

Laborator 08 - Thread-uri Linux

Materiale ajutătoare

- [lab08-slides.pdf](#)
- [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

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 să 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 să 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 - vezi IPC), 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 celorlalte:
 - 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:
 - socket-uri
 - 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 regiștri (deci și un contor de program - registrul (E) IP)

Procesele sunt folosite de SO pentru a grupa și alocă 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/distrugea unui fir de execuție durează mai puțin decât crearea/distrugea 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 fire 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

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 tipurilor de date necesare) și să utilizăm biblioteca `libpthread`.

Crearea firelor de execuție

Un fir de execuție este creat folosind [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 concurrent 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](#):

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

Așteptarea firelor de execuție

Firele de execuție se așteaptă folosind funcția [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](#) sau printr-un `return` din rutina utilizată la [pthread_create](#)).

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

Terminarea firelor de execuție

Un fir de execuție își încheie execuția:

- la un apel al funcției [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](#).

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 dispoziție de `libpthread`. Totuși, această metodă nu este recomandată, pentru că este greoaie, și pune probleme foarte delicate la clean-up. 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 o 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](#):

```
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](#):

```
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](#):

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

Pentru a determina valoarea unei variabile de tip TSD se folosește funcția [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.

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 poate specifica un atribut 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`: `int sched_yield(void);`

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

Alte operații

Compilare

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

Atenție! Nu link-ați 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.

[thread2.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++)
        <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("%c",
p->character);
        <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("\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 altul. 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 statica a unui mutex, cu atribute implicite
// NB: mutex-ul nu este eliberat, durata de viata a mutex-ului
// este durata de viata a programului.

```
pthread_mutex_t mutex_static = PTHREAD_MUTEX_INITIALIZER;
```

- inițializat cu **atribute implicite** ([pthread_mutex_init](#), [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); // attr=NULL -> atribute implicite

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

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

- inițializare cu **atribute explicite**

Atenție: 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** (*priority inheritance*) pentru a preveni **inversiunea de prioritate** (*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.

Ocuparea/eliberarea unui mutex

Funcțiile de ocupare blocantă/eliberare a unui mutex ([pthread_mutex_lock](#), [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 nebloantă 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](#):

```
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 ocured */
}
```

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 */
    <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("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.

Semafor

Semafoarele sunt obiecte de sincronizare ce reprezintă o generalizare a mutexurilor 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. Mutexurile pot fi privite, așadar, ca niște semafoare binare.

Semafoarele POSIX au fost prezentate în cadrul [Laboratorului 05 - IPC](#).

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 */
sem_t* sem_open(const char *name, int oflag, mode_t mode, unsigned int value);

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

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

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);
```

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** de execuție 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** mutexul asociat variabilei condiție. Condiția logică este testată sub protecția mutexului, iar dacă nu este îndeplinită, firul apelant se blochează la variabila condiție, eliberând atomic mutexul. În momentul deblocării, un fir de execuție va încerca să ocupe mutexul 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 mutexul asociat condiției. Restul vor aștepta ca acesta să-l elibereze, însă este posibil ca firul care a ocupat mutexul să **schimbe** valoarea predicatului logic pe durata deținerii mutexului. 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/distrușirea unei variabile de condiție

Inițializarea unei variabile de condiție se face folosind macro-ul `PTHREAD_COND_INITIALIZER` sau funcția [pthread_cond_init](#). Distrușirea unei variabile de condiție se face prin funcția [pthread_cond_destroy](#).

```
// initializare statica a unei variabile de condiție cu attribute implicite
// NB: variabila de conditie nu este eliberata,
// durata de viata a variabilei de condiție este durata de viata a programului.
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

// semnaturile functiilor de initializare si 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 nul, se folosesc attribute 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](#):

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

Firul de execuție apelant trebuie să fi **ocupat** deja mutexul asociat, în momentul apelului. Funcția `pthread_cond_wait` va **elibera** mutexul 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 mutexului asociat, și după **ocuparea** acestuia, apelul funcției va întoarce. Observați că firul de execuție apelant poate fi suspendat, după deblocare, în așteptarea ocupării mutexului 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](#):

```
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 mutexului 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](#):

```
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 **mutexul** 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);
    while (count == 0)
        pthread_cond_signal(&count_nonzero);
    count = count + 1;
    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](#):

```
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 mutexului asociat variabilei condiție. Firul de execuție apelant trebuie să dețină mutexul 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 ajutorul unei variabile de condiție.

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

#define NUM_THREADS 5

// implementarea unei bariere *nereentrante* cu variabile de conditie
struct my_barrier_t {
    // mutex folosit pentru a serializa accesese la datele interne ale barierei
    pthread_mutex_t lock;

    // variabila de conditie pe care se astepta sosirea tuturor firelor de executie
    pthread_cond_t cond;

    // numar de fire de executie care trebuie sa mai vina 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);
}
```

```

    // cate fire de executie sunt asteptate la bariera.
    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;

    // inainte de a lucra cu datele interne ale barierei trebuie sa preluam mutexul
    pthread_mutex_lock(&bar.lock);

    <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("thd %d: before the
barrier\n", thd_id);

    // suntem ultimul fir de executie care a sosit la bariera?
    int is_last_to_arrive = (bar.nr_still_to_come == 1);
    // decrementam numarul de fire de executie asteptate la bariera
    bar.nr_still_to_come --;

    // cat timp mai sunt fire de executie care nu au ajuns la bariera, asteptam.
    while (bar.nr_still_to_come != 0)
        // lockul se elibereaza automat inainte de a incepe asteptarea
        pthread_cond_wait(&bar.cond, &bar.lock);

    // ultimul fir de executie ajuns la bariera va semnaliza celelalte fire
    if (is_last_to_arrive) {
        <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("    let the flood
in\n");
        pthread_cond_broadcast(&bar.cond);
    }

    <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("thd %d: after the
barrier\n", thd_id);

    // la iesirea din functia de asteptare se preia automat mutexul, 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.

Bariera

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](#) și se va distruge folosind [pthread_barrier_destroy](#).

```
// pentru a folosi funcțiile de lucru cu bariere e nevoie să se definească
// _XOPEN_SOURCE la o valoare  $\geq 600$ . 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](#):

```
#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;

    <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("thd %d: before the barrier\n", thd_id);

    // toate firele de executie asteapta la bariera.
    rc = pthread_barrier_wait(&barrier);
    if (rc == PTHREAD_BARRIER_SERIAL_THREAD) {
        // un singur fir de execuție (posibil ultimul) va intoarce PTHREAD_BARRIER_SERIAL_THREAD
        // restul firelor de execuție întorc 0 în caz de succes.
        <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>(" let the flood in\n", thd_id);
```

```

    }

    <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("thd %d: after the
barrier\n", thd_id);

    return NULL;
}

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

    // bariera este initializata o singura data si folosita de toate firele de executie
    pthread_barrier_init(&barrier, NULL, NUM_THREADS);

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

    // asteptam ca toate firele de executie sa se termine
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tids[i], NULL);

    // eliberam 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

```

Exercitii de laborator

În rezolvarea laboratorului folosiți arhiva de sarcini [lab08-tasks.zip](#)

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

Linux

Pentru a instala paginile de manual pentru 'pthreads'

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

1. (1 punct) Thread Stack

- o Intrați în directorul 1- `th_stack` și inspectați sursa.
- o Compilați și rulați programul
- o Urmăriți cu `pmap` cum se modifică spațiul de adresă al programului.
 - Hint:
 - Puteți rula comanda: `watch -d pmap $(pidof th_stack)`
- o Ce reprezintă zonele de memorie care se creează după fiecare apel `pthread_create`?
- o Câte fire de execuție se creează? Ce cod execută fiecare fir?
- o Folosiți `Ctrl+C` pentru a termina programul.

2. (1 punct) Fire de execuție vs Procese

- o Intrați în directorul 2- `th_vs_proc` și inspectați sursele.
- o Ambele programe simulează un server care creează fire de execuție/procese.
- o Compilați și rulați pe rând ambele programe.
- o Afișați câte fire de execuție/procese s-au creat în ambele situații. Dar pentru procesul `init`?

- Hint:
 - Folosiți utilitarul `ps`
 - Căutați parametrul `'-L'`
- Ce credeți că se întâmplă dacă la un moment dat un fir de execuție moare? Dar dacă moare un proces?
 - Hint:
 - Testați utilizând funcția `do_bad_task` la fiecare al 4-lea fir de execuție/process
- 3. (2 puncte) Thread safe
 - Intrați în directorul `3-safety` și inspectați sursa `vars.c`
 - Sunt funcțiile `thread_function` și `main thread-safe` relativ la variabilele `a`, `b`, `c`?
 - Hint:
 - Revedeți ce înseamnă [thread safe](#)
 - Observați ce se întâmplă cu memoria alocată pentru variabila `rez` după ce se face `join`. Cum explicați?
 - Este funcția `malloc` thread-safe?
 - Hint:
 - Sursa `malloc.c` testează apelul `malloc` realizat din mai multe fire de execuție
- 4. (2 puncte) Parallel [fgrep](#)
 - Implementați un program similar cu [fgrep](#), care să realizeze numărarea aparițiilor unui string într-un fișier în paralel.
 - Porniți de la sursa `4-pfgrep.c` din directorul `pfgrep`
 - Hint:
 - Revedeți secțiunile `TODO`. Fiecare fir de execuție va căuta un șir într-o zonă de fișier, și va întoarce numărul de apariții. firul de execuție principal va colecta rezultatele și va afișa numărul total de apariții.
 - Fișierul este mapat înainte de pornirea firelor de execuție. Este nevoie de sincronizarea accesului la citire?
 - Comparați timpii de execuție obținuți cu varianta serială.
 - Hints:
 - Întâi generați un fișier mare, pe care să puteți testa: `ls -R / > big_file.txt`
 - Pentru a măsura timpul de execuție al unui program folosiți comanda `time`.
- 5. (1 punct) ? Blocked
 - Inspectați fișierul `blocked.c` din directorul `5-blocked`, compilați și executați binarul (repetăți până detectați blocarea programului).
 - Programul creează două fire de execuție care caută un număr magic, fiecare în intervalul propriu.
 - 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 celuiilalt fir de execuție, pentru a vedea dacă acesta a găsit deja numărul căutat.
 - Determinați cauza blocării, reparați programul și explicați soluția.
 - Hints:
 - puteți utiliza `helgrind`, unul din tool-urile `valgrind`, pentru a detecta problema: `$ valgrind --tool=helgrind ./blocked`
 - chiar dacă, aparent, programul nu se blochează, citiți mesajele afișate de `valgrind`
- 6. (1 punct) Implementare comportament [pthread_once\(\)](#)
 - Aveți o funcție de inițializare pe care vreți să o apelați o singură dată.
 - Pornind de la sursa `once.c` din directorul `6-once`, asigurați-vă că funcția `init_func()` este apelată o singură dată.
 - Nu modificați funcția `init_func()`
 - Hint:
 - Citiți despre funcționalitatea [pthread_once\(\)](#)
 - Revedeți secțiunea despre [mutex](#)
- 7. (2 puncte) Producător - Consumator
 - Intrați în directorul `7-prodcons`
 - Compilați și rulați sursa. Ce observați ?
 - 1. (1 punct) Sincronizați accesul la bufferul partajat folosind două semafoare.
 - Hints:
 - Revedeți secțiunea despre [semafoare](#)
 - 2. (1 punct) Sincronizați accesul folosind variabile de condiție
 - Hints:
 - Revedeți secțiunea despre [variabile de condiție](#)

BONUS

- (1 so-karma) `fork()` vs `pthread_create()`
 - Ce se întâmplă dacă într-un proces care a creat fire de execuție se apelează `fork()`?
 - Intrați în directorul `8-fork_thread` și inspectați sursa.
 - Programul creează un fir de execuție, care nu își termină execuția până la apelul `fork()`
 - Verificați ce se întâmplă rulând programul. Cum explicați?
 - Hint:

- Folosiți [strace](#) pentru a vedea ce apeluri de sistem de fac.
- Apare apelul de sistem fork()?
- Folosiți următoarea comandă pentru a urmări apelul de sistem [clone\(\)](#): `ltrace -S -n 8 ./ft`
- Citiți pagina de manual a apelului [clone\(\)](#).
- **(1 so-karma) 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:**
 - crearea de fire de execuție
 - așteptarea terminării acestora
 - cum se creează / folosește / șterge o variabilă specifică unui fir de execuție - `thread_log_key`
 - utilitatea unei funcții de cleanup - `close_thread_log`
 - De ce `thread_function` nu mai trebuie să închidă fișierele de log?
- **(1 so-karma) Mutex vs. Spinlock**
 - Care varianta este mai eficientă pentru a proteja incrementarea unei variabile?
 - Intrați în directorul `10-spin` și inspectați sursa `spin.c`
 - Compilați sursa. În urma compilării vor rezulta două executabile, unul care folosește mutex pentru sincronizare, respectiv spinlock.
 - Obțineți timpul de execuție pentru fiecare program. Cum explicați diferența?
 - Încercați să înlocuiți incrementarea variabilei cu un cod mai costisitor. Ce observați?
 - Hint:
 - Ce se întâmplă dacă un fir de execuție găsește mutex-ul ocupat?
 - Ce se întâmplă dacă un fir de execuție găsește spinlock-ul ocupat?

Extra

1. Descărcați [arhiva](#) cu scripturi Python
 - Rulați `tls.py`; ce observați? De ce cu TLS programul afișează rezultatul corect, iar cu o variabilă globală nu?
 - Rulați `profile_threads.py` și extindeți scriptul pentru a intercepta trace-ul firelor de execuție. Vezi funcția [settrace](#)
 - Analizați cu [ltrace](#) apelurile `pthread_*` le care le face Python când creați un fir de execuție; ce observați?

Soluții

[lab08-sol.zip](#)

Resurse utile

[LinuxTutorialPosixThreads](#)

[POSIX Threads Programming](#)

From:

<http://elf.cs.pub.ro/so/wiki/> - **Sisteme de Operare**

Permanent link:

<http://elf.cs.pub.ro/so/wiki/laboratoare/laborator-08>

Last update: 2012/05/20 00:53