

Introducere in firele de executie POSIX

Tom Wagner
Don Towsley
Departamentul de stiinta calculatoarelor
Universitatea din Amherst, Massachusetts

19 Iulie 1995

Traducere de Bogdan Manolache

1. Introducere: Ce este un fir de executie si la ce foloseste?

Firele de executie sunt adesea numite *procese usoare* si datorita faptului ca acest termen este simplificat, acesta este un bun punct de pornire. Firele de executie sunt foarte asemanatoare cu procesele din UNIX cu toate ca acestea nu sunt procese propriu-zise. Pentru a intelege diferenta trebuie sa examinam relatia dintre procesele UNIX si firele de executie si sarcinile de pe Mach. In UNIX, un proces contine in acelasi timp un program executabil si o colectie de resurse, asa cum ar fi o tabela descriptor de fisier si spatiul de adrese. Pe Mach, o sarcina contine doar colectia de resurse; firele de executie controleaza toate activitatile legate de executie. O sarcina de pe Mach poate avea oricate fire de executie asociate si toate firele de executie trebuie sa fie asociate cu o sarcina. Toate firele de executie asociate cu o anumita sarcina impart resursele sarcinii. Astfel, un fir de executie este in esenta un contor, o stiva, si o multime de registri – toate celelalte structuri de date apartin sarcinii. Un proces din UNIX este modelat pe Mach ca o sarcina cu un singur fir de executie.

Din moment ce firele de executie sunt foarte mici ca marime in comparatie cu procesele, crearea de fire de executie ocupa foarte putine resurse ale procesorului. Deoarece procesele au nevoie de propria colectie de resurse, iar firele de executie impart resursele, acestea din urma sunt foarte economice din punctul de vedere al consumului de memorie. Firele de executie de pe Mach ofera programatorilor posibilitatea de a scrie aplicatii concurente care ruleaza transparent atat pe calculatoarele cu un singur procesor cat si pe cele cu mai multe procesoare, folosind din plin avantajul mai multor procesoare, atunci cand acestea exista. In plus, firele de executie pot avea o performanta mai mare pe un calculator cu un singur procesor atunci cand aplicatia realizeaza operatii care au sanse mari de a se bloca sau de a intarzia, asa cum sunt fisierele sau socket-urile de intrare/iesire.

In urmatoarele sectiuni vom discuta despre firele de executie standard POSIX si implementarile specifice pe DEC OSF/1 OS, V3.0. Firele de executie POSIX sunt denumite *threads* si sunt asemanatoare cu firele de executie non-POSIX, denumite *cthread*s.

2. Hello World

Acum ca am terminat cu formalitatile, sa incepem. Functia *pthread_create* creeaza un nou fir de executie. Functia are patru argumente, o variabila pentru firul de executie, un atribut al firului de executie, functia care va fi apelata de catre firul de executie atunci cand acesta incepe sa se execute, si un argument pentru functie. De exemplu:

```
pthread_t a_thread;
pthread_attr_t a_thread_attribute;
void thread_function(void *argument);
char *some_argument;
pthread_create( &a_thread, a_thread_attribute, (void *)
&thread_function, (void *) &some_argument);
```

Momentan, atributul unui fir de executie specifica numai marimea minima a stivei care va fi folosita. Pe viitor attributele firelor de executie ar putea deveni mai interesante, dar pentru moment, majoritatea aplicatiilor functioneaza si asa, trimitand *pthread_attr_default* ca parametru pentru atributul firului de executie. Spre deosebire de procesele din UNIX create cu functia *fork* care isi incep executia din acelasi loc cu parintele, firele de executie isi incep executia la functia specificata de parametrul *pthread_create*. Motivul este clar; daca firele de executie nu si-au inceput executia in alta parte nu am avea mai multe fire de executie care sa execute aceleasi instructiuni cu *aceleasi* resurse. Reexecutand acele procese fiecare au propria colectie de resurse iar firele de executie impart propria colectie de resurse.

Odata ce stim sa cream fire de executie, suntem gata sa realizam prima aplicatie. Sa proiectam o aplicatie cu mai multe fire de executie care afiseaza mesajul "Hello World" la iesirea *stdout*. In primul rand avem nevoie de doua variabile pentru firele de executie si de o functie care va fi apelata de noile fire de executie cand vor fi executate. Avem de asemenea nevoie de o metoda pentru a specifica faptul ca fiecare fir de executie ar trebui sa afiseze un mesaj diferit. O metoda este aceea de a imparti cuvintele in siruri de caractere diferite si de a trimite fiecarui fir de executie cate un sir ca parametru de "pornire". Priviti urmatoarea secventa de cod:

```
void print_message_function( void *ptr );
main()
{
    pthread_t thread1, thread2;
```

```

char *message1 = "Hello";
char *message2 = "World";
pthread_create( &thread1, pthread_attr_defaul t,
               (voi d*)&print_message_functi on, (voi d*)
message1);
pthread_create(&thread2, pthread_attr_defaul t,
               (voi d*)&print_message_functi on, (voi d*)
message2);
exit(0);
}

voi d print_message_functi on( voi d *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s ", message);
}

```

Observati prototipul functiei *print_message_functi on* si operatorii de transformare care preceda argumentele apelului *pthread_create*. Programul creeaza primul fir de executie apeland *pthread_create* cu primul argument "Hello"; al doilea fir de executie este creat avand argumentul "World". Atunci cand se executa primul fir acesta isi incepe executia de la *print_message_functi on* cu argumentul "Hello". Acesta afiseaza "Hello" si ajunge la sfarsitul functiei. Un fir de executie se termina cand paraseste functia initiala deci primul fir de executie se termina dupa afisarea mesajului "Hello". Atunci cand se executa al doilea fir de executie acesta afiseaza "World" si se termina si el. Cu toate ca programul arata acceptabil, exista doua mari probleme.

Prima si cea mai importanta este aceea ca firele se executa concurrent. Din acest motiv nu exista nici o garantie ca primul fir de executie ajunge la functia *printf* inaintea celui de-al doilea fir de executie. De aceea poate fi afisat "World Hello" in loc de "Hello World". Nu exista alt punct mai subtil. Observati apelul functiei *exit* realizat de firul parinte¹ din blocul principal. Daca firul parinte executa functia *exit* inaintea executarii functiei *printf* de catre oricare dintre firele copil, nu se va afisa nimic. Aceasta se intampla deoarece functia *exit* iese din proces (elibereaza sarcina) prin aceasta terminand toate firele de executie. Orice fir de executie, parinte sau copil, care apeleaza functia *exit* poate termina toate celelalte fire de executie impreuna cu procesul. Firele de executie care vor sa se termine explicit trebuie sa apeleze functia *pthread_exit*.

¹ In timp ce toate firele de executie sunt la fel, vom numi firul de executie care incepe executia *firul parinte* pentru a-l deosebi de firele de executie create mai tarziu, pe care le vom numi *fire de executie copil* sau *copii*.

Astfel, micul nostru program hello world are doua conditii urgente. Urgenta apelarii functiei *exit* si urgenta de a vedea care fir copil apeleaza primul functia *printf*. Sa rezolvam problema conditiilor urgente cu putin lipici si banda adeziva. Din moment ce vrem ca fiecare fiu copil sa se termine inaintea firului parinte, sa inseram o pauza la fiul parinte care va oferi firelor copil ocazia sa ajunga la functia *printf*. Pentru a ne asigura ca primul fir copil ajunge la functia *printf* inaintea celui de-al doilea, sa inseram o pauza inaintea apelului *pthread_create* care creaza al doilea fir de executie. Codul rezultat este urmatorul:

```
main()
{
    pthread_t thread1, thread2;
    char *message1 = "Hello";
    char *message2 = "World";
    pthread_create( &thread1, pthread_attr_default,
        (void *) &print_message_function, (void *)
        message1);
    sleep(10);
    pthread_create(&thread2, pthread_attr_default,
        (void *) &print_message_function, (void *)
        message2);
    sleep(10);
    exit(0);
}

void print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s", message);
    pthread_exit(0);
}
```

Indeplineste acest program cerintele noastre? Nu chiar. Nu este niciodata recomandat sa ne bazam pe pauzele de timp pentru a realiza sincronizarea. Deoarece firele de executie sunt strans cuplate este tentant sa adoptam o atitudine mai putin riguroasa in ceea ce priveste sincronizarea, dar tentatia trebuie evitata. Conditia urgenta de aici este asemanatoare cu situatia unei aplicatii distribuite si a unei resurse partajate. Resursa este iesirea standard iar elementele de calcul distribuite sunt cele trei fire de executie. Primul fir de executie trebuie sa foloseasca functia *printf* *stdout* inaintea celui de-al doilea fir de executie si ambele trebuie sa isi indeplineasca actiunile inainte ca firul parinte sa apeleze functia *exit*.

Dupa incercarea noastra de a realiza sincronizarea folosind pauza, am facut o alta greseala. Functia *sleep*, ca si functia *exit* are legatura cu procesele. Atunci cand un fir de executie apeleaza functia *sleep*, procesul doarme, toate firele de executie dorm atunci cand procesul doarme. Astfel avem aceeași situatie fara apelurile functiei *sleep* dar programul ruleaza cu douazeci de secunde mai mult. Functia care este potrivita atunci cand avem nevoie sa intarziem un fir de executie este *pthread_delay_np* (np vine de la *not process* = nu este proces). De exemplu, pentru a intarzia un fir de executie pentru doua secunde putem folosi urmatoarea secventa de cod:

```
struct timespec delay;
delay.tv_sec = 2;
delay.tv_nsec = 0;
pthread_delay_np( &delay );
```

Functii folosite in aceasta sectiune: *pthread_create()*,
pthread_exit(),
pthread_delay_np().

3. Sincronizarea firelor de executie

POSIX ofera doua primitive pentru sincronizarea firelor de executie, variabila mutex si variabila conditie. Mutex-urile sunt primitive simple de blocare si pot fi folosite pentru a controla accesul la o resursa partajata. Tineti cont ca in cazul firelor de executie, tot spatiul de adrese este partajat astfel incat totul poate fi considerat resursa partajata. Oricum, in majoritatea cazurilor, firele de executie lucreaza individual cu (conceptual) variabile locale private, acelea create in interiorul functiei apelate de catre *pthread_create* si de catre functiile succesive, si combina eforturile acestora prin variabile globale. Accesul la articolele scrise in mod comun trebuie sa fie controlat.

Sa cream o aplicatie care foloseste fire de citire/scriere in care un singur fir care citeste si un singur fir care scrie comunica folosind un buffer partajat iar accesul este controlat folosind variabila mutex.

```
void reader_function(void);
void writer_function(void);
char buffer;
int buffer_has_item = 0;
pthread_mutex_t mutex;
struct timespec delay;
```

```
main()
{
    pthread_t reader;
```

```

    delay.tv_sec = 2;
    delay.tv_nsec = 0;
    pthread_mutex_init(&mutex,
pthread_mutexattr_default);
    pthread_create( &reader, pthread_attr_default,
(void*)&reader_function, NULL);
        writer_function();
}

void writer_function(void)
{
    while(1)
    {
        pthread_mutex_lock( &mutex );
        if ( buffer_has_item == 0 )
        {
            buffer = make_new_item();
            buffer_has_item = 1;
        }
        pthread_mutex_unlock( &mutex );
        pthread_delay_np( &delay );
    }
}

void reader_function(void)
{
    while(1)
    {
        pthread_mutex_lock( &mutex );
        if ( buffer_has_item == 1 )
        {
            consume_item( buffer );
            buffer_has_item = 0;
        }
        pthread_mutex_unlock( &mutex );
        pthread_delay_np( &delay );
    }
}

```

In acest program simplu consideram ca bufferul poate memora doar un articol deci se afla mereu in una din cele doua stari, ori contine un articol ori nu. Firul de scriere intai fixeaza variabila mutex, tinand-o blocata pana cand nu mai este fixata, in caz ca este deja fixata, apoi verifica daca bufferul este gol. Daca bufferul este gol, creaza un nou element si seteaza flag-ul *buffer_has_item* in asa fel incat firul care citeste sa stie ca in buffer se afla un element. Apoi elibereaza variabila mutex si face o pauza de doua secunde pentru a da destul timp firului care citeste sa se ocupe de element. Aceasta pauza difera de pauzele

precedente prin aceea ca este menita sa imbunatateasca eficienta programului. Fara aceasta pauza, firul care scrie va elibera elementul iar la instantierea urmatoare va incerca sa reobtina blocajul cu intentia de a crea un nou element. Este foarte probabil ca firul care citeste sa nu fi avut ocazia sa se ocupe de obiect asa de rapid deci folosirea pauzei este o idee buna.

Firul care citeste are aceeasi pozitie. Obtine blocajul, verifica daca a fost creat un obiect, iar in caz afirmativ se ocupa de acel obiect. Elibereaza blocajul si face o scurta pauza oferind firului care scrie ocazia de a crea un nou obiect. In acest exemplu firul care citeste si firul care scrie ruleaza la infinit, generand si consumand obiectele. In orice caz, daca o variabila mutex nu mai este necesara intr-un program, aceasta ar trebui eliberata folosind functia *pthread_mutex_destroy(&mutex)*. Observati ca in functia care initializeaza variabila mutex, care este necesara, am folosit atributul *pthread_mutexattr_default* pentru variabila mutex. In OSF/1 variabila mutex nu foloseste la nimic, deci este incurajata folosirea unei variabile predefinite.

Folosirea corecta a variabilelor mutex garanteaza eliminarea conditiilor de terminare. In orice caz, primitiva mutex singura nu are mare putere deoarece are doar doua stari, blocat sau neblocat. Variabilele conditionale POSIX suplimenteaza variabilele mutex permitand firelor de executie sa blocheze si sa astepte un semnal de la alt fir. Atunci cand este primit un semnal, firul blocat este trezit si incerca sa obtina un blocaj pe respectiva variabila mutex. In acest fel pot fi combinate semnalele si variabilele mutex pentru a elimina problema blocarii exemplificata in programul cu firele de scriere/citire. Am proiectat o biblioteca cu semafoare intregi simple folosind variabila mutex pthread si variabilele conditionale si de acum inainte vom discuta sincronizarea in acest context. Codul folosit pentru semafoare poate fi gasit in sectiunea Appendix A iar informatii detaliate despre variabilele conditionale pot fi gasite in paginile man.

Functii folosite in aceasta sectiune: *pthread_mutex_init()*,
pthread_mutex_lock(),
pthread_mutex_unlock(),
pthread_mutex_destroy().

4. Coordonarea activitatilor cu ajutorul semafoarelor

Sa revedem programul nostru de citire/scriere folosind semafoarele. Vom inlocui primitiva mutex cu un semafor intreg care este mult mai eficient si care elimina problema blocarii circulare. Operatiile cu semafoare sunt: *semaphore_up*, *semaphore_down*, *semaphore_init*, *semaphore_destroy*, si *semaphore_decrement*. Functiile up si down sunt conforme semanticii traditionale a semafoarelor – operatia down blocheaza daca semaforul are o valoare mai mica sau egala cu zero, iar operatia up incrementeaza semaforul.

Funcția `init` trebuie apelată înaintea folosirii semaforului și toate semafoarele sunt inițializate cu valoarea unu. Funcția `destroy` eliberează semaforul dacă nu mai este folosit. Toate funcțiile acceptă un singur argument care este pointer la un obiect de tip semafor.

Funcția `decrement` nu decrementează valoarea semaforului și este o funcție care nu blochează. Ea permite firelor de execuție să decrementeze valoarea semaforului până la valori negative ca parte a unui proces de inițializare. Ne vom uita la un exemplu care folosește funcția `semaphore_decrement` după programul de citire/scriere.

```
void reader_function(void);
void writer_function(void);
char buffer;
Semaphore writers_turn;
Semaphore readers_turn;

main()
{
    pthread_t reader;
    semaphore_init( &readers_turn );
    semaphore_init( &writers_turn );
    /* scriitorul trebuie sa fie primul */
    semaphore_down( &readers_turn );
    pthread_create( &reader, pthread_attr_default,
        (void *)&reader_function, NULL);
    writer_function();
}

void writer_function(void)
{
    while(1)
    {
        semaphore_down( &writers_turn );
        buffer = make_new_item();
        semaphore_up( &readers_turn );
    }
}

void reader_function(void)
{
    while(1)
    {
        semaphore_down( &readers_turn );
        consume_item( buffer );
        semaphore_up( &writers_turn );
    }
}
```

```
}
```

Acest exemplu nu foloseste pe deplin puterea semaforului intreg general. Sa revedem programul hello world din Sectiunea 2 si sa corectam conditia urgenta folosind semafoare.

```
void print_message_function( void *ptr );
```

```
Semaphore child_counter;
```

```
Semaphore worlds_turn;
```

```
main()
```

```
{  
    pthread_t thread1, thread2;  
    char *message1 = "Hello";  
    char *message2 = "World";  
    semaphore_init( &child_counter );  
    semaphore_init( &worlds_turn );  
    semaphore_down( &worlds_turn ); //world este al doilea  
    semaphore_decrement( &child_counter ); /* valoarea  
    este 0 */  
    semaphore_decrement( &child_counter ); /* valoarea  
    este -1 */  
    /* child_counter trebuie incrementat de 2 ori pentru  
    ca un fir care este blocat pe el sa fie eliberat*/  
  
    pthread_create( &thread1, pthread_attr_default,  
    (void *) &print_message_function, (void *) message1);  
    semaphore_down( &worlds_turn );  
    pthread_create(&thread2, pthread_attr_default,  
    (void *) &print_message_function, (void *) message2);  
    semaphore_down( &child_counter );  
    /* nu este necesar distrugerea din moment ce oricum  
    iesim */  
  
    semaphore_destroy ( &child_counter );  
    semaphore_destroy ( &worlds_turn );  
    exit(0);  
}
```

```
void print_message_function( void *ptr )
```

```
{  
    char *message;  
    message = (char *) ptr;  
    printf("%s ", message);  
    fflush(stdout);  
    semaphore_up( &worlds_turn );  
}
```

```

    semaphore_up( &child_counter );
    pthread_exit(0);
}

```

Cititorii pot sa se convinga foarte usor de faptul ca nu exista conditii urgente in aceasta versiune a programului hello world si de faptul ca se vor afisa cuvintele in ordinea corespunzatoare. Variabila semaforului *child_counter* este folosita pentru a forta firul parinte sa blocheze pana cand ambele fire de executie copil au executat comenzile *printf* si *semaphore_up(&child_counter)*.

Functii folosite in aceasta sectiune:

```

    semaphore_init(), semaphore_up(),
    semaphore_down(), semaphore_destroy(), si
    semaphore_decrement().

```

5. Practici

Pentru a compila cu fire de executie trebuie inclus fisierul header pthread, *#include <pthread.h>* si trebuie facuta legatura cu biblioteca pthread. De exemplu, *cc hello_world.c -o hello_world -lpthreads*. Pentru a folosi biblioteca asociata semafoarelor trebuie deasemenea inclus si fisierul header pentru aceasta si trebuie realizata legatura la biblioteca sau la fisierul respectiv.

Firele DEC sunt bazate pe standardul firelor POSIX IV, si nu pe standardul firelor POSIX VIII. Functia *pthread_join* permite ca un fir de executie sa astepte ca altul sa termine. In timp ce acest lucru putea fi folosit in programul hello world pentru a determina cand firele de executie copil au terminat, in loc de operatiile de decrementare/incrementare cu semafoare, implementarea DEC a functiei *pthread_join* are un comportament pe care nu ne putem baza in cazul in care obiectul fir de executie specificat nu mai exista. De exemplu, in secventa de cod urmatoare, daca functia *some_thread* nu mai exista, atunci functia *pthread_join* poate produce o eroare in loc sa realizeze intoarcerea din functie.

```

pthread_t some_thread;
void *exit_status;
pthread_join( some_thread, &exit_status );

```

Alte erori bizare care pot apare din cauza functiilor din afara rutinei firului de executie. In timp ce aceste erori sunt putine la numar, unele biblioteci fac presupuneri "uni-proces". De exemplu, am intalnit probleme de intrerupere cu functiile de intrare/iesire *fread* si *fwrite* care pot fi atribuite doar conditiilor urgente. In legatura cu problema erorilor, cu toate ca nu am verificat valorile returnate de apelurile firelor de executie din exemplele noastre, valorile returnate

ar trebuie atent verificate. Aproape toate functiile care se refera la firele de executie vor returna ca valoare de eroare -1. De exemplu:

```
pthread_t some_thread;
if ( pthread_create( &some_thread, ... ) == -1 )
{
    perror("Thread creation error");
    exit(1);
}
```

Biblioteca pentru semafoare va afisa un mesaj si va iesi in caz de erori. Cateva functii utile care nu au fost prezentate in exemple sunt:

pthread_yield(); Informeaza organizatorul ca firul de executie vrea sa ofere partea lui; nu are nevoie de argumente.

pthread_t me;
me = pthread_self(); Permite unui fir de executie sa obtina propriul identificator.

pthread_t thread;
pthread_detach(thread); Informeaza biblioteca ca starea de terminare a firelor de executie nu este necesara apelurilor ulterioare ale functiei *pthread_join* rezultand intr-o mai buna performanta a firelor de executie.

Pentru mai multe informatii consultati paginile man, de exemplu *man -k pthread*.

Apendix A – Libraria de cod pentru semafoare

```
/*
 * Fi si er: semaphore.h
 */
#include <stdio.h>
#include <pthread.h>
#ifndef SEMAPHORES
#define SEMAPHORES

typedef struct Semaphore
{
    int v;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
}
Semaphore;
```

```
int semaphore_down (Semaphore * s);
int semaphore_up (Semaphore * s);
void semaphore_destroy (Semaphore * s);
void semaphore_init (Semaphore * s);
int semaphore_value (Semaphore * s);
int tw_thread_cond_signal (pthread_cond_t * c);
int tw_thread_cond_wait (pthread_cond_t * c,
pthread_mutex_t * m);
int tw_thread_mutex_unlock (pthread_mutex_t * m);
int tw_thread_mutex_lock (pthread_mutex_t * m);
void do_error (char *msg);
#endif
```

```
/*
 * Fisier: semaphore.c
 *
 */
#include "semaphore.h"

/*
 * functia trebuie apelata inaintea folosirii semafoarelor -
 * - functia se ocupa de setari si de initializare;
 * functia destroy ar trebui apelata atunci cand semaforul
 * nu mai este folosit.
 */

void semaphore_init (Semaphore * s)
{
    s->v = 1;
    if (pthread_mutex_init (&(s->mutex),
pthread_mutexattr_default) == -1)
        do_error ("Error setting up semaphore mutex");
    if (pthread_cond_init (&(s->cond),
pthread_condattr_default) == -1)
        do_error ("Error setting up semaphore condition
signal");
}

/*
 * functia ar trebui apelata atunci cand nu mai este nevoie
 * de semafoare. se ocupa de dealocarea si eliberarea
 * resurselor.
 */

void semaphore_destroy (Semaphore * s)
{
```

```

    if (pthread_mutex_destroy (&(s->mutex)) == -1)
do_error ("Error destroying semaphore mutex");
    if (pthread_cond_destroy (&(s->cond)) == -1)
do_error ("Error destroying semaphore condition
signal");
}

/*
* functia incrementeaza semaforul si semnaleaza blocarea
* oricarui fir de executie sau existenta unui fir care
* trebuie schimbat in semafor.
*/

int semaphore_up (Semaphore * s)
{
    int value_after_op;
    tw_pthread_mutex_lock (&(s->mutex));
    (s->v)++;
    value_after_op = s->v;
    tw_pthread_mutex_unlock (&(s->mutex));
    tw_pthread_cond_signal (&(s->cond));
    return (value_after_op);
}

/*
* functia decrementeaza semaforul si blocheaza daca
* semaforul este mai mic sau egal cu 0 pana cand un alt
* fir de executie semnaleaza o schimbare.
*/

int semaphore_down (Semaphore * s)
{
    int value_after_op;
    tw_pthread_mutex_lock (&(s->mutex));
    while (s->v <= 0)
    {
        tw_pthread_cond_wait (&(s->cond), &(s->mutex));
    }
    (s->v)--;
    value_after_op = s->v;
    tw_pthread_mutex_unlock (&(s->mutex));
    return (value_after_op);
}

/*
* functia nu blocheaza ci doar decrementeaza semaforul.
* nu ar trebui folosita in locul functiei down - este

```

```
* recomandata doar pentru programele unde mai multe fire de  
* executie trebuie sa fie pe semafor inainte ca un alt fir  
* sa coboare; functia permite programatorului sa seteze o  
* valoare negativa pentru semafor inainte de a-l folosi  
* pentru sincronizare  
*/
```

```
int semaphore_decrement (Semaphore * s)  
{  
    int value_after_op;  
    tw_pthread_mutex_lock (&(s->mutex));  
    s->v--;  
    value_after_op = s->v;  
    tw_pthread_mutex_unlock (&(s->mutex));  
    return (value_after_op);  
}
```

```
/*  
* functia returneaza valoarea semaforului atunci cand este  
* accesata sectiunea critica. valoarea nu este garantata  
* dupa ce functia deblocheaza sectiunea critica. functia  
* este oferita doar pentru debugging-ul ocazional, dar o  
* practica mai buna pentru programator este ca acesta sa  
* protejeze un semafor cu altul iar apoi sa-i verifice  
* valoarea. o alternativa este memorarea valorii returnate  
* de functiile semaphore_up sau semaphore_down.  
*/
```

```
int semaphore_value (Semaphore * s)  
{  
    /* nu este pentru sincronizare */  
    int value_after_op;  
    tw_pthread_mutex_lock (&(s->mutex));  
    value_after_op = s->v;  
    tw_pthread_mutex_unlock (&(s->mutex));  
    return (value_after_op);  
}
```

```
/* ----- */  
/* Functiile urmatoare inlocuiesc functiile standard din  
* biblioteca prin aceea ca termina executia la orice eroare  
* returnata de apelurile sistem. Ne usureaza treaba prin  
* faptul ca nu mai trebuie sa verificam fiecare apel al  
* functiilor dinainte.  
*/  
/* ----- */
```

```

int tw_pthread_mutex_unlock (pthread_mutex_t * m)
{
    int return_value;
    if ((return_value = pthread_mutex_unlock (m)) == -1)
        do_error ("pthread_mutex_unlock");
    return (return_value);
}

```

```

int tw_pthread_mutex_lock (pthread_mutex_t * m)
{
    int return_value;
    if ((return_value = pthread_mutex_lock (m)) == -1)
        do_error ("pthread_mutex_lock");
    return (return_value);
}

```

```

int tw_pthread_cond_wait (pthread_cond_t * c,
pthread_mutex_t * m)
{
    int return_value;
    if ((return_value = pthread_cond_wait (c, m)) == -1)
        do_error ("pthread_cond_wait");
    return (return_value);
}

```

```

int tw_pthread_cond_signal (pthread_cond_t * c)
{
    int return_value;
    if ((return_value = pthread_cond_signal (c)) == -1)
        do_error ("pthread_cond_signal");
    return (return_value);
}

```

```

/*
* functia afiseaza un mesaj iar apoi termina executia
*/

```

```

void
do_error (char *msg)
{
    perror (msg);
    exit (1);
}

```