Anatomia unui apel de sistem în Linux

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

15 decembrie 1997

Subiect: Execuţia unui apel de sistem urmărită pas cu pas în sistemul de operare Linux;

Cunoştinţe necesare: Noţiuni elementare despre sisteme de operare, limbajul C foarte bine, noţiuni despre setul de instrucţiuni al microprocesoarelor Intel 80x86;

Cuvinte cheie: apel de sistem, trap, monitor, cod re-entrant, nucleu.

Cuprins

Acest articol este un "studiu de caz" (case study) în sisteme de operare. Vom urmări (aproape) pas cu pas operaţiile executate de microprocesor pentru execuţia unui foarte simplu apel de sistem; cobaiul experimentului nostru (înafară de cititor) este sistemul de operare Linux. Citirea codului unui sistem de operare "adevărat" este una dintre cele mai bune metode de a înţelege cum funcţionează măruntaiele acestuia. În definitiv ce poate fi mai concret de atîta?

1 Nucleul sistemului de operare

În această secţiune voi revizui pe scurt noţiunea de "nucleu al unui sistem de operare" (kernel); un tratament mai amplu al chestiunii poate fi găsit într-un articol din PC Report din septembrie–octombrie 1996, a cărui copie este disponibilă şi din pagina de web a autorului (ca şi toate celelalte articole ale sale la care face referinţă).

Ce este un sistem de operare? Un set de programe care tratează multe din funcţiile cel mai des utilizate de programele utilizatorilor (cum ar fi accesul la disc) şi permite simultan executarea pe un acelaşi calculator a unor programe independente. Cea mai importantă parte a unui sistem de operare este *nucleul* lui. Acesta este practic o colecţie de funcţii (numite "apeluri de sistem" — system calls) care pot fi executate de programele utilizatorilor şi care îndeplinesc funcţiuni utile. Nucleul unui sistem de operare se bucură de oarecare privilegii relativ la programele scrise de utilizatorii obişnuiţi, în sensul că anumite operaţii sunt permise numai nucleului, dar nu şi programelor care beneficiază de serviciile sale. De pildă utilizatorii nu pot accesa discul în nici un fel; ei au la dispoziţie însă un set de funcţii ale nucleului care fac (teoretic) tot ce utilizatorul ar avea nevoie într-un mod organizat: crează şi distrug fişiere, permit scrierea datelor şi citirea lor în fişiere, precum şi accesul controlat la aceste resurse.

Motivaţia pentru care accesul utilizatorului este interzis la disc este în principal legată de *integritatea* discului: dacă programe diferite ar vrea să folosească fiecare pentru sine discul într-un alt fel, ar putea să interfereze între ele. Nucleul oferă un acces limitat la disc, încercînd să garanteze anumite proprietăţi de consistenţă a datelor: de pildă dacă datele scrise în fişiere diferite nu au nici o legătura unele cu altele, pentru că creşterea ambelor fişiere este supervizată atent de nucleu[1].

Funcţiile nucleului mai sunt ciudate pentru că (pe lîngă faptul că pot folosi anumite operaţii privilegiate), ele sunt comune tuturor programelor care se execută pe acel calculator, fie că programele se execută unul după altul sau simultan. De fapt una din misiunile esenţiale ale nucleului este lansarea programelor în execuţie (şi atunci ele capătă denumirea de "procese") şi controlarea execuţiei lor. Toate nucleele moderne suportă execuţia "simultană" a mai multor procese (ceea ce se numeşte "multiprogramare"). Multiprogramarea poate fi "reală", în cazul în care calculatorul are mai multe procesoare, sau simulată prin ceea ce se numeşte "time-sharing" (punerea în comun a timpului): oprirea unor programe din execuţie temporar pentru a executa altele. Comutarea de la un proces la altul are numele englezesc de "context switch": comutarea contextului. Raţiunea principală pentru time-sharing este una economică: nu toate părţile unui calculator funcţionează cu aceeaşi viteză, deci dacă două dintre ele comunică cea mai rapidă trebuie să aştepte după cea mai lentă (discul de pildă este de cam un milion de ori mai lent decît procesorul). Cînd sunt mai multe lucruri de făcut putem executa unele dintre ele în timp ce altele aşteaptă după operaţiile lente.

[1] Ideea aceasta este binecunoscută în ingineria programării sub numele de "tipuri de date abstracte". Lăsăm cititorului sarcina explorării similitudinii.

1.1 Reentranţa

Nucleul este deci o colecţie de funcţii şi de structuri de date care oferă utilizatorului o sumedenie de operaţii utile. Vom vedea că nucleul are o singură colecţie de structuri de date pentru *toate* procesele care se execută.

Aceste două atribute (multiprogramarea şi unicitatea structurilor de date) puse cap la cap ridică o problemă foarte dificilă: să presupunem că un proces A execută un apel de sistem pentru un acces la disc. O astfel de operaţie este foarte costisitoare (în timp), aşa că nucleul roagă discul să-i trimită datele, şi pentru că are la dispoziţie timp pentru un milion de instrucţiuni suspendă procesul A şi porneşte procesul B. Ce facem însă dacă B face el însuşi un apel de sistem pentru operaţii pe fişiere în timp ce apelul lui A nu s-a terminat? Poate B şterge fişierul pe care A tocmai îl modifică sau altceva de genul ăsta.

Un astfel de cod, care se poate executa *simultan* în contextul mai multor procese se numeşte *cod re-entrant* (se poate intra din nou într-o funcţie în timp ce se execută). Codul re-entrant trebuie proiectat cu foarte multă grijă dintru început şi trebuie scris cu mare atenţie. Nucleele tuturor sistemelor de operare moderne sunt re-entrante.

Subiectul este extrem de interesant şi de subtil; toate cursurile universitare despre sisteme de operare îi consacră o parte relativ importantă. Noi nu ne vom izbi în acest articol explicit de re-entranţă, deşi ea este de fapt "ascunsă" undeva, şi o grămadă de funcţii (despre care *nu* vom discuta) colaborează la a ascunde natura re-entrantă a nucleului. Principala tehnică folosită pentru a scrie cod re-entrant este *regiunea critică*; aceasta este o regiune de cod care nu poate fi executată de mai multe procese simultan. O soluţie la problema de mai sus a proceselor A şi B ar fi de a nu permite nici unui proces să facă operaţii pe fişiere pînă A nu şi-a terminat-o pe a lui (atunci practic toate operaţiile pe fişiere ar fi constituit o regiune critică). În realitate nucleele încearcă să permită cît mai multă activitate concurentă, pentru că de obicei procesele au nevoie de resurse distincte. De pildă ar fi păcat să nu-l lăsăm pe B să şteargă alt fişier decît cel cu care operează A doar pentru că A nu şi-a terminat treaba.

Dar ştiu că sunteţi anxioşi să vedeţi cod, aşa că voi întrerupe aici discuţia despre secţiuni critice.

2 Linux

Voi baza discuţia mea pe sistemul de operare Linux. Linux este un sistem de operare de tip Unix[2], scrisă iniţial în 1991 de un student Finlandez pe nume Linus Torvalds. El este în continuare principalul "scriitor" al nucleului Linux, dar nu cantitativ, pentru că la cele peste 800 000 linii ale codului au contribuit deja mii de voluntari din întreaga lume. Trebuie spus că sistemul este de o calitate foarte bună, rivalizînd cu succes cu produse ale marilor firme care costă bani grei. Diferenţa este că Linux este disponibil în surse oricui îl doreşte; poate fi obţinut contra cost sau gratuit de pe Internet. Linux evoluează foarte rapid; noi versiuni ale nucleului apar la fiecare cîteva zile. Voi baza discuţia mea pe versiunea 2.0.30. Aceasta este ultima versiune mare stabilă a nucleului.

Dezvoltarea nucleului se face pe două linii: cele cu un număr par după'primul punct sunt versiuni stabile, care sunt recomandate celor care folosesc Linux pentru nevoile lor, iar versiunile cu un număr impar (2.1.x) conţin cod experimental, care nu a fost încă îndeajuns testat pentru a fi recomandabil celor care au nevoie de fiabilitate. Versiunile impare sunt folosite de cei care dezvoltă sistemul, sau care au neapărată nevoie de anumite lucruri neimplementate încă în celelalte versiuni.

---

[2] O scurtă istorie a evoluţiei Unix-ului, publicată mai demult în BYTE România, puteţi obţine din pagina de web a autorului.

Linux este suficient de bine scris încît poate rula pe calculatoare extrem de diferite; la ora aceasta el merge pe procesoare 80x86/Pentium (de la Intel), Sparc (SUN), Power PC (IBM), Alpha (Digital, cumpărat de curînd de Compaq), MIPS (acum la Silicon Graphics), M68K (Motorola). Noi ne vom referi la versiunea pentru procesoare Intel, pentru că este cea mai răspîndită. Principiile care emerg sunt însă valabile pentru toate celelalte procesoare.

2.1 Arborele de directoare al sursei

Este instructiv să aruncăm o scurtă privire asupra arborelui de directoare care constituie sursele nucleului. De obicei acesta este instalat în directorul `/usr/src/linux` pe maşinile Linux. În acest articol voi referi toate cărările de directoare relativ la acest punct.

Subdirectoarele principale sunt:

Director	Conţine	Linii de cod
fs	sisteme de fişiere (File System)	68 000
mm	memorie (Memory Management)	17 000
init	procesul init (nr 1, care porneşte maşina)	4000
kernel	funcţii esenţiale ale nucleului	7200
lib	utilitare diverse	1800
include	fişiere header cu declaraţii pentru compilarea nucleului şi programelor utilizatorilor	78 000
net	protocoalele reţelelor de calculatoare	56 000
ipc	mecanisme de comunicare între procese (Inter Process Communication)	2500
drivers	programe care mînuiesc perifericele	412 000
modules	nu conţine surse	0
arch	cod dependent de procesor	150 000[3]

După cum vedeţi mai mult de jumătate de cod este în drivere. Codul driverelor este însă pentru toate plăcile posibile; un anumit sistem va avea compilate numai driverele pentru hardware-ul instalat. Mulţimea aceasta de drivere se datoreşte popularităţii enorme a hardware-ului de PC, pentru care tot omul fabrică cîte o nouă placă.

Două cuvinte şi despre unele din subdirectoarele acestor directoare:

fs/* Linux suportă o mulţime de sisteme de fişiere (organizări ale fişierelor pe disc). Lista lor este include sistemul de fişiere din MS-DOS, din Amiga şi din OS/2, sistemul de fişiere de reţea (NFS, Network File System) de la Sun, sistemul vfat al lui Windows NT, sistemele de fişiere din Unix-ul original, System V (sysv) şi sistemul de fişiere din Berkeley Unix, UFS (numit şi Fast File System, FFS, în literatură), şi altele!. Intenţionez să consacru un articol special arhitecturii sistemelor de fişiere în nucleele Unix (două articole înrudite despre această temă au apărut deja în PC Report), aşa că nu voi divaga în continuare.

include/* conţine headere cu declaraţiile structurilor de date şi prototipurile funcţiilor publice din nucleu;

/drivers/net/* felurite plăci de reţea;

[3] Sunt cam 30 000 de linii dependente de arhitectură pentru Intel, şi în total 150 000 pentru toate arhitecturile.

drivers/block/* toate perifericele tratate de Unix drept colecţii de blocuri: discuri în special;

drivers/char/* majoritatea tuturor celorlalte periferice;

drivers/* alte periferice.

3 Un apel de sistem: `getpid(2)`

Am ales pentru vivisecţia noastră un apel de sistem foarte simplu; poate cel mai simplu. Cu toate acestea periplul nostru pînă la el va fi destul de lung, sî, sperăm, instructiv. Vom discuta despre funcţia `getpid`, "GET Process IDentifier". Nucleul Unix asignează fiecărui proces în curs de execuţie un număr unic între 0 şi 30000 care poate fi folosit pentru comunicarea între procese (semnalele se trimit indicînd acest pid). Pagina de manual Unix care descrie apelul de sistem este în secţiunea 2 a manualului (unde sunt toate celelalte apeluri de sistem); am indicat acest lucru în modul standard, punînd secţiunea între paranteze. Manualul poate fi citit tastînd comanda `man 2 getpid`. Pagina de manual ne spune ca `getpid` nu are argumente şi întoarce ca rezultat PID-ul procesului care face apelul. Iată mai jos şi un exemplu de folosire:

```
#include <unistd.h>
#include <stdio.h>

int main(void)
{
        int p = getpid();
        printf("Pid = %d\n", p);
}
```

Restul acestui articol va explora un singur lucru, şi anume, cum se execută prima linie a programului de mai sus.

3.1 Apelul funcţiei de bibliotecă

În primul rînd trebuie să răspundem la întrebarea: unde este codul funcţiei `getpid`? Cînd l-a scris şi de unde-l ia programul de mai sus.

Răspunsul este: codul este în biblioteca de funcţii a limbajului C care vine împreună cu compilatorul de C şi nucleul sistemului. Fiecare apel de sistem are o astfel de funcţie asociată în bibliotecă. Declaraţia funcţiei este în fişierul header `/usr/include/unistd.h`. Funcţia a fost compilată de cei care au scris compilatorul şi legată în biblioteca de funcţii C `/lib/libc.a`. Corpul funcţiei a fost generat anterior din următoarea sursă C:

```
#include <linux/unistd.h>

_syscall0(int, getpid)
```

3.1.1 Un macro ciudat

Fişierul `include/linux/unistd.h` conţine definiţia macro-ului `syscall0`, care este folosit pentru a genera funcţiile C care cheamă apeluri de sistem cu 0 argumente. În acelaşi fişier există şi codul macrourilor `syscall1`, etc, care generează corpul apelurilor de system cu mai multe argumente.

Iată cum arată cel care ne interesează:

```
#define _syscall0(type,name) \
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name)); \
if (__res >= 0) \
        return (type) __res; \
errno = -__res; \
return -1; \
}
```

Trebuie să recunoaşteţi ca nu vedeţi prea des astfel de cod C, nu? Codul foloseşte mai multe trăsături mai puţin cunoscute (dar absolut standard) ale preprocesorului de C, plus că amestecă asamblare cu C (ceea ce în standardul C nu există, dar Linux se compilează numai cu compilatorul gcc, aşa că nu contează prea tare ce zice standardul).

Ce e ciudat cu acest macro?

- Este un macro pe mai multe linii, scris folosind caracterul \ inainte de sfîrşitul liniei pentru a indica o continuare pe cea următoare;
- Acest macro nu generează o *expresie* cînd este expandat, ci corpul unei funcţii!
- Unul din argumente (`type`) apare în corpul macro-ului într-o poziţie în care trebuie să apară un tip;
- Alt argument (`name`) apare în poziţia unde trebuie să apară un nume de funcţie;
- Se foloseşte operatorul `##`, care concatenează simbolurile de la stînga şi de la dreapta sa.

Să vedem ce iese după pre-procesare din programul de o linie de mai sus:

```
#include <linux/unistd.h>

_syscall0(int, getpid)
```

dă naştere la (ţinînd contul că in acel header avem definiţia: `#define __NR_getpid 20`, ceea ce înseamnă că numărul apelului de sistem `getpid` este 20):

```
int getpid(void)
{
        long __res;
        __asm__ volatile ("int $0x80"
                : "=a" (__res)
                : "0" (20));
        if (__res >= 0)
                return (int) __res;
        errno = -__res;
        return -1;
}
```

Asta se traduce cam aşa:

> Definesc funcţia `getpid` fără argumente care dă ca rezultat un întreg. Funcţia va executa întîi instrucţiunea `int 0x80`, care generează o întrerupere, avînd în registrul 0 numărul `__NR_getpid`, adică 20, iar la sfîrşitul executării lui `int 0x80` rezultatul din registrul A trebuie pus în variabila `__res` (aşa se traduce linia care începe cu `__asm__`, care este scrisă într-un idiom special al compilatorului gcc).
>
> După aceea, dacă `__res` este pozitiv, acesta este rezultatul funcţiei; altfel rezultatul este -1, iar valoarea lui `__res` este pusă în variabila globală a procesului, `errno`.

Deja am învăţat un lucru interesant despre apelurile de sistem (dacă inspectaţi macrourile celelalte, pentru apeluri de sistem cu mai multe argumente, veţi observa aceeaşi comportare): nucleul va întoarce întotdeauna un număr pozitiv ca răspuns la un apel de sistem. O valoare negativă reprezintă codul unei erori. Funcţia de bibliotecă ia codul erorii şi îl pune într-o variabilă globală a procesului, `errno`. Răspunsul unui apel de sistem în caz de eroare este -1. Valorile pe care le poate lua `errno` sunt în fişierul header standard `errno.h`; studiaţi-l, căci este interesant. Variabila aceasta mai este folosită de funcţii ca `perror(3)` sau `strerror(3)` pentru a tipări mesaje de eroare.

3.2 Întreruperea

Am văzut deci că pentru a invoca serviciile nucleului programele pun un număr care descrie serviciul cerut (`getpid()` în cazul nostru) în registrul 0 (la 386 este registrul EAX), după care execută o întrerupere software, cea cu numărul 80 în hexazecimal (128).

Ce mai e şi cu întreruperea asta?

Dacă vă reamintiţi, am spus că programele utilizatorului nu au dreptul să execute orice operaţii, pe cînd nucleul da. Această segregare este realizată de microprocesor printr-un bit intern de stare, care indică dacă programul curent se execută *în mod nucleu (kernel mode)* (şi atunci este privilegiat), sau în *mod utilizator (user mode)*[4]. Microprocesorul trece automat în mod nucleu atunci cînd se întîmplă un eveniment excepţional, cum ar fi:

- Un periferic generează o întrerupere;

[4] De fapt familia x86 are nu 2 ci 4 moduri de privilegiu, dar Linux foloseşte numai 2 dintre ele.

- Un program execută o instrucţiune ilegală;
- Un program accesează zone de memorie interzise;
- Un program execută o întrerupere software;
- O eroare gravă este detectată (ex: a căzut curentul);
- etc.

Trecerea în mod nucleu înseamnă nu doar o schimbare a valorii bitului care indică modul, ci şi *un salt* la o adresă dinainte stabilită. Raţionamentul este următorul: cel care scrie sistemul de operare scrie pentru fiecare din cazurile de mai sus un program (handler) care ia acţiunile corespunzătoare pentru a remedia evenimentul excepţional. Aceste programe sunt instalate apoi în memorie la început (în procesul de boot-are al calculatorului), iar apoi avem garanţia că utilizatorul nu poate face nici o stricăciune (intenţionată sau nu), pentru că orice acţiune excepţională va transfera controlul la unul dintre aceste programe scrise dinainte şi în care avem mare încredere.

Pentru a fi şi mai precişi: fiecare eveniment excepţional are la nivelul microprocesorului un număr asociat. Instalarea handler-elor pentru excepţii constă în construirea unui vector de adrese de proceduri, care indică pentru fiecare excepţie ce procedură trebuie s-o trateze, cam aşa[5]:

```
codul exceptiei
  |
  |
  |      ---------
  |     0|       |------->procedura pentru tratarea impartirii la 0
  |      ---------
  \---->1|       |------->procedura pentru depanare
         ---------
         .........
         ---------
      128|       |------->procedura pentru tratarea unui apel de sistem
         ---------

         vector de
         exceptii
```

Vectorul de excepţii este construit imediat după pornirea sistemului; funcţia răspunzătoare de acest lucru este în fişierul `arch/i386/kernel/traps.c`, şi este numită `trap_init()`. Codul esenţial arată cam aşa:

```
void trap_init(void)
{
.....
        set_trap_gate(0,&divide_error);
```

[5] Procesoarele Intel disting mai multe tipuri de evenimente excepţionale, clasificînd separat întreruperile generate de hardware, erorile de execuţie, etc. Diferitele tipuri funcţionează însă la fel, doar că fiecare tip are alt vector de excepţii.

```
        set_trap_gate(1,&debug);
        set_trap_gate(2,&nmi);
        set_system_gate(3,&int3);       /* int3-5 can be called from all */
        set_system_gate(4,&overflow);
        set_system_gate(5,&bounds);
        set_trap_gate(6,&invalid_op);
        set_trap_gate(7,&device_not_available);
        set_trap_gate(8,&double_fault);
        set_trap_gate(9,&coprocessor_segment_overrun);
        set_trap_gate(10,&invalid_TSS);
        set_trap_gate(11,&segment_not_present);
        set_trap_gate(12,&stack_segment);
        set_trap_gate(13,&general_protection);
        set_trap_gate(14,&page_fault);
        set_trap_gate(15,&spurious_interrupt_bug);
        set_trap_gate(16,&coprocessor_error);
        set_trap_gate(17,&alignment_check);
        for (i=18;i<48;i++)
                set_trap_gate(i,&reserved);
        set_system_gate(0x80,&system_call);
.....
}
```

Asta e relativ simplu de ghicit ce înseamnă: excepţia nr 0, care se declanşează cînd se împarte la 0, va fi tratată de funcţia `divide_error`, care este undeva prin nucleu, excepţia 1 de funcţia `debug`, etc.

Cît despre codul macro-ului `set_trap_gate()`, îl puteţi găsi în fişierul `include/asm-i386/system.h`. Codul este încîlcit pentru că procesoarele x86 nu conţin în căsuţa din vectorul de excepţii doar adresa unei proceduri, ci şi o mulţime de alte informaţii, legate de privilegiile pe care le are un program în timp ce execută excepţia, de tipul excepţiei (excepţie, întrerupere), etc. Studiul detaliat al tabelei ne-ar îndepărta de la scopul nostru, şi anume de a vedea cum se execută un apel de sistem. Important este de reţinut:

1. După executarea întreruperii software execuţia sare la o procedură specificată de vectorul de excepţii (`system_call` pentru exemplu nostru concret);
2. Microprocesorul intră în mod nucleu;
3. Microprocesorul schimbă stiva curentă la cea indicată de noul privilegiu.

3.2.1 Stiva

Acest ultim punct merită o clarificare.

Cum se execută procedurile? Folosind o stivă pentru a-şi păstra variabilele locale; cînd o procedură o cheamă pe alta se contruieşte un nou *cadru de stivă* (stack frame) pentru procedura nouă, în care aceasta-şi ţine variabilele personale, argumentele şi alte lucruşoare. (Pe îndelete despre rolul stivei am scris în PC Report din ianuarie 1997, în articolul “Multithreading”.)

Nucleul însuşi este practic o colecţie de proceduri, care deci au nevoie de o stivă pentru a se putea executa. Dar de unde s-o ia pe aceasta? Nucleul nu poate folosi stiva pe care o foloseşte procesul în mod obişnuit, pentru că nu poate avea încredere în proces. Diferenţa este că dacă procesul manipulează stiva într-un mod eronat, nu poate face rău decît sieşi, datorită faptului că mecanismele de memorie virtuală împiedică un proces să acceseze memoria alocată altor procese. Cu nucleul lucrurile nu mai stau aşa: privilegiile lui ridicate i-ar putea permite să scrie oriunde, ştergînd orice.

Din cauza aceasta, la procesoarele x86, o schimbare de privilegiu a procesorului implică *automat* o schimbare de stivă. Cum se face asta? Fiecare proces are o tabelă cu pointeri către stive, cîte una pentru fiecare nivel de privilegiu. Acest lucru poate fi văzut în fişierul `include/asm-i386/processor.h`, unde cele 4 stive, corespunzînd celor 4 nivele de privilegiu ale familiei x86, sunt indicate în structura numită TSS (Task Segment Selector, terminologie Intel):

```
struct thread_struct {
        unsigned short  back_link,__blh;
        unsigned long   esp0;                  \
        unsigned short  ss0,__ss0h;            |
        unsigned long   esp1;                  | 3 stive
        unsigned short  ss1,__ss1h;            |
        unsigned long   esp2;                  |
        unsigned short  ss2,__ss2h;            /
        unsigned long   cr3;
        unsigned long   eip;
        unsigned long   eflags;
        unsigned long   eax,ecx,edx,ebx;
        unsigned long   esp;                   | virful stivei curente
        unsigned long   ebp;
        unsigned long   esi;
        unsigned long   edi;
        unsigned short  es, __esh;             | segmentul stivei curente
        unsigned short  cs, __csh;
        unsigned short  ss, __ssh;
        unsigned short  ds, __dsh;
        unsigned short  fs, __fsh;
        unsigned short  gs, __gsh;
        unsigned short  ldt, __ldth;
        unsigned short  trace, bitmap;
        unsigned long   io_bitmap[IO_BITMAP_SIZE+1];
        unsigned long   tr;
        unsigned long   cr2, trap_no, error_code;
/* floating point info */
        union i387_union i387;
/* virtual 86 mode info */
        struct vm86_struct * vm86_info;
        unsigned long screen_bitmap;
        unsigned long v86flags, v86mask, v86mode;
```

```
}
```

Cînd nucleul crează un proces îi alocă două stive: una pentru modul utilizator şi una pentru modul nucleu. (Celelalte două stive nu sunt niciodată folosite de Linux). Cînd microprocesorul îşi schimbă privilegiul îşi schimbă automat şi stiva curentă.

Stiva nucleului în general este mică (4K), pentru că nucleul este o bucată fixă de cod, care nu conţine apeluri recursive de funcţii, deci consumă relativ puţină stivă.

Deci funcţia `sys_call`, chemată indirect prin întrerupere, şi toate funcţiile chemate de ea, se vor executa pe stiva procesului curent care corespunde modului nucleu.

3.3 Poarta de intrare în nucleu

Să vedem ce se întîmplă mai departe. Codul funcţiei `system_call` este (din păcate) scris în asamblare. Se găseşte în fişierul `arch/i386/kernel/entry.S`, şi foloseşte din plin macro-uri foarte simple definite în alte părţi (cele mai interesante în `include/asm-i386/linkage.h`), (cum ar fi `ENTRY`, `SYMBOL_NAME`, `SAVE_ALL`, etc.). Zic "din păcate", pentru că dialectul de asamblare al compilatorului gcc nu este acelaşi sintactic cu cel al firmei Intel, aşa că acelaşi program se scrie în feluri diferite folosind cele două limbaje. Mă rog, nu o să ne împiedicăm noi de atîta lucru; să încercăm să ne facem o idee despre ce se întîmpla în codul următor:

```
ENTRY(system_call)
        pushl %eax                      # save orig_eax
        SAVE_ALL
#ifdef __SMP__
        ENTER_KERNEL
#endif
        movl $-ENOSYS,EAX(%esp)
        cmpl $(NR_syscalls),%eax
        jae ret_from_sys_call
        movl SYMBOL_NAME(sys_call_table)(,%eax,4),%eax
        testl %eax,%eax
        je ret_from_sys_call
#ifdef __SMP__
        GET_PROCESSOR_OFFSET(%edx)
        movl SYMBOL_NAME(current_set)(,%edx),%ebx
#else
        movl SYMBOL_NAME(current_set),%ebx
#endif
        andl $~CF_MASK,EFLAGS(%esp)     # clear carry - assume no errors
        movl %db6,%edx
        movl %edx,dbgreg6(%ebx)  # save current hardware debugging status
        testb $0x20,flags(%ebx)         # PF_TRACESYS
        jne 1f
        call *%eax
        movl %eax,EAX(%esp)             # save the return value
        jmp ret_from_sys_call
```

Paşii mari sunt următorii:

- Se salvează registrul AX, care conţine codul apelului de sistem;
- Se salvează toţi regiştrii (care au valorile pe care le aveau cînd s-a executat întreruperea 0x80);
- Dacă calculatorul este un calculator cu mai multe procesoare se execută un cod special pentru sincronizarea nucleelor de pe diferitele procesoare[6].
- Numărul apelului de sistem este comparat cu numărul total de apeluri existente (`NR_syscalls`, un macro definit în `include/asm-i386/unistd.h`).
- Dacă numărul este înafara limitelor atunci nucleul se întoarce imediat la utilizator (prin salt la `ret_from_sys_call`) cu eroarea `ENOSYS` ("nu avem un astfel de apel de sistem");
- Nucleul indexează cu codul apelului într-o tabelă care conţine adresele tuturor funcţiilor care tratează apeluri de sistem (tabela numită `sys_call_table` este discutată în secţiunea următoare); adresa funcţiei de tratare este pusă în registrul EAX;
- Dacă o înregistrarea din tabelă este 0, funcţia respectivă nu există, deci din nou ne întoarcem la utilizator cu eroare;
- Se fac felurite procesări legate de multiprocesoare şi eventuala depanare a procesului curent; le ignorăm;
- Instrucţiunea principală este `call *%eax`, adică salt la adresa din registrul EAX. Această instrucţiune execută funcţia corespunzătoare apelului de sistem; rezultatul acestei funcţii este întors prin convenţie tot în registrul EAX.
- Rezultatul din registrul EAX este pus pe stivă;
- Se sare la eticheta `ret_from_sys_call`, discutată mai jos.

3.4 Tabela de dispecerizare

Am văzut că funcţia de bibliotecă a pus în registrul EAX un cod de apel de sistem, că întreruperea a comutat privilegiul şi stiva, iar apoi că în nucleu s-a indexat într-o tabelă mare cu codul din EAX. Această tabelă este construită tot în fişierul `arch/i386/kernel/entry.S`, şi arată cam aşa:

```
.data
ENTRY(sys_call_table)
        .long SYMBOL_NAME(sys_setup)            /* 0 */
        .long SYMBOL_NAME(sys_exit)
        .long SYMBOL_NAME(sys_fork)
```

[6] SMP înseamnă Symmetric Multi Processing, şi este o tehnică în care pe un calculator cu mai multe procesoare fiecare procesor execută cod atît de proces utilizator cît şi de nucleu. Scrierea de cod pentru multiprocesoare simetrice este mult mai grea decît scrierea de cod re-entrant, din motive pe care nu avem timp să le explorăm acum, dar asupra cărora sperăm să revenim altădată. Oricum, Linux aici "trişează" un pic, nepremiţînd unui procesor să execute cod nucleu dacă un alt procesor execută deja cod nucleu pentru un alt proces.

```
        .long SYMBOL_NAME(sys_read)
        .long SYMBOL_NAME(sys_write)
        .long SYMBOL_NAME(sys_open)              /* 5 */
        .........................
        .long SYMBOL_NAME(sys_getpid)            /* 20 */
        .........................
        .space (NR_syscalls-165)*4               /* neimplementate */
```

După cum vedeţi în căsuţa 20 a tabelei se găseşte adresa unei funcţii, numită `sys_getpid`. Această funcţie va fi deci executată atunci cînd codul apelului de sistem este 20.

3.5 Funcţia `sys_getpid()`

Am ajuns în fine la funcţia din nucleu care face procesarea corespunzătoare. Codul ei este în fişierul `kernel/sched.c`, şi este banal; îl reproducem în întregime:

```
asmlinkage int sys_getpid(void)
{
        return current->pid;
}
```

Prin convenţie compilatorul gcc pune rezultatul unei funcţii C în registrul EAX; din această cauză valoarea întoarsă de această funcţie poate fi consumată de codul de mai sus.

Dar cine este `current`? Este nimeni altul decît "procesul" curent. Cum vine asta?

3.5.1 Structura Task

Pentru a răspunde la această întrebare trebuie să aflăm ce este un proces pentru nucleu. Ei bine, pentru nucleu un proces este nimic altceva decît o structură de date. Putem vedea această structură de date în fişierul `include/linux/sched.h`; unul dintre cîmpurile ei este structura TSS de care am vorbit mai sus. Ea arată cam aşa:

```
struct task_struct {
/* these are hardcoded - don't touch */
        volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
        long counter;
        long priority;
        unsigned long signal;
        unsigned long blocked;  /* bitmap of masked signals */
        unsigned long flags;    /* per process flags, defined below */
        int errno;
        long debugreg[8];  /* Hardware debugging registers */
        struct exec_domain *exec_domain;
/* various fields */
        struct linux_binfmt *binfmt;
        struct task_struct *next_task, *prev_task;
        struct task_struct *next_run,  *prev_run;
```

```
        unsigned long saved_kernel_stack;
        unsigned long kernel_stack_page;
        int exit_code, exit_signal;
        /* ??? */
        unsigned long personality;
        int dumpable:1;
        int did_exec:1;
        /* shouldn't this be pid_t? */
        int pid;
        int pgrp;
        int tty_old_pgrp;
        int session;
        /* boolean value for session group leader */
        int leader;
        int     groups[NGROUPS];
        /*
         * pointers to (original) parent process, youngest child, younger sibling,
         * older sibling, respectively.  (p->father can be replaced with
         * p->p_pptr->pid)
         */
        struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
        struct wait_queue *wait_chldexit;       /* for wait4() */
        unsigned short uid,euid,suid,fsuid;
        unsigned short gid,egid,sgid,fsgid;
        unsigned long timeout, policy, rt_priority;
        unsigned long it_real_value, it_prof_value, it_virt_value;
        unsigned long it_real_incr, it_prof_incr, it_virt_incr;
        struct timer_list real_timer;
        long utime, stime, cutime, cstime, start_time;
/* mm fault and swap info: this can arguably be seen as either
mm-specific or thread-specific */
        unsigned long min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
        int swappable:1;
        unsigned long swap_address;
        unsigned long old_maj_flt;      /* old value of maj_flt */
        unsigned long dec_flt;          /* page fault count of the last time */
        unsigned long swap_cnt;         /* number of pages to swap on next pass */
/* limits */
        struct rlimit rlim[RLIM_NLIMITS];
        unsigned short used_math;
        char comm[16];
/* file system info */
        int link_count;
        struct tty_struct *tty; /* NULL if no tty */
/* ipc stuff */
```

```
        struct sem_undo *semundo;
        struct sem_queue *semsleeping;
/* ldt for this task - used by Wine.  If NULL, default_ldt is used */
        struct desc_struct *ldt;
/* tss for this task */
        struct thread_struct tss;
/* filesystem information */
        struct fs_struct *fs;
/* open file information */
        struct files_struct *files;
/* memory management info */
        struct mm_struct *mm;
/* signal handlers */
        struct signal_struct *sig;
#ifdef __SMP__
        int processor;
        int last_processor;
        int lock_depth;         /* Lock depth. We can context switch
in and out of holding a syscall kernel lock... */
}
#endif
```

Nucleul manipulează în principiu două mari clase de structuri de date:

- Structuri de date private care aparţin unui singur proces; de exemplu pid-ul, prioritatea, pointeri spre fişierele deschise, etc.
- Structuri de date *globale* întregului sistem: fişiere, memorie, procesoare, etc.

Practic tot ce este per-proces este ţinut într-un array mare de structuri de tipul `struct task_struct`. Un array de pointeri spre aceste structuri este declarat în fişierul `kernel/sched.c`:

```
struct task_struct * task[NR_TASKS] = {&init_task, };
```

`current` este un macro definit în `include/linux/sched.h` spre un `task_struct`, care punctează spre procesul care tocmai se execută pe procesorul curent. Planificatorul (scheduler) are grijă ca de fiecare dată cînd comută de la procesul curent la un altul să schimbe valoarea acestui pointer.

3.6 Întoarcerea

Gata, am ajuns pînă în "centrul nucleului". Acum trebuie să ieşim la suprafaţă, cu valoarea calculată. Credeţi că nu poate fi decît mai simplu? Ehe, vă înşelaţi.

Reluăm periplul din fişierul `arch/i386/kernel/entry.S`; acum trebuie să vedem cum se execută funcţia `ret_from_sys_call`, a cărei misiune este să părăsească modul privilegiat. Codul este mai complicat decît ne aşteptăm pentru că această funcţie nu este chemată numai la sfîrşitul unui apel de sistem, ci şi la sfîrşitul unei întreruperi hardware. Problema este că întreruperile hardware pot surveni oricînd, chiar şi atunci cînd se execută deja un apel de sistem sau o altă întrerupere hardware. Din cauza asta nucleul trebuie întîi să verifice dacă trebuie să se întoarcă la modul utilizator sau trebuie să rămînă în mod nucleu; acţiunile sunt diferite în cele două cazuri.

```
        ALIGN
        .globl ret_from_sys_call
ret_from_sys_call:
        cmpl $0,SYMBOL_NAME(intr_count)
        jne 2f
9:      movl SYMBOL_NAME(bh_mask),%eax
        andl SYMBOL_NAME(bh_active),%eax
        jne handle_bottom_half
        movl EFLAGS(%esp),%eax          # check VM86 flag: CS/SS are
        testl $(VM_MASK),%eax           # different then
        jne 1f
        cmpw $(KERNEL_CS),CS(%esp)      # was old code segment supervisor ?
        je 2f
1:      sti
        orl $(IF_MASK),%eax             # these just try to make sure
        andl $~NT_MASK,%eax             # the program doesn't do anything
        movl %eax,EFLAGS(%esp)          # stupid
        cmpl $0,SYMBOL_NAME(need_resched)
        jne reschedule
#ifdef __SMP__
        GET_PROCESSOR_OFFSET(%eax)
        movl SYMBOL_NAME(current_set)(,%eax), %eax
#else
        movl SYMBOL_NAME(current_set),%eax
#endif
        cmpl SYMBOL_NAME(task),%eax     # task[0] cannot have signals
        je 2f
        movl blocked(%eax),%ecx
        movl %ecx,%ebx                  # save blocked in %ebx for signal handling
        notl %ecx
        andl signal(%eax),%ecx
        jne signal_return
2:      RESTORE_ALL
```

Pînă la eticheta "1:" în programul de mai sus asta se petrece: bazîndu-se pe felurite numere, cum ar fi numărul de întreruperi în curs de tratare sau numărul de drivere active în "partea de jos" (bh: bottom half), sau în funcţie de poziţia segmentului de stivă al apelantului se poate deduce din ce loc a fost chemat codul curent. Deşi este deosebit de instructiv de urmat calea în fiecare din aceste cazuri, noi o să pretindem încăpăţînaţi că tocmai de întoarcem în spaţiul utilizator.

Variabila `need_reschedule` este nenulă în cazul în care în timpul execuţiei procesului curent în nucleu s-au petrecut evenimente care cer întreruperea procesului curent şi comutarea la un altul. Hai să zicem că nu s-a întîmplat nimic de acest gen, ca să vedem cum ne întoarcem în spaţiul utilizator.

Dar înainte de acest pas se petrece un alt lucru foarte important: se verifică dacă procesul curent are semnale de primit.

3.7 Livrarea semnalelor

Semnalele sunt o metodă simplistă de comunicaţie inter-proces în Unix. Un semnal este un eveniment identificat printr-un nume şi printr-un număr asociat. Un proces poate trimite semnale altui proces folosind apelul de sistem `kill(2)`, cu care indică PID-ul şi numărul semnalului. Semnalele pot fi trimise spontan de nucleu în anumite circumstanţe.

Un proces poate reacţiona la un semnal în mai multe feluri, şi poate controla într-o oarecare măsură livrarea semnalelor folosind o serie de funcţii de bibliotecă şi apeluri de sistem (`signal, sigsuspend, sigpending, sigaction`, etc.). Am văzut nu demult în PC Report un articol amplu consacrat semnalelor, aşa ca nu voi discuta despre ce fac.

Ce înseamnă că nucleul “transmite un semnal”? Fiecare proces are un array de biţi, cîte unul pentru fiecare semnal. Cînd un proces primeşte un semnal nucleul nu face altceva decît să pună bitul corespunzător pe 1 şi să continue. Adevărata livrare a semnalului se va face mai tîrziu, cînd procesul destinatar se execută.

Din timp în timp un proces verifică dacă nu i-au fost trimise semnale. De obicei face asta înainte de a se bloca în aşteptarea unei activităţi care durează mult timp, şi întotdeauna verifică dacă nu are semnale în momentul cînd termină executarea unui apel de sistem.

Aici am ajuns şi noi cu explicaţiile; codul cu pricina este în fişierul `arch/i386/kernel/entry.S`:

```
signal_return:
        movl %esp,%ecx
        pushl %ecx
        testl $(VM_MASK),EFLAGS(%ecx)
        jne v86_signal_return
        pushl %ebx
        call SYMBOL_NAME(do_signal)
        popl %ebx
        popl %ebx
        RESTORE_ALL
```

Aici nu se întîmplă mare lucru; se cheamă doar funcţia `do_signal` cu felurite argumente pe stivă. Această funcţie se ocupă de tot ce trebuie, livrînd unul cîte unul toate semnalele acumulate între timp. Aceste semnale ar putea avea drept efect omorîrea procesului curent, şi atunci funcţia `do_signal` nu se mai întoarce niciodată.

3.8 Sfîrşitul întreruperii

Presupunînd că `do_signal()` se întoarce, execuţia în mod nucleu se termină cu codul lui `RESTORE_ALL`, care extrage regiştrii salvaţi pe stivă atunci cînd s-a început execuţia în mod nucleu. Codul este tot în fişierul `arch/i386/kernel/entry.S`.

```
#define RESTORE_ALL \
        cmpw $(KERNEL_CS),CS(%esp); \
        je 1f;   \
        GET_PROCESSOR_OFFSET(%edx) \
        movl SYMBOL_NAME(current_set)(,%edx), %eax ; ; \
        movl dbgreg7(%eax),%ebx; \
```

```
        movl %ebx,%db7; \
1:      LEAVE_KERNEL \
        popl %ebx; \
        popl %ecx; \
        popl %edx; \
        popl %esi; \
        popl %edi; \
        popl %ebp; \
        popl %eax; \
        pop %ds; \
        pop %es; \
        pop %fs; \
        pop %gs; \
        addl $4,%esp; \
        iret
```

Cea mai importantă instrucţiune aici este ultima, `iret`. Asta înseamnă "Interrupt RETurn", adică "întoarcere din întrerupere".

Această instrucţiune face exact opusul unei înteruperi, şi anume descreşte privilegiul, comută stivele şi se întoarce la programul întrerupt.

3.9 Terminarea funcţiei de bibliotecă

Iată cum periplul nostru prin nucleu s-a terminat. Ne-am întors înapoi în corpul funcţiei de bibliotecă `getpid()`, avînd în registrul EAX valoarea PID-ului pentru procesul curent. Funcţia aceasta vede dacă valoarea este negativă (nu ar avea nici un motiv să fie în cazul nostru), setează `errno` după cum am descris mai sus şi se întoarce la programul apelant.

4 Rezumat

Am încălecat pe program counter şi am străbătut împreună un periplu în grotele mai superficiale ale nucleului (alte apeluri de sistem au coduri infinit mai complexe, cu multe regiuni critice şi cu probleme grele de re-entranţă).

Să revedem etapele străbătute:

1. Utilizatorul cheamă o funcţie de bibliotecă (`getpid(2)`);
2. Funcţia de bibliotecă împachetează numărul apelului (20) într-un registru şi eventualele argumente în alţi regiştri;
3. Funcţia de bibliotecă generează o întrerupere software (0x80);
4. Automat întreruperea comută în mod nucleu, schimbă stivele şi sare la o procedură de intercepţie (handler);
5. Procedura de intercepţie extrage numărul apelului şi indexează într-o tabelă de apeluri de sistem;

6. Se sare la funcţia care execută cu adevărat apelul (`sys_getpid()`); folosind structurile de date ale nucleului funcţia calculează răspunsul;
7. Codul de întoarcere verifică dacă sunt semnale de livrat procesului curent; dacă da, acestea sunt procesate înainte de întoarcerea în mod utilizator;
8. Se execută o instrucţiune RETI care termină o întrerupere, restaurează privilegiile scăzute şi comută stivele înapoi;
9. Funcţia de bibliotecă despachetează răspunsul şi dacă este necesar setează variabila `errno` la eroarea survenită;
10. Funcţia de bibliotecă întoarce rezultatul primit de la nucleu. Execuţia apelului de sistem s-a terminat.

Cum vi s-a părut?