

# Laborator 09 - Thread-uri Windows

## Materiale ajutătoare

- [lab09-slides.pdf](#)
- [lab09-refcard.pdf](#)

## Nice to read

- WSP4 - Chapter 7, Threads and Scheduling

## Crearea firelor de execuție

Pentru a lansa un nou fir de execuție, există funcțiile [CreateThread](#) și [CreateRemoteThread](#) (a doua fiind folosită pentru a crea un fir de execuție în cadrul altui proces decât cel curent).

|  |   |
|--|---|
| <pre>HANDLE CreateThread (     LPSECURITY_ATTRIBUTES lpThreadAttributes,     SIZE_T dwStackSize,     LPTHREAD_START_ROUTINE lpStartAddress,     LPVOID lpParameter,     DWORD dwCreationFlags,     LPDWORD lpThreadId );</pre> | <pre>hthread = CreateThread(     NULL,     0,     ThreadFunc,     &amp;dwThreadParam,     0,     &amp;dwThreadId );</pre> |
|--|---|

Parametrul `dwStackSize` reprezintă mărimea inițială a stivei (în octeți). Sistemul rotunjește această valoare la cel mai apropiat multiplu de dimensiunea unei pagini. Dacă parametrul este 0, noul fir de execuție va folosi mărimea implicită (1 MB). `lpStartAddress` este un pointer la funcția ce trebuie executată de către firul de execuție. Această funcție are următorul prototip:

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

unde `lpParameter` reprezintă datele care sunt pasate firului în momentul execuției. 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 fir de execuție folosind funcția [GetExitCodeThread](#).

## Handle și identificador

Firele de execuție pot fi identificate în sistem în 3 moduri:

- printr-un HANDLE, obținut la crearea firului de execuție, sau folosind funcția [OpenThread](#), căreia i se dă ca parametru identificadorul firului de execuție:

```
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 firul de execuție 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 închis dacă nu mai este nevoie de el.
- printr-un identificador al firului de execuție, de tipul DWORD, întors la crearea firului, sau obținut folosind [GetCurrentThreadId](#). O diferență dintre identificador și HANDLE este faptul că nu trebuie să ne preocupăm să închidem un

identificator, pe când la HANDLE, pentru a evita *leak*-urile, trebuie să apelăm [CloseHandle](#).

Handle-ul obținut la crearea unui fir de execuție are implicit drepturi de acces nelimitate. El poate fi moștenit (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 când este închis, chiar dacă firul de execuție pe care îl reprezintă s-a terminat.

## Așteptarea firelor de execuție

Pe Windows, se poate aștepta terminarea unui fir de execuție folosind aceeași funcție ca pentru așteptarea oricărui obiect de sincronizare [WaitForSingleObject](#):

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

## Terminarea firelor de execuție

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

- el însuși 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  
);
- sau întregul proces se termină ca urmare a unui apel [ExitProcess](#) sau [TerminateProcess](#).

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

**Atenție!** Funcțiile [TerminateThread](#) și [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 firul de execuție care dorește să închidă un alt fir de execuție și firul care trebuie oprit.

Pentru aflarea codului de terminare a unui fir de execuție, folosim funcția [GetExitCodeThread](#).

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

- hThread - handle al firului de execuție 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 și-a terminat execuția, această valoare va fi STILL\_ACTIVE.

**Atenție!** Pot apărea probleme dacă firul de execuție returnează STILL\_ACTIVE (259), și anume aplicația care testează valoarea poate intra într-o buclă infinită.

## Suspend, Resume

## Cedarea procesorului

### Alte funcții utile

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

Rezultatul este un pseudo-handle 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 hThread);
```

Rezultatul este identificatorul firului ce corespunde handle-ului hThread.

## Thread Local Storage

Ca și în Linux, și 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 indecșii asociați acestora (corespunzători cheilor din Linux).

Pentru a crea un nou TLS, se apelează funcția [TlsAlloc](#):

```
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. Valoarea stocată în TLS este inițializată cu 0. În caz de eșec, funcția întoarce valoarea `TLS_OUT_OF_INDEXES`.

Pentru a stoca o nouă valoare într-un TLS, se folosește funcția [TlsSetValue](#):

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

Un fir de execuție poate afla valoarea specifică lui dintr-un TLS apelând funcția [TlsGetValue](#):

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

În caz de succes, funcția întoarce valoarea stocată în TLS, iar în caz de eșec, 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 folosește funcția [TlsFree](#):

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

Dacă firele de execuție au alocat memorie și au stocat în TLS un pointer la memoria alocată, această funcție nu va face dealocarea memoriei. Memoria trebuie dealocată de către fire înainte de apelul lui [TlsFree](#).

## Exemplu

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

### [ThreadTLS.c](#)

```
#include <stdio.h>  
#include <windows.h>  
#include "utils.h"  
  
#define NO_THREADS 2  
  
DWORD dwTlsIndex;
```

```

VOID TLSUse(VOID)
{
    LPVOID lpvData;

    /* get the pointer from TLS for current thread */
    lpvData = TlsGetValue(dwTlsIndex);
    DIE((lpvData == 0) && (GetLastError() != 0), "TlsGetValue");

    /* use this data */
    <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("thread %d: get
lpvData=%p\n", GetCurrentThreadId(), lpvData);

    Sleep(5000);
}

/* function executed by the threads */
DWORD WINAPI ThreadFunc(LPVOID lpParameter)
{
    LPVOID lpvData;
    DWORD dwReturn;

    /* TLS init for the current thread */
    lpvData = (LPVOID) LocalAlloc(LPTR, 256);
    DIE(lpvData == NULL, "LocalAlloc");

    dwReturn = TlsSetValue(dwTlsIndex, lpvData);
    DIE(dwReturn == FALSE, "TlsSetValue");

    <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("thread %d: set
lpvData=%p\n", GetCurrentThreadId(), lpvData);

    TLSUse();

    /* free dinamic memory */
    lpvData = TlsGetValue(dwTlsIndex);
    DIE((lpvData == 0) && (GetLastError() != 0), "TlsGetValue");

    LocalFree((HLOCAL) lpvData);

    return 0;
}

DWORD main(VOID)
{
    DWORD IDThread, dwReturn;
    HANDLE hThread[NO_THREADS];
    int i;

    /* allocate TLS index */
    dwTlsIndex = TlsAlloc();
    DIE(dwTlsIndex == TLS_OUT_OF_INDEXES, "Eroare la TlsAlloc");

    /* create threads */
    for (i = 0; i < NO_THREADS; i++) {
        hThread[i] = CreateThread(NULL, /* default security attributes */
                                0, /* default stack size */
                                (LPTHREAD_START_ROUTINE) ThreadFunc, /* routine to execute */
                                NULL, /* no thread parameter */
                                0, /* immediately run the thread */
                                &IDThread); /* thread id */
        DIE(hThread[i] == NULL, "CreateThread");
    }

    /* wait for threads completion */
    for (i = 0; i < NO_THREADS; i++) {
        dwReturn = WaitForSingleObject(hThread[i], INFINITE);
        DIE(dwReturn == WAIT_FAILED, "WaitForSingleObject");
    }

    /* free TLS index */
    dwReturn = TlsFree(dwTlsIndex);
    DIE(dwReturn == FALSE, "TlsFree");

    return 0;
}

```

## Fibre de execuție

Windows pune la dispoziție și o implementare de User-space Threads, numite **fibre**. Kernel-ul planifică un singur Kernel Level Thread (KLT) asociat cu un set de fibre, iar fibrele colaborează pentru a partaja timpul de procesor oferit acestuia. Deși 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 în cadrul [secțiunii suplimentare dedicate](#).

## Securitate și 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 de securitate implicit.

Pentru a obține acest descriptor este folosită funcția [GetSecurityInfo](#), iar pentru a-l schimba funcția [SetSecurityInfo](#).

```
DWORD WINAPI GetSecurityInfo(
    HANDLE handle,
    SE_OBJECT_TYPE ObjectType,
    SECURITY_INFORMATION SecurityInfo,
    PSID *ppsidOwner,
    PSID *ppsidGroup,
    PACL *ppDacl,
    PACL *ppSacl,
    PSECURITY_DESCRIPTOR *ppSecurityDescriptor
);

DWORD WINAPI SetSecurityInfo(
    HANDLE handle,
    SE_OBJECT_TYPE ObjectType,
    SECURITY_INFORMATION SecurityInfo,
    PSID psidOwner,
    PSID psidGroup,
    PACL pDacl,
    PACL pSacl
);
```

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

Drepturile de acces pentru un obiect de tip fir de execuție includ drepturile de acces standard : `DELETE`, `READ_CONTROL`, `SYNCHRONIZE`, `WRITE_DAC` și `WRITE_OWNER` la care se adaugă drepturi specifice, pe care le puteți găsi pe [MSDN](#).

## Sincronizarea firelor de execuție

Pentru sincronizarea firelor de execuție avem la dispoziție:

- **mutex**: [POSIX](#), [Win32](#)
- **semafoare**: [POSIX](#), [Win32](#)
- **secțiuni critice** (excludere mutuală în cadrul aceluiași proces): [Win32](#)
- **variabile de condiție**: [POSIX](#), [Win32](#) (începând cu Vista)
- **evenimente**: [Win32](#)
- **timere**: Win32.

Standardul POSIX specifică funcții de sincronizare pentru fiecare tip de obiect de sincronizare. API-ul Win32, fiind controlat de o singură entitate, permite ca toate obiectele de sincronizare să poată fi utilizate cu funcțiile standard de sincronizare: [WaitForSingleObject](#), [WaitForMultipleObjects](#) sau [SignalObjectAndWait](#).

Obiectele de sincronizare [Semaphore](#), [Mutex](#), [Event](#) și [WaitableTimer](#) pot fi folosite atât pentru sincronizarea proceselor, cât și a firelor de execuție. Ele au fost deja introduse în laboratoarele trecute.

În Windows mai există un mecanism de sincronizare care este disponibil doar pentru firele de execuție ale **aceluiași proces**, și anume [CriticalSection](#). Se recomandă folosirea [CriticalSection](#) pentru excluderea mutuală a firelor de execuție ale aceluiași proces, fiind mai **eficient** decât [Mutex](#) sau [Semaphore](#).

Win32 API pune la dispoziție un mecanism de acces sincronizat la variabile partajate între fire de execuție prin intermediul funcțiilor **interlocked** ([Interlocked Variable Access](#)), precum și operații atomice de inserare și ștergere în liste simplu înlănțuite ([Interlocked Singly Linked Lists](#)).

## Mutex Win32

Subiectul a fost tratat în laboratorul de [comunicație inter-proces](#).

```
/* creează un mutex */
HANDLE CreateMutex(
    LPSECURITY_ATTRIBUTES lpMutexAttributes,
    BOOL bInitialOwner,
    LPCTSTR lpName
);

/* deschide un mutex (identificat prin nume) */
HANDLE OpenMutex(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    LPCTSTR lpName
);

/* eliberează un mutex ocupat */
BOOL ReleaseMutex(
    HANDLE hMutex
);
```

## Semafor Win32

Subiectul a fost tratat în laboratorul de [comunicație inter-proces](#).

```
/* creează un semafor */
HANDLE CreateSemaphore(
    LPSECURITY_ATTRIBUTES semattr,
    LONG initial_count,
    LONG maximum_count,
    LPCTSTR name
);

/* deschide un semafor existent */
HANDLE OpenSemaphore(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    LPCTSTR name
);

/* incrementare contor semafor cu 'lReleaseCount' */
BOOL ReleaseSemaphore(
    HANDLE hSemaphore,
    LONG lReleaseCount,
    LPLONG lpPreviousCount
);
```

## Secțiune critică

Obiectele [CriticalSection](#) sunt echivalente mutex-urilor POSIX de tip RECURSIVE. Acestea sunt folosite pentru excluderea mutuală a accesului firelor de execuție ale **aceluiași proces** la o secțiune critică de cod care conține operații asupra unor date partajate. Un singur fir de execuție va fi activ la un moment dat în interiorul secțiunii critice. Dacă mai multe fire așteaptă să intre, nu este garantată ordinea lor de intrare, totuși sistemul va fi echitabil față de toate.

Operațiile care se pot efectua asupra unei secțiuni critice sunt: intrarea, intrarea neblokantă, ieșirea din secțiunea critică, inițializarea și distrugerea.

Pentru serializarea accesului la o secțiune critică, fiecare fir de execuție va trebui să intre într-un obiect `CriticalSection` la începutul secțiunii și să-l părăsească la sfârșitul ei. În acest fel, dacă două fire de execuție încearcă să intre în `CriticalSection` simultan, doar unul dintre ele va reuși, și își va continua execuția în interiorul secțiunii critice, iar celălalt se va bloca pînă când obiectul `CriticalSection` va fi părăsit de primul fir. Așadar, la sfârșitul secțiunii, primul fir trebuie să părăsească obiectul `CriticalSection`, permițându-i celuiilalt intrarea.

Pentru excluderea mutuală se pot folosi atât obiecte [Mutex](#), cât și obiecte [CriticalSection](#); dacă sincronizarea trebuie făcută doar între firele de execuție ale aceluiași proces este recomandată folosirea `CriticalSection`, fiind un mecanism mai **eficient**. Operația de intrare în `CriticalSection` se traduce într-o singură instrucțiune de asamblare de tip

*test-and-set-lock* (TSL). `CriticalSection` este **echivalentul** `futex`-ului din Linux.

### Inițializarea/distrugerea unei secțiuni critice

Alocarea memoriei pentru o secțiune critică se face prin declararea unui obiect `CRITICAL_SECTION`. Acesta nu va putea fi folosit, totuși, înainte de a fi inițializat ([InitializeCriticalSection](#), [InitializeCriticalSectionAndSpinCount](#), [SetCriticalSectionSpinCount](#), [DeleteCriticalSection](#)).

```
void InitializeCriticalSection(
    LPCRITICAL_SECTION pcrit_sect
);

BOOL InitializeCriticalSectionAndSpinCount(
    LPCRITICAL_SECTION pcrit_sect,
    DWORD dwSpinCount
);

DWORD SetCriticalSectionSpinCount(
    LPCRITICAL_SECTION pcrit_sect,
    DWORD dwSpinCount
);

void DeleteCriticalSection(
    LPCRITICAL_SECTION pcrit_sect
);
```

**Atenție!** Un obiect `CRITICAL_SECTION` nu poate fi copiat sau modificat după inițializare. De asemenea, un obiect `CRITICAL_SECTION` nu trebuie inițializat de două ori, în caz contrar, comportamentul său fiind nedefinit.

**Contorul de spin** (Spin Count) are sens doar pe sistemele **multiprocesor** (SMP) (este ignorat pe sisteme uniprocessor). Contorul de spin reprezintă numărul de cicluri pe care îl petrece un fir de execuție pe un procesor în busy-waiting, înainte de a-și suspenda execuția la un semafor asociat secțiunii critice, în așteptarea eliberării acesteia. Scopul așteptării unui număr de cicluri în busy-waiting este evitarea blocării la semafor în cazul în care secțiunea critică se eliberează în intervalul respectiv, deoarece blocarea la semafor are impact asupra performanțelor. Folosirea contorului de spin este recomandată mai ales în cazul unei secțiuni critice scurte, accesate foarte des.

### Utilizarea secțiunilor critice

Secțiunile critice Windows au comportamentul mutex-urilor POSIX de tip `RECURSIVE`. Un fir de execuție care se află deja în secțiunea critică nu se va bloca dacă apelează din nou [EnterCriticalSection](#), însă va trebui să părăsească secțiunea critică de un număr de ori **egal** cu cel al ocupărilor, pentru a o elibera.

Pentru a încerca intrarea într-o secțiune critică fără a se bloca, un fir de execuție trebuie să apeleze [TryEnterCriticalSection](#).

```
void EnterCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection
);

void LeaveCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection
);

/* pentru TryEnterCriticalSection _WIN32_WINNT >= 0x0400 înainte de include <windows.h> */

#define _WIN32_WINNT 0x0400
#include <windows.h>
BOOL TryEnterCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection
);
```

În cadrul unui fir de execuție, numărul apelurilor [LeaveCriticalSection](#) trebuie să fie **egal** cu numărul apelurilor [EnterCriticalSection](#), pentru a elibera în final secțiunea critică. Dacă un fir de execuție care nu a intrat în secțiunea critică apelează [LeaveCriticalSection](#), se va produce o eroare care va face ca firele care au apelat [EnterCriticalSection](#) să aștepte pentru o perioadă nedefinită de timp.

### Exemplu secțiuni critice

```
/* global critical section */
CRITICAL_SECTION CriticalSection;

DWORD ThreadProc(LPVOID *param)
{
```

```

/* only one thread enters the critical section, the rest are blocked */
EnterCriticalSection(&CriticalSection);

/* use of protected data */

/* leaves the critical section, allowing another thread to enter */
LeaveCriticalSection(&CriticalSection);
}

int main()
{
/* initialize only one time */
InitializeCriticalSection(&CriticalSection);

/* the threads execution ... */

DeleteCriticalSection(&CriticalSection);

return 0;
}

```

## Evenimente

Evenimentele reprezintă un mecanism prin care un fir de execuție poate semnaliza unul sau mai multe fire că o anumită condiție este îndeplinită. Ce e important este faptul că pot fi deblocate mai multe fire de execuție prin semnalarea unui singur eveniment.

Evenimentele sunt de două tipuri, în funcție de modul în care sunt resetate:

- resetare manuală - după alertarea mai multor fire de execuție, evenimentul trebuie resetat
- resetare automată - după alertarea unui singur fir de execuție, evenimentul se resetează automat

Un eveniment este creat folosind funcția [CreateEvent](#):

|   |   |
|---|---|
| HANDLE WINAPI<br>CreateEvent(<br>LPSECURITY_ATTRIBUTES,<br>BOOL<br>bManualReset,<br>BOOL<br>bInitialState,<br>LPCTSTR<br>lpName<br>); | hEvent =<br>CreateEvent(<br>NULL,<br>TRUE, /*<br>Manual Reset<br>*/<br>FALSE, /*<br>Non-signaled<br>state */<br>NULL /*<br>Private<br>variable */<br>); |
|---|---|

Pentru a controla un eveniment se folosesc funcțiile:

- [SetEvent](#) - pentru semnalizarea evenimentului. Dacă evenimentul este de tip auto-reset, atunci **un singur** fir de execuție va fi trezit, iar evenimentul se resetează automat. Dacă evenimentul este de tip manual-reset, atunci evenimentul rămâne semnalizat până când un fir de execuție apelează [ResetEvent](#). Altfel, orice fir de execuție care încearcă să aștepte pe eveniment va fi automat deblocat.
- [ResetEvent](#) - asigură trecerea evenimentului în starea non-signaled. Se utilizează împreună cu un eveniment de tip manual-reset.
- [PulseEvent](#) - deblochează toate firele de execuție care așteaptă la un eveniment de tip manual-reset, iar evenimentul este apoi resetat. Dacă funcția este folosită în conjuncție cu un eveniment auto-reset, atunci va debloca un singur fir de execuție.

## Operații atomice cu variabile partajate (Interlocked Variable Access)

Funcțiile **interlocked** pun la dispoziție un mecanism de sincronizare a accesului la variabile partajate între mai multe fire de execuție. Funcțiile pot fi apelate de fire de execuție ale unor procese diferite, pentru variabile aflate într-un spațiu de memorie partajată. Funcțiile interlocked reprezintă cel mai simplu mod de evitare a race-ului care apare când două fire de



execuție modifică aceeași variabilă.

Operațiile atomice asupra variabilelor partajate:

- **incrementare / decrementare** (ambele funcții întorc noua valoare)

```
LONG InterlockedIncrement(
    LONG volatile *lpAddend
);

LONG InterlockedDecrement(
    LONG volatile *lpDecend
);
```

- **atribuirea** atomică a unei valori unei variabile partajate (primele două funcții întorc vechea valoare)

```
LONG InterlockedExchange(
    LONG volatile *Target,
    LONG Value
);

LONG InterlockedExchangeAdd(
    LPLONG volatile Addend,
    LONG Value
);

PVOID InterlockedExchangePointer(
    PVOID volatile *Target,
    PVOID Value
);
```

- **atribuirea** atomică după **testarea** valorii variabilei partajate

```
LONG InterlockedCompareExchange(
    LONG volatile * dest,
    LONG exchange,
    LONG comp
);

PVOID InterlockedCompareExchangePointer(
    PVOID volatile * dest,
    PVOID exchange,
    PVOID comp
);
```

[InterlockedCompareExchange](#) va compara `dest` cu `comp`; dacă sunt egale, îi va atribui lui `dest` valoarea `exchange`. Testul și atribuirea vor fi executate într-o singură operație **atomică**. Pentru variabile de tip pointer se va folosi [InterlockedCompareExchangePointer](#). Comportamentul este echivalent cu:

```
atomicly_do {
    tmp = *dest; // execută atomic tot blocul următor
    if (tmp == comp) { // copiază valoarea din *dest
        *dest = exchange; // dacă e egală cu valoarea lui 'comp'
    } // atunci scrie valoarea 'exchange' în *dest
}
```

## Windows Thread Pooling

Programele cu un număr mare de fire de execuție pot aduce probleme de performanță dincolo de cele de locking:

- Fiecare fir de execuție are o stivă proprie (default 1MB). Astfel, 1000 de fire vor consuma 1GB de spațiu virtual.
- Context-switch-urile între fire de execuție pot cauza page-fault-uri la accesarea stivei.
- Crearea și terminarea firelor de execuție presupun calcule suplimentare.

Pentru a facilita dezvoltarea de aplicații eficiente bazate pe fire de execuție, sistemul de operare Windows pune la dispoziție mecanismul **thread pooling**. Utilizarea acestuia este benefică în cazul unei aplicații bazată pe fire de execuție care au de îndeplinit taskuri relativ scurte. Prin utilizarea thread pooling, fiecare task de efectuat va fi atribuit unui fir de execuție din pool (un task este o procedură executată de un fir de execuție din thread pool).

Există două modalități prin care o aplicație poate specifica task-urile pe care le dorește executate de fire de execuție din thread pool:

- se pot adăuga taskuri ce vor fi executate **imediat** ce se eliberează un fir de execuție din thread pool
- se pot adăuga operații de așteptare care au asociată o funcție callback ce urmează a fi executată la sfârșitul unui **timeout** de unul dintre firele de execuție din thread pool. Din această categorie fac parte operațiile de așteptare a terminării unei intrări/ieșiri asincrone, operațiile de așteptare a expirării unui Timer-Queue Timer și funcțiile de așteptare înregistrate.

Dacă vreuna dintre funcțiile executate într-un thread-pool apelează `TerminateThread`, comportamentul nu este definit.

Un exemplu practic pentru Windows ThreadPools, ce folosește noul API, se găsește [aici](#).

## Adăugarea de taskuri la thread pool

### Așteptarea unei operații de intrare/ieșire asincrone

Pentru a adăuga la *thread pool* un task care se va executa la finalul unei operații de intrare/ieșire asincrone pe un anumit *file handle*, se va apela funcția:

```
// înregistrează o funcție ce va fi chemată când se încheie o
// operație de IO asincron pe fișierul identificat prin FileHandle.
// Pot fi înregistrate mai multe funcții și vor fi chemate toate
// când se încheie operația IO asincronă. Ordinea în care sunt apelate
// nu este specificată.
BOOL BindIoCompletionCallback(
    HANDLE FileHandle,
    LPOVERLAPPED_COMPLETION_ROUTINE Function,
    ULONG Flags
);

// semnătura funcției înregistrate să fie executată la încheierea operației AIO
VOID CALLBACK FileIOCompletionRoutine(
    DWORD dwErrorCode,
    DWORD dwNumberOfBytesTransferred,
    LPOVERLAPPED lpOverlapped
);
```

### Adăugarea unui task pentru execuție imediată

Pentru a adăuga la *thread pool* un task care să fie executat imediat se va apela funcția:

```
BOOL QueueUserWorkItem(
    LPTHREAD_START_ROUTINE Function, // funcția de executat
    PVOID Context, // pointer ce va fi pasat funcției ca argument
    ULONG Flags); // tipul rutinei (IO, NON-IO, funcția așteaptă mult, etc.)

// Semnătura funcției e identică cu semnătura funcțiilor executate cu CreateThread
DWORD WINAPI ThreadProc(
    LPVOID param
);
```

## Timer Queues

Obiectele *TimerQueue* reprezintă **cozi de timere**. Ele conțin obiecte *Timer-Queue Timer* care au asociată o funcție *callback*, ce va fi executată de un fir de execuție din *thread pool* la expirarea timerului.

### Crearea/distrugerea unei cozi de timere

```
#define _WIN32_WINNT 0x0500
#include <windows.h>

HANDLE CreateTimerQueue(void);

// marchează coada pentru ștergere, dar *NU* așteaptă
// ca toate callbackurile asociate cozii să se termine
BOOL DeleteTimerQueue(
    HANDLE TimerQueue
);

/**
 * CompletionEvent = NULL - marchează coada pentru ștergere și iese imediat (ca DeleteTimerQueue)
 * CompletionEvent = INVALID_HANDLE_VALUE - funcția așteaptă să se încheie toate callbackurile.
 * CompletionEvent = un handle de tip Event - un obiect Event care va fi
 * trecut în starea SIGNALED când se încheie toate callbackurile.
```

```

*/
BOOL DeleteTimerQueueEx(
    HANDLE TimerQueue,
    HANDLE CompletionEvent
);

```

## Crearea unui timer

Pentru crearea unui timer se va apela funcția:

```

BOOL CreateTimerQueueTimer(
    PHANDLE phNewTimer, // aici întoarce un HANDLE la timerul nou creat
    HANDLE TimerQueue, // coada la care este adăugat timerul.
    // Dacă e NULL se folosește o coadă implicită.
    WAITORTIMERCALLBACK Callback, // callback de executat
    PVOID Parameter, // parametru trimis callbackului
    DWORD DueTime, // timerul va expira prima dată după 'DueTime' milisec.
    DWORD Period, // apoi timerul va expira periodic după 'Period' milisec.
    ULONG Flags // tipul callbackului: IO/NonIO, EXECUTEONLYONCE, ș.a.
);

// semnătura unui callback
VOID WaitOrTimerCallback(
    PVOID lpParameter,
    BOOLEAN TimerOrWaitFired
);

// modificarea timpului de expirare al unui timer
BOOL ChangeTimerQueueTimer(
    HANDLE TimerQueue, // coada la care este adăugat timerul.
    // Dacă e NULL se folosește o coadă implicită.
    HANDLE Timer, // HANDLE la timerul de modificat
    ULONG DueTime, // timerul va expira prima dată după 'DueTime' milisec.
    ULONG Period // apoi timerul va expira periodic după 'Period' milisec.
);

// dezactivarea unui timer
BOOL CancelTimerQueueTimer(
    HANDLE TimerQueue,
    HANDLE Timer
);

// dezactivarea ȘI distrugerea unui timer.
// CompletionEvent e similar cu cel din DeleteTimerQueueEx.
BOOL DeleteTimerQueueTimer(
    HANDLE TimerQueue,
    HANDLE Timer,
    HANDLE CompletionEvent
);

```

## Registered Wait Functions

Funcțiile de așteptare înregistrate sunt funcții de așteptare executate de un fir de execuție din *thread pool*. În momentul în care obiectul de sincronizare după care se așteaptă trece în starea *signaled*, se va executa rutina *callback* asociată funcției de așteptare înregistrate, de un fir de execuție din *thread pool*. În mod implicit, funcțiile de așteptare înregistrate se **rearmează automat** și rutinele *callback* sunt executate de fiecare dată când obiectul de sincronizare după care se așteaptă trece în starea ***signaled***, sau intervalul de timeout **expiră**. Acest lucru se repetă până când înregistrarea funcției de așteptare este anulată. Se poate seta, însă, ca funcția de așteptare înregistrată să se execute **o singură dată**.

## Înregistrarea unei funcții de așteptare

Pentru înregistrarea în *thread pool* a unei funcții de așteptare se va apela funcția:

```

BOOL RegisterWaitForSingleObject(
    PHANDLE phNewWaitObject,
    HANDLE hObject,
    WAITORTIMERCALLBACK Callback,
    PVOID Context,
    ULONG dwMilliseconds,
    ULONG dwFlags
);

```

De fiecare dată când *hObject* trece în starea *signaled*, și la fiecare *dwMilliseconds*, rutina *Callback* va fi executată cu parametrul *Context*, de un fir de execuție din *thread pool*. Rutina *Callback* trebuie să nu apeleze *TerminateThread* și să aibă următoarea semnătură:

```
VOID CALLBACK WaitOrTimerCallback(  
    PVOID lpParameter,  
    BOOLEAN TimerOrWaitFired  
);
```

Parametrul `TimerOrWaitFired` va specifica dacă execuția rutinei `Callback` s-a declanșat în urma trecerii în starea **signaled** a obiectului de sincronizare, sau în urma **expirării** intervalului de timeout specificat.

Prin intermediul parametrului `dwFlags` se pot transmite caracteristici ale firului de execuție care va executa rutina `Callback`, precum și dacă funcția de așteptare trebuie să se execute doar o singură dată. Funcția va întoarce, prin parametrul `phNewWaitObject`, un *handle* ce va fi folosit pentru deînregistrarea funcției de așteptare.

### Deînregistrarea unei funcții de așteptare

Pentru a anula înregistrarea unei funcții de așteptare se va apela una dintre funcțiile:

```
BOOL UnregisterWait (HANDLE WaitHandle);  
BOOL UnregisterWaitEx(HANDLE WaitHandle, HANDLE CompletionEvent);
```

Orice funcție de așteptare înregistrată va trebui deînregistrată prin apelul uneia dintre funcțiile de mai sus.

Funcția `UnregisterWaitEx` va semnaliza *event*-ul `CompletionEvent` în cazul în care se termină cu succes și rutina de *callback* s-a terminat cu succes. Dacă valoarea lui `CompletionEvent` nu este `NULL`, atunci funcția va aștepta finalizarea operației de așteptare și terminarea rutinei asociate.

## Exerciții

În rezolvarea laboratorului folosiți arhiva de sarcini [lab09-tasks.zip](#)

Pentru a deschide proiectul VS conținând exercițiile, deschideți fișierul `lab09.sln`.

### 1. (1 punct) Threading și priorități

- Încărcați proiectul `1-threading` și setați-l ca *Startup Project*
- Compilați și rulați programul. Câte fire de execuție există în total?
- Lansați `Process Explorer` (check Desktop) și verificați răspunsul de la întrebarea de mai sus.
  - Hint: View Select Columns Process Performance Threads
- Ce prioritate are procesul `threading.exe`?
  - Hint: View Select Columns Process Performance Base Priority
- Experimentați schimbând prioritatea procesului. Pentru ce caz procesul `threading.exe` primește mai mult timp de procesor?
  - Hint: click-dreapta pe numele procesului Set Priority
- **Atenție** - dacă setați ca prioritate `real-time`, cel mai probabil vi se va bloca mașina virtuală. De ce credeți că se întâmplă asta?

### 2. (1.5 puncte) Thread debugging

- Deschideți sursa `2-debug.c` din proiectul `2-debug`
- Completați funcția `StartThread` pentru a implementa crearea unui fir de execuție.
  - Hint: Urmăriți în cod secțiunea marcată cu *TODO*
- Compilați și rulați sursa. De ce se blochează?
  - Rezolvați problema (soluția nu implică comentarea funcției `Sleep`)

### 3. (2 puncte) interlocked

- Deschideți sursa `interlocked.c` din proiectul `3-interlocked`
- 1. Creați `THREAD_NO` fire de execuție, care incrementează circular o variabilă (când se ajunge la o limită se resetează la 0).
  - Hint: Folosiți [Interlocked Variables](#) deoarece mecanismul e mai rapid decât o incrementare normală protejată cu `Mutex` sau `CRITICAL_SECTION`.
  - Incrementarea circulară se va face în funcția `thread_function`.
    - Hints:
      - Folosiți `InterlockedCompareExchange`
      - Urmăriți comentariile cu *TODO 1*
    - **Observați vreo problemă cu această abordare?**
- 2. Comparați timpul de execuție al programului precedent în cazul în care se folosește un mutex care să sincronizeze accesul la variabila `counter`
  - Completați funcția `thread_function_mutex ( TODO 2 )`
  - Pentru ambele implementări, **decomentați** apelul funcției `Sleep`
- Cum explicați diferența de timp?

### 4. (2 puncte) TLS

- Deschideți sursa `tls.c` din proiectul `4-tls`.

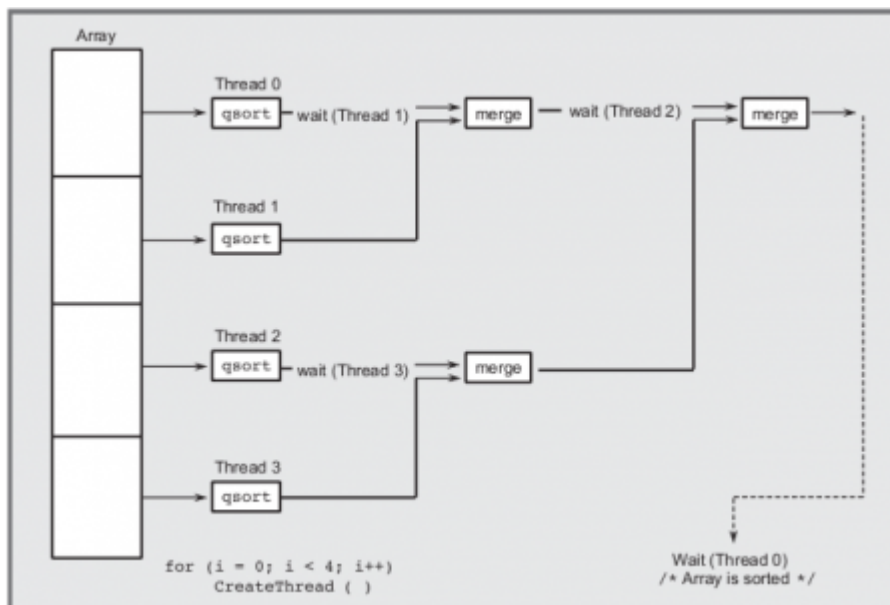
- Dorim să simulăm o implementare a funcției `perror`.
  - Pentru aceasta vom avea variabila globală `myErrno`, dar cu valori specifice (diferite) pentru fiecare fir de execuție.
  - **Hints:**
    - Urmăriți comentariile marcate cu *TODO*
    - Revedeți secțiunea despre [TLS](#)
5. (2 puncte) TimerQueue
- Deschideți sursa `timer.c` din proiectul 5-timer.
  - Creați un Timer-Queue `Timer`, a cărui rutină *callback* să fie declanșată de exact 3 ori, din secundă în secundă. După 3 declanșări se va dezactiva timerul și se vor distruge toate resursele create.
  - **Hints:**
    - Trebuie să sincronizați rutina *timer*-ului cu funcția *main* care va dezactiva timer-ul; pentru aceasta puteți folosi orice mecanism de semnalizare: mutex, semafor, event
6. (2 puncte) Barrier
- Deschideți sursa `barrier.c` din proiectul 6-Barrier.
  - Implementați o barieră reutilizabilă folosind un mutex și o variabilă de tip eveniment.
  - Bariera va fi reprezentată prin structura: `typedef struct {`

```

HANDLE hGuard;          /* mutex to protect internal variable access */
HANDLE hEvent;          /* auto-resatable event */
DWORD dwCount;          /* number of threads to have reached the barrier */
DWORD dwThreshold;     /* barrier limit */
}THRESHOLD_BARRIER, *THB_OBJECT;
```
  - Completați funcțiile de lucru cu bariera pentru a obține funcționalitatea dorită.
  - Pentru ce folosiți mutexul? Dar variabila de tip eveniment.
  - Hints:
    - Completați comentariile marcate cu *TODO*
    - De ce ar fi nevoie de mutex?
    - De ce ar fi nevoie de event? Cum semnalati toate firele de execuție care așteaptă?
    - Pentru a putea semnaliza un semafor și a aștepta la un alt obiect de sincronizare în același timp, puteți folosi funcția [SignalObjectAndWait](#).
    - Revedeți secțiunile despre lucrul cu [mutex-uri](#) și [evenimente](#).

## BONUS

1. (2 so karma) Parallel Sort
- Deschideți sursa `sort.c` din proiectul 7-sort.
  - Se dorește realizarea sortării unui șir de numere aleatoare dintr-un fișier în următorul mod:
    - Se împarte vectorul în bucăți către fiecare fir de execuție
    - Un fir de execuție sortează bucata proprie folosind quicksort
    - Se face merge la bucăți, în următorul fel:



- Realizați partea de creare a firelor de execuție și împărțire a taskurilor în funcția `init_setup()`
  - Urmăriți *TODO*
- După ce toate firele de execuție sortează chunk-ul static, unele vor începe să facă merge la chunk-urile sortate.

- Completați funcția `ThreadFunc` pentru ca, în funcție de id, un fir de execuție să apeleze funcția `MergeChunks` (care realizează interclasarea a doi vectori sortați)
  - Urmăriți *TODO*
- Hints:
  - Șirul de numere este dat sub forma unui fișier binar care poate fi generat cu programul `generator.exe`.
  - Citirea șirului într-un vector este deja realizată în funcția `init_setup`
  - Fiecare fir de execuție primește o structură `CHUNK` care reprezintă dimensiunea unui vector de sortat, cât și adresa inițială a vectorului.
  - Interclasarea a două structuri `CHUNK` în care vectorii sunt deja sortați se realizează cu funcția `MergeChunks`
- **Pot apărea race-uri ?**
- 2. **(2 so karma) The dorm room problem**
  - Deschideți sursa `dorm_room.c` din proiectul 8-dean.
  - Se dorește simularea/modelarea următoarei probleme: decanul și studenții
  - Se dau următoarele constrângeri:
    - Orice număr de studenți poate intra într-o cameră în același moment
    - Decanul poate intra într-o cameră doar dacă nu sunt studenți acolo (pentru a realiza o percheziție) sau dacă sunt mai mult de 25 de studenți (pentru a sparge petrecerea)
    - Cât timp Decanul este în cameră, studenții pot doar ieși, nu și intra
    - Decanul nu poate părăsi camera până când nu au ieșit toți studenții (s-a terminat sigur petrecerea :P)
    - Există un singur Decan.
  - Cerință:
    - Rezolvați problema scriind cod pentru entitățile respective: decan și student
    - Hint:
      - Pentru firele de execuție *studenți* completați funcțiile "enter\_room" și "party"
      - Pentru firul de execuție *decan* completați funcția "break\_party"
      - Revedeți secțiunea despre [mutex-uri](#) și [semafoare](#)
      - Folosiți funcțiile "dbg\_student" și "dbg\_decan" pentru a afișa mesaje corespunzătoare de fiecare dată când un fir de execuție își schimbă starea (ex: decanul intră în cameră, un student nu poate intra deoarece decanul e deja în cameră etc.)

## Extra

- Sincronizarea firelor de execuție în Python și PyGTK (exemplu vizual)
  - Citiți fișierul README și rulați scriptul din [această arhivă](#) (TODO: Arhivă goală)
  - Studiați comportamentul aplicației când utilizatorul apasă pe butoanele de Release și Signal
  - De ce este firele de execuție se sincronizează cu obiectele de pe ecran? (componentele GTK)
  - Implementați vizualizarea unui timer; spre exemplu, un fir de execuție care alternează două operații: draw și sleep. (desenează, se oprește, desenează, iar se oprește și tot așa; intervalul unei operații poate fi același)

## Soluții

[lab09-sol.zip](#)

From:

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

Permanent link:

<http://elf.cs.pub.ro/so/wiki/laboratoare/laborator-09>

Last update: 2012/04/22 23:25