Thread-uri

Contents

- 1 Prezentare teoretică
  - 1.1 Introducere
    - 1.1.1 Diferente dintre thread-uri şi procese
  - 1.2 Avantajele thread-urilor
  - 1.3 Tipuri de thread-uri
    - 1.3.1 Kernel Level Threads
    - 1.3.2 User Level Threads
    - 1.3.3 Fire de execuţie hibride
- 2 Funcţii în Linux
  - 2.1 Suport POSIX
  - 2.2 Crearea firelor de execuţie
  - 2.3 Ateptarea firelor de execuţie
  - 2.4 Terminarea firelor de execuţie
  - 2.5 Thread Specific Data
    - 2.5.1 Crearea şi ştergerea unei variabile
    - 2.5.2 Modificarea şi citirea unei variabile
  - 2.6 Funcţii pentru cleanup
  - 2.7 Atributele unui thread
  - 2.8 Cedarea procesorului
  - 2.9 Alte operaţii
  - 2.10 Compilare
  - 2.11 Exemplu
- 3 Funcţii în Windows
  - 3.1 Crearea firelor de execuţie
  - 3.2 Handle i identificator
  - 3.3 Aşteptarea firelor de execuţie
  - 3.4 Terminarea firelor de execuţie
  - 3.5 Suspend, Resume
  - 3.6 Cedarea procesorului
  - 3.7 Alte funcţii utile
  - 3.8 Thread Local Storage
  - 3.9 Fibre de execuţie
  - 3.10 Securitate i drepturi de acces
  - 3.11 Exemplu
- 4 Quiz
- 5 Exerciţii
  - 5.1 Warning
  - 5.2 Prezentare
  - 5.3 Linux
  - 5.4 Windows
- 6 Soluţii

Prezentare teoretică

Introducere

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

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

Diferente 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)
    - ♦ socketii
- fiecare fir are un context de execuție 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 execuție pentru a planifica execuția de cod care accesează (în mod partajat) aceste resurse.

Avantajele thread-urilor

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

- crearea/distrugerea unui thread durează mai puțin 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 spațiul de adrese (pentru mai multe informații, căutați "TLB flush" pe google)
- comunicarea între thread-uri are un overhead minim (practic se face prin modificarea unor zone de memorie din spațiul de adresă)

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 thread diferit de cel care afişează interfața.

De asemenea, ele simplifică structura unui program i 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

Există 3 categorii de thread-uri :

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

Kernel Level Threads

Managementul thread-urilor este făcut de kernel, şi programele user-space pot crea/distruge thread-uri printr-un set de apeluri de sistem. Kernel-ul menține informații de context atât pentru procese cât i pentru thread-urile din cadrul proceselor, iar planificarea pentru execuție se face la nivel de thread.

Avantaje :

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

Dezavantaje :

- comutarea de context o face kernelul, deci pentru fiecare schimbare de context se trece din firul de execuție în kernel i apoi se mai face încă o schimbare din kernel în alt fir de execuție, deci viteza de comutare este mică.

User Level Threads

Kernel-ul nu este contient de existența lor, şi managementul lor este făcut de procesul în care ele există, folosind de obicei o bibliotecă. Astfel, schimbarea contextului nu necesită intervenția kernel-ului, iar algoritmul de planificare depinde de aplicație.

Avantaje :

- schimbarea de context nu implică kernelul, deci avem o comutare rapidă
- planificarea poate fi aleasă de aplicație si deci se poate alege una care să favorizeze creşterea vitezei aplicației noastre
- thread-urile pot rula pe orice SO, deci şi pe cele care nu suportă thread-uri (au nevoie doar de biblioteca ce le implementează)

Dezavantaje :

- kernel-ul nu tie de thread-uri, deci dacă un thread apelează ceva blocant toate thread-urile planificate de aplicație vor fi blocate. Cele mai multe apeluri de sistem sunt blocante
- kernel-ul planifică thread-urile de care ştie, fiecare pe un singur procesor la un moment dat. În cazul user-level threads, el va vedea un singur thread. Astfel, chiar dacă 2 thread-uri user-level sunt implementate folosind un singur thread "văzut" de kernel, ele nu vor putea folosi eficient resursele sistemului (vor împărți amândouă un acelaşi procesor).

Fire de execuție hibride

Aceste fire încearcă să combine avantajele thread-urilor user-level cu cele ale thread-urilor 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 thread-urilor se face în user space i tot aici se face aproape toată planificarea şi sincronizarea. Kernel-ul ştie doar de KLT-urile pe care sunt multiplexate ULT, şi doar pe acestea le planifică. Programatorul poate schimba eventual numărul de KLT alocate unui proces.

Funcții în Linux

Suport POSIX

În ceea ce privete 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 spațiul 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 declarațiile funcțiilor i tipurilor de date necesare şi să utilizăm biblioteca *libpthread*.

Crearea firelor de execuție

Pentru crearea unui nou fir de execuție se foloseste funcția `pthread_create` :

```
#include <pthread.h>
int pthread_create(pthread_t *tid, const pthread_attr_t *tattr,
                   void*(*start_routine)(void *), void *arg);
```

Noul fir creat se va executa 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`. Folosind `arg` se poate transmite firului de execuție un pointer la o structură care sa conțină toți "parametrii" necesari acestuia.

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

Ateptarea firelor de execuție

La fel ca la procese, un părinte îi poate atepta fiul apelând `pthread_join` (înlocuiete `waitpid`).

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

Primul parametru specifică identificatorul firului de execuție ateptat, iar al doilea parametru specifică unde se va plasa codul întors de funcția copil (printr-un `pthread_exit` sau printr-un `return`).

În caz de succes se întoarce valoarea 0, altfel se întoarce o valoare negativă reprezentând un cod de eroare.

Thread-urile se împart în două categorii :

- *unificabile* :
  - permit unificarea cu alte threaduri care apelează `pthread_join`.
  - resursele ocupate de thread nu sunt eliberate imediat după terminarea threadului, ci mai sunt păstrate până când un alt thread va executa `pthread_join` (analog cu procesele zombie)
  - threadurile sunt implicit unificabile

- *detaşabile*
  - un thread este detaşabil dacă :
    - a fost creat detaşabil.
    - i s-a schimbat acest atribut în timpul execuției prin apelul `pthread_detach`.
  - nu se poate executa un `pthread_join` pe ele
  - vor elibera resursele imediat ce se vor termina (analog cu ignorarea semnalului SIGCHLD în părinte atunci când se termină procesele copil)

Terminarea firelor de execuție

Un fir de execuție se termină la un apel al funcției `pthread_exit` :

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

Dacă nu există un astfel de apel este adăugat unul, în mod automat, la sfârșitul codului firului de execuție.

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

Metodele ca un fir de execuție să termine un alt 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, consultați 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 locația 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 operațiile care necesită multă sincronizare între thread-uri: fiecare thread calculează informația 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 locația de pe stivă unde este memorată variabila respectivă). Descriem în continuare operațiile disponibile cu variabilele din TSD:

Crearea şi ştergerea unei variabile

O variabilă se crează folosind:

```
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 clean-up care se execută la terminarea thread-ului

Pentru ştergerea unei variabile se apelează:

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

Ea nu apelează funcția de cleanup asociată acesteia.

Modificarea şi citirea unei variabile

După crearea cheii, fiecare fir de execuție poate modifica propria copie a variabilei asociate folosind funcția `pthread_setspecific`:

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

Primul parametru reprezintă cheia, iar al doilea parametru reprezintă valoarea specifică ce trebuie stocată i care este de tipul `void*`.

Pentru a determina valoarea unei variabile de tip TSD se folosete funcția :

```
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 thread specific data. Pentru acest scop exista funcțiile de cleanup.

O astfel de funcție de cleanup este o funcție care este apelată când un thread se termină. Ea primete 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` sau este terminat de un alt fir folosind `pthread_cancel`. În circumstanțe normale, atunci când un fir nu se termină în mod forțat, resursa trebuie eliberată explicit, iar funcția de cleanup trebuie

sa fie scoasă.

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

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

Aceasta funcție primete ca parametri un pointer la funcția care este înregistrată i 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` trebuie să existe i apelul corespunzător `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, i dacă parametrul `execute` este nenul o va i executa.

Atentie! Un apel `pthread_cleanup_push` trebuie să aibă un apel corespunzător `pthread_cleanup_pop` în **aceeai funcție** i la **acelai nivel de imbricare**.

Un mic exemplu de folosire a funcțiilor de cleanup :

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

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

/* functia apelata de un thread */

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 f

/* deinregistrarea functiei de cleanup si executia ei (parametrul dat este nenul) */
pthread_cleanup_pop(1);
}
```

Atributele unui thread

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 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 puteți găsi la secțiunea suplimentară dedicată.

Cedarea procesorului

Un thread cedează dreptul de executie 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 funcția:

```
#include <sched.h>
int sched_yield(void);
```

Dacă există alte procese interesate de procesor acesta li se oferă, iar dacă nu există nici un alt proces în ateptare pentru procesor, firul curent îi 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 execute o singură dată. Argumentul `once_control` este un pointer la o variabilă inițializată cu PTHREAD_ONCE_INIT. Prima oară când această funcție este apelată ea va apela funcția `init_routine` i va schimba valoarea variabilei `once_control` pentru a ține minte că inițializarea a avut loc. Următoarele apeluri ale acestei funcții cu acelai `once_control` nu vor face nimic.

Funcția `pthread_once` întoarce întotdeauna 0.

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

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

Pentru a determina dacă doi identificatori se referă la acelai thread 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 i biblioteca *libpthread* (deci se va folosi argumentul *-lpthread*).

Atentie! Nu link-ați un program single-threaded cu această bibliotecă. Daca faceți aa ceva se vor stabili nite mecanisme multithreading care vor fi inițializate la execuție. Atunci programul va fi mult mai lent, va ocupa mult mai multe resurse i va fi mult mai dificil de debug-at.

Exemplu

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

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

/* structura ce contine parametrii transmisi fiecarui thread */
struct parametri {
char caracter; /* caracterul afisat */
int numar;     /* de cate ori va fi afisat */
};

/* functia executata de thread-uri */
void* afisare_caracter(void *params)
{
struct parametri* p = (struct parametri*) params;
int i;

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

return NULL;
}

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

/* cream un thread care va afisa 'x' de 11 ori */
        fir1_args.caracter = 'x';
        fir1_args.numar = 11;
if (pthread_create(&fir1, NULL, &afisare_caracter, &fir1_args)) {
perror("pthread_create");
exit(1);
}

/* cream un thread care va afisa 'y' de 13 ori */
        fir2_args.caracter = 'y';
        fir2_args.numar = 13;
if (pthread_create(&fir2, NULL, &afisare_caracter, &fir2_args)) {
perror("pthread_create");
exit(1);
```

```
}

/* asteptam terminarea celor doua fire de executie */
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
```

Funcții în Windows

Crearea firelor de execuție

Pentru a lansa un nou fir de execuție există funcțiile `CreateThread` i `CreateRemoteThread` (a doua fiind folosită pentru a crea un fir de execuție în cadrul altui proces decât cel curent).

```
HANDLE CreateThread (
LPSECURITY_ATTRIBUTES  lpThreadAttributes,
SIZE_T dwStackSize,
        LPTHREAD_START_ROUTINE lpStartAddress,
LPVOID lpParameter,
DWORD dwCreationFlags,
LPDWORD lpThreadId
);
```

`dwStackSize` reprezintă mărimea inițială a stivei, în bytes. Sistemul rotunjete această valoare la cel mai apropiat multiplu de dimensiunea unei pagini. Dacă parametrul este 0, noul thread va folosi mărimea implicită. `lpStartAddress` este un pointer la funcția ce trebuie executată de către thread. Această funcție are următorul prototip:

```
DWORD WINAPI ThreadProc(
LPVOID lpParameter
);
```

unde `lpParameter` reprezintă datele care sunt pasate firului de execuție la execuție. La fel ca pe Linux, se poate transmite un pointer la o structură, care conține toți parametrii necesari. Rezultatul întors poate fi obținut de un alt thread folosind funcția `GetExitCodeThread`.

Un mic exemplu :

```
HANDLE hthread;
hthread = CreateThread(NULL, 0, ThreadFunc, &dwThreadParam, 0, &dwThreadId);
```

La crearea unui nou fir de execuție parametrii cei mai importanți sunt funcția pe care acesta o va executa şi parametrul care este pasat acesteia.

Handle i identificator

Thread-urile pot fi identificate în sistem în 3 moduri:

- printr-un `HANDLE`, obținut la crearea thread-ului, sau folosind funcția `OpenThread`, căreia i se dă ca parametru identificatorul thread-ului:

```
HANDLE OpenThread(
DWORD dwDesiredAccess,
BOOL bInheritHandle,
DWORD dwThreadId
);
```

- printr-un pseudo-HANDLE, o valoare specială care indică funcțiilor de lucru cu HANDLE-uri că este vorba de HANDLE-ul asociat cu thread-ul curent (obținut, de exemplu, apelând `GetCurrentThread`). Pentru a converti un pseudo-HANDLE într-un HANDLE veritabil, trebuie folosită funcția `DuplicateHandle`. De asemenea, nu are sens să facem `CloseHandle` pe un pseudo-HANDLE. Pe de altă parte, handle-ul obținut cu `DuplicateHandle` trebuie inchis daca nu mai este nevoie de el.
- printr-un identificator de thread, de tipul DWORD, întors la crearea thread-ului, sau obținut folosind `GetCurrentThreadId`. O diferență dintre identificator şi HANDLE este faptul că nu trebuie să ne preocupăm sa *închidem* un identificator, pe când la HANDLE, pentru a evita leak-urile, trebuie să apelăm `CloseHandle`

Handle-ul obținut la crearea unui thread are implicit drepturi de acces nelimitate. El poate fi moştenit sau nu de procesele copil ale procesului curent în funcție de flag-urile specificate la crearea lui. Prin funcția `DuplicateHandle`, se poate crea un nou handle cu mai puține drepturi. Handle-ul este valid până ce este închis, chiar dacă firul de execuție pe care îl reprezintă s-a terminat.

Aşteptarea firelor de execuție

Pe Windows, se poate aştepta terminarea unui fir de execuție folosind aceeaşi funcție ca pentru aşteptarea oricărui obiect de sincronizare:

```
DWORD WINAPI WaitForSingleObject(
     HANDLE hHandle,
     DWORD dwMilliseconds
);
```

Terminarea firelor de execuție

Un fir de execuție se termină în unul din următoarele cazuri :

- el însui apelează funcția `ExitThread` :

```
void ExitThread(DWORD dwExitCode);
```

- funcția asociată firului de execuție execută un `return`.
- un fir de execuție ce deține un handle cu dreptul `THREAD_TERMINATE` asupra firului de execuție, execută un apel `TerminateThread` pe acest handle :

```
BOOL TerminateThread(HANDLE hThread, DWORD dwExitCode);
```

unde `dwExitCode` specifică codul de terminare al threadului.

- sau întregul proces se termină ca urmare a unui apel `ExitProcess` sau `TerminateProcess`.

Pentru aflarea codului de terminare a unui fir de execuție folosim funcția `GetExitCodeThread`, i acest cod poate fi:

- `STILL_ACTIVE` daca firul de execuție nu s-a terminat.
- valoarea întoarsă de funcția asociată firului de execuție.
- valoarea specificată la apelul uneia din funcțiile `TerminateThread`, `TerminateProcess`, `ExitThread` sau `ExitProcess`.

Atenție! Funcțiile `TerminateThread` i `TerminateProcess` nu trebuie folosite decât în cazuri extreme (pentru că nu eliberează resursele folosite de firul de execuție, iar unele resurse pot fi VITALE). Metoda preferată de a termina un fir de execuție este `ExitThread`, sau folosirea unui protocol de oprire între thread-ul care doreşte să închidă un alt thread şi thread-ul care trebuie oprit.

La terminarea ultimului fir de execuție al unui proces se termină i procesul.

```
BOOL GetExitCodeThread(HANDLE hThread, LPDWORD lpExitCode);
```

- `hThread` - handle la firul de execuție în discuție ce trebuie să aibă dreptul de acces `THREAD_QUERY_INFORMATION`.
- `lpExitCode` - pointer la o variabilă în care va fi plasat codul de terminare al firului. Dacă firul nu i-a terminat execuția, această valoare va fi `STILL_ACTIVE`.

  Atentie! Pot apărea probleme dacă firul de execuție returnează chiar `STILL_ACTIVE` (259), i anume aplicația care testează valoarea poate intra într-o buclă infinită.

Dacă funcția se termină cu succes va întoarce o valoare nenulă. Altfel întoarce 0, iar eroarea poate fi aflată folosind `GetLastError`.

Suspend, Resume

```
DWORD SuspendThread(HANDLE hThread);
DWORD ResumeThread(HANDLE hThread);
```

Prin intermediul acestor două funcții un fir de execuție poate suspenda/relua execuția unui alt fir de execuție.

Un fir de execuție suspendat nu mai este planificat pentru a obține timp pe procesor.

Cele doua funcții manipulează un *contor de suspendare* (prin incrementare, respectiv decrementare - în limitele 0 - `MAXIMUM_SUSPEND_COUNT`).

În cazul în care contorul de suspendare este mai mare strict decât 0, firul de execuție este suspendat.

Un fir de execuție poate fi creat în starea suspendat folosind flag-ul `CREATE_SUSPENDED`.

Aceste funcții nu pot fi folosite pentru sincronizare (pentru ca nu controlează punctul în care firul de execuție îi va suspenda execuția), dar sunt utile pentru programe de debug.

Cedarea procesorului

Un fir de execuție poate renunța de bună voie la procesor.

În urma apelului funcției `Sleep` un fir de execuție este suspendat pentru cel puțin o anumită perioadă de timp (`dwMilliseconds`).

```
void Sleep(DWORD dwMilliseconds);
```

Există de asemenea funcția `SleepEx` care este un `Sleep` alertabil (ceea ce înseamnă că se pot prelucra APC-uri - Asynchronous Procedure Call - pe durata execuției lui SleepEx).

Funcția `SwitchToThread` este asemănătoare cu `Sleep` doar că nu este specificat intervalul de timp, astfel firul de execuție renunță doar la timpul pe care îl avea pe procesor în momentul respectiv (time-slice).

```
BOOL SwitchToThread(void);
```

Funcția întoarce `TRUE` dacă procesorul este cedat unui alt thread şi `FALSE` dacă nu există alte thread-uri gata de execuție.

Alte funcții utile

```
HANDLE GetCurrentThread(void);
```

Rezultatul este un pseudohandle pentru firul curent ce nu poate fi folosit decât de firul apelant. Acest handle are maximum de drepturi de acces asupra obiectului pe care îl reprezintă.

```
DWORD GetCurrentThreadId(void);
```

Rezultatul este identificatorul firului de execuție.

```
DWORD GetThreadId(HANDLE Thread);
```

Rezultatul este identificatorul firului ce corespunde handle-ului `Thread`.

Thread Local Storage

Ca i în Linux, i în Windows există un mecanism prin care fiecare fir de execuție să aibă anumite date specifice. Acest mecanism poartă numele de *thread local storage* (TLS). În Windows, pentru a accesa datele din TLS se folosesc indecii asociați acestora (corespunzători cheilor din Linux).

Pentru a crea un nou TLS se apelează funcția :

```
DWORD TlsAlloc(void);
```

Funcția întoarce în caz de succes indexul asociat TLS-ului, prin intermediul căruia fiecare fir de execuție va putea accesa datele specifice.

În caz de eec funcția întoarce valoarea `TLS_OUT_OF_INDEXES`.

Pentru a stoca o nouă valoare într-un TLS se folosete funcția :

```
BOOL TlsSetValue(
DWORD dwTlsIndex,
LPVOID lpTlsValue
);
```

Un thread poate afla valoarea specifică lui dintr-un TLS apelând funcția :

```
LPVOID TlsGetValue(
DWORD dwTlsIndex
);
```

unde `dwTlsIndex` este indexul asociat TLS-ului, alocat cu `TlsAlloc`.

În caz de succes funcția întoarce valoarea stocată în TLS, iar în caz de eec întoarce 0. Dacă data stocată în TLS are valoarea 0 atunci valoarea întoarsă este tot 0, dar `GetLastError` va întoarce `NO_ERROR`. Deci trebuie verificată eroarea întoarsă de `GetLastError`.

Pentru a elibera un index asociat unui TLS se folosete funcția :

```
BOOL TlsFree(
DWORD dwTlsIndex
);
```

unde `dwTlsIndex` este indexul asociat TLS-ului.

Dacă firele de execuție au alocat memorie i au stocat în TLS un pointer la memoria alocată, această funcție nu va face dezalocarea memoriei. Memoria trebuie dezalocată de către fire înainte de apelul lui `TlsFree`.

Fibre de execuție

Windows pune la dispoziție şi o implementare de user-space threads, numite `fibre`. Kernel-ul planifică un singur KLT asociat cu un set de fibre, iar fibrele colaborează pentru a partaja timpul de procesor oferit acestuia. Deşi viteza de execuție este mai bună (pentru context-switch, nu mai este necesară interacțiunea cu kernel-ul), programele scrise folosind fibre pot deveni complexe. Mai multe informații puteți găsi la secțiunea suplimentară dedicată.

Securitate i drepturi de acces

Modelul de securitate Windows NT ne permite să controlăm accesul la obiectele de tip fir de execuție.

Descriptorul de securitate pentru un fir de execuție se poate specifica la apelul uneia dintre funcțiile `CreateProcess`, `CreateProcessAsUser`, `CreateProcessWithLogonW`, `CreateThread` sau `CreateRemoteThread`.

Dacă în locul acestui descriptor este pasată valoarea NULL, firul de execuție va avea un descriptor implicit.

Pentru a obține acest descriptor este folosită funcția `GetSecurityInfo`, iar pentru a-l schimba funcția `SetSecurityInfo`.

Handle-ul întors de funcția `CreateThread` are `THREAD_ALL_ACCESS`. La apelul `GetCurrentThread`, sistemul întoarce un pseudohandle cu maximul de drepturi de acces pe care descriptorul de securitate al firului de execuție îl permite apelantului.

Drepturile de acces pentru un obiect fir de execuție includ drepturile de acces standard : `DELETE`, `READ_CONTROL`, `SYNCHRONIZE`, `WRITE_DAC` i `WRITE_OWNER` la care se adaugă drepturi specifice, pe care le puteți găsi pe MSDN.

Exemplu

Exemplul prezintă crearea a 2 fire de execuție ce vor folosi un TLS.

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

#define NUMAR_FIRE 2

DWORD dwTlsIndex;

VOID ErrorExit (LPTSTR lpszMessage)
{
fprintf(stderr, "%s\n", lpszMessage);
        ExitProcess(1);
}

VOID FolosireTLS(VOID)
{
LPVOID lpvData;

// obtin pointer-ul stocat in TLS pentru firul curent

        lpvData = TlsGetValue(dwTlsIndex);
if ((lpvData == 0) && (GetLastError() != 0))
         ("Eroare la TlsGetValue");

// folosire date stocate

printf("thread %d: lpvData=%lx\n", GetCurrentThreadId(), lpvData);

Sleep(5000);
}

// functia executata de cele doua fire

DWORD WINAPI ThreadFunc(LPVOID)
{
LPVOID lpvData;

// intializare TLS pentru acest thread.

        lpvData = (LPVOID) LocalAlloc(LPTR, 256);
if (! TlsSetValue(dwTlsIndex, lpvData))
```

```
        ("Eroare la TlsSetValue");

printf("thread %d: lpvData=%lx\n", GetCurrentThreadId(), lpvData);

        Sleep(0);

// eliberare memorie alocata dinamic

        lpvData = TlsGetValue(dwTlsIndex);
if (lpvData != 0)
        LocalFree((HLOCAL) lpvData);

return 0;
}

DWORD main(VOID)
{
DWORD IDThread;
HANDLE hThread[NUMAR_FIRE];
int i;

// alocare index TLS

if ((dwTlsIndex = TlsAlloc()) == -1)
        ("Eroare la TlsAlloc");

// creare thread-uri

for (i=0;i<NUMAR_FIRE;i++)
{
       [i] = CreateThread(NULL,              // fara atribute de securitate
0,                                  // utilizare dimensiune implicita pentru stiva
(LPTHREAD_START_ROUTINE) ThreadFunc, // functia executata
NULL,                               // functia nu primeste nici un argument
0,                                  // se folosesc flag-urile implicite
&IDThread);                         // aici va fi stocat identificatorul firului

if (hThread[i] == NULL)
        ("Eroare la CreateThread\n");
}

// asteptare terminare thread-uri

for (i=0;i<NUMAR_FIRE;i++)
               WaitForSingleObject(hThread[i], INFINITE);

// eliberare index TLS

       (dwTlsIndex);

return 0;
}
```

Quiz

Pentru autoevaluare răspundei întrebările din acest quiz.

Exerciții

Warning

Pentru că nu ați parcurs încă noțiunile necesare pentru a sincroniza thread-urile între ele, în cadrul acestui laborator vom folosi apeluri `sleep()` acolo unde e nevoie de sincronizare.

Prezentare

Pentru a urmări mai uor noiunile expuse la începutul laboratorului folosii această prezentare (pdf) (odp).

Linux

Folosiți macro-ul CHECK pentru a verifica valorile întoarse de apelurile de sistem.

1. **(1.5 puncte)** Întreesere thread-uri (directorul `lin/1-shared/` din arhiva de sarcini a laboratorului)
   - Realizaţi un program care creează 2 thread-uri.
   - Thread-urile create vor partaja un descriptor de fiiere, modificat de către fiecare din ele, la momente diferite.
   - Thread-urile afiează mesaje specifice la ieirea standard. Explicai succesiunea mesajelor.
   - Programul principal va atepta încheierea execuiei celor două thread-uri.

     Hint-uri:
     - ◊ Trebuie adăugat codul de creare a thread-urilor i apelul către funcia care întoarce id-ul thread-ului curent.
     - ◊ Consultai seciunile Crearea firelor de execuie i Ateptarea firelor de execuie
2. **(2 puncte)** Thread Specific Data (directorul `lin/2-time/` din arhiva de sarcini a laboratorului)
   - Realizaţi un program care creează 2 thread-uri CPU-bound.
   - Thread-urile calculează suma numerelor prime mai mici decât o limită dată (constanta `MAX` din schelet).
   - Thread-urile vor stoca suma în TSD (Thread Specific Data) i o vor afia după calcularea acesteia.
   - Rezultatul întors de fiecare thread este durata de execuie a calculului.
   - Programul principal va afia timpul total de procesare (suma rezultatelor thread-urilor).
   - Testai programului se va face cu ajutorul comenzii `time` (`time ./time`). Explicaţi diferena dintre durata afiată de program i timpii măsurai folosind `time`.

     Hint-uri:
     - ◊ Trebuie să completai codul de creare / aşteptare a thread-urilor.
     - ◊ Trebuie să obţineţi rezultatul întors de fiecare thread.
     - ◊ Pentru a folosi Thread Specific Data, consultai secţiunea asociată
     - ◊ Pentru a explica diferena dintre timpi, gândii-vă ce ar putea genera overhead în programul respectiv.
     - ◊ Formatul de afişare pentru "long long" atunci când folosiţi printf este "%lld".
3. **(2.5 puncte)** Thread workers (directorul `lin/3-bgrep/` din arhiva de sarcini a laboratorului)
   - Realizaţi un program numit "bgrep" (binary grep), care caută un caracter într-un fişier.
   - Fişierul va fi mapat în întregime în memorie, va fi împărţit în bucăi, şi fiecare bucată va fi atribuită unui thread.

- Rezultatul întors de un thread va fi numărul de apariii ale caracterului în bucata atribuită thread-ului.
- Rezultatul final afiat va fi numărul de apariii ale caracterului în întregul fişier (suma rezultatelor thread-urilor).

Hint-uri:

◊ Trebuie să completați codul de creare / aşteptare a thread-urilor.
◊ Calculați limitele bucăților de fişier pentru fiecare thread.
◊ Adunați numărul de apariții întors de fiecare thread.
◊ Formula pentru chunk_size este corectă; încercați câteva exemple pentru a vă convinge. De ce nu am împărțit direct `len` la `NUM_THREADS`?

Windows

1. **(1.5 puncte)** Întreesere thread-uri (directorul `win/1-shared/` din arhiva de sarcini a laboratorului)
   - Creați 2 thread-uri: primul va aloca o zonă de memorie i va scrie în ea, iar al doilea va citi din ea.

   Hint-uri:

   ◊ Trebuie să completai în schelet codul de creare al thread-urilor.
   ◊ Consultai secțiunea dedicată creării thread-urilor
   ◊ Observai că alocările de memorie sunt partajate între thread-uri.
2. **(1.5 puncte)** Terminarea unui thread i întoarcerea unei valori (directorul `win/2-display/` din arhiva de sarcini a laboratorului)
   - Creați un thread care primeşte ca parametru o structură, definită în schelet (`struct ThreadParam`).
   - Thread-ul va afia vectorul din structura primită ca parametru, va calcula suma numerelor din acesta, i va întoarce suma obinută.
   - Programul principal afiează suma întoarsă de vector.

   Hint-uri:

   ◊ Folosiți funcția `DisplayParam` (din schelet) pentru a afişa vectorul.
   ◊ Pentru a verifica codul întors de un thread, revedeți secțiunea relevantă din laborator.
3. **(2 puncte)** Thread Local Storage (directorul `win/3-sum/` din arhiva de sarcini a laboratorului)
   - Creați 2 thread-uri care vor calcula suma numerelor dintr-un vector (primul thread, suma pentru prima jumătate; al doilea thread, suma pentru a doua jumătate).
   - Suma va fi stocată in TLS (Thread Local Storage).
   - Fiecare thread întoarce suma pentru jumătatea corespunzătoare, iar programul principal afiează suma rezultatelor.

Soluții

Soluții exerciții laborator 8