Multithreading

Mihai BUDIU — budiu@cs.cornell.edu, mihaib@pub.ro

decembrie 1996

Subiect: Implementarea thread-urilor fără ajutorul SO.
Cuvinte cheie: thread, proces, stivă, cadru (stack frame), context.
Cunoştinţe necesare: limbajul C, funcţii recursive, elemente de programare în limbaj de asamblare.

Contents

1 Vedere de ansamblu

Acest articol îşi propune să explice unul dintre conceptele esenţiale ale sistemelor de operare, "thread-urile". (Din păcate nu ştiu nici un termen românesc foarte convenabil care să fie larg acceptat pentru a traduce "thread"; numele folosit este de obicei "fir de execuţie", dar îl găsesc prea lung. Vă rog să toleraţi acest barbarism, şi probabil şi alte cîteva.) Bazîndu-se pe un minim de cunoştinţe, constînd în programarea cu funcţii recursive şi elemente de bază în C, articolul construieşte toate conceptele necesare, şi apoi şi o implementare, a unui pachet de funcţii care permite simularea execuţiei simultane a mai multor programe, fără nici un fel de suport special din partea sistemului de operare.

Din păcate implementarea se bazează pe folosirea în mod neconvenţional a unor instrucţiuni standard C (ANSI), şi s-ar putea să nu poată funcţiona întotdeauna. Am testat programele pe trei sisteme diferite şi au mers, dar am avut într-un fel noroc: m-am bazat pe faptul că (pentru că este mai simplu pentru cei care scriu compilatoare) anumite instrucţiuni fac mai mult decît spune manualul că trebuie să facă.

Structura de ansamblu a textului este următoarea: întîi noţiunile de thread şi proces sunt explicate, iar modul în care pot fi implementate schiţat. Apoi urmează o discuţie despre rolul stivei în executarea programelor — vedem faptul că o parte importantă despre starea unui program este ţinută pe stivă.

Explorăm apoi funcţionalitatea unor instrucţiuni C standard, dar care se comportă extrem de ciudat; vedem că ele acţionează în mod dramatic asupra stivei. Urmează un experiment cu aceste instrucţiuni, a cărui menire este să verifice dacă nu cumva ele îşi îndeplinesc misiunea într-un mod de care ne putem folosi pentru a implementa thread-uri.

În fine, în caz că experimentul s-a dovedit reuşit, trecem la a pune la lucru aceste instrucţiuni pentru a scrie un set de funcţii care simulează multi-procesarea.

Programele au fost testate cu succes pe următoarele platforme: Windows NT, folosind compilatorul MS Visual C++ 4.0, SunOS 4.x, şi 5.x cu gcc, Linux 2.x cu gcc. Se pare că merg şi pe HP-UX.

Extensiile cu "întreruperi" ale programelor au fost testate numai pe sistemele Unix (Windows NT nu oferă funcţia `alarm()` pe care ele se bazează).

Practic textul prezintă un mini-nucleu de sistem de operare în esenţa lui; în orice sistem modern partea care se ocupă de manipularea proceselor are o structură asemănătoare.

2 Procese şi "thread"-uri

"Ce este un program" probabil că toată lumea ştie. Ce este un proces este un pic mai delicat, însă nu greu de înţeles. Cea mai scurtă definiţie ar fi: "un proces este un program în curs de execuţie". E o mică diferenţă între cele două noţiuni: în timp ce un program este un concept "static", imuabil, un proces este un obiect care se schimbă într-una. De exemplu, atunci cînd un program se execută, el modifică valorile variabilelor sale. La fiecare moment de timp procesul deci arată altfel. Distincţia este aceeaşi ca cea dintre un drum trasat pe o hartă (programul) şi o excursie reală pe acel traseu (procesul). Nici măcar nu este obligatoriu ca la rulări diferite un acelaşi program să parcurgă aceleaşi faze: execuţia lui poate depinde de evenimente externe, care sunt diferite (ca de exemplu interacţiunea cu utilizatorul).

Putem vorbi despre "starea" unui proces la un anumit moment de timp. Starea ar trebui să ne spună cam "în ce fază de execuţie a programului am ajuns". Cum se poate descrie starea unui proces?

Ei bine, starea trebuie să cuprindă o indicaţie despre următoarea instrucţiune care se va executa. La nivelul programelor în limbaj-maşină instrucţiunea următoare este indicată prin valoarea unui registru special, numit PC de obicei ("Program Counter") (numai Intel se încăpăţînează să-i zică

IP – "Instruction Pointer"), care arată adresa în memorie unde se află următoarea instrucţiune de executat.

PC-ul singur însă nu ajută: dacă suntem în mijlocul unei bucle, PC-ul nu ne poate spune de cîte ori s-a executat bucla. Descrierea starea unui proces mai trebuie să conţină şi descrierea *valorii* tuturor variabilelor cu care operează programul.

Din păcate nici atîta nu ajunge pentru a descrie starea unui proces. Ca să va convingeţi iată un exemplu:

```
int f(int a)
{
  /* PC==> */ printf("%d\n", a);
}

int g(int a)
{  f(a); }

int main(void)
{  int x = 5;  f(x); g(x); return 0; }
```

Chiar dacă vă spun că PC-ul punctează unde am pus comentariul, în corpul funcţiei `f()`, şi vă dau valorile variabilelor, nu veţi putea spune ce se va întîmpla mai departe dacă nu ştiţi *cine* a chemat pe `f()`: `main()` sau `g()`?

O descrierea a stării procesului trebuie să includă deci şi *istoricul* apelurilor de funcţii: ce funcţii au început să se execute şi nu s-au terminat încă. Vom vedea că această informaţie este de obicei menţinută pe o *stivă* de către calculator.

Să rezumăm: starea unui proces este descrisă de

1. programul care este executat
2. valoarea PC
3. valorile variabilelor
4. funcţiile care tocmai se execută şi locurile în care ele s-au întrerupt.

Calculatoarele moderne ţin aceste informaţii în memorie (cu excepţia PC), în zone diferite, astfel:

Informaţia	Unde
programul	"segmentul de cod"
variabile	"segmentul de date"
funcţiile în curs de execuţie	"segmentul de stivă"

Practic dacă vreau să execut un program (să-l transform într-un proces), calculatorul trebuie să aloce cîte o zonă de memorie pentru fiecare din aceste părţi şi să pună în PC adresa primei instrucţiuni.

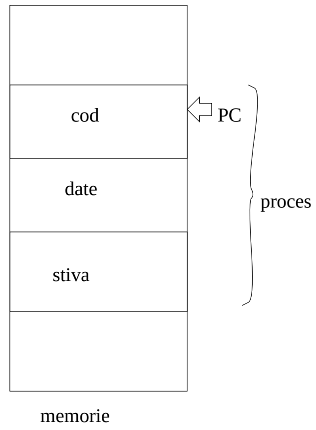

Figura 1: Un proces în memorie

Partea interesantă este că informaţia de mai sus este nu numai necesară ci şi suficientă pentru a descrie starea unui proces. Mai exact, dacă “salvăm” cumva această descriere, stingem calculatorul, şi după o săptămînă restaurăm starea aşa cum era salvată, procesul îşi va continua nestingherit execuţia din locul în care a fost salvat, ca şi cum nimic nu s-ar fi întîmplat.

Tehnica asta este familiară de cînd există congelatoare: o găină pusă la îngheţat nu îşi schimbă starea de loc pînă o scoţi de acolo. Toate variabilele care descriu starea ei (de descompunere) sunt înţepenite la valoarea pe care o aveau în momentul congelării.

Pe acest lucru se bazează şi sistemele de operare moderne cînd dau impresia executării simultane a mai multor procese: (time-sharing) de fapt ele execută un proces o vreme (cîteva milisecunde), apoi pun deoparte starea procesului, iau starea altui proces, îl execută pe ăsta o vreme, după aceea se întorc din nou la primul şi tot aşa.

Dacă cele două procese pe care le execuţi în “paralel” încap simultan în memorie treaba este şi mai simplă: nu ai decît de mutat PC de la unul la altul, şi un indicator pentru care segment de date şi stiva se foloseşte în momentul curent! Faptul că un proces foloseşte numai propriile lui zone de memorie asigură faptul că nu îl “deranjează” pe celălalt.

În dorinţa lor de perfecţiune, proiectanţii au observat că pot îmbunătăţi schema în cazul în care cele două procese execută acelaşi program (nu este o presupunere complet absurdă, mai ales dacă pe acelaşi calculator pot lucra mai multe persoane simultan). Îmbunătăţirea se bazează pe faptul că (practic niciodată) segmentul de cod nu se schimbă. Din cauza asta dacă ai două procese care au acelaşi segment de cod, le poţi aranja să folosească împreună acea parte de memorie. Observaţi că asta nu înseamnă că fac exact acelaşi lucru la un moment dat: PC-urile lor pot avea valori complet diferite.

Odată ideea asta născută, a fost împinsă şi mai departe: datorită faptului că procesele nu au nici un fel de memorie modificabilă în comun, este destul de greu pentru două procese să schimbe informaţii (trebuie să folosească fişiere, ceea ce e complicat şi lent). Ce-ar fi dacă am permite proceselor să aibă şi ceva memorie în comun? Ele vor trebui să fie atente să nu se calce pe picioare (să nu modifice fiecare starea altuia cînd nu trebuie), dar ar putea cîştiga mult prin eficienţa mare a comunicării.

Aşa s-a născut noţiunea de “thread”: două thread-uri sunt procese care au în comun şi seg-

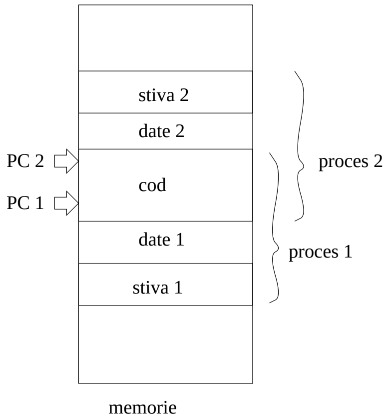

Figura 2: Procese cu zona de cod comună

mentul de date, pe lîngă cel de cod. Din cauză că thread-urile au în comun atît de mult, cîteodată conceptual sunt privite ca parte a aceluiaşi proces, dar care de data asta poate avea mai multe "fire de execuţie" simultan (traducerea termenului "thread" este fir). Pentru utilizator e ca şi cum ai scrie un program care poate executa mai multe funcţii simultan (de altfel asta vom obţine şi noi în final).

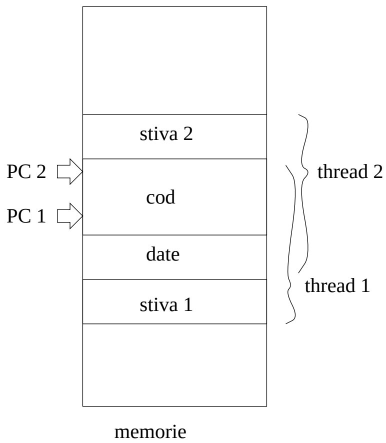

Figura 3: Folosirea memoriei de thread-uri

Distincţia dintre un proces şi un thread este doar de natură a folosirii în comun a memoriei: practic două procese care execută acelaşi program au memorii complet disjuncte, dar două thread-uri care execută acelaşi program folosesc segmentul de date în comun.

3 Recursie şi stive

În secţiunea de faţă o să ne concentrăm privirea spre una singură din zonele de memorie de care are nevoie un proces (sau thread): stiva. Vom vedea de ce e nevoie de ea, şi cum este ea folosită. Vom vedea că unul dintre cele mai utile concepte ale limbajelor de programare moderne, recursia, este implementată folosind mecanismul de stivă.

Să considerăm un exemplu simplu, folosit de toată lumea:

```
int factorial(int n)
{
    if (n == 1)
        return 1;
    else
        return
               factorial(n-1)  /* <= PC1 */
               * n;
}

int main(void)
{
        int l;
        l  = factorial(2);     /* <= PC2 */
        printf("%d\n", l);
        l = factorial(4);
        printf("%d\n", l);
        return 0;
}
```

Cum merge? Păi execuţia se desfăşoară aşa: începem cu prima instrucţiune din `main()`, care este chiar un apel al funcţiei `factorial`. Pentru a executa `factorial()` programul pune bine la păstrare valoarea curentă a lui PC; cînd funcţia `factorial()` va termina valoarea păstrată va fi folosită pentru a continua execuţia. (Dacă nu am salva valoarea lui PC cînd chemăm `factorial()`, nu am putea şti atunci cînd se va termina dacă a fost chemat prima oară, cu argumentul 2, sau a doua oară, cu 4.)

În continuare se procedează astfel: PC-ul este pus să arate la prima instrucţiune din corpul funcţiei `factorial`; am început s-o executăm. Valoarea argumentului `n` este luată din valoarea cu care a fost chemat factorial, 2. Programul se execută de sus în jos, `n != 1`, deci se execută `return`-ul după `else`. Însă `return` are nevoie de valoarea calculată de `factorial(n-1)` = `factorial(1)`! Înainte de a termina prima invocare a funcţiei factorial, trebuie să începem o a doua. Trebuie să punem deoparte valoarea pe care o are `n` acum (2), ca `factorial(1)` să nu o strice. Trebuie să ne amintim şi PC-ul curent, ca după ce `factorial(1)` se termină, să putem face înmulţirea şi să terminăm `return`-ul.

Deja figura se complică: ţinem minte locul de unde `factorial` a fost chemat din `main`, dar şi locul de unde s-a chemat singur, şi valorile variabilelor sale locale din acea clipă... Ne încurcăm. Avem nevoie de o structură care să facă ordine.

Observaţi două proprietăţi ale funcţiilor de care vom profita:

1. funcţiile se termină întotdeauna în ordinea inversă decît cea în care sunt invocate. Dacă A cheamă pe B, B se va termina înaintea lui A.

2. de fiecare data cînd funcţia se execută se crează o nouă copie a variabilelor locale (putem include în această categorie şi argumentele funcţiei, cum este `n` al lui `factorial`).

De fapt noi folosim acelaşi nume pentru o mulţime de obiecte: la un moment dat pot exista multe `n`-uri.

Funcţiile recursive se implementează folosind *cadre* ("(stack) frames" în engleză). Un cadru este o structură de date care ţine în ea informaţiile necesare executării unei instanţe a unei funcţii — fiecare factorial care se va executa va avea propriul lui cadru.

Un cadru de stivă arată cam aşa:

argument 1 argument 2 ... argument n PC pentru întoarcere
variabile locale spaţiu de lucru temporar

Cadrele sunt construite în două etape: prima parte (cea de deasupra liniei) este făcută de funcţia care o cheamă pe cea care posedă cadrul; prin această zonă chemătorul şi chematul comunică.

Partea a doua este construită de funcţia care foloseşte cadrul cînd începe să se execute. Acolo îşi pune ea lucruşoarele pe care vrea să le folosească singură.

Ca o consecinţă a faptului că funcţiile se termină invers de cum au început, putem pune cadrele lor pe o stivă (stiva, ţineţi minte=ultimul venit–primul ieşit). În fiecare moment funcţia care tocmai se execută foloseşte cadrul din vîrful stivei.

acţiune	operaţie pe stiva
apel de funcţie return execuţia funcţiei	se creaza un nou cadru în vîrf se distruge cadrul din vîrf se foloseşte cadrul din vîrful stivei

Ca să fie şi mai clar, să vedem cum arată stiva în momente succesive, în timpul execuţiei programului de mai sus:

Figura 4 ne arată stiva in 4 momente succesive: la început, înainte ca `main()` să cheme `factorial(2)`, după aceea cu cadrul lui `factorial(2)` construit. Observaţi cum cadrul lui `factorial(2)` conţine adresa de întoarcere (PC2) în `main()`. Funcţiile noastre nu au variabile temporare, iar singura variabilă locală este `l`, în `main`.

Apoi `factorial(2)` cheamă `factorial(1)`: cadrul lui `factorial(2)` continuă să existe, cu n=2, şi un nou cadru se crează, în care n=1. Observaţi că `factorial(1)` ţine minte în cadrul lui că a fost chemat de `factorial(2)`, deci PC-ul după terminare trebuie să fie PC1. Observaţi şi că pe stivă se află două valori ale lui n, una în cadrul lui `factorial(1)`, una în cadrul lui `factorial(2)`. Regulile limbajului C spun că, dacă funcţia curentă vrea să ştie cît este n, valoarea cu care va lucra este cea din cadrul curent.

`factorial(1)` se termină repede, returnînd valoarea 1. Cadrul lui `factorial(1)` este distrus. Cadrul curent devine cel vechi, al lui `factorial(2)`, a cărui execuţie se reia de după PC1. Ce se întîmplă apoi: `factorial(2)` tocmai a aflat cît e `factorial(1)` (=1), aşa că va continua

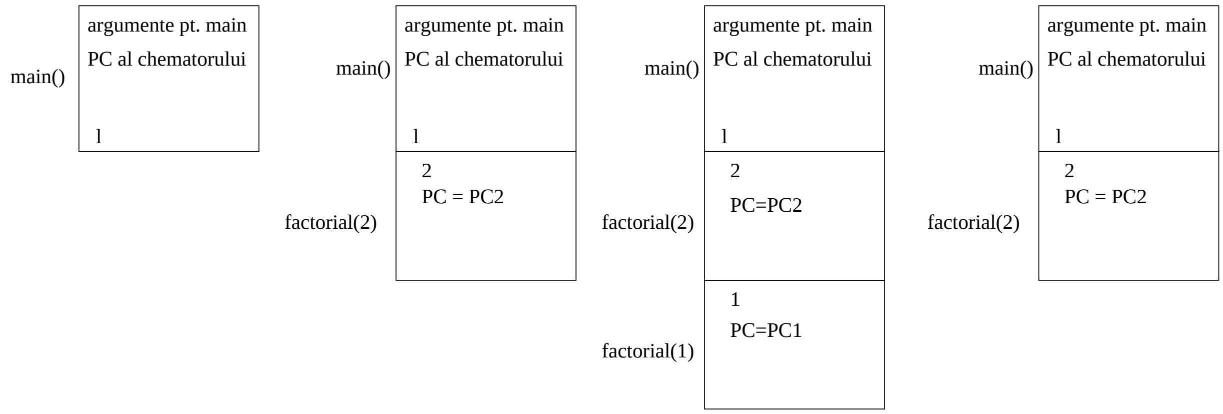

Figura 4: Cadrele în timpul execuţiei

evaluarea expresiei `n * factorial(n-1)`. Valoarea folosită pentru n este 2, deci expresia este 2*1 = 1. Aceasta este şi valoarea returnată de `factorial(2)`, care se întoarce la `main()`, la adresa salvată în cadrul propriu, PC2.

După ce `main()` tipăreşte "2", se va efectua un şir asemănător de operaţii, numai că la momentul de vîrf se vor afla pe stivă 4 cadre diferite pentru factorial. Încercaţi să urmăriţi evoluţia.

4 C exotic: `longjmp` şi `setjmp`

Standardul ANSI pentru limbajul C conţine printre instrucţiunile mai mult sau mai puţin ezoterice şi cerinţa ca orice compilator de C regulamentar să implementeze două foarte ciudate "funcţii", numite `setjmp` şi `longjmp`. Prototipurile acestor funcţii sunt în headerul `<setjmp.h>`.

Poate cel mai bine ar fi să începem iarăşi printr-un exemplu; rulaţi următorul program:

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

jmp_buf cadru;

void fun(void)
{
   printf("fun se executa\n");
   longjmp(cadru, 1);
   printf("Nu ajung aici!\n");
}

int main(void)
{
   if (setjmp(cadru) == 0) {    /* <= PC salvat */
      printf("setjmp s-a intors prima oara\n");
      printf("Chem acum fun\n");
```

```
        fun();
        printf("Nu ajung aici!\n");
    }
    else {   /* setjmp != 0 */
        printf("setjmp s-a intors a doua oara!\n");
    }
    printf("Gata.\n");
    return 0;
}
```

Acum să vedem cum funcţionează...Ca mai întotdeauna în cazul limbajelor de programare, cel mai simplu mod pentru a explica ce face un program este de a explica fiecare instrucţiune componentă.

4.1 `setjmp`

Instrucţiunea `setjmp` are ca argument un obiect de tip `jmp_buf` (tip declarat în headerul `setjmp.h`). (E mai simplu să ne gîndim la această instrucţiune ca la un macro, nu ca la o funcţie.) Execuţia lui `setjmp(x)` pune în structura `x` următoarele informaţii:

1. o descriere a *poziţiei* pe stivă a cadrului funcţiei curente (cea care conţine instrucţiunea `setjmp()`);
2. valoarea curentă a PC (care punctează tocmai la instrucţiunea `setjmp()`, nu?);
3. În plus, `setjmp()` dă ca rezultat o valoare 0

4.2 `longjmp`

Instrucţiunea `longjmp()` are două argumente: unul de tip `jmp_buf`, în care un `setjmp()` anterior a scris ceva, şi o valoare numerică. Efectul executării lui `longjmp(x, v)` este:

1. cadrul descris de `x` devine cel curent;
2. valoarea salvată a PC in `x` devine cea curentă;
3. se continuă execuţia; asta înseamnă că se re-execută `setjmp()` (pentru că acolo a fost salvat PC!). De data asta însă rezultatul "evaluării" lui `setjmp()` este `v`!

Să ne uităm la programul de mai sus din nou. El va tipări:

1. `setjmp s-a intors prima oara`
2. `Chem acum fun`
3. `fun se executa`
4. `setjmp s-a intors a doua oara!`

5. Gata.

Execuţia începe cu `setjmp(cadru)`. Asta salvează în variabila `cadru` descrierea cadrului funcţiei curente, `main()`, şi valoarea PC, care punctează la linia cu comentariul. `setjmp()` întoarce 0, aşa că `if`-ul este "adevărat". Prima linie se tipăreşte acum, ca şi a doua.

Se cheamă `fun()`, care scrie linia a treia.

Partea interesantă începe: se execută `longjmp(cadru, 1)`. Aplicînd definiţia de mai sus, asta înseamnă:

1. cadrul curent devine cel salvat, al funcţiei `main()`. Practic cadrul lui `fun()` este "şters" de pe stivă.
2. PC capătă valoarea din momentul lui `setjmp`: punctează la linia cu comentariul din nou; practic `setjmp` se execută din nou!
3. valoarea execuţiei lui `setjmp` este 1, cum a indicat `longjmp`-ul.

Deci `if`-ul se execută din nou şi el, pe ramura `else` de data asta. Asta cauzează apariţia liniei 4.

Linia 5 apare apoi natural.

Interesant, nu?

Practic `longjmp` face o "întoarcere în timp", la situaţia pe care o avea execuţia unui program în momentul cînd a executat `setjmp`.

Care e folosul unor asemenea instrucţiuni?

`setjmp` şi `longjmp` au fost create pentru a uşura tratarea erorilor: dacă ceva se întîmplă undeva în adîncurile misterioase ale unor funcţii recursive, poţi imediat "ieşi la suprafaţă" cu un `longjmp`, fără să trebuiască să scrii fiecare funcţie de pe drum în aşa fel încît să verifice dacă nu cumva funcţiile pe care le cheamă au produs vreo eroare.

(Modul acesta de a trata erorile poate fi îmbunătăţit folosind limbaje care permit definirea şi tratarea excepţiilor, care funcţionează într-un mod asemănător, dar mai elegant. De altfel compilatorul de C de la Microsoft pentru Windows NT chiar extinde limbajul C pentru a încorpora excepţii; însuşi sistemul de operare Windows NT ajută la asta).

Să mai observăm că în momentul executării instrucţiunii `longjmp` *cadrul de stivă* indicat de `jmp_buf`-ul dat ca argument lui `longjmp` *trebuie* sa existe! (Dacă nu, dezastre se vor întîmpla.) Din această cauză standardul ANSI spune despre `longjmp`: "`longjmp` poate fi executată numai într-o funcţie care este un descendent direct din funcţia care a executat `setjmp`". Datorită modului în care se crează şi distrug cadre de stivă, această cerinţă îndeplinită va garanta existenţa cadrului la care se sare.

[Cu alte cuvinte nu se poate face aşa:

```
jmp_buf x;

int g()
{  setjmp(x); }

int h()
{  longjmp(x); }
```

```
int main()
{  g(); h(); }
```

pentru că atunci cînd h() face longjmp, cadrul salvat în x, al funcţiei g() nu mai este pe stivă, pentru că funcţia g() şi-a terminat execuţia — h() nu este un descendent direct al lui g().]

5 Un abuz

Am spus mai sus ca longjmp "şterge" de pe stiva toate cadrele pîna la cel salvat. De fapt acest lucru este foarte costisitor, şi este mult mai simplu pentru longjmp doar să "mute" vîrful stivei la cadrul indicat. Este o şansă destul de mare ca stiva între cadrul la care se sare şi cel de la care se sare să rămînă neatinsă.

Mai trebuie să vedem şi dacă longjmp poate fi "prostită" să sară în alte direcţii decît scrie la carte; să încercăm următorul program:

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

jmp_buf b[2];
int setup = 0;

void fun1(void);

void delay(void) { for (long i=0; i < 5000000; i++); }

void fun0(void)
{
        if (setjmp(b[0]) == 0)
                fun1();
        else {
                printf("Fun 0\n");
                delay();
                longjmp(b[1], 1);
        }
}

void fun1(void)
{
        if (!setup) setjmp(b[1]);
        setup=1;
        printf("Fun 1\n");
        delay();
        longjmp(b[0], 1);
}
```

```
int main(void)
{
        fun0();
        return 0;
}
```

Funcţiile `fun0()` şi `fun1()` nu au variabile locale şi nici argumente; practic cadrele lor nu conţin nici un fel de informaţii interesante.

Evoluţia programului nu este prea greu de urmărit, cel puţin pînă la executarea lui `fun1()`. Practic `main()` cheamă `fun0()`, acesta îşi salvează cadrul în `b[0]`, `fun0()` cheamă apoi `fun1()`, care salvează cadrul lui personal in `b[1]`, şi scrie ceva pe ecran.

Partea interesantă începe: `fun1()` face un `longjmp` la `b[0]`; dacă avem noroc, aceasta mută PC la loc în `fun0()`, care se re-execută. Vom obţine pe ecran:

```
Fun1
Fun0
Fun1
...
```

Dacă programul compilat de dumneavoastră nu se comportă astfel, înseamnă că `longjmp` nu vrea să sară cînd cadrul descris de `jmp_buf` nu este anterior celui curent pe stivă. Puteţi continua să citiţi textul mai departe, dar programele pe care le prezentăm nu vor funcţiona nici ele; desigur, se pot scrie altfel, fără a folosi `setjmp` şi `longjmp`, în limbaj de asamblare. Însă detaliile ar deveni mai proeminente decît tabloul însuşi.

6 Stive în stivă

Presupunînd că

1. `longjmp` poate sări în orice direcţie;

2. `longjmp` nu strică stiva atunci cînd se execută;

vom implementa acum pas cu pas un set de funcţii care simulează multithreading. Ideea centrală este următoarea: pe stivă creăm mai multe cadre, a căror descriere o salvăm într-un vector de `jmp_buf`. După aceea, să executăm thread-ul 10 înseamnă să facem cadrul din căsuţa 10 a vectorului cadrul curent (cu un `longjmp`). Să întrerupem thread-ul 5 din execuţie înseamnă să punem în căsuţa 5 a vectorului descrierea cadrului curent (cu un `setjmp`).

Avem o singură problemă reală: cum facem ca un thread să nu dea cu stiva peste cadrele care se află pe stivă “sub” el? (Stiva este una: va trebui să înghesuim pe ea mai multe cadre potenţial active simultan.)

Soluţia este extrem de ingenioasă şi simplă: pentru a crea două cadre la distanţă mare pe stivă, între ele trebuie să se afle un cadru (sau mai multe) foarte mari.

Dacă ne uităm puţin mai sus, vom vedea că un cadru conţine *toate* variabilele locale ale unei funcţii; deci o funcţie de genul:

```
void f(void) { char c[100000]; }
```

va avea un cadru de 100000 de octeţi cel puţin!

Avem deci la îndemînă următoarele operaţii pe stivă:

1. apelul de funcţie, care crează un cadru;
2. terminarea unei funcţii, care distruge cadrul curent;
3. `longjmp`, care face un cadru indicat de pe stivă curent;
4. `setjmp` care salvează descrierea unui cadru.

Le putem combina pentru a obţine o serie de cadre, pe care să le salvăm, plasate la mari distanţe unele de altele. Codul arată cam aşa:

```
#include <setjmp.h>

#define NR_THREADS 2
#define SPATIU_STIVA 1024

jmp_buf contexte[NR_THREADS];
jmp_buf main_context;

void salveaza(int);   /* declaratie forward */

void aloca_stiva(int thread_no)
    /* aloca spatiu pe stiva pentru thread-urile thread_no -> NR_THREADS */
{
    char space[SPATIU_STIVA];     /* alocare esentiala a stivei */

    if (thread_no >= NR_THREADS) /* am terminat? */
        longjmp(main_context, 1);
    salveaza(thread_no);
}

void salveaza(int thread_no)
    /* salveaza contexte pentru thread_no -> NR_THREADS */
{
    if (setjmp(contexte[thread_no])) {
        /* vom adauga cod aici */
    }
    else
        aloca_stiva(thread_no+1);
}
```

```
int main(void)
{
    if (!setjmp(main_context))
        aloca_stiva(0);
    return 0;
}
```

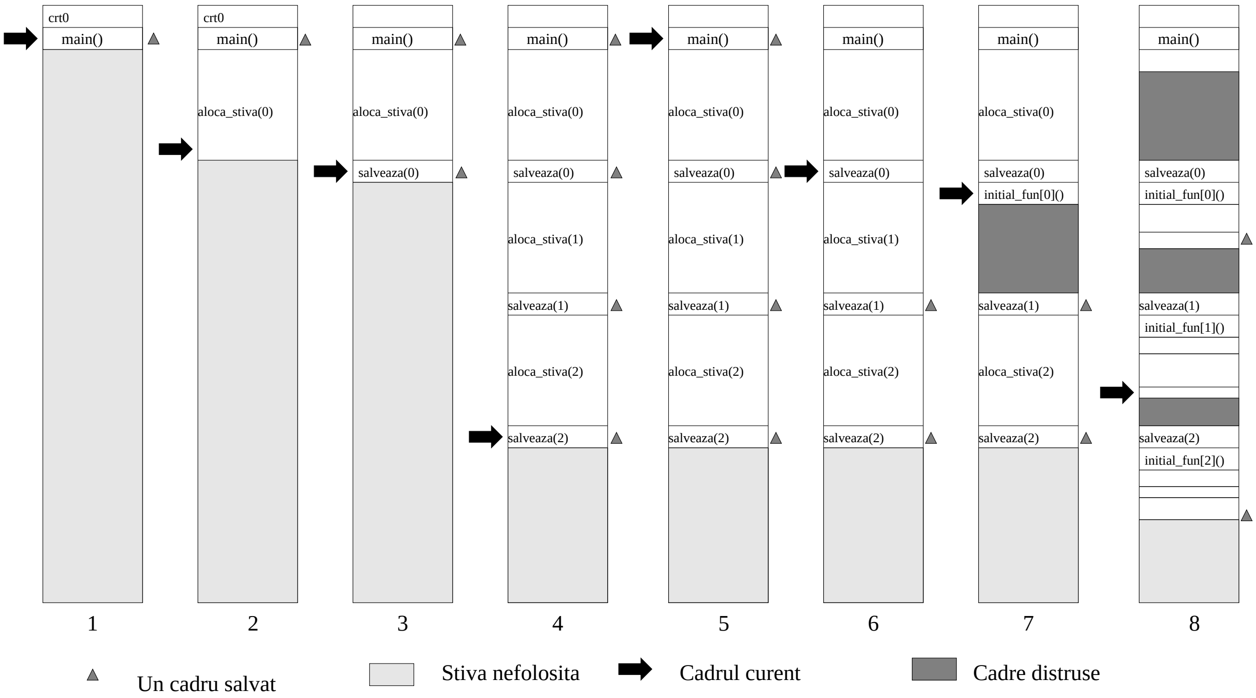

Figura 5: Cadre de stivă succesive

Desenele 1-5 din figura 5 arată evoluţia stivei pe măsură ce `aloca_stiva()` şi `salveaza()` se execută. Desenul 5 arată starea după execuţia lui `longjmp(main_context, 1)`, din funcţia `aloca_stiva`. (Ne vom referi la desenele 6-8 mai tîrziu puţin.)

O descriere sumară a situaţiei: avem 3 cadre ale funcţiei `salveaza` descrise de vectorul `contexte`, şi cadrul curent este cel al funcţiei `main()`. Dacă am fost norocoşi, faptul că am făcut un `longjmp` la `main_context` nu a stricat cu nimic cadrele pe care le-am pus la păstrare pe stivă. (Norocoşi în sensul că implementarea lui `longjmp` funcţionează ne-distructiv, deşi standardul de C nu o obligă!)

Vom vedea că funcţia `aloca_stiva` construieşte nişte "cadre de sacrificiu", pe care le vom folosi pentru a chema alte funcţii (spaţiul ocupat de cadrele lui `aloca_stiva` va fi re-folosit).

7 Funcţiile iniţiale

Mai avem de făcut puţini paşi pentru a obţine mai multe thread-uri: trebuie ca fiecare din cadrele (salvate în vectorul `contexte`) ale lui `salveaza` să poată invoca o altă funcţie atunci cînd va fi pornit. Avem mai multe soluţii, dar cea mai simplă pare să construim un vector de pointeri spre funcţii: cîte unul pentru fiecare cadru salvat. Cînd vom face acel cadru curent, el va invoca funcţia respectivă:

```
/* modificari ale codului precedent */
typedef void (*functie_initiala)(void);

functie_initiala functie_start[NR_THREADS];

void salveaza(int thread_no)
    /* salveaza contexte pentru thread_no -> NR_THREADS */
{
    if (setjmp(contexte[thread_no])) {
        (functie_start[thread_no])();   /* invoca functia */
    }
    else
        aloca_stiva(thread_no+1);
}

int main(void)
{
    extern functie_initiala f, g;    /* functiile care vor fi
                                        executate de threaduri;
                                        undeva in alt fisier */
    if (!setjmp(main_context))
        aloca_stiva(0);
    functie_start[0] = f;
    functie_start[1] = g;
    longjmp(contexte[0], 1);         /* porneste threadul 0 */
    return 0;
}
```

Am văzut că starea stivei după executarea apelului lui `aloca_stiva` din `main()` este cea din figura 5, poza 5. După `longjmp`-ul din main situaţia devine cea din poza 6: cadrul curent este cel care a fost creat de `salveaza(0)` şi salvat în `contexte[0]`. Acum `salveaza()` va chema funcţia din căsuţa 0 a vectorului `functie_start`; asta va transforma starea stivei ca în poza 7.

Primul thread a pornit! Funcţia `f`, care este de fapt cea invocată, poate face acum orice.

Mai trebuie să concepem un mecanism prin care un thread se întrerupe şi dă drumul altuia. Asta e deja o sarcină relativ simplă: va implica două acţiuni:

- salvarea stării thread-ului curent (pentru a putea fi continuat mai tîrziu) se face cu `setjmp`;
- transformarea stării celuilalt în starea curentă folosind `longjmp`.

8 Comutarea

Pentru a fi civilizaţi vom împacheta operaţia de "comutare" a thread-ului într-o funcţie. Vom introduce şi o variabilă globală care ştie cine este thread-ul curent; se va dovedi foarte utilă:

```
/* modificari la codul anterior */

int thread_curent;

void sari_la(int th)
{
    if (!setjmp(contexte[thread_curent])) {
        thread_curent = th;
        longjmp(contexte[thread_curent], 1);
    }
}
```

Deja puteţi testa pachetul de thread-uri scriind ceva de genul pentru funcţiile iniţiale:

```
void f(void)
{
   int x=0;

   while (1) {
        x++;
        printf("Functia f, thread %d, x=%d\n", thread_curent, x);
        sari_la(1-thread_curent);
   }
}

void g(void)
{
   int x=0;

   while (1) {
        x++;
        printf("Functia g, thread %d, x=%d\n", thread_curent, x);
        sari_la(1-thread_curent);
   }
}

int thread_curent = 0;  /* incepem cu thread-ul 0 */
```

Faptul că în acest program nu se cheamă pur şi simplu `f()` şi `g()` alternativ, ci se *continuă* execuţia lor se vede din faptul că valorile pe care le folosesc pentru `x` cresc.

Într-un scenariu mai complicat, cu mai multe funcţii iniţiale mai sofisticate, situaţia stivei la un moment dat ar putea arăta ca în figura 5, poza 8: practic funcţiile iniţiale au chemat alte funcţii la rîndul lor (cum `f()` cheamă `printf()`), care la un moment dat şi-au întrerupt execuţia. Cadrele lor sunt salvate (în figură asta este valabil pentru thread-urile 0 şi 2). Unul dintre thread-uri tocmai se execută (în figură thread-ul nr. 1), şi cadrul lui este cel curent.

De aici încolo nu mai avem mare lucru de făcut; ar fi elegant să concepem un set de funcţii care construieşte thread-uri noi în mod dinamic (să nu trebuiască să punem cu mîna pointerii la început), funcţii care să omoare thread-uri, şi poate o funcţie care să aleagă cine să se execute.

Pentru a ţine mai uşor socoteala despre starea fiecărui thread ar fi bine să strîngem toate informaţiile care ţin de el într-o singură structură (care va include cadrul şi funcţia iniţială); asta va necesita o "periere" a codului anterior, dar mai bine s-o facem acum, cît încă nu e prea mare. Iată declaraţiile funcţiilor pe care ni le propunem:

```
/* threads.h */
#ifndef THREADSH
#define THREADSH

#include <setjmp.h>

#define PUBLIC
#define PRIVAT static
/* cuvintul cheie 'static' este folosit in doua sensuri in C
 * a) pentru a indica variabile locale ale unei functii care
 *    nu se aloca pe stiva ci in segmentul de date
 * b) pentru a indica faptul ca o definitie dintr-un fisier
 *    nu este vizibila in exterior
 * Definim macro-ul PRIVAT pentru aceasta a doua semnificatie:
 * va face programul mai usor de citit
 */

#define MAX_THREADS 10          /* numarul maxim */
#define SPATIU_STIVA (32 * 1024)/* in octeti */

typedef void (*functie_initiala)(void);

struct thread {
    jmp_buf context;
    jmp_buf context_original;   /* contextul din salveaza() */
    int stare;
#define THREAD_GOL       0
#define THREAD_GATA      1      /* gata de executie */
#define THREAD_MORT      2
#define THREAD_ASTEAPTA  3      /* nefolosit, deocamdata */
    functie_initiala functie_start;
};

void initializeaza(void);
```

```
int creaza_thread(functie_initiala);
/* creaza un thread; intoarce un numar (-1 = eroare) */

void omoara_thread(int);

void porneste(int th);
/* porneste executia incepind cu acest thread */

void sari_la(int th);

void alege(void);
/* alege un thread pentru executie si sari la el */

#endif   /* THREADSH */
```

Funcţia `alege()` este ceea ce în engleză se cheamă `scheduler()`: partea care decide cine cînd se execută. Vom folosi cea mai simplă implementare pentru ea, care în terminologia sistemelor de operare se cheamă "round-robin": o coadă circulară.

O implementare ar putea arăta cam aşa (am mai modificat şi funcţiile vechi pe ici-pe colo):

```
/* threads.c */

#include "threads.h"
#include <stdio.h>     /* pentru printf */
#include <assert.h>    /* pentru erori */

#define pt_orice_thread(t) \
  for ((t) = tabela_thread; (t) < tabela_thread + MAX_THREADS; (t)++)
#define NU_E_THREAD (tabela_thread + MAX_THREADS)

PRIVAT struct thread tabela_thread[MAX_THREADS];
PRIVAT int thread_curent;
PRIVAT jmp_buf context_main;           /* folosit la initializare */

PRIVAT void
panica(char * s, int d)
    /* eroare fatala */
{
    fprintf(stderr, "** panica: ");
    fprintf(stderr, s, d);
    fflush(stderr);
    exit(1);
}

PRIVAT void
```

```
aloca_stiva(int thread_no, int maxthreads)
    /* creaza spatiu pe stiva pentru thread_no pina la maxthreads */
{
    char space[SPATIU_STIVA];           /* spatiul de pe stiva */
    void salveaza(int, int);

    if (thread_no >= maxthreads)
        longjmp(context_main, 1);       /* le-am facut pe toate */
    salveaza(thread_no, maxthreads);
}

PRIVAT void
salveaza(int thread_no, int max)
{
    char c;

    if (setjmp(tabela_thread[thread_no].context)) {
        tabela_thread[thread_no].functie_start();

        /* daca se ajunge aici functia s-a terminat; omoara-l */
        omoara_thread(thread_no);
    }
    else
        aloca_stiva(thread_no+1, max); /*  nu se mai intoarce.. */
}

PRIVAT void
copiaza_context(jmp_buf s, jmp_buf d)
   /* muta date dintr-un jmp_buf intr-altul
      (se bazeaza pe faptul ca sunt vectori:
       a se vedea definitia din setjmp.h) */
{
    int i;
    char *sb, *db;

    sb = (char *) s;
    db = (char *) d;
    for (i = 0; i < sizeof(jmp_buf); i++)
        *db++ = *sb++;
}

PUBLIC void
initializeaza(void)
{
```

```
    struct thread* t;

    if (!setjmp(context_main))
        aloca_stiva(0, MAX_THREADS);

    /* pastram o copie a cadrelor initiale de acum, ca sa le punem
       la loc cind moare un thread */
    pt_orice_thread(t)
        copiaza_context(t->context, t->context_original);
}

PUBLIC int
creaza_thread(functie_initiala start)
   /* facem un thread; intoarcem tid */
{
    struct thread *t;

    /* cautam un loc gol in tabela */
    pt_orice_thread(t)
        if (t->stare == THREAD_GOL ||
            t->stare == THREAD_MORT) break;
    if (t == NU_E_THREAD) return -1;        /* n-am gasit */
    t->functie_start = start;
    t->stare = THREAD_GATA;
    return t - tabela_thread;
}

#define thread_corect(x) ((x) >= 0 && (x) < MAX_THREADS)
/* verifica daca un thread are un numar corect */

PUBLIC void
omoara_thread(int th)
{
    struct thread *t;

    if (!thread_corect(th)) {
        return;
    }
    t = tabela_thread + th;
    t->stare = THREAD_MORT;
    copiaza_context(t->context_original, t->context);
    /* contextul acestui thread devine cel original, al functiei
       salveaza(th); in acest fel putem construi un nou thread
       folosind stiva acestuia */
```

```
    alege();
}

PUBLIC void
sari_la(int th)
{
    struct thread *t, *c;

    if (! thread_corect(th)) return;
    t = tabela_thread + th;
    if (t->stare != THREAD_GATA) return;
    assert(thread_corect(thread_curent));
    c = tabela_thread + thread_curent;

    /* salvam starea curenta */
    if (!setjmp(c->context)) {
        thread_curent = th;
        longjmp(t->context, 1);
    }
    else {
        /* continuam chematorul (asta poate lipsi) */
        thread_curent = c - tabela_thread;
    }
}

PUBLIC void
alege(void)
{
    struct thread *t;
    static int ultimul = 0;
    int i;

    for (i=0; i < MAX_THREADS; i++) {
        t = tabela_thread + ultimul;
        ultimul = (ultimul+1) % MAX_THREADS;
        if (t->stare == THREAD_GATA) break;
    }
    if (i == MAX_THREADS)
        panica("Nu mai e nimeni gata\n", 0);
    else sari_la(t - tabela_thread);

    /* ne intoarcem la chemator */
}
```

```
PUBLIC void
porneste(int th)
{
    struct thread *t;

    if (! thread_corect(th)) return;
    t = tabela_thread + th;
    if (t->stare != THREAD_GATA) return;
    thread_curent = th;
    longjmp(t->context, 1);
}
```

9 Întreruperi

Deja avem un pachet de funcţii prin care putem simula execuţia independentă "paralelă" a mai multor programe. Singura neplăcere constă din faptul că programele trebuie să-şi paseze reciproc controlul, chemînd `sari_la()` sau `alege()`.

Am dori ca un mecanism extern să se ocupe de asta fără să trebuiască să ne batem capul prea tare. Vom schiţa în continuare o soluţie, dar trebuie să atragem atenţia asupra faptului că este practic inutilă.

Cea mai simplă metodă este să avem un "ceas" care să întrerupă din timp în timp execuţia programului curent şi să zică: "destul: este rîndul altcuiva". La nivel hardware asta se întîmplă pe orice calculator modern, prin *întreruperi*.

Pentru sistemul de operare Unix avem la dispoziţie toate mecanismele de care avem nevoie: semnale generate de nucleu. Problema cea mare cu folosirea semnalelor este că dacă survin în timp ce procesul execută un apel de sistem (cum ar fi scrierea într-un fişier), ele opresc apelul din execuţie. Din cauza asta codul care urmează este util numai pentru a ilustra cum se fac lucrurile în realitate în interiorul nucleului, şi nu şi pentru a scrie programe reale cu thread-uri care se comută singure. Pentru că programul nu este lipsit de posibile învăţăminte, iată cam cum ar arăta:

Fiecare proces are în Unix pentru fiecare semnal asociată o funcţie care "tratează semnalul" (signal handler). Apariţia unui semnal se soldează cu executarea funcţiei. Mai precis, cînd nucleul sistemului are de trimis un semnal pentru un proces, construieşte pe stivă cadrul funcţiei de tratament, pe care după aceea o execută. Funcţia asta se poate executa practic în orice moment, fără a fi chemată de program.

Cu funcţia de bibliotecă `signal()` se declară "handler"-e.

Unix ne mai pune la dispoziţie funcţia de bibliotecă `alarm(timp)`, care la `timp` secunde de la invocare va genera un semnal de tip `SIGALRM`.

O primă schiţă a tratamentului semnalelor ar fi: (indicăm adăugările faţă de funcţiile existente deja în codul precedent cu `/**/`)

```
#include <unistd.h>     /* pentru alarm() */
#include <signal.h>     /* pentru signal() */

#define CUANTA 1        /* timp in sec. intre doua intreruperi */
```

```
/* #definitia trebuie pusa in threads.h */

PRIVAT int tratam_intrerupere;

PRIVAT void
intrerupere(int ignor)
{
   if (tratam_intrerupere)
        panica("intrerupere in tratament de intrerupere", 0);
   tratam_intrerupere++;
   alarm(CUANTA);
   signal(SIGALRM, intrerupere);
   tratam_intrerupere--;
   alege();
}

PUBLIC void
porneste(int th)
{
    struct thread *t;

    if (! thread_corect(th)) return;
    t = tabela_thread + th;
    if (t->stare != THREAD_GATA) return;
    thread_curent = th;
/**/signal(SIGALRM, intrerupere);
/**/alarm(CUANTA);              /* intrerupere */
    longjmp(t->context, 1);
}
```

Aceste funcţii vor face ca periodic să fie invocat `alege()`. Din păcate soluţia nu este suficient de grijulie, pentru că există riscul ca întreruperea să vină atunci cînd facem ceva important (de exemplu lucrăm în funcţia `creaza_thread()` şi am făcut jumătate din schimbări în a pune la punct starea procesului cel nou). În general întreruperile nu trebuie să lase structurile de date ne-consistente.

Există o soluţie foarte elegantă pentru asta: funcţia `intrerupere()` va evita să cheme `alege()` dacă s-a ivit tocmai în mijlocul unei funcţii importante, şi va ruga chiar funcţia importantă să cheme `alege()` cînd se termină.

Ca să simplificăm povestea, vom face funcţiile `sari_la()` şi `alege()` private; oricum utilizatorul nu mai are nevoie de ele, pentru că sunt chemate automat.

```
/* threads.h -- adaugiri sau modificari */

#define CUANTA 2   /* sec */
```

```
#if 0
   /* functiile astea nu mai sunt publice */
void sari_la(int th);
void alege(void); /* alege un thread pentru executie si sari la el */
#endif
```

Modificările din `threads.c` sunt indicate cu /**/ (celelalte funcţii rămîn neschimbate):

```
/* threads.c -- adaugiri sau modificari */

#include <unistd.h>
#include <signal.h>

PRIVAT int tratam_intrerupere;  /* cind tratam intreruperi != 0 */
PRIVAT int mod_nucleu ;         /* in functii importante != 0 */
PRIVAT int realege;             /* intrerupere intirziata */

PRIVAT int
creaza_thread(functie_initiala start)
   /* facem un thread; intoarcem tid */
{
    struct thread *t;

    /* cautam un loc gol in tabela */
/**/mod_nucleu++;
    pt_orice_thread(t)
        if (t->stare == THREAD_GOL ||
            t->stare == THREAD_MORT) break;
    if (t == NU_E_THREAD) {
/**/    mod_nucleu--;
        return -1;          /* n-am gasit */
    }
    t->functie_start = start;
    t->stare = THREAD_GATA;
/**/if (realege) alege();
/**/mod_nucleu--;
    return t - tabela_thread;
}

PUBLIC void
omoara_thread(int th)
{
    struct thread *t;

/**/mod_nucleu++;
```

```
    if (!thread_corect(th)) {
/**/    mod_nucleu--;
        return;
    }
    t = tabela_thread + th;
    t->stare = THREAD_MORT;
    copiaza_context(t->context_original, t->context);
    alege();
/**/mod_nucleu--;
}

/* PRIVAT void sari_la(int th)... in loc de PUBLIC */

PRIVAT void
alege(void)
{
    struct thread *t;
    static int ultimul = 0;
    int i;

/**/realege=0;
    for (i=0; i < MAX_THREADS; i++) {
        t = tabela_thread + ultimul;
        ultimul = (ultimul+1) % MAX_THREADS;
        if (t->stare == THREAD_GATA) break;
    }
    if (i == MAX_THREADS)
        panica("Nu e nimeni gata!\n", 0);
    else sari_la(t - tabela_thread);

    /* ne intoarcem la chemator */
}

PRIVAT void
salveaza(int thread_no, int max)
{
    char c;

    if (setjmp(tabela_thread[thread_no].context)) {
/**/    mod_nucleu=0;
        tabela_thread[thread_no].functie_start();
        omoara_thread(thread_no);
    }
    else
```

```
        aloca_stiva(thread_no+1, max); /*  nu se mai intoarce.. */
}


PRIVAT void
intrerupere(int ignor)
{
    tratam_intrerupere++;
    alarm(CUANTA);
    signal(SIGALRM, intrerupere);
    if (tratam_intrerupere > 1) {
        tratam_intrerupere--;
        return;
    }
/**/if (mod_nucleu) {         /* aminam alegerea */
/**/    tratam_intrerupere--;
/**/    realege++;
/**/    return;
/**/}
/**/else {
/**/    tratam_intrerupere--;
/**/    mod_nucleu++;
        alege();
/**/    mod_nucleu--;
/**/}
}
```

Variabila `mod_nucleu` indică execuţia curentă a unei proceduri ne-intreruptibile. O întrerupere venită în acest caz nu va face `alege()`, dar îşi va indica prezenţa prin `realege++`. Funcţiile importante care pot fi chemate de utilizator `creaza_thread()` şi `omoara_thread()` vor verifica înainte de a termina dacă nu a venit o întrerupere între timp, şi eventual vor chema ele `alege()`.

10 Rezumat

Am văzut că programele se execută în nişte universuri închise, pe care dacă le păstrăm neschimbate, putem întrerupe rularea programelor fără nici un fel de consecinţe. Am văzut cum putem folosi acest lucru pentru a executa mai multe programe pe o singură maşină.

Stiva este un mecanism extrem de simplu, de care ne folosim pentru a implementa funcţii recursive. Un cadru de stivă este micro-universul în care se execută o funcţie.

Limbajul C ne oferă instrucţiuni prin care putem comuta între cadre diferite de pe o aceeaşi stivă: astfel obţinem mai multe thread-uri.

Prezenţa unui ceas extern ne ajută să comutăm automat între mai multe procese.