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

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 sǎ 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 sǎ 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:
    - sockeți
    - 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 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

Detalii despre categoriile de fire de execuţie

**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 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 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*.

Detalii despre fire de execuţie unificabile şi detaşabile

* *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 clean-up. Pentru mai multe detalii: [Terminarea thread-urilor](http://ocw.cs.pub.ro/courses/so/laboratoare-2013/resurse/threaduri_extra)

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.

Detalii despre funcţii 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) înaceeași funcție și la același nivel de imbricare.

Un mic exemplu de folosire a funcțiilor de cleanup :

[th\_cleanup.c](http://ocw.cs.pub.ro/courses/_export/code/so/laboratoare/laborator-08?codeblock=10)

void \*alocare\_buffer(int size)

{

return [malloc](http://www.opengroup.org/onlinepubs/009695399/functions/malloc.html)(size);

}

void dealocare\_buffer(void \*buffer)

{

[free](http://www.opengroup.org/onlinepubs/009695399/functions/free.html)(buffer);

}

*/\* functia apelata de un fir de execuție \*/*

void functie()

{

void \*buffer = alocare\_buffer(512);

*/\* inregistrarea functiei de cleanup \*/*

pthread\_cleanup\_push(dealocare\_buffer, buffer);

*/\* aici au loc prelucrari, si se poate apela pthread\_exit*

*sau firul poate fi terminat de un alt fir \*/*

*/\* deinregistrarea functiei de cleanup si executia 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 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](http://ocw.cs.pub.ro/courses/so/laboratoare-2013/resurse/threaduri_extra#lucrul_cu_atributele_unui_thread).

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 nici un 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 cuPTHREAD\_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 întotdeauna 0.

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.

[thread2.c](http://ocw.cs.pub.ro/courses/_export/code/so/laboratoare/laborator-08?codeblock=15)

*#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](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("%c", p->character);

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("\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](http://www.opengroup.org/onlinepubs/009695399/functions/perror.html)("pthread\_create");

[exit](http://www.opengroup.org/onlinepubs/009695399/functions/exit.html)(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](http://www.opengroup.org/onlinepubs/009695399/functions/perror.html)("pthread\_create");

[exit](http://www.opengroup.org/onlinepubs/009695399/functions/exit.html)(1);

}

*/\* wait for completion \*/*

if (pthread\_join(fir1, NULL))

[perror](http://www.opengroup.org/onlinepubs/009695399/functions/perror.html)("pthread\_join");

if (pthread\_join(fir2, NULL))

[perror](http://www.opengroup.org/onlinepubs/009695399/functions/perror.html)("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](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-08#mutex)
* [semafoare](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-08#semafor)
* [variabile de condiție](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-08#variabile_conditie)
* [bariere](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-08#bariera)

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**
* *// initializare 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](http://linux.die.net/man/3/pthread_mutex_init), [pthread\_mutex\_destroy](http://linux.die.net/man/3/pthread_mutex_destroy))
* *// 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 mutex-ului ...*
* *// eliberare mutex*
* pthread\_mutex\_destroy(&mutex\_implicit);

}

* inițializare cu **atribute explicite**

Detalii despre iniţializare 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 atribute, 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.

Click to hide ⇱

*#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](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("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.

Detalii despre futex-uri

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 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 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);

Detalii despre funcţiile sem\_open

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](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("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** 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/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).

*// initializare statica a unei variabile de condiție cu atribute 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 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 mutexul asociat, în momentul apelului. Funcțiapthread\_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](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 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](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 **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);

if (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](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 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 ajutorului unei variabile de condiție.

*#include <stdio.h>*

*#include <pthread.h>*

*#define NUM\_THREADS 5*

*// implementarea unei bariere \*non-reutilizabile\* 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);

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("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 execuție 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 execuție ajuns la bariera va semnaliza celelalte fire*

if (is\_last\_to\_arrive) {

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)(" let the flood in\n");

pthread\_cond\_broadcast(&bar.cond);

}

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("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](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. Funcțiapthread\_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](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("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.*

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)(" let the flood in\n");

}

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("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

Exerciţii de laborator

Exercițiul 0 - Joc interactiv (2p)

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

Linux (9p)

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 (1p)

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:

student@spook:~$ **watch** -d pmap $(**pidof** th\_stack)

student@spook:~$ **watch** -d **cat** /proc/$(**pidof** th\_stack)/maps

Zonele de memorie cu dimensiunea de 8MB (8192KB) care se creează după fiecare apel pthread\_createreprezintă 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) care are rolul de "pagină de gardă".

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

Exercițiul 2 - Fire de execuție vs. Procese (1p)

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.

Afișați câte fire de execuție/procese s-au creat în ambele situații folosind comanda ps -L -C <nume\_program>.

student@spook:~$ **ps** -L -C threads

student@spook:~$ **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 - Thread safe (1p)

Intrați în directorul 3-safety și inspectați sursa vars.c. Funcțiile thread\_function și main **NU** sunt thread-safe relativ la variabilele a și b (revedeți semnificația lui [thread safe](http://en.wikipedia.org/wiki/Thread_safety)). Există o condiție de cursă între cele două threaduri create la incrementarea variabilei b, declarate în funcția thread\_function, și o altă condiție de cursă între toate threadurile procesului la incrementarea variabilei globale a. Datorită introducerii artificiale a apelurilor sleep(), manifestarea condițiilor de cursă poate fi diminuată (dar nu **eliminată**).

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

student@spook:~$ **valgrind** --tool=helgrind ./vars

Observați ce se întâmplă cu memoria alocată pentru variabila rez după ce se face join. Folosiți valgrind pentru a investiga:

student@spook:~$ **valgrind** --leak-check=full ./vars

În fișierul malloc.c se creează NUM\_THREADS thread-uri care alocă în NUM\_ROUNDS runde memorie. Șansele sunt mari ca thread-urile să execute apeluri malloc concurente. După compilare și rulare de mai multe ori, observăm că programul rulează cu succes. Pentru a face verificări suplimentare, rulăm din nou helgrind.

student@spook:~$ **valgrind** --tool=helgrind ./malloc

Observăm că nici helgrind nu raportează vreo eroare, lucru care conduce la faptul că funcția malloc ar fi thread-safe.

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

Funcția malloc în 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).

Exercițiul 4 - Parallel fgrep (1.5p)

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

Implementați un program similar cu [fgrep](http://linux.die.net/man/1/fgrep), care să realizeze numărarea aparițiilor unui string într-un fișier în paralel pornind de la sursa pfgrep.c din directorul 4-pfgrep. Urmăriți 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. Observați că nu este nevoie de sincronizarea accesului la citire. Citirile se pot executa în paralel, fără condiții de cursă, atâta timp cât nu există scrieri.

Varianta serială a fgrep este implementată în fișierul myfgrep.c. Comparați timpii de execuție obținuți cu varianta serială. Întâi generați un fișier mare, pe care să puteți testa:

student@eg306:~$ **cat** <(**find** /var 2> /dev/null){,}{,}{,}{,}{,}{,}{,}{,}{,} > big\_file.txt

Pentru a măsura timpul de execuție al unui program folosiți comanda time.

student@eg306:~$ time ./myfgrep spool big\_file.txt

student@eg306:~$ time ./pfgrep spool big\_file.txt

Dacă implementarea este corectă, ar trebui să observați un timp de rulare considerabil mai mic pentru implementarea paralelă.

Exercițiul 5 - Blocked (1.5p)

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-urilevalgrind, pentru a detecta problema:

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

Chiar dacă, aparent, programul nu se blochează, citiți mesajele afișate de valgrind.

Așa cum ne arată și helgrind, problema constă în faptul că cele două threaduri iau cele două mutexuri în ordinea inversă, situație foarte probabilă în a cauza un [deadlock](https://en.wikipedia.org/wiki/Deadlock).

Exercițiul 6 - Implementare comportament pthread\_once (1p)

Aveți o funcție de inițializare pe care vreți să o apelați o singură dată. Pornind de la sursa once.c din directorul6-once, asigurați-vă că funcția init\_func() este apelată o singură dată. Nu modificați funcția init\_func() si nu folosiţi pthread\_once(). Citiți despre funcționalitatea [pthread\_once()](http://linux.die.net/man/3/pthread_once) și revedeți secțiunea despre [mutex](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-08#mutex).

Exercițiul 7 - Producător - Consumator (2p)

Intrați în directorul 7-prodcons. Compilați și rulați sursa.

Sincronizați accesul folosind variabile de condiție (revedeți secțiunea despre [variabile de condiție](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-08#variabile_conditie)).

BONUS

1 so karma - fork() vs pthread\_create()

Ce se întâmplă dacă într-un proces care a creat fire de execuție se apelează [fork()](http://linux.die.net/man/2/fork)? Intrați în directorul8-fork\_thread și inspectați sursa. Programul creează un fir de execuție, care nu își termină execuția până la apelul [fork()](http://linux.die.net/man/2/fork). Verificați ce se întamplă rulând programul și folosiți [strace](http://linux.die.net/man/1/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()](http://linux.die.net/man/2/clone):

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

Citiți pagina de manual a apelului [clone()](http://linux.die.net/man/2/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 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.

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, inspectați și compilați sursa spin.c. 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 se întâmplă dacă un fir de execuție găsește mutex-ul ocupat? Ce se întâmplă dacă un fir de execuție găsește spinlock-ul ocupat?