ecranul. Cînd ai de desenat un punct, o linie, un caracter sau un "bitmap", trebuie să discuţi cu această entitate. Transmiterea unui mesaj spre un alt proces (la 3.5) sau un apel de sistem (la 4.0) costă, iar cînd vrei să trasezi zeci de mii de elemente (aplicaţiile Windows sunt în mod normal risipitoare în grafică) costul se înmulţeşte cu acest factor. Din cauza asta, proiectanţii lui NT au aplicat tehnica mutării serviciilor grafice în bibliotecile utilizatorului.

Pe lîngă faptul că au aplicat tehnica "buffer"-ului, de a strînge cît mai multe operaţii la un loc înainte de a le trimite serverului de ecran, au ajuns la adevărate "exagerări": de pildă culoarea curentă este menţinută atît de serverul ecranului cît şi de bibliotecă; atunci cînd utilizatorul cheamă o funcţie pentru a afla valoarea culorii, biblioteca răspunde direct, fără a interoga serverul, economisind o comunicaţie.

Atunci cînd utilizatorul schimbă culoarea curentă, biblioteca nu transmite modificarea spre server decît în momentul în care utilizatorul face şi o desenare; în acest fel se mai economiseşte un mesaj, fără ca vreun efect vizual să arate schimbarea.

2.1.4 Cazul exokernel

PC Report din Septembrie 1996 a găzduit un articol despre un nou prototip de arhitectură a sistemului de operare, "exokernel" -ul. Exokernel-ul este o idee împinsă la extrem, încarnată într-un prototip de sistem de operare numit ExOS, dezvoltat la MIT. Ideea este de a elimina aproape complet nucleul (mai mult chiar decît în cazul sistemelor micro-nucleu), mutînd absolut toate funcţiile acestuia, (mai puţin o oarecare funcţie de arbitrare a accesului la resurse de nivel foarte jos, cum ar fi pagini de memorie) în biblioteci gigantice legate de procesele utilizatorilor. În acest caz toate operaţiile se pot executa la întreaga viteză a microprocesorului, evitînd costisitoarele apeluri de sistem. (De aici şi numele: kernel (nucleu) "exterior": la purtător.)

Ideea este foarte tentantă, sî anumite aplicaţii dezvoltate special pentru exokernel au într-adevăr viteze uluitoare. Anumite probleme rămîn însă foarte greu de rezolvat în contextul exokernel-ului. Una dintre ele, extrem de spinoasă, este tratată sumar în secţiunea consacrată semanticii operaţiilor, aflată puţin mai jos.

2.2 Nucleu

Al doilea loc unde poate fi plasat un serviciu este în nucleu. Sistemele de operare monolitice pun în nucleu mai toate serviciile care au o utilizare frecventă.

Sisteme monolitice tipice sunt (şi veţi recunoaşte toate sistemele dominante pe piaţă): Windows 3.1, Windows 95, Unix, VMS, JavaOS[8]. Windows NT este considerat în continuare un sistem micro-nucleu, deşi conţine în nucleu servicii care la Unix (un monolit tipic) sunt înafara nucleului, cum

[8]Informaţiile mele în legătură cu JavaOS nu sunt foarte ample, dar cred că poate fi categorisit ca monolitic.

ar fi sistemul de ferestre sau serverul de fişiere. După cum vedeţi, graniţele sunt difuze între categorii...

Sistemele monolitice sunt comercial cele mai răspîndite.

Pentru ilustraţie, să vedem care sunt categoriile de servicii oferite de un sistem Unix tipic:

* Operaţii cu procese (creare, distrugere, etc.);
* Depanarea şi măsurarea (profiling) proceselor;
* Planificarea şi execuţia proceselor;
* Accounting şi tarifare după consumul resurselor;
- Operaţii cu fişiere;
* Comunicaţie inter-proces: ţevi (pipes), semnale, memorie partajată, semafoare, mesaje;
- Protocoale de comunicaţie în reţea (TCP/IP);
* Gestiunea memoriei virtuale;
* Alocarea şi eliberarea memoriei;
- Timere şi alarme;
* Mecanisme de protecţie şi securitate;
- Legarea dinamică;
- Managementul perifericelor.

Am marcat cu asterisc serviciile care în orice implementare a unui sistem de operare, fie ea monolit sau micro-nucleu, trebuie să fie oferite de nucleu. (Cum se descurcă exokernel-ul fără ele, mie personal nu îmi este foarte clar.)

Sistemele monolitice sunt destul de greu de scris şi relativ inflexibile: o schimbare a serviciilor oferite se poate face în mod tradiţional doar oprind sistemul şi recompilînd o imagine a nucleului[9]. Există însă o cantitate considerabilă de experienţă în folosirea şi manipularea acestor sisteme.

Cea mai dezirabilă trăsătură a acestor sisteme (comparate cu bibliotecile) este separaţia netă între spaţiul de adrese al nucleului şi cel al proceselor utilizator. Aceasta permite nucleului să aibă un control foarte strîns[10] asupra operaţiilor care pot fi efectuate de procese, şi îi permite să forţeze cu uşurinţă respectarea politicilor de folosire a resurselor.

2.3 Server(e)

În fine, putem lua o resursă partajată şi o putem depune în braţele unui proces, care să aibă grijă de ea; procesul care ne “serveşte” cu această resursă se va numi “server”. Nucleul trebuie să pună la dispoziţia proceselor o metodă eficace prin care să comunice între ele; de îndată ce au această metodă la dispoziţie, procesele care au nevoie de resursa deţinută de server devin “clienţii” lui, trimiţîndu-i un mesaj cu cererea lor. Serverul le răspunde clienţilor cu datele cerute.

[9] Sistemele moderne Unix pot încărca dinamic unele porţiuni de nucleu.

[10] Cel puţin teoretic; faptul că se raportează mereu noi bug-uri în securitatea sistemelor Unix nu zguduie de loc încrederea adepţilor în această teză.

Marele avantaj al acestei formule este că teoretic se poate aplica şi în cazul în care clientul şi serverul sunt pe maşini diferite şi comunică printr-o reţea. Într-adevăr, majoritatea covîrşitoare a aplicaţiilor din reţea folosesc această arhitectură.

De aici apar însă şi problemele, după cum vom vedea într-o secţiune ulterioară consacrată în mod special sistemelor micro-nucleu, care tind să exploateze tehnologia client-server.

Pentru a ilustra diferenţa de performanţă, pe acelaşi calculator pe care am făcut măsurătorile anterioare, un nucleu experimental extrem de simplu (PicOS — implementat de autor), fără memorie virtuală, cu procese integral rezidente în RAM, permite schimbarea a circa 5000 de mesaje/sec între două procese pe aceeaşi maşină. Asta înseamnă deja 200 de microsecunde pentru un mesaj, adică 400 pentru un apel complet cerere/răspuns. Comparaţi cu performanţa unui apel de sistem şi cu a unui apel de procedură.

Vom petrece restul rîndurilor din această secţiune chiorîndu-ne la cîteva exemple reale de folosire a serverelor pentru gestiunea resurselor.

2.3.1 Servere în Unix

În Unix cele mai faimoase servere care rulează ca procese sunt:

- pentru accesul fişierelor la distanţă (NFS: Network File System), care în mod normal rulează ca o combinaţie proces/nucleu;
- "super-serverul" inetd, care porneşte la cerere alte cîteva sute de servere; lista acestora se poate găsi într-un fişier `/etc/services` pe o maşină Unix; printre ele: servere pentru acces la distanţă (telnet, rlogin, rsh, rexec, ssh, kerberos), poştă electronică (smtp, uucp), transfer de fişiere (ftp), etc., etc.;
- serverul de www (world wide web);
- serverul de nume (named);
- serverul de raportare a erorilor (syslogd, klogd);
- serverul de ferestre (X).

2.3.2 Windows NT

Windows NT are o arhitectură în care serverele ocupă un loc central; probabil din această cauză este asimilat cu un micro-nucleu, deşi cele două noţiuni sunt oarecum independente, după cum se poate vedea din exemplele citate. Ideea de bază a lui NT a fost "furată" de arhitectul lui şef, David Cutler (la rîndul lui "furat" de la Digital de către Microsoft) de la sistemul de operare Mach, un prototip de cercetare micro-nucleu construit la Universitatea Carnegie Mellon şi acum dezvoltat la Universitatea din Utah.

Ideea, extrem de elegantă, este de a construi un mediu foarte bogat în proprietăţi pe care să se poată apoi construi servere independente care să emuleze sisteme de operare distincte. Astfel, pe o maşină NT pot rula simultan aplicaţii Windows 95, MS-DOS, OS/2 şi Unix, fiecare avînd impresia că foloseşte sistemul de operare respectiv, dar discutînd de fapt cu un server, ca în figura 4.

În mod normal un server lucrează astfel: un thread[11] aşteaptă blocat să vină cereri de operaţii de la clienţi. Cînd vine o cerere, acest thread crează un nou thread, căruia îi pasează mesajul

[11] Dacă noţiunea nu vă este familiară, puteţi asimila un thread cu un proces; o descriere amplă a conceptului şi o implementare a unui pachet de thread-uri am publicat în PC Report din ianuarie 1997. De altfel în Unix-ul tradiţional, ne-existînd thread-uri, se folosesc chiar procese pentru servere.

```
          | server  | proces   |  server   | server  | proces |
          | Windows | Windows  | securitate| Unix    | Unix   |
          |         |          |           |         |        |
          |         |          |           |         |        |
          ===|=|==========|===================|=|========|====
          |   \ \_________/    |grafica|____/   \_______/   |
Nucleu    |    \_________________| Win95 |                  |
 (4.0)    |                    |_______|                    |
          |                                                 |
```

Figura 4: Arhitectura lui Windows NT.

cu operaţia cerută, după care se duce din nou la culcare, aşteptînd noi mesaje. Thread-ul fiu decodifică mesajul, execută operaţia, răspunde şi moare.

În acest fel server-ul poate executa simultan mai multe cereri. Pe de altă parte, dacă nu există cereri, server-ul nu consumă aproape nici un fel de resurse, pentru că are un singur thread, care doarme.

2.4 Resurse fără servere (serverless); Memoria distribuită partajată

Să notăm în treacăt că toate cele trei soluţii citate dau fiecare resursă pe seama cuiva: o bibliotecă, un nucleu, un proces. Există şi o soluţie "democratică", în care nimeni nu posedă un obiect; acest stil de proiectare se numeşte "fără servere" (serverless). Din păcate algoritmii folosiţi pentru acest fel de probleme sunt în general puţin robuşti şi extrem de complicaţi, şi trădează adesea tocmai cauza pentru care erau creaţi: evitarea unei "gîtuituri" (bottleneck) în accesul la resursă, reprezentată de serverul care o gestionează.

Să notăm totuşi o aplicaţie a acestui gen de algoritmi în sistemele de calcul paralele şi distribuite, mai ales a celor care implementează ceea ce se numeşte *memorie distribuită partajată (Distributed Shared Memory, DSM)*. Ideea centrală este de a avea pentru toate calculatoarele dintr-o reţea un singur spaţiu de adrese uriaş, în care toate scriu şi citesc, dar care nu este memorat fizic în vreun loc fixat, ci ale cărui "locaţii" se "plimbă" după necesităţi între maşinile care le folosesc.

Există şi sisteme de fişiere implementate după această schemă, dar progresele comerciale sunt (încă) slăbuţe. Nici noi nu o să le consacrăm deci prea mare importanţă în acest articol, care iar a început să ia proporţii mai mari decît cele anticipate iniţial de autor.

2.5 Semantica operaţiilor

"Unde să plasăm un serviciu, în care din cele trei posturi?" Aparent răspunsul la această întrebare depinde doar de consideraţii de eficienţă şi estetică, precum şi de extravaganţa design-erului. Dar lucrurile nu stau chiar aşa!

Vom vedea că anumite însuşiri ale unui serviciu depind esenţial de plasarea sa, şi că fiecare din cele trei scheme are proprietăţi speciale, care nu pot fi simulate nicicum în întregime de celelalte. (Vom ignora asemenea consideraţii elementare cum ar fi că nu putem plasa servicii de creare a proceselor într-o bibliotecă, pentru că atunci cine crează procesul în care se află biblioteca?)

Lista de diferenţe care urmează nu este în nici un caz exhaustivă, ci doar vrea să ilustreze prin exemple problema.

2.5.1 Diferenţe semantice bibliotecă–nucleu

Deosebirea dintre bibliotecă şi nucleu este cu siguranţă familiară oricărui programator în C care frustrat a încercat să-şi depaneze programele punînd `printf()`-uri pe ici-colo, dar care nu dădeau nici un efect!

Explicaţia este simplă: `printf()` scrie, după cum am văzut, într-un buffer, care este golit numai în anumite circumstanţe. Dacă o eroare survine înainte ca apelul de sistem `write()` să fie executat şi procesul moare, conţinutul din buffer este definitiv pierdut! (Soluţia este, fireşte, să forţăm golirea buffer-ului folosind funcţia `fflush()`.)

Aşa ceva nu se va întîmpla dacă folosim direct `write()`, pentru că odată apelul de sistem executat, datele au fost copiate de nucleu şi vor ajunge pînă la urmă la perifericul-destinaţie.

Diferenţa esenţială între cele două cazuri este de durată de viaţă a informaţiei: dacă informaţia este într-o bibliotecă, locală unui proces, atunci ea nu poate supravieţui morţii procesului[12].

Nucleul în schimb supravieţuieşte tuturor proceselor (teoretic), deci poate menţine în siguranţă informaţiile globale pentru întregul sistem.

Un alt exemplu faimos este implementarea protocoalelor de reţea în biblioteci, care pune infernale dificultăţi (de altfel acesta este unul dintre motivele atacurilor la exokernel, care, vă amintiţi, este o bibliotecă mare): de exemplu standardul TCP/IP impune ca după închiderea unei conexiuni de reţea, unul din capete să reţină identificatorul conexiunii pentru o vreme nefolosit ("30 de secunde" scrie la carte), în aşa fel încît să nu fie însuşit de o altă conexiune (în acel caz, pachete întîrziate ale conexiunii precedente care mai rătăcesc pe reţea ar putea fi incorect primite pe conexiunea nouă). Puteţi vedea în Unix lista tuturor conexiunilor cu comanda `netstat`. Cele care sunt în starea `TIME_WAIT` sunt conexiuni de acest gen: terminate, dar memorate. Pe un server de web trebuie să fie o mulţime de astfel de conexiuni la un moment dat[13].

Ei bine, pe un proces care are protocoalele de comunicaţie implementate în biblioteci îl vor trece toate sudorile să menţină identificatorul conexiunii după moartea procesului.

2.5.2 Diferenţe semantice nucleu–server

O deosebire de acelaşi gen de semantică (semnificaţie) de acelaşi gen a serviciilor subzistă între serviciile oferite de nucleu şi cele oferite de servere aflate la distanţă: în mod normal pe o maşină ori merge nucleul şi procesele, ori, dacă nucleul nu merge, nu merge nimic. Cu alte cuvinte, dacă procesele pot cere servicii de la nucleu sunt sigure că gestionarul lor este sănătos. Acest lucru nu mai este adevărat în cazul proceselor care cer servicii de la distanţă: cînd scuipi nişte informaţie în reţea şi nu primeşti răspuns este greu de zis dacă informaţia a ajuns şi răspunsul nu, sau informaţia s-a pierdut, sau a ajuns şi serverul a murit înainte sau după ce a primit-o. De aici şi complexitatea enormă a protocoalelor de reţea şi a algoritmilor distribuiţi.

O altă diferenţa greu de mascat între soluţia unei probleme cu nucleu şi soluţia cu servere este în încredere. Un nucleu ştie că toate procesele care rulează pe maşina lui au fost create de el şi sunt "legitime". Pe de altă parte cînd un server primeşte o cerere de la distanţă (sau chiar pe aceeaşi maşină), el nu are la dispoziţie mijloacele nucleului de verificare. Un mecanism complet diferit trebuie inventat pentru a asigura serverul de identitatea clienţilor, un lucru nenecesar pentru un serviciu plasat în nucleu.

[12] Fireşte, o bibliotecă partajată poate servi de repozitoriu de informaţie

[13] Dacă nu aveţi la îndemînă un server de web, creaţi o conexiune cu `telnet localhost`, vedeţi ambele capete cu `netstat`, închideţi conexiunea cu `exit` şi apoi vedeţi informaţia rămasă din nou cu `netstat`.

3 Promisiunile micro-nucleelor

Iată care sunt unele din avantajele (reale sau fictive) ale arhitecturii micro-nucleu; unele din aceste merite sunt atributabile arhitecturii client-server, altele arhitecturii de micro-nucleu, dar am văzut că a doua o implică pe prima.

Modularitate Într-un nucleu monolitic diferitele părţi comunică foarte adesea folosind variabile globale, ceea ce ridică probleme dificile privitoare la corectitudinea codului. Mai ales pe multiprocesoare, unde mai multe programe pot acţiona simultan asupra aceleiaşi structuri de date, se pot ivi tot felul de comportamente ciudate. Prin contrast, în soluţia cu servere, un server gestionează o cantitate redusă de resurse, iar interacţiunea cu alte programe se face prin interfeţe foarte bine precizate (mesaje). E clar, micro-nucleele încurajează modularitatea.

Scalabilitate Vrei un serviciu mai puternic: mai adaugi nişte servere sau nişte clienţi, înlocuieşti serverele cu altele mai performante. O maşină este prea încărcată: muţi din servere pe alta. Toate acestea sunt aspecte ale creşterii incrementale a unui sistem, sau creştere în raport cu resursele şi necesităţile disponibile.

Distribuire facilă Dacă primitiva ta de bază este `trimite_mesaj()`, atunci eşti încurajat să dezvolţi soluţii pentru programe în care nu contează unde se află server-ul. Mediile de calcul distribuit (DCE: Distributed Computing Environment al lui OSF: Open Software Foundation) sunt un exemplu de standard de scriere a unei aplicaţii independent de numărul de calculatoare pe care se execută.

Adaptabilitate (customizability) Nucleul nu poate fi schimbat decît în mică măsură, cu mare grijă, şi arar fără a opri sistemul. Pe de altă parte, serverele sunt simple procese, care pot fi create şi omorîte dinamic, fără a avea nevoie de privilegii administrative extraordinare. Cu alte cuvinte fiecare utilizator al unui micro-nucleu ar putea să-şi construiască mediul care îi convine; cine nu foloseşte fişiere nu porneşte nici un server de fişiere, şi are mai multe resurse pentru alte ocupaţii.

Dimensiune redusă Vîrful de lance al creatorilor de micro-nuclee este: plăteşti numai pentru ce foloseşti. Nu ai nevoie de ceva: nu primeşti. Un nucleu monolit conţine toate serviciile, fie că le vrei, fie că nu.

Comunicaţie puternică inter-proces Aceasta este o condiţie necesară pentru viabilitatea unui micro-nucleu. Comunicaţia trebuie să fie eficientă, pentru că fiecare serviciu implică cel puţin un schimb de două mesaje (cerere/răspuns), şi flexibilă pentru a permite transmiterea unei variate game de mesaje (de la un număr, la un fişier de megaocteţi cu imagini).

3.1 RPC

Orice discuţie serioasă despre arhitectura client-server trebuie să atingă măcar în trecere subiectul apelului procedurilor la distanţă (remote procedure calls). Subiectul este fascinant şi merită o tratare mult mai amplă.

Observaţi că în cazul folosirii serverelor nu numai că am mutat un serviciu într-un proces separat, dar am schimbat şi *natura* modului în care serviciul este invocat: înainte era printr-un apel de funcţie (sau de sistem), dar acum este prin *trimiterea* unui mesaj. E aceeaşi diferenţă de perspectivă ca între o procedură şi un fişier; pentru un programator paradigma mesajului este mai incomodă.

Din cauza asta a fost inventată împachetarea mesajelor în proceduri. Practic programatorul cheamă o procedură (dintr-o bibliotecă), care procedură construieşte mesajul, îl trimite, aşteaptă răspunsul şi se întoarce în mod uzual. În acest fel programatorul operează din nou cu conceptul familiar de procedură.

O astfel de procedură chemată la distanţă se numeşte în engleză "remote procedure call", prescurtat RPC. Un pachet RPC face multe lucruri: dintr-o specificare de nivel înalt a procedurii oferite de server[14] el construieşte automat funcţiile pentru client şi server. Funcţiile acestea, care manipulează mesajul se numesc în engleză *stub*. Misiunea lor este de a împacheta argumentele procedurii într-un mesaj (operaţie numită *marshalling*), de a trimite mesajul şi a despacheta valorile la recepţia unui mesaj. Funcţionarea este prezentată schematic în figura 5. Procedurile stub sunt generate de compilatoare speciale din simpla descriere a interfeţei lor (tipul argumentelor şi al rezultatelor).

```
------------           ------------
|  client  |           |  server  |
|          |           |          |
| 11^  |1  |           |   _6_    |
|===|==v===|           |=5|===|7==|
|   |  2\__|____3_>>>__|__/4  |   |
|  10\_____|___________|_____/ 8  |
|          |    9 <<<  |          |
| stub     |           |     stub |
------------           ------------
```

```
 1. clientul cheama stub-ul
 2. împachetarea (marshalling)
 3. stub-ul client trimite mesaj
 4. despachetarea (demarshalling)
 5. apelul procedurii în server
 6. execuţia procedurii în server
 7. procedura serverului se întoarce
 8. stub împachetează rezultatele
 9. rezultatele transmise clientului
10. stub-ul client despachetează
11. stub-ul client se întoarce
```

Figura 5: Operaţiile într-un RPC.

O altă problemă rezolvată de un pachet RPC este de a *localiza* serverele care oferă servicii. Cînd un client porneşte el ştie doar că undeva ar trebui să se afle un server care exportă serviciile care îl interesează. Operaţia de descoperire a server-ului se numeşte tot "legare", dar în engleză foloseşte termenul *binding*, care este un sinonim parţial pentru "linking" (corespunzînd acestei faze din compilarea tradiţională).

O calitate a unui pachet RPC bun este că permite invocarea de proceduri pe maşini diferite arhitectural de cea pe care se află clientul. Pentru asta procedurile stub de împachetare trebuie să folosească un standard comun de reprezentare a datelor, astfel încît calculatoare care folosesc reprezentări diferite (ex. big/little endian) diferite să se înţeleagă totuşi între ele.

Există o mulţime de probleme cu RPC, care scad oarecum din meritele metodei; în principal o serie de diferenţe între semantica unui apel de procedură obişnuită şi al unei proceduri distante sunt aproape de nedepăşit. De exemplu cum trimiţi un pointer la distanţă şi la ce-i foloseşte server-ului, care are alt spaţiu de adrese?

4 Problema micro-nucleelor

Din cît am divagat, probabil că problema sare-n ochi: într-un micro-nucleu costul unui serviciu este prea ridicat. Asta pentru că transmiterea unui mesaj implică:

1. Apeluri de sistem pentru trimitere şi recepţie de mesaje, atît de partea clientului cît şi a serverului;

[14]Într-un limbaj de descriere a interfeţei, "interface definition language", prescurtat IDL.

2. Cel puţin o comutare de procese;

3. Pentru RPC împachetarea şi despachetarea argumentelor;

4. Copierea argumentelor din spaţiul de adrese al clientului, în nucleu, eventual prin reţea, şi apoi în spaţiul serverului;

5. Crearea unui thread în server pentru a trata cererea;

6. Dacă mesajul trece prin reţea este posibil să fie copiat de mai multe ori: de pildă din nucleu pe placa de reţea şi invers la recepţie;

7. Copierea răspunsului pe traseul invers server $\rightarrow$ client.

O altă observaţie interesantă este că în general în micro-nuclee tendinţa este de a fragmenta un serviciu oferit de un nucleu monolit în *mai multe* servicii oferite de servere diferite. De exemplu, apelul `open(fisier)` din Unix de obicei devine un apel pentru a traduce numele de fişier într-un identificator unic, executat de serverul de directoare (sau, mai rău, traducerea fiecărei părţi din “cărare” (path) independent, printr-un apel separat al unui alt server!), şi apoi “deschiderea” fişierului la serverul de fişiere propriu-zis (figura 6 ilustrează acest fapt). În acest caz, ceea ce iniţial era un singur apel de sistem poate deveni o suită de zeci de mesaje schimbate cu mai multe servere!

```
              1         __________                          Mesaje schimbate
        /---------->>--|          | server-ul          1. unde e serverul usr?
        | /--------<<--|__________| directorului /     2. la adresa xx
        | |   2
------------       3    __________
|  client  |____/--->>-|          | server-ul (xx)     3. unde e ‘bin’?
|          |--------<<-|__________| directorului /usr  4. la adresa yy
|          |____   4
|          |--\ \     5 __________
------------   \ \-->>-|          | server-ul (yy)     5. unde e ‘bash’?
        | |     \---<<-|__________| directorului /bin  6. la zz, cod ww
        | |           6
        | |             __________
        | \__7__/--->>-|          | server-ul (zz)     7. scoate ‘ww’!
        \-----------<<-|__________| de fisiere         8. uite-l! (date)
             8
```

Figura 6: Deschiderea fişierului `/usr/bin/bash` într-un sistem distribuit.

5 În loc de concluzie: arhitectura NT

Vom încheia acest articol urmărind cîteva din trucurile făcute de proiectanţii sistemului Windows NT în lupta cu microsecundele. Trebuie spus din capul locului că pentru a păstra compatibilitatea cu atîtea sisteme existente (Windows 95, OS/2, etc.) ei nu aveau de ales între prea multe variante arhitecturale, şi că soluţia cu serverele din figura 4 este cea mai flexibilă.

5.1 Local procedure call (LPC)

Un pachet RPC face o grămadă de operaţii de care nu este nevoie în cazul în care clientul şi serverul sunt pe aceeaşi maşină. De exemplu conversia datelor într-un format standard şi înapoi este curată sinucidere, din moment ce procesorul este acelaşi. Pe de altă parte o cantitate impresionantă de mesaje în NT tind să fie între programe locale; de exemplu tot ce ţine de afişare pe ecran se plimbă între unul din serverele de emulare şi serverul Windows 95, care singur are în mînă gestiunea ecranului. Din cauza asta între NT 3.54 şi NT 4.0 serverul de ecran a coborît în nucleu, cum apare şi în figura 4.

Mai apar şi alte mesaje locale, de exemplu între un program Unix şi serverul de emulare Unix.

Proiectanţii NT au optimizat în mod special acest gen de comunicare, numind-o "apel de procedură locală" (Local Procedure Call, LPC). Cînd apelul unei funcţii dintr-un server este pe aceeaşi maşină, majoritatea operaţiilor costisitoare sunt pur şi simplu evitate. Din păcate nici atîta nu este suficient pentru a atinge performanţa necesară. Iată alte două trucuri care sunt folosite cu succes în cazul NT, pentru servere locale.

5.2 Memoria partajată

Dacă se elimină operaţiile care *transformă* datele (marshalling), atunci cel mai mare cost nu este dat nici de apelul de sistem, nici de comutarea proceselor, ci de *copierea* argumentelor mari. Foarte adesea serverele de fişiere sunt implementate folosind RPC; operaţii tipice vor transfera cantităţi mari de date din/spre fişier. Copierea datelor din client în nucleu şi apoi în server pentru un `write()` (sau invers la citirea unui fişier) este o risipă majoră (overhead) fără nici un beneficiu real.

Proiectanţii lui Windows NT au folosit un mod special de transmitere a datelor între un client şi un server, care foloseşte mecanismul de memorie virtuală oferit de nucleu. Astfel, clientul alocă o zonă de memorie partajată (pagini din RAM care sunt vizibile în mai multe spaţii virtuale) şi trimite server-ului doar o descriere a zonei partajate. De pildă pentru un `write()` clientul alocă zona (printr-un apel special de sistem), scrie datele în zonă, face un LPC `write()` la server, dînd descrierea zonei partajate. Serverul poate accesa acum direct memoria comună cu clientul. Nici un octet nu a fost mutat în bufferele din nucleu şi înapoi! Singura operaţie este modificarea tabelelor de translatare a adreselor pentru a face zona vizibilă şi în server. Zona partajată este făcută vizibilă în spaţiul server-ului în timp ce server-ul execută apelul, după care devine din nou invizibilă pentru server.

Aceeaşi tehnologie, a memoriei partajate, este folosită şi în anumite implementări ale serverelor de ferestre X Windows, în care clienţii locali pot da direct zone de memorie, şi nu conţinutul lor.

5.3 Prealocarea resurselor

Amintiţi-va, din secţiunea dedicată serviciilor oferite de servere, cum funcţionează un server în NT: face "pui" (thread-uri) pentru fiecare nouă cerere, care mor după execuţia cererii. Această creare de thread-uri pentru fiecare serviciu este de asemenea o sursă importantă de ineficienţă. De aceea Windows NT mai încalcă încă odată regulile bunelor maniere şi face o optimizare specială. Proiectanţii spun că această optimizare este atît de urîtă încît nici nu este accesibilă utilizatorului obişnuit; ea este folosită numai între serverele de emulare şi modulul grafic.

Pe scurt între un client foarte activ şi un server se stabileşte o cale extrem de rapidă de comunicare la cererea clientului. Astfel server-ul alocă un thread special pentru acel client, care va executa numai cererile acestui client. Mai există o zonă partajată alocată permanent pentru uzul celor doi, şi o pereche de "evenimente" (event pair). Perechea de evenimente este de fapt un întrerupător prin care clientul şi thread-ul alocat din server îşi semnalează reciproc (mai exact îi semnalează planificatorului din nucleu) cînd au nevoie de serviciile celuilalt.

Acest mecanism simplifică multe operaţii: thread-ul nu mai este creat/distrus, tabela de pagini pentru memoria partajată nu mai este modificată (pentru că zona partajată va fi folosită pentru totdeauna de două thread-uri fixate), planificatorul (scheduler-ul) nu mai are de făcut nici o decizie de alegere, pentru că va rula server-ul în cuanta de timp neconsumată a client-ului.

6 Concluzii

Plasarea serviciilor în mîinile unor procese face gestiunea mult mai simplă, dar este un factor mare de ineficienţă datorită *graniţelor numeroase de trecut între sisteme autonome (procese şi nucleu)*. Diferenţa este semnificativă şi forţează programatorii de sistem la tot felul de soluţii extravagante optimizate pentru cazuri particulare.

Windows NT, care este considerat sistemul de operare al viitoarei decade, a constatat pe propria piele dezavantajele arhitecturilor micro-nucleu, căpătînd caracteristici accentuate de monolit, şi folosind tehnici extreme pentru a recupera eficienţa.

Una peste alta, în ziua de astăzi în sisteme de operare viaţa nu este prea simplă pentru programatori!