Laborator 08 - Thread-uri Linux

Materiale ajutătoare

- lab08-slides.pdf [http://elf.cs.pub.ro/so/res/laboratoare/lab08-slides.pdf]
- lab08-refcard.pdf [http://elf.cs.pub.ro/so/res/laboratoare/lab08-refcard.pdf]

Nice to read

- TLPI - Chapter 29, Threads: Introduction
- TLPI - Chapter 30, Threads: Thread Synchronization
- TLPI - Chapter 31, Threads: Thread Safety and Per-Thread Storage

Link-uri către secțiuni utile

Linux

- Fire de execuție - creare, așteptare, terminare
- Sincronizarea firelor de execuție
- Thread Specific Data (TSD)
- Mutex-uri
- Futex-uri
- Semafoare
- Variabile condiție
- Bariere

Prezentare teoretică

În laboratoarele anterioare a fost prezentat conceptul de **proces**, acesta fiind unitatea elementară de alocare a resurselor utilizatorilor. În cadrul acestui laborator este prezentat conceptul de **fir de execuție** (sau **thread**), acesta fiind unitatea elementară de planificare într-un sistem. Ca și procesele, firele de execuție reprezintă un mecanism prin care un calculator poate sa˘ ruleze mai multe task-uri simultan.

Un fir de execuție există în cadrul unui proces, și reprezintă o unitate de execuție mai fină decât acesta. În momentul în care un proces este creat, în cadrul lui există un singur fir de execuție, care execută programul secvențial. Acest fir poate la rândul lui sa˘ creeze alte fire de execuție; aceste fire vor rula porțiuni ale binarului asociat cu procesul curent, posibil aceleași cu firul inițial (care le-a creat).

Diferențe dintre fire de execuție și procese

- procesele nu partajează resurse între ele (decât dacă programatorul folosește un mecanism special pentru asta - shared memory spre exemplu), pe când firele de execuție partajează în mod implicit majoritatea resurselor unui proces. Modificarea unei astfel de resurse dintr-un fir este vizibilă instantaneu și din celelalte fire:
  - segmentele de memorie precum `.heap`, `.data` și `.bss` (deci și variabilele stocate în ele)

    - descriptorii de fișiere (așadar, închiderea unui fișier este vizibilă imediat pentru toate firele de execuție), indiferent de tipul fișierului:
        - socketi
        - fișiere normale
        - pipe-uri
        - fișiere ce reprezintă dispozitive hardware (de ex. `/dev/sda1`).
- fiecare fir are un context de execuție propriu, format din:
    - stivă
    - set de registre (deci și un contor de program - registrul `(E)IP`)

Procesele sunt folosite de SO pentru a grupa și aloca resurse, iar firele de execuție pentru a planifica execuția de cod care accesează (în mod partajat) aceste resurse.

Avantajele firelor de execuție

Deoarece toate firele de execuție ale unui proces folosesc spațiul de adrese al procesului de care aparțin, folosirea lor are o serie de avantaje:

- crearea/distrugerea unui fir de execuție durează mai puțin decât crearea/distrugerea unui proces
- durata context switch-ului între firele de execuție aceluiași proces este foarte mică, întrucât nu e necesar să se "comute" și spațiul de adrese (pentru mai multe informații, căutați „TLB flush")
- comunicarea între firele de execuție are un overhead mai mic (realizată prin modificarea unor zone de memorie din spațiul comun de adrese)

Firele de execuție se pot dovedi utile în multe situații, de exemplu, pentru a îmbunătăți timpul de răspuns al aplicațiilor cu interfețe grafice (GUI), unde prelucrările CPU-intensive se fac de obicei într-un fir de execuție diferit de cel care afișează interfața.

De asemenea, ele simplifică structura unui program și conduc la utilizarea unui număr mai mic de resurse (pentru că nu mai este nevoie de diversele forme de IPC pentru a comunica).

Tipuri de fire de execuție

Din punctul de vedere al implementării, există 3 categorii de fire de execuție:

- Kernel Level Threads (KLT)
- User Level Threads (ULT)
- Fire de execuție hibride

Kernel Level Threads

Managementul și planificarea firelor de execuție sunt realizate în kernel; programele creează/distrug fire de execuție prin apeluri de sistem. Kernel-ul menține informații de context, atât pentru procese, cât și pentru firele de execuție din cadrul proceselor, iar planificarea execuției se face la nivel de fir.

Avantaje :

- dacă avem mai multe procesoare putem lansa în execuție simultană mai multe fire de execuție ale aceluiași proces;
- blocarea unui fir nu înseamnă blocarea întregului proces;
- putem scrie cod în kernel care să se bazeze pe fire de execuție.

Dezavantaje :

- comutarea contextului este efectuată de kernel (cu o viteză de comutare mai mică):
  - se trece dintr-un fir de execuție în kernel
  - kernelul întoarce controlul unui alt fir de execuție.

User Level Threads

Kernel-ul nu este conștient de existența firelor de execuție, iar managementul acestora este realizat de procesul în care ele există (implementarea managementului firelor de execuție este realizată de obicei în biblioteci). Schimbarea contextului nu necesită intervenția kernel-ului, iar algoritmul de planificare depinde de aplicație.

Avantaje :

- schimbarea de context nu implică kernelul ⇒ comutare rapidă
- planificarea poate fi aleasă de aplicație; aplicația poate folosi acea planificare care favorizează creșterea performanțelor
- firele de execuție pot rula pe orice SO, inclusiv pe SO-uri care nu suportă fire de execuție la nivel kernel (au nevoie doar de biblioteca care implementează firele de execuție la nivel utilizator).

Dezavantaje :

- kernel-ul nu știe de fire de execuție ⇒ dacă un fir de execuție face un apel blocant toate firele de execuție planificate de aplicație vor fi blocate. Acest lucru poate fi un impediment întrucât majoritatea apelurilor de sistem sunt blocante. O soluție este utilizarea unor variante non-blocante pentru apelurile de sistem.
- nu se pot utiliza la maximum resursele hardware: kernelul planifică firele de execuție de care știe, câte unul pe fiecare procesor. Kernelul nu este conștient de existența firelor de execuție user-level ⇒ el va vedea un singur fir de execuție ⇒ va planifica procesul respectiv pe maximum un procesor, chiar dacă aplicația ar avea mai multe fire de execuție planificabile în același timp.

Fire de execuție hibride

Aceste fire încearcă să combine avantajele firelor de execuție user-level cu cele ale firelor de execuție kernel-level. O modalitate de a face acest lucru este de a utiliza fire kernel-level pe care să fie multiplexate fire user-level. KLT sunt unitățile elementare care pot fi distribuite pe procesoare. De regulă, crearea firelor de execuție se face în user space și tot aici se face aproape toată planificarea și sincronizarea. Kernel-ul știe doar de KLT-urile pe care sunt multiplexate ULT, și doar pe acestea le planifică. Programatorul poate schimba eventual numărul de KLT alocate unui proces.

Suport POSIX

În ceea ce privește firele de execuție, POSIX nu specifică dacă acestea trebuie implementate în user-space sau kernel-space. Linux le implementează în kernel-space, dar nu diferențiază firele de execuție de procese decât prin faptul că firele de execuție partajează spațiul de adresă (atât firele de execuție, cât și procesele, sunt un caz particular de "task"). Pentru folosirea firelor de execuție în Linux trebuie să includem header-ul `pthread.h` (unde se găsesc declarațiile funcțiilor și ale tipurilor de date necesare) și să utilizăm biblioteca `libpthread`.

Crearea firelor de execuție

Un fir de execuție este creat folosind pthread_create [http://linux.die.net/man/3/pthread_create]:

```
int pthread_create(pthread_t *tid, const pthread_attr_t *tattr,
                   void*(*start_routine)(void *), void *arg);
```

Noul fir creat va avea identificatorul `tid` și va rula concurent cu firul de execuție din care a fost creat. Acesta va executa codul specificat de funcția `start_routine` căreia i se va pasa argumentul `arg`. Dacă funcția de executat are nevoie de mai mulți parametri, aceștia pot fi agregați într-o structură, în câmpul `arg` punându-se un pointer către acea structură.

Prin parametrul `tattr` se stabilesc atributele noului fir de execuție. Dacă transmitem valoarea `NULL` firul de execuție va fi creat cu atributele implicite.

Pentru a determina identificatorul firului de execuție curent se poate folosi funcția pthread_self [http://linux.die.net/man/3/pthread_self]:

```
pthread_t pthread_self(void);
```

Așteptarea firelor de execuție

Firele de execuție se așteaptă folosind funcția pthread_join [http://linux.die.net/man/3/pthread_join]:

```
int pthread_join(pthread_t th, void **thread_return);
```

Primul parametru specifică identificatorul firului de execuție așteptat, iar al doilea parametru specifică unde se va plasa valoarea întoarsă de funcția copil (printr-un pthread_exit [http://linux.die.net/man/3/pthread_exit] sau printr-un `return` din rutina utilizată la pthread_create [http://linux.die.net/man/3/pthread_create]).

Firele de execuție se împart în două categorii: *unificabile* și *detașabile*.

- *unificabile* :
  - permit unificarea cu alte fire de execuție care apelează pthread_join [http://linux.die.net/man/3/pthread_join].
  - resursele ocupate de fir nu sunt eliberate imediat după terminarea firului, ci sunt păstrate până când un alt fir de execuție va executa pthread_join [http://linux.die.net/man/3/pthread_join] (analog proceselor *zombie*)
  - implicit firele de execuție sunt unificabile
- *detașabile*
  - un fir de execuție este detașabil dacă :
    - a fost creat detașabil.
    - i s-a schimbat acest atribut în timpul execuției prin apelul pthread_detach [http://linux.die.net/man/3/pthread_detach].
  - nu se poate executa un pthread_join [http://linux.die.net/man/3/pthread_join] pe ele
  - vor elibera resursele imediat ce se vor termina (analog cu ignorarea semnalului SIGCHLD în părinte la încheierea execuției proceselor copil)

Terminarea firelor de execuție

Un fir de execuție își încheie execuția:

- la un apel al funcției pthread_exit [http://linux.die.net/man/3/pthread_exit]:

  ```
  void pthread_exit(void *retval);
  ```

- în mod automat, la sfârșitul codului firului de execuție.

Prin parametrul `retval` se comunică părintelui un mesaj despre modul de terminare al copilului. Această valoare va fi preluată de funcția pthread_join [http://linux.die.net/man/3/pthread_join].

Metodele ca un fir de execuție să termine un alt fir sunt:

- stabilirea unui protocol de terminare (spre exemplu, firul **master** setează o variabilă globală, pe care firul **slave** o verifică periodic).
- mecanismul de "**thread cancellation**", pus la dispozitie de `libpthread`. Totuși, această metodă nu este recomandată, pentru că este greoaie, și pune probleme foarte delicate la cleanup. Pentru mai multe detalii: Terminarea thread-urilor

Thread Specific Data (TSD)

Uneori este util ca o variabilă să fie specifică unui fir de execuție (invizibilă pentru celelalte fire). Linux permite memorarea de perechi (cheie, valoare) într-o zonă special desemnată din stiva fiecărui fir de execuție al procesului curent. Cheia are același rol pe care îl are numele unei variabile: desemnează locația de memorie la care se află valoarea.

Fiecare fir de execuție va avea propria copie a unei "variabile" corespunzătoare unei chei `k`, pe care o poate modifica, fără ca acest lucru să fie observat de celelalte fire, sau să necesite sincronizare. De aceea, TSD este folosită uneori pentru a optimiza operațiile care necesită multă sincronizare între fire de execuție: fiecare fir calculează informația specifică, și există un singur pas de sincronizare la sfârșit, necesar pentru reunirea rezultatelor tuturor firelor de execuție.

Cheile sunt de tipul `pthread_key_t`, iar valorile asociate cu ele, de tipul generic `void *` (pointeri către locația de pe stivă unde este memorată variabila respectivă). Descriem în continuare operațiile disponibile cu variabilele din TSD:

Crearea și ștergerea unei variabile

O variabilă se creează folosind pthread_key_create [http://linux.die.net/man/3/pthread_key_create]:

```
int pthread_key_create(pthread_key_t *key, void (*destr_function) (void *));
```

Al doilea parametru reprezintă o funcție de cleanup. Acesta poate avea una din valorile:

- `NULL` și este ignorat
- pointer către o funcție de cleanup care se execută la terminarea firului de execuție

Pentru ștergerea unei variabile se apelează pthread_key_delete [http://linux.die.net/man/3/pthread_key_delete]:

```
int pthread_key_delete(pthread_key_t key);
```

Funcția nu apelează funcția de cleanup asociată variabilei.

Modificarea și citirea unei variabile

După crearea cheii, fiecare fir de execuție poate modifica propria copie a variabilei asociate folosind funcția pthread_setspecific [http://linux.die.net/man/3/pthread_setspecific]:

```
int pthread_setspecific(pthread_key_t key, const void *pointer);
```

Pentru a determina valoarea unei variabile de tip TSD se folosește funcția pthread_getspecific [http://linux.die.net/man/3/pthread_getspecific]:

```
void* pthread_getspecific(pthread_key_t key);
```

Funcții pentru cleanup

Funcțiile de cleanup asociate TSD-urilor pot fi foarte utile pentru a asigura faptul că resursele sunt eliberate atunci când un fir se termină singur sau este terminat de către un alt fir. Uneori poate fi util să se poată specifica astfel de funcții fără a crea neapărat un TSD. Pentru acest scop există funcțiile de cleanup.

O astfel de funcție de cleanup este o funcție care este apelată când un fir de execuție se termină. Ea primește un singur parametru de tipul `void *` care este specificat la înregistrarea funcției.

O funcție de cleanup este folosită pentru a elibera o resursă numai în cazul în care un fir de execuție apelează pthread_exit [http://linux.die.net/man/3/pthread_exit] sau este terminat de un alt fir folosind pthread_cancel [http://linux.die.net/man/3/pthread_cancel]. În circumstanțe normale, atunci când un fir nu se termină în mod forțat, resursa trebuie eliberată explicit, iar funcția de cleanup nu trebuie să fie apelată.

Pentru a înregistra o astfel de funcție de cleanup se folosește :

```
void pthread_cleanup_push(void (*routine) (void *), void *arg);
```

Aceasta funcție primește ca parametri un pointer la funcția care este înregistrată și valoarea argumentului care va fi transmis acesteia. Funcția `routine` va fi apelată cu argumentul `arg` atunci când firul este terminat forțat. Daca sunt înregistrate mai multe funcții de cleanup, ele vor fi apelate în ordine LIFO (cea mai recent instalată va fi prima apelată).

Pentru fiecare apel pthread_cleanup_push [http://linux.die.net/man/3/pthread_cleanup_push] trebuie să existe și apelul corespunzător pthread_cleanup_pop [http://linux.die.net/man/3/pthread_cleanup_pop] care deînregistrează o funcție de cleanup:

```
void pthread_cleanup_pop(int execute);
```

Această funcție va deînregistra cea mai recent instalată funcție de cleanup, și dacă parametrul `execute` este nenul o va și executa.

Atentie! Un apel pthread_cleanup_push [http://linux.die.net/man/3/pthread_cleanup_push] trebuie să aibă un apel corespunzător pthread_cleanup_pop [http://linux.die.net/man/3/pthread_cleanup_pop] în aceeași funcție și la același nivel de imbricare.

Un mic exemplu de folosire a funcțiilor de cleanup :

th_cleanup.c

```
void *alocare_buffer(int size)
{
        return malloc(size);
}

void dealocare_buffer(void *buffer)
{
        free(buffer);
}

/* functia apelata de un fir de execuție */

void functie()
{
        void *buffer = alocare_buffer(512);

        /* înregistrarea funcției de cleanup */
        pthread_cleanup_push(dealocare_buffer, buffer);

        /* aici au loc prelucrari, și se poate apela pthread_exit
        sau firul poate fi terminat de un alt fir */
```

```
        /* deînregistrarea functiei de cleanup și execuția ei
        (parametrul dat este nenul) */

        pthread_cleanup_pop(1);
}
```

Atributele unui fir de execuție

Atributele reprezintă o modalitate de specificare a unui comportament diferit de comportamentul implicit. Atunci când un fir de execuție este creat cu `pthread_create` se pot specifica atributele pentru respectivul fir de execuție. Atributele implicite sunt suficiente pentru marea majoritate a aplicațiilor. Cu ajutorul unui atribut se pot schimba:

- starea: unificabil sau detașabil
- politica de alocare a procesorului pentru firul de execuție respectiv (round robin, FIFO, sau system default)
- prioritatea (cele cu prioritate mai mare vor fi planificate, în medie, mai des)
- dimensiunea și adresa de start a stivei

Mai multe detalii puteți găsi în secțiunea suplimentară dedicată.

Cedarea procesorului

Un fir de execuție cedează dreptul de execuție unui alt fir, în urma unuia din următoarele evenimente:

- efectuează un apel blocant (cerere de I/O, sincronizare cu un alt fir de execuție) și kernel-ul decide că este *rentabil* să facă un context switch
- i-a expirat cuanta de timp alocată de către kernel
- cedează voluntar dreptul, folosind funcția sched_yield [http://linux.die.net/man/2/sched_yield]:

```
int sched_yield(void);
```

Dacă există alte procese interesate de procesor, unul dintre procese va acapara procesorul, iar dacă nu există niciun alt proces în așteptare pentru procesor, firul curent își continuă execuția.

Alte operații

Dacă dorim să fim siguri că un cod de inițializare se execută o singură dată putem folosi funcția:

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;
int pthread_once(pthread_once_t *once_control, void (*init_routine) (void));
```

Scopul funcției `pthread_once` este de a asigura că o bucată de cod (de obicei folosită pentru inițializări) se execută o singură dată. Argumentul `once_control` este un pointer la o variabilă inițializată cu `PTHREAD_ONCE_INIT`. Prima oară când această funcție este apelată ea va apela funcția `init_routine` și va schimba valoarea variabilei `once_control` pentru a ține minte că inițializarea a avut loc. Următoarele apeluri ale acestei funcții cu același `once_control` nu vor face nimic.

Funcția `pthread_once` întoarce 0 în caz de succes sau cod de eroare în caz de eșec.

Pentru a determina dacă doi identificatori se referă la același fir de execuție se poate folosi:

```
int pthread_equal(pthread_t thread1, pthread_t thread2);
```

Pentru aflarea/modificarea priorităților sunt disponibile următoarele apeluri:

```
int pthread_setschedparam(pthread_t target_thread, int policy, const struct sched_param *param);
int pthread_getschedparam(pthread_t target_thread, int *policy, struct sched_param *param);
```

Compilare

La compilare trebuie specificată și biblioteca `libpthread` (deci se va folosi argumentul `-lpthread`).

Nu legați un program single-threaded cu această bibliotecă. Anumite apeluri din bibliotecile standard pot avea implementări mai ineficiente sau mai greu de depanat când se utilizează această bibliotecă.

Exemplu

În continuare, este prezentat un exemplu simplu în care sunt create 2 fire de execuție, fiecare afișând un caracter de un anumit număr de ori pe ecran.

exemplu.c

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

/* parameter structure for every thread */
struct parameter {
        char character; /* printed character */
        int number;     /* how many times */
};

/* the function performed by every thread */
void* print_character(void *params)
{
        struct parameter *p = (struct parameter *) params;
        int i;

        for (i = 0; i < p->number; i++)
                printf("%c", p->character);
        printf("\n");

        return NULL;
}

int main()
{
        pthread_t fir1, fir2;
        struct parameter fir1_args, fir2_args;

        /* create one thread that will print 'x' 11 times */
        fir1_args.character = 'x';
        fir1_args.number = 11;
        if (pthread_create(&fir1, NULL, &print_character, &fir1_args)) {
                perror("pthread_create");
                exit(1);
        }

        /* create one thread that will print 'y' 13 times */
        fir2_args.character = 'y';
        fir2_args.number = 13;
        if (pthread_create(&fir2, NULL, &print_character, &fir2_args)) {
                perror("pthread_create");
                exit(1);
        }

        /* wait for completion */
        if (pthread_join(fir1, NULL))
                perror("pthread_join");
        if (pthread_join(fir2, NULL))
                perror("pthread_join");
```

```
        return 0;
}
```

Comanda utilizată pentru a compila acest exemplu va fi:

```
gcc -o exemplu exemplu.c -lpthread
```

Sincronizarea firelor de execuție

Pentru sincronizarea firelor de execuție, avem la dispoziție:

- mutex
- semafoare
- variabile de condiție
- bariere

Mutex

Mutex-urile (mutual exclusion locks) sunt obiecte de sincronizare utilizate pentru a asigura **accesul exclusiv** într-o secțiune de cod în care se utilizează **date partajate** între două sau mai multe fire de execuție. Un mutex are două stări posibile: **ocupat** și **liber**. Un mutex poate fi ocupat de un **singur fir** de execuție la un moment dat. Atunci când un mutex este ocupat de un fir de execuție, el nu mai poate fi ocupat de niciun alt fir. În acest caz, o cerere de ocupare venită din partea unui alt fir, în general, va **bloca** firul până în momentul în care mutex-ul devine liber.

Inițializarea/distrugerea unui mutex

Un mutex poate fi inițializat/distrus în mai multe moduri:

- folosind o **macrodefiniție**

```
// inițializare statică a unui mutex, cu atribute implicite
// NB: mutex-ul nu este eliberat, durata de viață a mutex-ului
//     este durata de viață a programului.
pthread_mutex_t mutex_static = PTHREAD_MUTEX_INITIALIZER;
```

- inițializare cu **atribute implicite** (pthread_mutex_init [http://linux.die.net/man/3/pthread_mutex_init], pthread_mutex_destroy [http://linux.die.net/man/3/pthread_mutex_destroy])

```
// semnăturile funcțiilor de inițializare și distrugere de mutex:
int pthread_mutex_init   (pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);


void initializare_mutex_cu_atribute_implicite() {
    pthread_mutex_t mutex_implicit;
    pthread_mutex_init(&mutex_implicit, NULL); // atrr = NULL -> atribute implicite


    // ... folosirea mutex-ului ...


    // eliberare mutex
```

```
    pthread_mutex_destroy(&mutex_implicit);
}
```

- inițializare cu **atribute explicite**

```
// NB: funcția pthread_mutexattr_settype și macro-ul PTHREAD_MUTEX_RECURSIVE
//     sunt disponibile doar dacă se definește _XOPEN_SOURCE la o valoare >= 500
//     **ÎNAINTE** de a include <pthread.h>.
//     Pentru mai multe detalii consultați feature_test_macros(7).

#define _XOPEN_SOURCE 500
#include <pthread.h>

void initializare_mutex_recursiv() {
    // definim atributele, le inițializăm și marcăm tipul ca fiind recursiv.
    pthread_mutexattr_t attr;
    pthread_mutexattr_init(&attr);
    pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);

    // definim un mutex recursiv, îl inițializăm cu atributele definite anterior
    pthread_mutex_t mutex_recursiv;
    pthread_mutex_init(&mutex_recursiv, &attr);

    // eliberăm resursele atributului după crearea mutex-ului
    pthread_mutexattr_destroy(&attr);

    // ... folosirea mutex-ului ...

    // eliberare mutex
    pthread_mutex_destroy(&mutex_recursiv);
}
```

Mutex-ul trebuie să fie **liber** pentru a putea fi **distrus**. În caz contrar, funcția va întoarce codul de eroare EBUSY. Întoarcerea valorii `0` semnifică succesul apelului.

Tipuri de mutex-uri

Folosind atributele de inițializare se pot crea mutex-uri cu proprietăți speciale:

- activarea **moștenirii de prioritate [http://en.wikipedia.org/wiki/Priority_inheritance]** (*priority inheritance*) pentru a preveni **inversiunea de prioritate [http://en.wikipedia.org/wiki/Priority_inversion]** (*priority inversion*). Există trei protocoale de moștenire a priorității:
  - `PTHREAD_PRIO_NONE` – **nu** se moștenește prioritatea când deținem mutex-ul creat cu acest atribut
  - `PTHREAD_PRIO_INHERIT` – dacă deținem un mutex creat cu acest atribut și dacă există fire de execuție blocate pe acel mutex, se moștenește prioritatea firului de execuție cu **cea mai mare prioritate**
  - `PTHREAD_PRIO_PROTECT` – dacă firul de execuție curent deține unul sau mai multe mutex-uri, acesta va executa la **maximul priorităților** specificate pentru toate mutex-urile deținute.

```
#define _XOPEN_SOURCE 500
#include <pthread.h>

int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *attr, int *protocol);
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol);
```

- modul de comportare la **preluări recursive** ale mutex-ului
  - PTHREAD_MUTEX_NORMAL – **nu** se fac verificări, preluarea recursivă duce la ***deadlock***
  - PTHREAD_MUTEX_ERRORCHECK – se fac verificări, preluarea recursivă duce la întoarcerea unei **erori**
  - PTHREAD_MUTEX_RECURSIVE – mutex-urile pot fi preluate recursiv, dar trebuie **eliberate de același număr de ori**.

```
#define _XOPEN_SOURCE 500
#include <pthread.h>

pthread_mutexattr_gettype(const pthread_mutexattr_t *attr, int *protocol);
pthread_mutexattr_settype(pthread_mutexattr_t *attr, int protocol);
```

Ocuparea/eliberarea unui mutex

Funcțiile de ocupare blocantă/eliberare a unui mutex (pthread_mutex_lock [http://linux.die.net/man/3/pthread_mutex_lock], pthread_mutex_unlock [http://linux.die.net/man/3/pthread_mutex_unlock]):

```
int pthread_mutex_lock  (pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Dacă mutex-ul este **liber** în momentul apelului, acesta va fi ocupat de firul apelant și funcția va întoarce imediat. Dacă mutex-ul este ocupat de un **alt fir**, apelul va bloca până la eliberarea mutex-ului. Dacă mutex-ul este deja ocupat de **firul curent** de execuție (lock recursiv), comportamentul funcției este dictat de tipul mutex-ului:

Tip mutex	Lock recursiv	Unlock
PTHREAD_MUTEX_NORMAL	deadlock	eliberează mutex-ul
PTHREAD_MUTEX_ERRORCHECK	returnează eroare	eliberează mutex-ul
PTHREAD_MUTEX_RECURSIVE	incrementează contorul de ocupări	decrementează contorul de ocupări (la zero eliberează mutex-ul)
PTHREAD_MUTEX_DEFAULT	deadlock	eliberează mutex-ul

Nu este garantată o ordine FIFO de ocupare a unui mutex. **Oricare din firele** aflate în așteptare la deblocarea unui mutex pot să-l acapareze.

Încercarea neblocantă de ocupare a unui mutex

Pentru a încerca ocuparea unui mutex **fără a aștepta** eliberarea acestuia în cazul în care este deja ocupat, se va apela funcția pthread_mutex_trylock [http://linux.die.net/man/3/pthread_mutex_trylock]:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Exemplu:

```
int rc = pthread_mutex_trylock(&mutex);
if (rc == 0) {
    /* successfully aquired the free mutex */
} else if (rc == EBUSY) {
    /* mutex was held by someone else
       instead of blocking we return EBUSY */
} else {
    /* some other error occured */
}
```

Exemplu de utilizare a mutex-urilor

Un exemplu de utilizare a unui mutex pentru a serializa accesul la variabila globală `global_counter`:

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

#define NUM_THREADS 5

/* global mutex */
pthread_mutex_t mutex;
int global_counter = 0;

void *thread_routine(void *arg)
{
    /* acquire global mutex */
    pthread_mutex_lock(&mutex);

    /* print and modify global_counter */
    printf("Thread %d says global_counter=%d\n", (int) arg, global_counter);
    global_counter++;

    /* release mutex - now other threads can modify global_counter */
    pthread_mutex_unlock(&mutex);

    return NULL;
}

int main(void)
{
    int i;
    pthread_t tids[NUM_THREADS];

    /* init mutex once, but use it in every thread */
    pthread_mutex_init(&mutex, NULL);

    /* all threads execute thread_routine
       as args to the thread send a thread id
       represented by a pointer to an integer */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tids[i], NULL, thread_routine, (void *) i);

    /* wait for all threads to finish */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tids[i], NULL);

    /* dispose mutex */
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

```
so@spook$ gcc -Wall mutex.c -lpthread
so@spook$ ./a.out
Thread 1 says global_counter=0
Thread 2 says global_counter=1
Thread 3 says global_counter=2
Thread 4 says global_counter=3
Thread 0 says global_counter=4
```

Futex-uri

Mutex-urile din firele de execuție POSIX sunt implementate cu ajutorul ***futex***-urilor, din considerente de performanță.

Optimizarea constă în testarea și setarea atomică a valorii mutex-ului (printr-o instrucțiune de tip *test-and-set-lock*) în user-space, **eliminându-se trap-ul în kernel** în cazul în care **nu** este necesară blocarea.

Numele de *futex* vine de la *Fast User-space muTEX*. Ideea de la care a plecat implementarea *futex*-urilor a fost aceea de a **optimiza** operația de ocupare a unui mutex în cazul în care acesta **nu este deja ocupat**. Dacă mutex-ul nu este ocupat, el va fi ocupat fără ca procesul care îl ocupă să se blocheze. În acest caz, nefiind necesară blocarea, nu este necesar ca procesul să intre în kernel-mode (pentru a intra într-o stare de așteptare). Optimizarea constă în testarea și setarea atomică a valorii mutex-ului (printr-o instrucțiune de tip *test-and-set-lock*) în user-space, **eliminându-se trap-ul în kernel** în cazul în care **nu** este necesară blocarea.

Futex-ul poate fi orice variabilă dintr-o zonă de memorie partajată între mai multe fire de execuție sau procese. Așadar, operațiile efective cu *futex*-urile se fac prin intermediul funcției `do_futex`, disponibilă prin includerea headerului `linux/futex.h`. Signatura ei arată astfel:

```
long do_futex(unsigned long uaddr, int op,
              int val, unsigned long timeout, unsigned long uaddr2, int val2);
```

În cazul în care este necesară blocarea, `do_futex` va face un apel de sistem - `sys_futex`. *Futex*-urile pot fi utile (și poate fi necesară utilizarea lor explicită) în cazul sincronizării proceselor, fiind alocate în variabile din zone de memorie partajată între procesele respective.

Semafor

Semafoarele sunt obiecte de sincronizare ce reprezintă o generalizare a mutex-urilor prin aceea că **salvează numărul de operații de eliberare** (incrementare) efectuate asupra lor. Practic, un semafor reprezintă un întreg care se incrementează/decrementează atomic. Valoarea unui semafor nu poate scădea sub 0. Dacă semaforul are valoarea 0, operația de decrementare se va bloca până când valoarea semaforului devine strict pozitivă. Mutex-urile pot fi privite, așadar, ca niște semafoare binare.

Semafoarele POSIX sunt de 2 tipuri:

- `cu nume` - folosite în general pentru sincronizare între procese distincte;
- `fără nume` - ce pot fi folosite pentru sincronizarea între firele de execuție ale aceluiași proces, sau între procese - cu condiția ca semaforul să fie într-o zonă de memorie partajată.

Diferențele dintre semafoarele cu nume față şi cele fără nume apar în funcțiile de creare și distrugere, celelalte funcții fiind identice.

- ambele tipuri de semafoare sunt reprezentate în cod prin tipul `sem_t`.
- semafoarele cu nume sunt identificate la nivel de sistem printr-un șir de forma ”/nume”.
- fișierele antet necesare sunt `<fcntl.h>`, `<sys/types.h>` și `<semaphore.h>`.

Operațiile care pot fi efectuate asupra semafoarelor POSIX sunt multiple:

Semafoare cu nume - Inițializare/deinițializare

```
/* use named semaphore to synchronize processes */
/* open */
sem_t* sem_open(const char *name, int oflag);
/* create if oflag has O_CREAT set */
sem_t* sem_open(const char *name, int oflag, mode_t mode, unsigned int value);

/* close named semaphore */
int sem_close(sem_t *sem);

/* delete a named semaphore from system */
int sem_unlink(const char *name);
```

Comportamentul este similar cu cel de la deschiderea fișierelor. Dacă flag-ul O_CREAT este prezent, trebuie folosită a doua formă a funcției, specificând permisiunile și valoarea inițială.

Singurele posibilități pentru al doilea argument sunt:

- 0 - se deschide semaforul dacă există
- O_CREAT - se creează semaforul dacă nu există; se deschide dacă există
- O_CREAT | O_EXCL - se creează semaforul **numai** dacă nu există; se întoarce eroare dacă există

Semafoare anonime - Inițializare/deinițializare

```
int sem_init(sem_t *sem, int pshared, unsigned int value);

/* close unnamed semaphore */
int sem_destroy(sem_t *sem);
```

Operații comune pe semafoare

```
/* increment/release semaphore (V) */
int sem_post(sem_t *sem);

/* decrement/acquire semaphore (P) */
int sem_wait(sem_t *sem);

/* non-blocking decrement/acquire */
int sem_trywait(sem_t *sem);

/* getting the semaphore count */
int sem_getvalue(sem_t *sem, int *pvalue);
```

Exemplu de utilizare semafor cu nume

```
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <semaphore.h>

#include "utils.h"

#define SEM_NAME        "/my_semaphore"

int main(void)
{
        sem_t *my_sem;
        int rc, pvalue;

        /* create semaphore with initial value of 1 */
        my_sem = sem_open(SEM_NAME, O_CREAT, 0644, 1);
        DIE(my_sem == SEM_FAILED, "sem_open failed");


        /* get the semaphore */
        sem_wait(my_sem);

        /* do important stuff protected by the semaphore */
        rc = sem_getvalue(my_sem, &pvalue);
        DIE(rc == -1, "sem_getvalue");
        printf("sem is %d\n", pvalue);

        /* release the lock */
        sem_post(my_sem);

        rc = sem_close(my_sem);
        DIE(rc == -1, "sem_close");
```

```
        rc = sem_unlink(SEM_NAME);
        DIE(rc == -1, "sem_unlink");

        return 0;
}
```

Semaforul va fi creat în `/dev/shm` și va avea numele `sem.my_semaphore`.

Variabile condiție

Variabilele condiție pun la dispoziție un sistem de notificare pentru fire de execuție, permițându-i unui fir să se blocheze în așteptarea unui semnal din partea unui alt fir. Folosirea corectă a variabilelor condiție presupune un protocol cooperativ între firele de execuție.

Mutex-urile și semafoarele permit blocarea **altor fire** de execuție. Variabilele de condiție se folosesc pentru a bloca **firul curent** până la îndeplinirea unei condiții.

Variabilele condiție sunt obiecte de sincronizare care-i permit unui fir de execuție să-și suspende execuția până când o condiție (predicat logic) **devine adevărată**. Când un fir de execuție determină că predicatul a devenit adevărat, va semnala variabila condiție, deblocând astfel unul sau toate firele de execuție blocate la acea variabilă condiție (în funcție de intenție).

O variabilă condiție trebuie întotdeauna folosită **împreună cu un mutex** pentru evitarea race-ului care se produce când un fir se pregătește să aștepte la variabila condiție în urma evaluării predicatului logic, iar alt fir semnalizează variabila condiție chiar înainte ca primul fir să se blocheze, pierzându-se astfel semnalul. Așadar, operațiile de semnalizare, testare a condiției logice și blocare la variabila condiție trebuie efectuate având **ocupat** mutex-ul asociat variabilei condiție. Condiția logică este testată sub protecția mutex-ului, iar dacă nu este îndeplinită, firul apelant se blochează la variabila condiție, eliberând atomic mutex-ul. În momentul deblocării, un fir de execuție va încerca să ocupe mutex-ul asociat variabilei condiție. De asemenea, testarea predicatului logic trebuie făcută într-o **buclă**, deoarece, dacă sunt eliberate mai multe fire deodată, doar unul va reuși să ocupe mutex-ul asociat condiției. Restul vor aștepta ca acesta să-l elibereze, însă este posibil ca firul care a ocupat mutex-ul să **schimbe** valoarea predicatului logic pe durata deținerii mutex-ului. Din acest motiv celelalte fire trebuie să testeze din nou predicatul pentru că, altfel, și-ar începe execuția presupunând predicatul adevărat, când el este, de fapt, fals.

Inițializarea/distrugerea unei variabile de condiție

Inițializarea unei variabile de condiție se face folosind macro-ul PTHREAD_COND_INITIALIZER sau funcția pthread_cond_init [http://linux.die.net/man/3/pthread_cond_init]. Distrugerea unei variabile de condiție se face prin funcția pthread_cond_destroy [http://linux.die.net/man/3/pthread_cond_destroy].

```
// inițializare statică a unei variabile de condiție cu atribute implicite
// NB: variabila de condiție nu este eliberată,
//     durata de viață a variabilei de condiție este durata de viață a programului.
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

// semnăturile funcțiilor de inițializare și eliberare de variabile de condiție:
int pthread_cond_init   (pthread_cond_t *cond, pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Ca și la mutex-uri:

- dacă parametrul `attr` este NULL, se folosesc atribute implicite
- trebuie să nu existe nici un fir de execuție în așteptare pe variabila de condiție atunci când aceasta este distrusă, altfel se întoarce EBUSY.

Blocarea la o variabilă condiție

Pentru a-și suspenda execuția și a aștepta la o variabilă condiție, un fir de execuție va apela funcția pthread_cond_wait [http://linux.die.net/man/3/pthread_cond_wait]:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

Firul de execuție apelant trebuie să fi **ocupat** deja mutex-ul asociat, în momentul apelului. Funcția `pthread_cond_wait` va **elibera** mutex-ul și se va **bloca**, așteptând ca variabila condiție să fie **semnalizată** de un alt fir de execuție. Cele două operații sunt efectuate **atomic**. În momentul în care variabila condiție este semnalizată, se va încerca ocuparea mutex-ului asociat, și după **ocuparea** acestuia, apelul funcției va întoarce. Observați că firul de execuție apelant poate fi suspendat, după deblocare, în așteptarea ocupării mutex-ului asociat, timp în care predicatul logic, adevărat în momentul deblocării firului, poate fi modificat de alte fire. De aceea, apelul `pthread_cond_wait` trebuie efectuat într-o buclă în care se testează valoarea de adevăr a predicatului logic asociat variabilei condiție, pentru a asigura o serializare corectă a firelor de execuție. Un alt argument pentru testarea în buclă a predicatului logic este acela că un apel `pthread_cond_wait` poate fi **întrerupt** de un semnal asincron (vezi laboratorul de semnale), înainte ca predicatul logic să devină adevărat. Dacă firele de execuție care așteptau la variabila condiție nu ar testa din nou predicatul logic, și-ar continua execuția presupunând greșit că acesta e adevărat.

Blocarea la o variabilă condiție cu timeout

Pentru a-și suspenda execuția și a aștepta la o variabilă condiție, nu mai târziu de un moment specificat de timp, un fir de execuție va apela pthread_cond_timedwait [http://linux.die.net/man/3/pthread_cond_timedwait]:

```
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
                           const struct timespec *abstime);
```

Funcția se comportă la fel ca `pthread_cond_wait`, cu excepția faptului că, dacă variabila condiție nu este semnalizată mai devreme de `abstime`, firul apelant este deblocat, și, după ocuparea mutex-ului asociat, funcția se întoarce cu eroarea `ETIMEDOUT`. Parametrul `abstime` este absolut și reprezintă numărul de secunde trecute de la 1 ianuarie 1970, ora 00:00.

Deblocarea unui singur fir blocat la o variabilă condiție

Pentru a debloca un singur fir de execuție blocat la o variabilă condiție se va semnaliza variabila condiție folosind pthread_cond_signal [http://linux.die.net/man/3/pthread_cond_signal]:

```
int pthread_cond_signal(pthread_cond_t *cond);
```

Dacă la variabila condiție nu așteaptă niciun fir de execuție, apelul funcției nu are efect și semnalizarea se va **pierde**. Dacă la variabila condiție așteaptă mai multe fire de execuție, va fi deblocat doar unul dintre acestea. Alegerea firului care va fi deblocat este făcută de planificatorul de fire de execuție. Nu se poate presupune că firele care așteaptă vor fi deblocate în ordinea în care și-au început așteptarea. Firul de execuție apelant trebuie să dețină **mutex-ul** asociat variabilei condiție în momentul apelului acestei funcții.

Exemplu:

```
pthread_mutex_t count_lock;
pthread_cond_t  count_nonzero;
unsigned        count;

void decrement_count() {
    pthread_mutex_lock(&count_lock);
    while (count == 0)
        pthread_cond_wait(&count_nonzero, &count_lock);
    count = count - 1;
    pthread_mutex_unlock(&count_lock);
}
```

```
void increment_count() {
    pthread_mutex_lock(&count_lock);
    count = count + 1;
    pthread_cond_signal(&count_nonzero);
    pthread_mutex_unlock(&count_lock);
}
```

Deblocarea tuturor firelor blocate la o variabilă condiție

Pentru a debloca toate firele de execuție blocate la o variabilă condiție, se semnalizează variabila condiție folosind pthread_cond_broadcast [http://linux.die.net/man/3/pthread_cond_broadcast]:

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Dacă la variabila condiție nu așteaptă niciun fir de execuție, apelul funcției nu are efect și semnalizarea se va **pierde**. Dacă la variabila condiție așteaptă fire de execuție, toate acestea vor fi deblocate, dar vor **concura** pentru ocuparea mutex-ului asociat variabilei condiție. Firul de execuție apelant trebuie să dețină mutex-ul asociat variabilei condiție în momentul apelului acestei funcții.

Exemplu de utilizare a variabilelor de condiție

În următorul program se utilizează o barieră pentru a sincroniza firele de execuție ale programului. Bariera este implementată cu ajutorului unei variabile de condiție.

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

#define NUM_THREADS 5

// implementarea unei bariere *non-reutilizabile* cu variabile de condiție
struct my_barrier_t {
    // mutex folosit pentru a serializa accesele la datele interne ale barierei
    pthread_mutex_t lock;

    // variabila de condiție pe care se așteptă sosirea tuturor firelor de execuție
    pthread_cond_t  cond;

    // număr de fire de execuție care trebuie să mai vină pentru a elibera bariera
    int nr_still_to_come;
};

struct my_barrier_t bar;

void my_barrier_init(struct my_barrier_t *bar, int nr_still_to_come) {
    pthread_mutex_init(&bar->lock, NULL);
    pthread_cond_init(&bar->cond, NULL);

    // câte fire de execuție sunt așteptate la barieră
    bar->nr_still_to_come = nr_still_to_come;
}

void my_barrier_destroy(struct my_barrier_t *bar) {
    pthread_cond_destroy(&bar->cond);
    pthread_mutex_destroy(&bar->lock);
}

void *thread_routine(void *arg) {
    int thd_id = (int) arg;

    // înainte de a lucra cu datele interne ale barierei trebuie să preluam mutex-ul
    pthread_mutex_lock(&bar.lock);

    printf("thd %d: before the barrier\n", thd_id);

    // suntem ultimul fir de execuție care a sosit la barieră?
    int is_last_to_arrive = (bar.nr_still_to_come == 1);
```

```
    // decrementăm numarul de fire de execuție așteptate la barieră
    bar.nr_still_to_come --;

    // cât timp mai sunt fire de execuție care nu au ajuns la barieră, așteptăm.
    while (bar.nr_still_to_come != 0)
        // mutex-ul se eliberează automat înainte de a incepe așteptarea
        pthread_cond_wait(&bar.cond, &bar.lock);

    // ultimul fir de execuție ajuns la barieră va semnaliza celelalte fire
    if (is_last_to_arrive) {
        printf("    let the flood in\n");
        pthread_cond_broadcast(&bar.cond);
    }

    printf("thd %d: after the barrier\n", thd_id);

    // la ieșirea din funcția de așteptare se preia automat mutex-ul, care trebuie eliberat
    pthread_mutex_unlock(&bar.lock);

    return NULL;
}

int main(void) {
    int i;
    pthread_t tids[NUM_THREADS];

    my_barrier_init(&bar, NUM_THREADS);

    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tids[i], NULL, thread_routine, (void *) i);

    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tids[i], NULL);

    my_barrier_destroy(&bar);

    return 0;
}
```

```
so@spook$ gcc -Wall cond_var.c -pthread
so@spook$ ./a.out
thd 0: before the barrier
thd 2: before the barrier
thd 3: before the barrier
thd 4: before the barrier
thd 1: before the barrier
    let the flood in
thd 1: after the barrier
thd 2: after the barrier
thd 3: after the barrier
thd 4: after the barrier
thd 0: after the barrier
```

Din execuția programului se observă:

- ordinea în care sunt planificate firele de execuție **nu** este neapărat cea a creării lor
- ordinea în care sunt trezite firele de execuție ce așteaptă la o variabilă de condiție **nu** este neapărat ordinea în care acestea au intrat în așteptare.

Barieră

Standardul POSIX definește și un set de funcții și structuri de date de lucru cu bariere. Aceste funcții sunt disponibile dacă se definește macro-ul _XOPEN_SOURCE la o valoare >= 600.

Inițializarea/distrugerea unei bariere

Bariera se va inițializa folosind pthread_barrier_init [http://linux.die.net/man/3/pthread_barrier_init] și se va distruge folosind pthread_barrier_destroy [http://linux.die.net/man/3/pthread_barrier_destroy].

```
// pentru a folosi funcțiile de lucru cu bariere e nevoie să se definească
// _XOPEN_SOURCE la o valoare >= 600. Pentru detalii consultați feature_test_macros(7).
#define _XOPEN_SOURCE 600
#include <pthread.h>

// attr    -> un set de atribute, poate fi NULL (se folosesc atribute implicite)
// count   -> numărul de fire de execuție care trebuie să ajungă
//            la barieră pentru ca aceasta să fie eliberată
int pthread_barrier_init(pthread_barrier_t *barrier,
                         const pthread_barrierattr_t *attr,
                         unsigned count);

// trebuie să nu existe fire de execuție în așteptare la barieră
// înainte de a apela funcția _destroy, altfel, se întoarce EBUSY
// și nu se distruge bariera.
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

Așteptarea la o barieră

Așteptarea la barieră se face prin apelul pthread_barrier_wait [http://linux.die.net/man/3/pthread_barrier_wait]:

```
#define _XOPEN_SOURCE 600
#include <pthread.h>
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

Dacă bariera a fost creată cu `count=N`, primele `N-1` fire de execuție care apelează `pthread_barrier_wait` se blochează. Când sosește **ultimul** (al N-lea), va debloca toate cele `N-1` fire de execuție. Funcția `pthread_barrier_wait` întoarce trei valori:

- `EINVAL` – în cazul în care bariera nu este inițializată (singura eroare definită)
- `PTHREAD_BARRIER_SERIAL_THREAD` – în caz de succes, un singur fir de execuție va întoarce valoarea aceasta – nu e specificat care este acel fir de execuție (nu e obligatoriu să fie ultimul ajuns la barieră)
- `0` – valoare întoarsă în caz de succes de celelalte `N-1` fire de execuție.

Exemplu de utilizare a barierei

Cu bariere POSIX, programul de mai sus poate fi simplificat:

```
#define _XOPEN_SOURCE 600
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS 5

pthread_barrier_t barrier;

void *thread_routine(void *arg) {
    int thd_id = (int) arg;
    int rc;

    printf("thd %d: before the barrier\n", thd_id);

    // toate firele de execuție așteaptă la barieră.
    rc = pthread_barrier_wait(&barrier);
    if (rc == PTHREAD_BARRIER_SERIAL_THREAD) {
        // un singur fir de execuție (posibil ultimul) va întoarce PTHREAD_BARRIER_SERIAL_THREAD
        // restul firelor de execuție întorc 0 în caz de succes.
        printf("   let the flood in\n");
```

```
    }

    printf("thd %d: after the barrier\n", thd_id);

    return NULL;
}

int main(void)
{
    int i;
    pthread_t tids[NUM_THREADS];

    // bariera este inițializată o singură dată și folosită de toate firele de execuție
    pthread_barrier_init(&barrier, NULL, NUM_THREADS);

    // firele de execuție vor executa codul funcției 'thread_routine'.
    // în locul unui pointer la date utile, se trimite în ultimul argument
    // un întreg - identificatorul firului de execuție
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tids[i], NULL, thread_routine, (void *) i);

    // așteptăm ca toate firele de execuție să se termine
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tids[i], NULL);

    // eliberăm resursele barierei
    pthread_barrier_destroy(&barrier);

    return 0;
}
```

```
so@spook$ gcc -Wall barrier.c -lpthread
so@spook$ ./a.out
thd 0: before the barrier
thd 2: before the barrier
thd 1: before the barrier
thd 3: before the barrier
thd 4: before the barrier
   let the flood in
thd 4: after the barrier
thd 2: after the barrier
thd 3: after the barrier
thd 0: after the barrier
thd 1: after the barrier
```

Exerciţii de laborator

Exercițiul 0 - Joc interactiv

- Detalii desfășurare joc [http://ocw.cs.pub.ro/courses/so/meta/notare#joc_interactiv].

Linux

În rezolvarea laboratorului folosiți arhiva de sarcini lab08-tasks.zip [http://elf.cs.pub.ro/so/res/laboratoare/lab08-tasks.zip]

Pentru a vă ajuta la implementarea exercițiilor din laborator, în directorul `utils` din arhivă există un fișier `utils.h` cu funcții utile.

Pentru a instala paginile de manual pentru 'pthreads'

```
sudo apt-get install manpages-posix manpages-posix-dev
```

Exercițiul 1 - Thread Stack

Intrați în directorul `1-th_stack` și inspectați sursa, apoi compilați și rulați programul. Urmăriți cu `pmap` sau folosind `procfs` cum se modifică spațiul de adresă al programului:

```
watch -d pmap $(pidof th_stack)
watch -d cat /proc/$(pidof th_stack)/maps
```

Zonele de memorie cu dimensiunea de 8MB (8192KB) care se creează după fiecare apel `pthread_create` reprezintă noile *stive* alocate de către biblioteca `libpthread` pentru fiecare thread în parte. Observați că, în plus, se mai mapează de fiecare dată o pagină (4KB) cu protecția `---p` (PROT_NONE, private - vizibil în `procfs`) care are rolul de "pagină de gardă".

Motivul pentru care nu se termină programul este prezența unui `while(1)` în funcția thread-urilor. Folosiți `Ctrl+C` pentru a termina programul.

Exercițiul 2 - Fire de execuție vs Procese

Intrați în directorul `2-th_vs_proc` și inspectați sursele. Ambele programe simulează un server care creează fire de execuție/procese. Compilați și rulați pe rând ambele programe.

În timp ce rulează, afișați, într-o altă consolă, câte fire de execuție/procese sunt create în ambele situații folosind comanda `ps -L -C <nume_program>`.

```
ps -L -C threads
ps -L -C processes
```

Verificați ce se întâmplă dacă la un moment dat un fir de execuție moare (sau un proces, în funcție de ce executabil testați). Testați utilizând funcția `do_bad_task` la fiecare al 4-lea fir de execuție/process.

Exercițiul 3 - Generator de numere pseudoaleatoare

Datorită faptului că mașina virtuală `spook` are un singur core virtual, exercițiul următor trebuie realizat pe mașina fizică pentru a permite mai multor thread-uri să ruleze în același moment de timp.

Intrați în directorul `3-safety` și inspectați sursa `malloc.c`. Funcțiile `thread_function` și `main` **NU** sunt thread-safe relativ la variabilele `global_storage` și `function_global_storage` (revedeți semnificația lui `thread safety` [http://en.wikipedia.org/wiki/Thread_safety]). Există o condiție de cursă [https://en.wikipedia.org/wiki/Race_condition] între cele două thread-uri create la incrementarea variabilei `function_global_storage`, declarată în funcția `thread_function`, și o altă condiție de cursă între toate thread-urile procesului la incrementarea variabilei globale `global_storage`.

Un utilitar foarte folositor este `helgrind`, care poate detecta automat aceste condiții de cursă. Îl putem folosi în cazul nostru așa:

```
valgrind --tool=helgrind ./mutex
```

Sincronizare acces variabile

Rezolvați comentariile `TODO1` din `malloc.c`. Pentru a rezolva aceste două condiții de cursă apelați functia `increase_numbers` intr-un mod thread safe cu ajutor API-ului pus la dispozitie de `critical.h`. De asemenea, sincronizați accesul în funcția `print_stats` (**de ce?**).

Pentru testare, folosiți executabilul `mutex`.

În fișierul `malloc.c` se creează `NUM_THREADS` thread-uri care alocă o matrice de `NUM_ROWS X NUM_COLUMNS` întregi. Sunt șanse mari ca thread-urile să execute apeluri `malloc` concurente.

După ce ați rezolvat `TODO1`, compilați și rulați de mai multe ori. Observăm că programul rulează cu succes. Pentru a face verificări suplimentare, rulăm din nou `helgrind`:

```
valgrind --tool=helgrind ./mutex
```

Observăm că nici `helgrind` nu raportează vreo eroare, lucru care conduce la faptul că funcția `malloc` ar fi thread-safe. (chiar daca acesta nu este protejat de API-ul pus la dispoziție).

Pentru a putea fi siguri trebuie să consultăm paginile de manual și codul sursă.

Este important de știut că anumite funcții sunt thread-safe iar altele nu. Găsiți o listă cu funcțiile care nu sunt thread-safe în pagina de manual pthreads(7) [http://man7.org/linux/man-pages/man7/pthreads.7.html], în secțiunea `Thread-safe functions`.

Funcția `malloc` din implementarea GLIBC **este thread-safe**, lucru indicat în pagina de manual malloc(3) [http://man7.org/linux/man-pages/man3/malloc.3.html#NOTES] (al treilea paragraf din secțiunea `NOTES`) și vizibil în codul sursă prin prezența câmpului `mutex` în structura malloc_state [https://sourceware.org/git/?p=glibc.git;a=blob;f=malloc/malloc.c;h=f361bad636167cf1680cb75b5098232c9232d771;hb=HEAD#l1672].

Spinlocks

`TODO 2`: Implementati un spinlock folosindu-va de operatii atomice. Operatiile atomice existente in standardul GCC le gasiti la __atomic functions [https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html]

In fisierul `critical.c` trebuie sa completati in dreptul comentarilor asociate `TODO 2`, avand la dispozitie hint-uri.

Pentru testare, folosiți executabilul `spin`.

Aici, la verificarea corectitudinii nu vă mai poate ajuta `helgrind` pentru că nu are implementat suportul nativ. Există adnotări cu care poate fi ajutat să nu dea false positives, dar pentru asta e nevoie sa compilați și să linkați o versiune modificată de helgrind. Detalii aici, secțiunea 7.5 [http://valgrind.org/docs/manual/hg-manual.html].

Exercițiul 4 - Parallel fgrep

Datorită faptului că mașina virtuală `spook` are un singur core virtual, exercițiul următor trebuie realizat pe mașina fizică pentru a permite mai multor thread-uri să ruleze în același moment de timp.

Implementați un program similar cu fgrep [http://linux.die.net/man/1/fgrep], care să realizeze numărarea în paralel a aparițiilor unui string într-un fișier. Porniți de la sursa `parallel_fgrep.c` din directorul `4-pfgrep` și urmăriți secțiunile *TODO*. Fiecare fir de execuție va căuta șirul într-o anumită zonă din fișier și va întoarce numărul de apariții găsite. Firul de execuție principal va colecta rezultatele și va afișa numărul total de apariții.

Fișierul este mapat înainte de pornirea firelor de execuție. Observați că nu este nevoie de sincronizarea accesului la citire. Citirile se pot executa în paralel, fără condiții de cursă, atât timp cât nu există scrieri.

Varianta serială a `fgrep` este implementată în fișierul `serial_fgrep.c`. Generați un fișier mare și comparați timpii de execuție:

```
cat Makefile{,}{,}{,}{,}{,}{,}{,}{,}{,}{,}{,}{,}{,}{,}{,}{,}{,}{,} > 2^16_Makefiles
```

```
time ./serial_fgrep grep 2^16_Makefiles
time ./parallel_fgrep grep 2^16_Makefiles
```

În exemplul anterior, se numără aparițiile șirului "grep" în fișierul `2^16_Makefiles`, obținut prin concatenarea conținutului fișierului Makefile de 2^16 ori. Ar trebui să observați un timp de rulare mai mic pentru implementarea paralelă.

Corectitudinea rezultatului se poate testa cu comanda:

```
fgrep -o grep 2^16_Makefiles | wc -l
```

Exercițiul 5 - Blocked

Inspectați fișierul `blocked.c` din directorul `5-blocked`, compilați și executați binarul (repetați până detectați blocarea programului). Programul creează două fire de execuție care caută un număr magic, fiecare în intervalul propriu (nu este neapărat necesar ca numărul să fie găsit). Fiecare fir de execuție, pentru fiecare valoare din intervalul propriu, verifică dacă este valoarea căutată:

- dacă da, marchează un câmp `found` pentru a înștiința și celălalt fir de execuție că a găsit numărul căutat.
- dacă nu, inspectează câmpul `found` al structurii celuilalt fir de execuție, pentru a vedea dacă acesta a găsit deja numărul căutat.

Determinați cauza blocării, reparați programul și explicați soluția. Puteți utiliza `helgrind`, unul din tool-urile `valgrind`, pentru a detecta problema:

```
$ valgrind --tool=helgrind ./blocked
```

Așa cum ne arată și `helgrind`, problema constă în faptul că cele două thread-uri iau cele două mutex-uri în ordinea inversă, situație foarte probabilă în a cauza un deadlock [https://en.wikipedia.org/wiki/Deadlock].

Exercițiul 6 - Single time start / stop functions

Aveți un număr de thread-uri, și două funcții:

- O funcție de `init` apelată de toate thread-urile, dar rulată doar de primul care ajunge să execute.
- O funcție de `deinit` apelată de toate thread-urile, dar rulată doar de ultimul care ajunge să execute.

Pornind de la sursa `once.c` din directorul `6-once`, asigurați-vă că funcțile `init_func` / `deinit_func` sunt apelate o singură dată, după scenariul descris mai sus. **Nu** aveți voie să modificați codul în afara funcțiilor marcate cu `TODO`.

Thread-urile apelează o funcție `thread_func` care printează câte un caracter la consolă.

- Ce se întâmplă dacă **nu** se mai face `fflush(stdout)` după fiecare printare?

Thread-urile apelează `usleep(5000)` pentru a încuraja invocarea scheduler-ului.

- Ce se întâmplă dacă perioada de `usleep` este micșorată?

Revedeți secțiunea despre mutex.

Exercițiul 7 - Producător - Consumator

Intrați în directorul `7-prodcons`. Completați *TODO*-urile din cod pentru a implementa sincronizarea unui producător cu un consumator, ce folosesc în comun un buffer.

Producătorul va pune obiecte în buffer atât timp cât buffer-ul nu este plin și se va bloca atunci când buffer-ul este plin. Producătorul va fi trezit de consumator după ce buffer-ul nu va mai fi plin.

Consumatorul va scoate obiecte din buffer atât timp cât buffer-ul nu este gol și se va bloca atunci când buffer-ul este gol. Consumatorul va fi trezit de producător după ce buffer-ul nu va mai fi gol.

Sincronizați accesul la buffer-ul comun folosind variabile de condiție (revedeți secțiunea despre variabile de condiție).

BONUS

fork vs pthread_create

Vrem să aflăm ce apeluri de sistem sunt realizate în urma apelurilor funcțiilor `fork` și `pthread_create`.

Intrați în directorul `8-fork_thread` și inspectați sursa. Programul creează un fir de execuție și un proces copil.

Folosiți ltrace [https://linux.die.net/man/1/ltrace] pentru a urmări ce apeluri de bibliotecă se fac și ce apeluri de sistem sunt folosite mai departe de către apelurile de bibliotecă:

```
ltrace -S -n 8 ./ft
```

Observați că atât `fork` cât și `pthread_create` folosesc apelul de sistem clone [http://linux.die.net/man/2/clone]. Urmăriți argumentele apelurilor de sistem `clone` folosind strace [http://linux.die.net/man/1/strace]:

```
strace -e clone ./ft
```

Observați alocarea unei stive separate pentru noul thread (argumentul `child_stack`) cât și partajarea resurselor procesului cu acesta (`flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND`).

Thread Specific Data

Fișierul `9-tsd/tsd.c` conține o aplicație ce împarte un task între mai multe fire de execuție. Fiecare fir de execuție are un fișier de log în care va înregistra mesaje despre progresul său. Observați următoarele aspecte:

- crearea de fire de execuție
- așteptarea terminării acestora
- modul în care se creează / folosește / șterge o variabilă specifică unui fir de execuție - `thread_log_key`
- utilitatea unei funcții de cleanup - `close_thread_log`

Mutex vs Spinlock

Dorim să testăm care varianta este mai eficientă pentru a proteja incrementarea unei variabile.

Intrați în directorul `10-spin`, inspectați și compilați sursa `spin.c`. În urma compilării vor rezulta două executabile, unul care folosește un mutex pentru sincronizare, iar altul un spinlock.

Comparați timpii de execuție:

```
time ./mutex
time ./spin
```

Atunci când un fir de execuție găsește mutex-ul ocupat se va bloca. Atunci când un fir de execuție găsește spinlock-ul ocupat va face busy-waiting.

Soluții

lab08-sol.zip [http://elf.cs.pub.ro/so/res/laboratoare/lab08-sol.zip]

Resurse utile

LinuxTutorialPosixThreads [http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html]

POSIX Threads Programming [https://computing.llnl.gov/tutorials/pthreads/]

so/laboratoare/laborator-08.txt · Last modified: 2019/04/17 12:14 by maria.mihailescu