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 execuţie** (sau **thread**), acesta fiind unitatea elementară de planificare într-un sistem. Ca şi procesele, thread-urile reprezintă un mecanism prin care un calculator poate s? ruleze mai multe lucruri simultan.

Un fir de execuţie există în cadrul unui proces, şi reprezintă o unitate de execuţie mai fină decât acesta. În momentul în care un proces este creat, în cadrul lui există un singur fir de execuţie, care execută programul secvenţial. Acest fir poate la rândul lui s? creeze alte fire de execuţie; aceste fire vor rula porţiuni ale binarului asociat cu procesul curent, posibil aceleaşi cu firul iniţial (care le-a creat).

Diferenţe dintre thread-uri şi procese

- procesele nu partajează resurse între ele (decât dacă programatorul foloseşte un mecanism special pentru asta - vezi IPC), pe când thread-urile partajează în mod implicit majoritatea resurselor unui proces. Modificarea unei astfel de resurse dintr-un fir este vizibilă instantaneu şi celorlalte:
  - segmentele de memorie precum `.heap`, `.data` şi `.bss` (deci şi variabilele stocate în ele)
  - descriptorii de fişiere (aşadar, închiderea unui fişier este vizibilă imediat pentru toate thread-urile), indiferent de tipul fişierului.:
    - socketi
    - fişiere normale
    - pipe-uri
    - fişiere ce reprezintă dispozitive hardware (de ex. `/dev/sda1`).
- fiecare fir are un context de execuţie propriu, format din
  - stivă
  - set de regiştri (deci şi un contor de program - registrul `(E)IP`)

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

Avantajele thread-urilor

Deoarece thread-urile aceluiaşi proces folosesc toate spaţiul de adrese al procesului de care aparţin, folosirea lor are o serie de avantaje:

- crearea/distrugerea unui thread durează mai puţin decât crearea/distrugerea unui proces
- timpul context switch-ului între thread-urile aceluiaşi proces este foarte mic, întrucât nu e necesar să se "comute" şi spaţiul de adrese (pentru mai multe informaţii, căutaţi informaţii despre ?TLB flush").
- comunicarea între thread-uri are un overhead mai mic (realizată prin modificarea unor zone de memorie din spaţiul comun de adrese)

Firele de execuţie se pot dovedi utile în multe situaţii, de exemplu, pentru a îmbunătăţi timpul de răspuns al aplicaţiilor cu interfeţe grafice (GUI), unde prelucrările CPU-intensive se fac de obicei într-un thread diferit de cel care afişează interfaţa.

De asemenea, ele simplifică structura unui program şi 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 thread-uri

Din punctul de vedere al implementării, există 3 categorii de thread-uri :

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

Suport POSIX

În ceea ce priveşte thread-urile, POSIX nu specifică dacă acestea trebuie implementate în user-space sau kernel-space. Linux le implementează în kernel-space, dar nu diferenţiază thread-urile de procese decât prin faptul că thread-urile partajează spaţiul de adresă (atât thread-urile, cât şi procesele, sunt un caz particular de "task"). Pentru folosirea thread-urilor în Linux trebuie să includem header-ul `pthread.h` unde se găsesc declaraţiile funcţiilor şi tipurilor de date necesare şi să utilizăm biblioteca `libpthread`.

Crearea firelor de execuţie

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

Noul fir creat va executa concurent cu firul de execuţie din care a fost creat. Acesta va executa codul specificat de funcţia `start_routine` căreia i se va pasa argumentul `arg`. Dacă funcţia de executat are nevoie de mai mulţi parametri, aceştia pot fi agregaţi într-o structură, în câmpul `arg` punându-se un pointer către acea structură.

Prin parametrul `tattr` se stabilesc atributele noului fir de execuţie. Dacă transmitem valoarea `NULL` thread-ul va fi creat cu atributele implicite.

Pentru a determina identificatorul thread-ului curent se poate folosi funcţia :

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

Aşteptarea firelor de execuţie

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

Primul parametru specifică identificatorul firului de execuţie aşteptat, iar al doilea parametru specifică unde se va plasa valoarea întoarsă de funcţia copil (printr-un pthread_exit sau printr-un `return` din rutina utilizată la pthread_create).

Thread-urile se împart în două categorii: *unificabile şi detaşabile. Mai multe detalii în "Show" următor.*

Terminarea firelor de execuţie

Un fir de execuţie îşi încheie execuţia:

- la un apel al funcţiei `pthread_exit`:void pthread_exit(void *retval);
- în mod automat, la sfârşitul codului firului de execuţie.

Prin parametrul `retval` se comunică părintelui un mesaj despre modul de terminare al copilului. Această valoare va fi preluată de funcţia pthread_join.

Metodele ca un fir de execuţie să termine un alt thread 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`. Totuşi, această metodă nu este recomandată, pentru că este greoaie, şi pune probleme foarte delicate la clean-up. Pentru mai multe detalii, consultaţi urmatorul material scris de echipa SO: Terminarea thread-urilor

Thread Specific Data

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

Fiecare thread 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 thread-uri, sau să necesite sincronizare. De aceea, TSD este folosită uneori pentru a optimiza operaţiile care necesită multă sincronizare între thread-uri: fiecare thread calculează informaţia specifică, şi există un singur pas de sincronizare la sfârşit, necesar pentru reunirea rezultatelor tuturor thread-urilor.

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

Crearea şi ştergerea unei variabile

O variabilă se crează folosind pthread_key_create:

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

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

- `NULL` şi este ignorat
- pointer către o funcţie de cleanup care se execută la terminarea thread-ului

Pentru ştergerea unei variabile se apelează pthread_key_delete:

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

Ea nu apelează funcţia de cleanup asociată acesteia.

Modificarea şi citirea unei variabile

După crearea cheii, fiecare fir de execuţie poate modifica propria copie a variabilei asociate folosind funcţia pthread_setspecific:

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

Pentru a determina valoarea unei variabile de tip TSD se foloseşte funcţia pthread_getspecific:

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

Funcţii pentru cleanup

Funcţiile 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 funcţii fără a crea neapărat un thread specific data. Pentru acest scop există funcţiile de cleanup.

Atributele unui thread

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

- starea: unificabil sau detaşabil
- politica de alocare a procesorului pentru thread-ul respectiv (round robin, FIFO, sau system default)

- prioritatea (cele cu prioritate mai mare vor fi planificate, în medie, mai des)
- dimensiunea şi adresa de start a stivei

Mai multe detalii puteţi găsi în secţiunea suplimentară dedicată.

Cedarea procesorului

Un thread cedează dreptul de execuţie unui alt thread, în urma unuia din următoarele evenimente:

- efectuează un apel blocant (cerere de I/O, sincronizare cu un alt thread) şi kernel-ul decide că este *rentabil* să faca un context switch
- i-a expirat cuanta de timp alocată de către kernel
- cedează voluntar dreptul, folosind funcţia:int sched_yield(void);

Dacă există alte procese interesate de procesor acesta li se oferă, iar dacă nu există nici un alt proces în aşteptare pentru procesor, firul curent îşi continuă execuţia.

Alte operaţii

Compilare

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

Atentie! Nu link-aţi 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 execuţie, fiecare afişând 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 thread-urilor

Pentru sincronizarea firelor de execuţie avem la dispoziţie:

- mutex
- semafoare
- variabile de condiţie
- bariere

Mutex

Mutexurile sunt obiecte de sincronizare utilizate pentru a asigura **accesul exclusiv** într-o secţiune de cod în care se utilizează **date partajate** între două sau mai multe fire de execuţie. Un mutex are două stări posibile: **ocupat** şi **liber**. Un mutex poate fi ocupat de un **singur fir** de execuţie la un moment dat. Atunci când un mutex este ocupat de un fir de execuţie, 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 mutexul devine liber.

Iniţializarea/distrugerea unui mutex

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

- folosind o **macrodefiniţie** // initializare statica a unui mutex, cu atribute implicite
  // NB: mutexul nu este eliberat, durata de viata a mutexului
  // este durata de viata a programului.
  pthread_mutex_t mutex_static = PTHREAD_MUTEX_INITIALIZER;

- iniţializat cu **atribute implicite** // semnaturile functiilor de initializare si 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 mutexului ...

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

- iniţializare cu **atribute explicite**

NB: Mutexul trebuie să fie **liber** pentru a putea fi **distrus**. În caz contrar, funcţia va întoarce codul de eroare `EBUSY`. Întoarcerea valorii `0` semnifică succesul apelului.

Tipuri de mutexuri

Folosind atributele de iniţializare se pot crea mutexuri cu proprietăţi speciale:

- activarea **moştenirii de prioritate** (*priority inharitance*) pentru a preveni **inversiunea de prioritate** (*priority invesion*). Există trei protocoale de moştenire a priorităţii:
  - `PTHREAD_PRIO_NONE` ? **nu** se moşteneşte prioritatea când deţinem mutexul creat cu acest atribut
  - `PTHREAD_PRIO_INHERIT` ? dacă deţinem un mutex creat cu acest atribut şi dacă există fire de execuţie blocate pe acel mutex se moşteneşte prioritatea firului de execuţie cu **cea mai mare prioritate**
  - `PTHREAD_PRIO_PROTECT` ? dacă firul de execuţie curent deţine unul sau mai multe mutexuri, acesta va executa la **maximul priorităţilor** specificate pentru toţi mutecşii deţinuţi.

Ocuparea/eliberearea unui mutex

Funcţiile de ocupare blocantă/eliberare a unui mutex:

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

Dacă mutexul este **liber** în momentul apelului, acesta va fi ocupat de firul apelant şi funcţia va întoarce imediat. Dacă mutexul este ocupat de un **alt fir**, apelul va bloca până la eliberarea mutexului. Dacă mutexul este deja ocupat de **firul curent** de execuţie (lock recursiv), comportamentul funcţiei este dictat de tipul mutexului:

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

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

Încercarea neblocantă de ocupare a unui mutex

Pentru a încerca ocuparea unui mutex **fără a aştepta** eliberarea acestuia în cazul în care este deja ocupat, se va apela funcţia:

```
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 mutexurilor

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

Mutexurile din firele de execuţie POSIX sunt implementate cu ajutorul ***futex***-urilor, din considerente de performanţă.

Optimizarea constă în testarea şi setarea atomică a valorii mutexului (printr-o instrucţiune 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 operaţii 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, operaţia de decrementare se va bloca până când valoarea semaforului devine strict pozitivă. Mutexurile pot fi privite, aşadar, ca nişte semafoare binare.

Operaţiile care pot fi efectuate asupra semafoarelor POSIX sunt multiple:

Semafoare cu nume - Iniţializare/deiniţializare

```
/* 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 - Iniţializare/deiniţializare

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

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

Operaţii 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);
```

Semafoarele POSIX au fost prezentate în cadrul laboratorului de comunicare interproces.

Variabile condiţie

Variabilele condiţie pun la dispoziţie un sistem de notificare pentru fire de execuţie, permiţându-i unui fir să se blocheze în aşteptarea unui semnal din partea unui alt fir. Folosirea corectă a variabilelor condiţie presupune un protocol cooperativ între firele de execuţie.

Mutexurile (mutual exclusion locks) şi semafoarele permit blocarea **altor fire** de execuţie. Variabilele de condiţie se folosesc pentru a bloca **firul curent** de execuţie până la îndeplinirea unei condiţii.

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

de intenţie).

O variabilă condiţie trebuie întotdeauna folosită **împreună cu un mutex** pentru evitarea race-ului care se produce când un fir se pregăteşte să aştepte la variabila condiţie în urma evaluării predicatului logic, iar alt fir semnalizează variabila condiţie chiar înainte ca primul fir să se blocheze, pierzându-se astfel semnalul. Aşadar, operaţiile de semnalizare, testare a condiţiei logice şi blocare la variabila condiţie trebuie efectuate având **ocupat** mutexul asociat variabilei condiţie. Condiţia logică este testată sub protecţia mutexului, iar dacă nu este îndeplinită, firul apelant se blochează la variabila condiţie, eliberând atomic mutexul. În momentul deblocării, un fir de execuţie va încerca să ocupe mutexul asociat variabilei condiţie. De asemenea, testarea predicatului logic trebuie făcută într-o **buclă**, deoarece, dacă sunt eliberate mai multe fire deodată, doar unul va reuşi să ocupe mutexul asociat condiţiei. Restul vor aştepta ca acesta să-l elibereze, însă este posibil ca firul care a ocupat mutexul să **schimbe** valoarea predicatului logic pe durata deţinerii mutexului. Din acest motiv celelalte fire trebuie să testeze din nou predicatul pentru că altfel şi-ar începe execuţia presupunând predicatul adevărat, când el este, de fapt, fals.

Iniţializarea/distrugerea unei variabile de condiţie

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

// semnaturile functiilor de initializare si eliberare de variabile de condiţie:
int pthread_cond_init   (pthread_cond_t *cond, pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Ca şi la mutex-uri:

- dacă parametrul `attr` este nul, se folosesc atribute implicite
- trebuie să nu existe nici un fir de execuţie în aşteptare pe variabila de condiţie atunci când aceasta este distrusă, altfel se întoarce `EBUSY`.

Blocarea la o variabilă condiţie

Pentru a-şi suspenda execuţia şi a aştepta la o variabilă condiţie, un fir de execuţie va apela:

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

Firul de execuţie apelant trebuie să fi **ocupat** deja mutexul asociat, în momentul apelului. Funcţia `pthread_cond_wait` va **elibera** mutexul şi se va **bloca**, aşteptând ca variabila condiţie să fie **semnalizată** de un alt fir de execuţie. Cele două operaţii sunt efectuate **atomic**. În momentul în care variabila condiţie este semnalizată, se va încerca ocuparea mutexului asociat, şi după **ocuparea** acestuia, apelul funcţiei va întoarce. Observaţi că firul de execuţie apelant poate fi suspendat, după deblocare, în aşteptarea 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 condiţie, pentru a asigura o serializare corectă a firelor de execuţie. 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 execuţie care aşteptau la variabila condiţie nu ar testa din nou predicatul logic, şi-ar continua execuţia presupunând greşit că acesta e adevărat.

Blocarea la o variabilă condiţie cu timeout

Pentru a-şi suspenda execuţia şi a aştepta la o variabilă condiţie, nu mai târziu de un moment specificat de timp, un fir de execuţie va apela:

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

Funcţia se comportă la fel ca `pthread_cond_wait`, cu excepţia faptului că, dacă variabila condiţie nu este semnalizată mai devreme de `abstime`, firul apelant este deblocat, şi, după ocuparea mutexului asociat, funcţia se întoarce cu eroarea `ETIMEDOUT`. Parametrul `abstime` este absolut şi reprezintă numărul de secunde trecute de la 1 ianuarie 1970, ora 00:00.

Deblocarea unui singur fir blocat la o variabilă condiţie

Pentru a debloca un singur fir de execuţie blocat la o variabilă condiţie se va semnaliza variabila condiţie astfel:

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

Dacă la variabila condiţie nu aşteaptă niciun fir de execuţie, apelul funcţiei nu are efect şi semnalizarea se va **pierde**. Dacă la variabila condiţie aşteaptă mai multe fire de execuţie, va fi deblocat doar unul dintre acestea. Alegerea firului care va fi deblocat este făcută de planificatorul de fire de execuţie. Nu se poate presupune că firele care aşteaptă vor fi deblocate în ordinea în care şi-au început aşteptarea. Firul de execuţie apelant trebuie să deţină **mutexul** asociat variabilei condiţie în momentul apelului acestei funcţii.

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 condiţie

Pentru a debloca toate firele de execuţie blocate la o variabilă condiţie, se semnalizează variabila condiţie astfel:

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

Dacă la variabila condiţie nu aşteaptă niciun fir de execuţie, apelul funcţiei nu are efect şi semnalizarea se va **pierde**. Dacă la variabila condiţie aşteaptă fire de execuţie, toate acestea vor fi deblocate, dar vor **concura** pentru ocuparea mutexului asociat variabilei condiţie. Firul de execuţie apelant trebuie să deţină mutexul asociat variabilei condiţie în momentul apelului acestei funcţii.

Exemplu de utilizare a variabilelor de condiţie

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

```
#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 accesele 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 teh
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 threaduri 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 thread ajuns la bariera va semnaliza celelalte threaduri
    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 teh
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 teh barrier
thd 2: before teh barrier
thd 3: before teh barrier
thd 4: before teh barrier
thd 1: before teh barrier
    let the flood in
thd 1: after teh barrier
thd 2: after teh barrier
thd 3: after teh barrier
thd 4: after teh barrier
thd 0: after teh barrier
```

Din execuţia programului se observă:

- ordinea în care sunt planificate firele de execuţie **nu** este neapărat cea a creării lor
- ordinea în care sunt trezite firele de execuţie ce aşteaptă la o variabilă de condiţie **nu** este neapărat ordinea în care acestea au intrat în aşteptare.

Bariera

Standardul POSIX defineşte şi un set de funcţii şi structuri de date de lucru cu bariere. Aceste funcţii sunt disponibile dacă se defineşte macro-ul `_XOPEN_SOURCE` la o valoare >= 600.

Exercitii de laborator

În rezolvarea laboratorului folosiţi arhiva de sarcini lab08-tasks.zip

Observaţii: Pentru a vă ajuta la implementarea exerciţiilor din laborator, în directorul `utils` din arhivă există un fişier `utils.h` cu funcţii utile.

Linux

Pentru a instala paginile de manual pentru 'pthreads'

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

1. (**1 punct**) Thread Stack
   - Intraţi în directorul `th_stack` şi inspectaţi sursa.
   - Rulaţi programul şi urmăriţi cu `pmap` cum se modifică spaţiul de adresă al programului.
   - Hint:
     - Puteţi rula comanda: watch -d pmap $(pidof th_stack)
   - Ce reprezintă zonele care se creează după fiecare apel `pthread_create`?
   - Câte threaduri se crează? Ce cod execută fiecare thread? Folosiţi `Ctrl+C` pentru a termina programul.
2. (**1 punct**) Threaduri vs Procese
   - Intraţi în directorul `th_vs_proc` şi inspectaţi sursele.
   - Ambele programe simulează un server care creează threaduri/procese.
   - Afişaţi câte threaduri/procese s-au creat folosind `ps`. Dar pentru procesul `init` ? (hint: căutaţi parametrul '-L')
   - Ce credeţi că se întâmplă dacă la un moment dat moare un thread? Dar dacă moare un proces?
   - Hint:
     - Testaţi utilizând funcţia `do_bad_task` la fiecare al 4-lea thread/process
3. (**2 puncte**) Thread safe
   - Intraţi în directorul `safety` şi inspectaţi sursa `vars.c`
   - Sunt funcţiile `thread_function` şi `main` thread-safe relativ la variabilele a, b, c?
   - Hint:
     - Revedeţi ce înseamnă `thread safe`
   - Observaţi ce se întâmplă cu memoria alocată pentru variabila `rez` după ce se face join. Cum explicaţi?
   - Este funcţia `malloc` thread-safe?
   - Hint:
     - Sursa `malloc.c` testează apelul malloc realizat din mai multe threaduri
4. (**2 puncte**) Parallel fgrep
   - Implementaţi un progam similar cu fgrep, care să realizeze numărarea apariţiilor unui string într-un fişier în paralel.
   - Porniţi de la sursa `pfgrep.c` din directorul `pfgrep`
   - Hint:
     - Revedeţi secţiunile *TODO* . Fiecare thread va căuta un şir într-o zonă de fişier, şi va întoarce numărul de apariţii. Thread-ul principal va colecta rezultatele şi va afişa numărul total de apariţii.
     - Mapaţi înainte fişierul. Este nevoie de sincronizarea accesului la citire?
     - Comparaţi timpii de execuţie obţinuţi cu varianta serială.
   - Întâi generaţi un fişier mare, pe care să puteţi testa: ls -R / > big_file.txt
   - Pentru a măsura timpul de execuţie al unui program folosiţi comanda `time`.
5. (**1 punct**) ? `Blocked`
   - Inspectaţi fişierul `blocked.c`, compilaţi şi executaţi binarul (repetaţi până detectaţi blocarea programului). Programul crează două fire de execuţie care caută un număr magic, fiecare în intervalul propriu.
   - Fiecare fir de execuţie, pentru fiecare valoare din intervalul propriu, verifică dacă este valoarea căutată:
     - dacă da, marchează un câmp found pentru a înştiinţa şi celălalt thread că a găsit numărul căutat,
     - dacă nu, inspectează câmpul `found` al structurii celuilalt fir de execuţie, pentru a vedea dacă acesta a găsit deja numărul căutat.
   - Determinaţi cauza blocării, reparaţi programul şi explicaţi soluţia.
   - Hints:
     - puteţi utiliza `helgrind`, unul din tool-urile `valgrind`, pentru a detecta problema: $ valgrind --tool=helgrind ./blocked
     - Chiar daca programul aparent nu se blochează, citiţi mesajele afişate de valgrind
6. (**1 punct**) Implementare comportament `pthread_once()`
   - Aveţi o funcţie de iniţializare pe care vreţi să o apelaţi o singură data.
   - Pornind de la sursa `once.c` din directorul `once`, asiguraţi-vă că funcţia `init_func()` este apelată o singură data.
   - Nu modificaţi funcţia `init_func()`
   - Hint:
     - Cititi despre funcţionalitatea pthread_once()
     - Revedeţi secţiunea despre mutex
7. (**2 puncte**)

- Intraţi în directorul `prodcons`
- Compilaţi si rulaţi sursa. Ce observaţi ?

1. (**1 punct**) Sincronizaţi accesul la bufferul partajat folosind două semafoare.
   - Hints:
     - Revedeţi secţiunea despre semafoare
2. (**1 punct**) Sincronizaţi accesul folosind variabile de condiţie
   - Hints:
     - Revedeţi sectiunea despre variabile de condiţie

BONUS

- (**1 so-karma**) fork() vs pthread_create()
  - Ce se întâmplă dacă într-un proces care a creat threaduri se apelează fork()?
  - Intraţi în directorul `fork_thread` şi inspectaţi sursa.
  - Programul creeaza lansează un thread, care nu îşi termină execuţia până la apelul fork()
  - Verificaţi ce se întamplă rulând programul. Cum explicaţi ?
  - Hint:
    - Folosiţi strace pentru a vedea ce apeluri de sistem de fac.
    - Apare apelul de sistem fork() ?
    - Folosiţi următoarea comandă pentru a urmări apelul de sistem clone(): ltrace -S -n 8 ./ft
    - Citiţi pagina de manual pentru apelul de sistem clone()
- (**1 so-karma**) Thread Specific Data
  - Fişierul `9-tsd/tsd.c` conţine o aplicaţie ce împarte taskul între mai multe threaduri.
  - Fiecare thread are un fişier de log în care va înregistra mesaje despre progresul său.
  - **Observaţi**
    - creare de thread-uri
    - aşteptarea terminării acestora
    - cum se creează / foloseşte / şterge o variabilă specifică unui thread - `thread_log_key`
    - utilitatea unei funcţii de cleanup - `close_thread_log`
  - De ce `thread_function` nu mai trebuie să închidă fişierele de log ?

- (**1 so-karma**) Mutex vs Spinlock
  - Care varianta este mai eficienta pentru a proteja incrementarea unei variabile?
  - Intraţi în directorul `10-spin` şi inspectaţi sursa `spin.c`
  - Compilaţi sursa. În urma compilării vor rezulta doua executabile, unul care foloseşte mutex pentru sincronizare, respectiv spinlock.
  - Testaţi timpii de execuţie. Cum explicaţi diferenţa ?
  - Încercaţi să înlocuiţi incrementarea variabilei cu un cod mai costisitor. Ce observaţi?
  - Hint:
    - Ce se întâmplă dacă un thread găseşte mutex-ul ocupat?
    - Ce se întâmplă dacă un thread găseşte spinlock-ul ocupat?

Extra

1. Descărcaţi arhiva cu scripturi Python
   - Rulaţi tls.py; ce observaţi? De ce cu TLS programul afişează rezultatul corect, iar cu o variabilă globală nu?
   - Rulaţi profile_threads.py şi extindeţi scriptul pentru a intercepta trace-ului thread-urilor. Vezi funcţia settrace
   - Analizaţi cu ltrace apelurile pthread_* le care le face Python când creaţi un thread; ce observaţi?

Soluţii

Resurse utile

LinuxTutorialPosixThreads

POSIX Threads Programming