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 execuţie

Pentru a lansa un nou fir de execuţie există funcţiile CreateThread şi CreateRemoteThread (a doua fiind folosită pentru a crea un fir de execuţie 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
);
```

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

`dwStackSize` reprezintă mărimea iniţială a stivei(în bytes). Sistemul rotunjeşte 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 funcţia ce trebuie executată de către thread. Această funcţie are următorul prototip:

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

unde `lpParameter` reprezintă datele care sunt pasate firului de execuţie la execuţie. La fel ca pe Linux, se poate transmite un pointer la o structură, care conţine toţi parametrii necesari. Rezultatul întors poate fi obţinut de un alt thread folosind funcţia GetExitCodeThread.

Handle şi identificator

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

- printr-un `HANDLE`, obţinut la crearea thread-ului, sau folosind funcţia 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 funcţiilor de lucru cu `HANDLE`-uri că este vorba de `HANDLE`-ul asociat cu thread-ul curent (obţinut, de exemplu, apelând GetCurrentThread). Pentru a converti un `pseudo-HANDLE` într-un `HANDLE` veritabil, trebuie folosită funcţia DuplicateHandle. De asemenea, nu are sens să facem CloseHandle pe un `pseudo-HANDLE`. Pe de altă parte, handle-ul obţinut cu DuplicateHandle trebuie închis dacă nu mai este nevoie de el.

- printr-un identificator de thread, de tipul `DWORD`, întors la crearea thread-ului, sau obţinut folosind GetCurrentThreadId. O diferenţă dintre identificator şi `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 obţinut la crearea unui thread are implicit drepturi de acces nelimitate. El poate fi moştenit sau nu de procesele copil ale procesului curent în funcţie de flag-urile specificate la crearea lui. Prin funcţia DuplicateHandle, se poate crea un nou handle cu mai puţine drepturi. Handle-ul este valid până când este închis, chiar dacă firul de execuţie pe care îl reprezintă s-a terminat.

Aşteptarea firelor de execuţie

Pe Windows, se poate aştepta terminarea unui fir de execuţie folosind aceeaşi funcţie ca pentru aşteptarea oricărui obiect de sincronizare WaitForSingleObject:

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

Terminarea firelor de execuţie

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

- el însuşi apelează funcţia ExitThread :void ExitThread(DWORD dwExitCode);
- funcţia asociată firului de execuţie execută un `return`.
- un fir de execuţie ce deţine un handle cu dreptul `THREAD_TERMINATE` asupra firului de execuţie, 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 execuţie al unui proces se termină şi procesul.

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

Pentru aflarea codului de terminare a unui fir de execuţie folosim funcţia GetExitCodeThread

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

- `hThread` - handle la firul de execuţie în discuţie 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 şi-a terminat execuţia, această valoare va fi `STILL_ACTIVE`.

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

Suspend, Resume

Cedarea procesorului

Alte funcţii 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 execuţie.

```
DWORD GetThreadId(HANDLE hThread);
```

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

Thread Local Storage

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

Pentru a crea un nou TLS se apelează funcţia TlsAlloc:

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

Funcţia întoarce în caz de succes indexul asociat TLS-ului, prin intermediul căruia fiecare fir de execuţie va putea accesa datele specifice. În caz de eşec funcţia întoarce valoarea `TLS_OUT_OF_INDEXES`.

Pentru a stoca o nouă valoare într-un TLS se foloseşte funcţia TlsSetValue:

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

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

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

În caz de succes funcţia întoarce valoarea stocată în TLS, iar în caz de eşec 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 foloseşte funcţia TlsFree:

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

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

Exemplu

Exemplul prezintă crearea a 2 fire de execuţie 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, "LocallAloc");

        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 execuţie

Windows pune la dispoziţie 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 execuţie este mai bună (pentru context-switch, nu mai este necesară interacţiunea cu kernel-ul), programele scrise folosind fibre pot deveni complexe. Mai multe informaţii puteţi găsi la secţiunea suplimentară dedicată.

Securitate şi drepturi de acces

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

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

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

Pentru a obţine acest descriptor este folosită funcţia GetSecurityInfo, iar pentru a-l schimba funcţia SetSecurityInfo.

Handle-ul întors de funcţia 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 execuţie îl permite apelantului.

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

Sincronizarea thread-urilor

Pentru sincronizarea firelor de execuţie avem la dispoziţie:

- **mutex**: POSIX, Win32
- **semafoare**: POSIX, Win32
- **secţiuni critice**: (excludere mutuală în cadrul aceluiasi proces) ? doar Win32
- **variabile de condiţie**: POSIX, Win32 (începând cu Vista)
- **evenimente**: doar Win32
- **timere**: doar Win32.

Standardul POSIX specifică funcţii 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 funcţiile standard de sincronizare: WaitForSingleObject, WaitForMultipleObjects sau SignalObjectAndWait.

Obiectele de sincronizare Semaphore, Mutex, Event şi WaitableTimer pot fi folosite atât pentru sincronizarea proceselor, cât şi a firelor de execuţie. Ele au fost deja introduse în laboratoarele trecute.

În Windows mai există un mecanism de sincronizare care este disponibil doar pentru firele de execuţie ale **aceluiaşi proces**, şi anume CriticalSection. Se recomandă folosirea CriticalSection pentru excluderea mutuală a firelor de execuţie ale aceluiaşi proces, fiind mai **eficient** decât Mutex sau Semaphore.

Win32 API pune la dispoziţie un mecanism de acces sincronizat la variabile partajate între fire de execuţie prin intermediul funcţiilor **interlocked** (Interlocked Variable Access), precum şi operaţii atomice de inserare şi ştergere în liste simplu înlănţuite (Interlocked Singly Linked Lists).

Mutex Win32

Subiectul a fost tratat în laboratorul de comunicaţie 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 comunicaţie 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);

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

Secţiune critică

Obiectele CriticalSection sunt echivalente mutex-urilor POSIX de tip `RECURSIVE`. Acestea sunt folosite pentru excluderea mutuală a accesului firelor de execuţie ale **aceluiaşi proces** la o secţiune critică de cod care conţine operaţii asupra unor date partajate. Un singur fir de execuţie va fi activ la un moment dat în interiorul secţiunii critice, şi dacă mai multe fire aşteaptă să intre, nu este garantată ordinea lor de intrare, totuşi sistemul va fi echitabil faţă de toate.

Operaţiile care se pot efectua asupra unei secţiuni critice sunt: intrarea, intrarea neblocantă, ieşirea din secţiunea critică, iniţializarea şi distrugerea.

Pentru serializarea accesului la o secţiune critică, fiecare fir de execuţie va trebui să intre într-un obiect `CriticalSection` la începutul secţiunii şi să-l părăsească la sfârşitul ei. În acest fel, dacă două fire de execuţie încearcă să intre în `CriticalSection` simultan, doar unul dintre ele va reuşi, şi îşi va continua execuţia în interiorul secţiunii critice, iar celălalt se va bloca pînă când obiectul `CriticalSection` va fi părăsit de primul fir. Aşadar, la sfârşitul secţiunii, primul fir trebuie să părăsească obiectul `CriticalSection`, permiţându-i celuilalt intrarea.

Pentru excluderea mutuală se pot folosi atât obiecte Mutex, cât şi obiecte CriticalSection; dacă sincronizarea trebuie făcută doar între firele de execuţie ale aceluiaşi proces este recomandată folosirea `CriticalSection`, fiind mai un mecanism mai **eficient**. Operaţia de intrare în `CriticalSection` se traduce într-o singură instrucţiune de asamblare de tip *test-and-set-lock* (`TSL`). `CriticalSection` este **echivalentul** futex-ului din Linux.

Iniţializarea/distrugerea unei secţiuni critice

Alocarea memoriei pentru o secţiune critică se face prin declararea unui obiect `CRITICAL_SECTION`. Acesta nu va putea fi folosit, totuşi, înainte de a fi iniţializat.

```
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);
```

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

Contorul de spin are sens doar pe sistemele **multiprocesor** (SMP) (este ignorat pe sisteme uniprocesor). Contorul de spin reprezintă numărul de cicli pe care îi petrece un fir de execuţie pe un procesor în busy-waiting, înainte de a-şi suspenda execuţia la un semafor asociat secţiunii critice, în aşteptarea eliberării acesteia. Scopul aşteptării unui număr de cicli în busy-waiting este evitarea blocării la semafor în cazul în care secţiunea critică se eliberează în intervalul respectiv, deoarece blocarea la semafor are impact asupra performanţelor. Folosirea contorului de spin este recomandată mai ales în cazul unei secţiuni critice scurte, accesate foarte des.

Utilizarea secţiunilor critice

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

```
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 execuţie, numărul apelurilor LeaveCriticalSection trebuie să fie **egal** cu numărul apelurilor EnterCriticalSection, pentru a elibera în final secţiunea critică. Dacă un fir de execuţie care nu a intrat în secţiunea critică apelează LeaveCriticalSection, se va produce o eroare care va face ca firele care au apelat EnterCriticalSection să aştepte pentru o perioadă nedefinită de timp.

Exemplu secţiuni 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 thread poate semnaliza unul sau mai multe threaduri că o anumita condiţie este îndeplintă. Ce e important este pot fi deblocate mai multe threaduri prin semnalarea unui singur eveniment. Evenimentele sunt de două tipuri, în funcţie de modul în care sunt resetate:

- resetare manuală - după alertarea mai multor threaduri, evenimentul trebuie resetat
- resetare automată - dupa alertarea unui singur thread, evenimentul se resetează automat

Un eveniment este create folosind funcţia CreateEvent:

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

Pentru a controla un eveniment se folosesc funcţiile:

- SetEvent - pentru semnalizarea evenimentului. Dacă Evenimentul este de tip auto-reset, atunci **un singur** thread va fi trezit, iar evenimentul se resetează automat. Dacă evenimentul este de tip manual-reset, atunci evenimentul rămâne semnalizat până când un thread apelează ResetEvent. Altfel, orice thread care încearcă să aştepte 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 threadurile care aşteaptă la un eveniment de tip manual-reset, iar evenimentul este apoi resetat. Dacă funcţia este folosită în conjucţie cu un eveniment auto-reset, atunci va debloca un singur thread.

Operaţii atomice cu variabile partajate (Interlocked Variable Access)

Funcţiile **interlocked** pun la dispoziţie un mecanism de sincronizare a accesului la variabile partajate între mai multe fire de execuţie. Funcţiile pot fi apelate de fire de execuţie ale unor procese diferite, pentru variabile aflate într-un spaţiu de memorie partajată. Funcţiile interlocked reprezintă cel mai simplu mod de evitare a race-ului care apare când două fire de execuţie modifică aceeaşi variabilă.

Operaţiile atomice asupra variabilelor partajate:

- **incrementare** / **decrementare** LONG InterlockedIncrement(LONG volatile *lpAddend);

  LONG InterlockedDecrement(LONG volatile *lpDecend);

ambele întorc noua valoare.

- **atribuirea** atomică a unei valori unei variabile partajate LONG InterlockedExchange(LONG volatile *Target, LONG Value);

  LONG InterlockedExchangeAdd(LPLONG volatile Addend, LONG Value);

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

primele două întorc vechea valoare.

- **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 şi atribuirea vor fi executate într-o singură operaţie **atomică**. Pentru variabile de tip pointer se va folosi InterlockedCompareExchangePointer. Comportamentul este echivalent cu:

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

Windows Thread Pooling

Programele cu un număr mare de threaduri pot aduce probleme de performanţă dincolo de cele de locking:

- Fiecare thread are o stivă proprie (default 1MB). Astfel, 100 de threaduri vor consuma 1GB de spaţiu virtual.
- Context-switch-urile între threaduri poate cazua page-fault-uri la accesarea stivei.
- Crearea şi terminarea thread-urilor presupune calcule suplimentare.

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

Există două modalităţi prin care o aplicaţie poate specifica task-urile pe care le doreşte executate de fire de execuţie din thread pool:

- se pot adăuga taskuri ce vor fi executate **imediat** ce se eliberează un fir de execuţie din thread pool
- se pot adăuga operaţii de aşteptare care au asociată o funcţie callback ce urmează a fi executată la sfârşitul unui **timeout** de unul din firele de execuţie din thread pool. Din această categorie fac parte operaţiile de aşteptare a

terminării unei intrări/ieşiri asincrone, operaţiile de aşteptare a expirării unui `TimerQueue Timer` şi funcţiile de aşteptare înregistrate.

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

Un exemplu de folosire pentru Windows ThreadPools, foloseşte noul API, se găseşte aici.

Adăugarea de taskuri la thread pool

Aşteptarea unei operaţii de intrare/ieşire asincrone

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

```
// înregistrează o funcţie ce va fi chemată când se încheie o
// operaţie de IO asincron pe fişierul identificat prin FileHandle.
// Pot fi înregistrate mai multe funcţii şi vor fi chemate toate
// când se încheie operaţia IO asincronă. Ordinea în care sunt apelate
// nu este specificată.
BOOL BindIoCompletionCallback(
        HANDLE                          FileHandle,
        LPOVERLAPPED_COMPLETION_ROUTINE Function,
        ULONG                           Flags);

// semnătura funcţiei înregistrate să fie executată la încheierea operaţia AIO
VOID CALLBACK FileIOCompletionRoutine(
        DWORD dwErrorCode,
        DWORD dwNumberOfBytesTransfered,
        LPOVERLAPPED lpOverlapped);
```

Adăugarea unui task pentru execuţie imediată

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

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

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

Timer Queues

Obiectele *TimerQueue* reprezintă **cozi de timere**. Ele conţin obiecte *TimerQueueTimer* care au asociată o funcţie *callback*, ce va fi executată de un fir de execuţie 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 ştergere, dar *NU* aşteaptă
// ca toate callbackurile asociate cozii să se termine
BOOL DeleteTimerQueue(HANDLE TimerQueue);

/**
 * CompletionEvent = NULL - marchează coada pentru ştergere şi iese imediat (ca DeleteTimerQueue)
 * CompletionEvent = INVALID_HANDLE_VALUE - funcţia aşteaptă 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 funcţia:

```
BOOL CreateTimerQueueTimer(
        PHANDLE phNewTimer, // aici întoarce un HANDLE la timerul nou creat
```

```
        HANDLE TimerQueue,  // coada la care este adăugat timerul.
                            // Dacă e NULL se foloseşte 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, ş.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 foloseşte 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 ŞI distrugerea unui timer.
// CompletionEvent e similar cu cel din DeleteTimerQueueEx.
BOOL DeleteTimerQueueTimer(HANDLE TimerQueue, HANDLE Timer, HANDLE CompletionEvent);
```

Registered Wait Functions

Funcţiile de aşteptare înregistrate sunt funcţii de aşteptare executate de un fir de execuţie din *thread pool*. În momentul în care obiectul de sincronizare după care se aşteaptă trece în starea *signaled*, se va executa rutina *callback* asociată funcţiei de aşteptare înregistrate, de un fir de execuţie din *thread pool*. În mod implicit, funcţiile de aşteptare înregistrate se **rearmează automat** şi rutinele *callback* sunt executate de fiecare dată când obiectul de sincronizare după care se aşteaptă trece în starea ***signaled***, sau intervalul de timeout **expiră**. Acest lucru se repetă până când înregistrarea funcţiei de aşteptare este anulată. Se poate seta, însă, ca funcţia de aşteptare înregistrată să se execute **o singură dată**.

Înregistrarea unei funcţii de aşteptare

Pentru înregistrarea în *thread pool* a unei funcţii de aşteptare se va apela funcţia:

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

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

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

Parametrul `TimerOrWaitFired` va specifica dacă execuţia rutinei `Callback` s-a declanşat 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 execuţie care va executa rutina `Callback`, precum şi dacă funcţia de aşteptare trebuie să se execute doar o singură dată. Funcţia va întoarce, prin parametrul `phNewWaitObject`, un *handle* ce va fi folosit pentru deînregistrarea funcţiei de aşteptare.

Deînregistrarea unei funcţii de aşteptare

Pentru a anula înregistrarea unei funcţii de aşteptare se va apela una din funcţiile:

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

Orice funcţie de aşteptare înregistrată va trebui deînregistrată prin apelul uneia din funcţiile de mai sus.

Funcţia `UnregisterWaitEx` va semnaliza *event*-ul `CompletionEvent` în cazul în care se termină cu succes şi rutina de

callback s-a terminat cu succes. Dacă valoarea lui `CompletionEvent` nu este `NULL`, atunci funcţia va aştepta finalizarea operaţiei de aşteptare şi terminarea rutinei asociate.

Exerciţii

În rezolvarea laboratorului folosiţi arhiva de sarcini [lab09-tasks.zip]

Pentru a deschide proiectul VS conţinând exerciţiile, deschideţi fişierul lab09.sln.

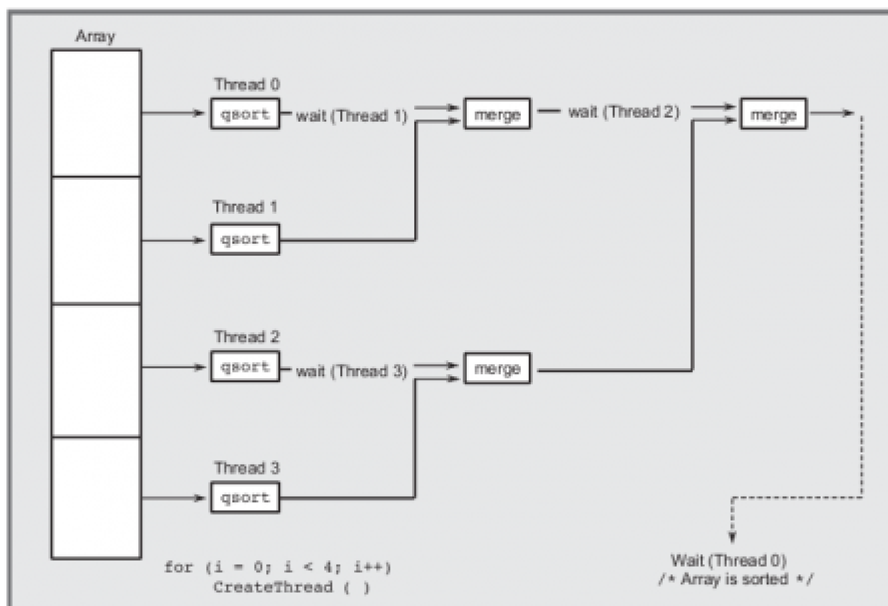
1. (**1 punct**) Threading şi priorităţi
   - Încărcaţi proiectul `threading` şi setaţi-l ca *start-up project*
   - Compilaţi şi rulaţi programul. Câte threaduri exista in total?
   - Lansaţi `ProcessExplorer` (check Desktop) şi verificaţi răspunsul de la întrebarea de mai sus.
   - Hint:
     - View Select Columns Process Performance Threads
   - Ce prioritate are procesul `threading.exe`?
   - Hint:
     - click-dreapta pe numele procesului Set Priority
   - Experimentaţi schimbând prioritatea procesului. Pentru ce caz procesul `threading.exe` primeşte mai mult timp de procesor?
     - **Atenţie** - dacă setaţi ca prioritate `real-time`, cel mai probabil vi se va bloca maşina virtuală. De ce credeţi că se întâmplă asta?
2. (**1.5 puncte**) Thread debugging
   - Deschideţi sursa `debug.c` din proiectul `2-debug`
   - Completaţi funcţia `StartThread` pentru a implementa crearea unui thread.
     - Hints: Urmariţi în cod secţiunea marcată cu *TODO*
   - Compilaţi şi rulaţi sursa. De ce se blochează?
     - Rezolvaţi problema (soluţia nu implică comentarea funcţiei Sleep)
3. (**2 puncte**) `interlocked`
   1. Creaţi `THREAD_NO` fire de execuţie, care incrementează circular o variabilă (când se ajunge la o limită se resetează la 0).
      - Folosiţi *Interlocked Variables* deoarece mecanismul e mai rapid decât o incrementare normală protejată cu `Mutex` sau `CRITICAL_SECTION`.
      - Incrementarea circulară se va face în funcţia `thread_function`.
        - Hints:
          - Folosiţi `InterlockedCompareExchange`
          - Urmăriţi comentariile cu *TODO 1*
      - **Observaţi vreo problemă cu această abordare?**
   2. Comparaţi timpul de execuţie al programului precedent în cazul în care se foloseşte un mutex care să sincronizeze accesul la variabila `counter`
      - Completaţi funcţia `thread_function_mutex` (*TODO 2*)
      - Pentru ambele implementări, **decomentaţi** apelul funcţiei `Sleep`
   - Cum explicaţi diferenţa de timp?
4. (**2 puncte**) ? `TLS`
   - Deschideţi proiectul `4-tls`.
   - Dorim să simulăm o implementare a funcţiei `perror`.
   - Pentru aceasta vom avea variabila globala `myErrno`, dar cu valoari specifice(diferite) pentru fiecare thread.
   - **Hints:**
     - Urmăriţi comentariile marcate cu *TODO*
     - Revedeţi secţiunea despre [TLS]
5. (**2 puncte**) ? `timerQueue`
   - Creaţi un `TimerQueueTimer`, a cărui rutină *callback* să fie declanşată de exact 3 ori, din secundă în secundă. După 3 declanşări se va dezactiva timerul şi se vor distruge toate resursele create.
   - **Hints**:
     - Trebuie să sincronizaţi rutina *timer*-ului cu funţia `main` care va dezactiva timer-ul; pentru aceasta puteţi folosi orice mecanism de semanlizare: mutex, samafor, event
6. (**2 puncte**) ? `Barrier`
   - Implementaţi o barieră reutilizabilă folosind un mutex şi 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;
```

   - Completaţi funcţiile de lucru cu bariera pentru a obţine funcţionalitatea dorită.

- Pentru ce folosiţi mutexul? Dar variabila de tip eveniment.
- Hint:
  - Completaţi comentariile marcate cu *TODO*
  - De ce ar fi nevoie de mutex?
  - De ce ar fi nevoie de event? Cum semnalaţi toate threadurile care aşteaptă?
  - Pentru a putea semnaliza un semafor şi a astepta la un alt obiect de sincronizare în acelaşi timp, puteţi folosi funcţia SignalObjectAndWait
  - Revedeţi secţiunile de lucru de mutex şi evenimente.

BONUS

1. (**2 so karma**) Parallel Sort
   - Se doreşte realizarea sortării unui şir de numere aleatoare dintr-un fişier în următorul mod:
     - Se împarte vectorul în bucăţi către fiecare thread
     - Un thread sortează bucata proprie folosind quicksort
     - Se face merge la bucăţi, în următorul fel:
     - 

     - Realizati partea de creare a threadurilor si impartire a taskurilor in functia `init_setup()`
       - Urmariti *TODO 1*
     - Dupa ce toate threadurile sorteaza chunk-ul static, unele vor incepe sa faca merge la chunk-urile sortate.
       - In prima etapa, threadurile 1, vor face merge
     - Completati functia `ThreadFunc` pentru ca, in functie de id, un thread sa apeleze functia `MergeChunks` (care realizeaza interclasarea a doi vectori sortati)
       - Urmariti *TODO2*
   - Hints:
     - Porniţi de la proiectul `4-sort`
     - Şirul de numere este dat sub forma unui fişier binar care poate fi generat cu programul `generator.exe`.
     - Citirea şirului într-un vector este deja realizată în funcţia `init_setup`
     - Fiecare thread primeşte o structură `CHUNK` care reprezintă dimensiunea unui vector de stortat, cât şi adresa iniţială a vectorului.
     - Interclasarea a două structuri CHUNK în care vectorii sunt deja sortaţi se realizează cu funcţia `MergeChunks`
   - **Pot apărea race-uri ?**

1. (**2 so karma**) ? `The dorm room problem`
   - Se doreşte simularea/modelarea următoarei probleme - Decanul şi studenţii
   - Se dau următoarele constrângeri:
     - Orice număr de studenţi poate intra într-o camera în acelaşi moment
     - Decanul poate intra într-o cameră doar daca nu sunt studenţi acolo (pentru a realiza o perchezitie) sau dacă sunt mai multe de 25 de studenţi (pentru a sparge petrecerea)
     - Cât timp Decanul este în cameră, studenţii pot doar ieşi, nu şi intra
     - Decanul nu poate părăsi camera până când toţi studenţii au ieşit (s-a terminat sigur petrecerea :P)
     - Există un singur Decan.
   - Cerinţă:

- Rezolvati problema scriind cod pentru entitatile respective: decan si student
- Plecaţi de la implementarea din proiectul `6-party`
- Hint:
  - Pentru threadurile *studenţi* completaţi funcţiile "enter_room" şi "party"
  - Pentru threadul decan completaţi funcţia "break_party"
  - Revedeţi secţiunea despre mutex şi semafoare
  - Folosiţi funcţiile "dbg_student" şi "dbg_decan" pentru a afişa mesaje corespunzătoare de fiecare dată când un thread îşi schimbă starea (ex: decanul intră în caremă, un student nu poate intra deoarece decanul e deja în cameră etc.)

Extra

- Sincronizarea firelor de execuţie în Python şi PyGTK (exemplu vizual)
  - Citiţi fişierul README şi rulaţi scriptul din această arhivă
  - Studiaţi comportamentul aplicaţiei când utilizatorul apasă pe butoanele de Release şi Signal
  - De ce este firele de execuţie se sincronizează cu obiectele de pe ecran? (componentele GTK)
  - Implementaţi vizualizarea unui timer; spre exemplu, un fir de execuţie care alternează două operaţii: draw şi sleep. (desenează, se opreşte, desenează, iar se opreşte şi tot aşa; intervalul unei operaţii poate fi acelaşi)

Soluţii

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: **2011/04/19 13:19**