Windows Programming
Threads and Concurrency

Vassos Tziongouros
Christos Constantinou

Ιστορική Αναδρομή

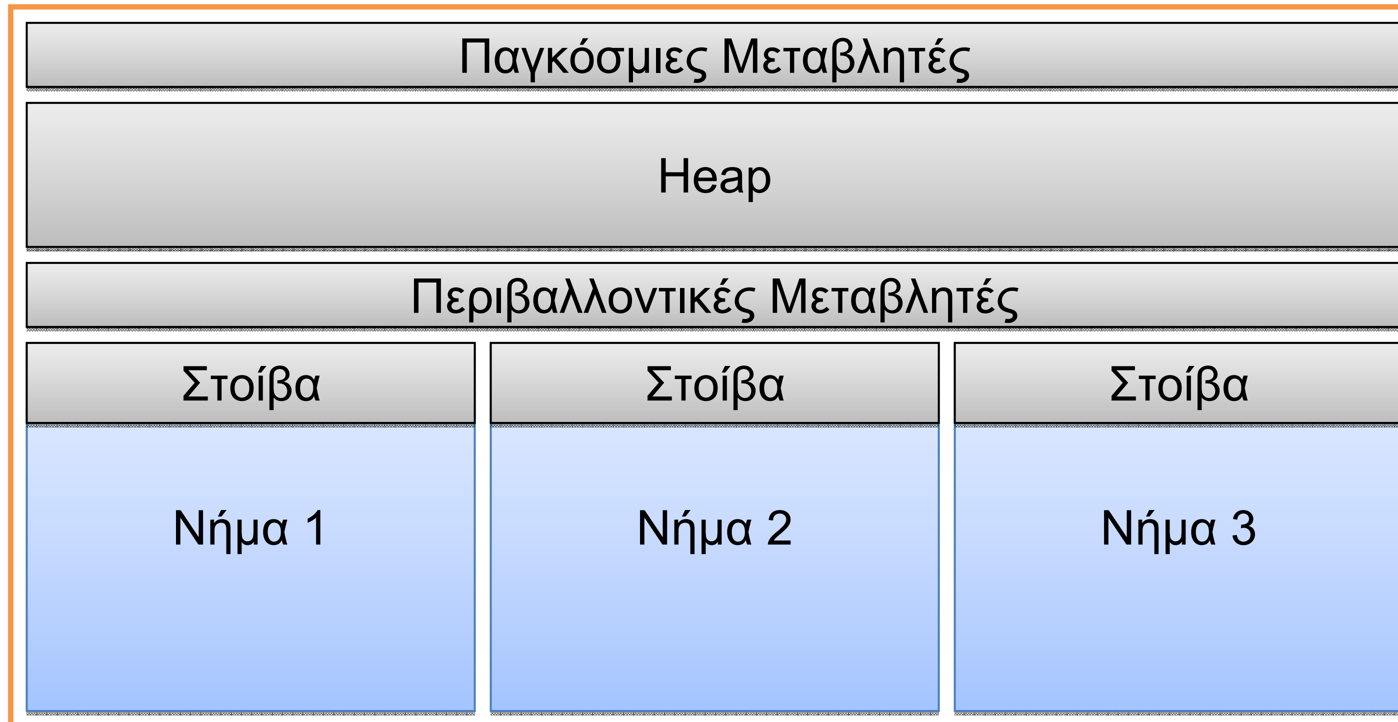
- Microsoft Windows 3.1: Cooperative Multitasking
  - Cooperative σημαίνει ότι εξαρτάται από την διεργασία να ελευθερώσει τον έλεγχο από τον επεξεργαστή την κατάλληλη στιγμή έτσι ώστε να φαίνεται ότι δουλεύει παράλληλα με τις υπόλοιπες διεργασίες. Χρονοβόρα I/O αποτελούν πρόβλημα στην συνεργασία διεργασιών.
- Microsoft Windows 98: Preemptive Multi-tasking, Multi-threaded, Single-CPU
  - Όπως και στα συστήματα UNIX, το λειτουργικό έχει τον έλεγχο χρονοδρομολόγησης διεργασιών. Το σύστημα λειτουργά ομαλά με πολλές ταυτόχρονες διεργασίες. Υπολογιστές με πολλούς επεξεργαστές ακόμα δεν μπορούν να αξιοποιηθούν πλήρως.
- Microsoft Windows 2000: Preemptive Multi-tasking, Multi-threading, Multi-CPU
  - Μια διεργασία μπορεί να σπάσει σε πολλά νήματα, τα οποία το σύστημα μπορεί να χρονοδρομολογήσει σε πολλαπλούς επεξεργαστές.

Αναγκαιότητα Νημάτων

- Σε αντίθεση με τα UNIX, μια καινούρια διεργασία σε Windows δεν κληρονομά τον χώρο μεταβλητών του πατέρα. Μπορεί να κληρονομήσει το πολύ μερικά handles μέσω του CreateProcess(). Δεν υπάρχει αντίστοιχη υλοποίηση του fork().
- Κάθε καινούρια διεργασία αποτελείται από τον δικό της ανεξάρτητο χώρο στην μνήμη και 1 νήμα. Η διεργασία απλά περιέχει τις παγκόσμιες μεταβλητές, περιβαλλοντικές μεταβλητές και τον αντίστοιχο χώρο στο heap. Το νήμα είναι αυτό που εκτελεί τον κώδικα, όχι η διεργασία.

Επικοινωνία Νημάτων

- Τα νήματα μοιράζονται τον χώρο παγκόσμιων μεταβλητών και το heap του πατέρα. Το κάθε νήμα έχει το δικό του stack.



Χρήση Νημάτων

- Για να χρησιμοποιήσουμε νήματα σε Windows πρέπει να συμπεριλάβουμε το header **windows.h** και να κάνουμε δυναμικό σύνδεσμο με το **kernel32.dll**
- Το header windows.h είναι μέρος του Platform SDK (Software Development Kit), το οποίο έρχεται μαζί με το Visual Studio. Για να χρησιμοποιηθεί με άλλους μεταγλωττιστές πρέπει να το κατεβάσετε ξεχωριστά από την ιστοσελίδα http://www.microsoft.com/downloads/details.aspx?familyid=E15438AC-60BE-41BD-AA14-7F1E0F19CA0D&displaylang=en
- Για δημιουργία δυναμικών συνδέσμων με βιβλιοθήκες στο Visual Studio μπορείτε να χρησιμοποιήσετε το preprocessor directive **#pragma**

Δημιουργία Νήματος

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

- *lpThreadAttributes: Παράμετροι ασφάλειας νήματος (Χρησιμοποιείται σε εφαρμογές DCOM)*
- *dwStackSize: Μέγεθος στοίβας για το καινούριο νήμα (όταν είναι 0, η στοίβα έχει το ίδιο μέγεθος με του πατέρα. Με τις τοπικές μεταβλητές η στοίβα μεγαλώνει ανάλογα.)*
- *lpStartAddress: Η συνάρτηση που θα εκτελεστεί για το νήμα*
- *lpParameter: Παράμετροι που μπορεί να θέλουμε να περάσουμε στην συνάρτηση*
- *dwCreationFlags: Παράμετροι δημιουργίας νήματος* (0 ή CREATE_SUSPENDED).
- *lpThreadId: Μεταβλητή που θα παραλάβει το ID του καινούριου νήματος, το οποίο είναι μοναδικό σε ολόκληρο το σύστημα. Επιστρέφεται 0 σε περίπτωση προβλήματος δημιουργίας.*

Μεταβλητές volatile

- Όταν δηλώνουμε τις μεταβλητές με *volatile*, λέμε στον μεταγλωττιστή ότι η μεταβλητή θα χρησιμοποιείται ταυτόχρονα από πολλά νήματα και ότι δεν θέλουμε να βελτιστοποιήσει τον μεταγλωττισμένο κώδικα. Π.χ. Όταν εκτελείται το count++, δεν θέλουμε να κρατά το count μέσα σε cpu register μέχρι το τέλος του while loop, θέλουμε να ανανεώνεται συνεχώς η μνήμη.

Παράδειγμα 1

```
#include <windows.h>
#include <iostream>
#pragma comment(lib,"kernel32.lib")

using namespace std;

volatile UINT count = 0;

void CountThread() {
	while(1) {
		count++;
		Sleep(100);
	}
}

int main() {
	HANDLE countHandle;
	DWORD threadID;

	countHandle = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) CountThread, 0, 0, &threadID);
	if (countHandle==0)
		cout << "Cannot create thread: " << GetLastError() << endl;

	while(1) {
		cout << "Press <ENTER> to display the count... ";
		cin.get();
		cout << "The count is: " << count << endl << endl;
	}
}
```

```
Press <ENTER> to display the count...
The count is: 13

Press <ENTER> to display the count...
The count is: 23

Press <ENTER> to display the count...
The count is: 40

Press <ENTER> to display the count...
The count is: 52

Press <ENTER> to display the count...
```

Η συνάρτηση GetLastError() επιστρέφει το τελευταίο μήνυμα λάθους του τρέχοντος νήματος. Πολλαπλά νήματα **δεν** επηρεάζουν τα μηνύματα λάθους των άλλων νημάτων.

Αναμονή τερματισμού νημάτων

- Εάν η διεργασία τερματίσει πριν τα νήματα που δημιούργησε, τότε πεθαίνουν και αυτά. Η συνάρτηση WaitForSingleObject() περιμένει ένα νήμα να τερματίσει πριν συνεχίσει η εκτέλεση του προγράμματος.

DWORD WaitForSingleObject(HANDLE *hHandle*, DWORD *dwMilliseconds*)

- Δέχεται ως παράμετρο το handle του νήματος το οποίο θέλουμε να περιμένει, και τον μέγιστο χρόνο σε milliseconds για τον οποίο θέλουμε να περιμένει. Επιστρέφει είτε κωδικό λάθους, ή τον λόγο για τον οποίο σταμάτησε.

Παράδειγμα 2

```
#include <windows.h>
#include <iostream>
#pragma comment(lib,"kernel32.lib")

using namespace std;

volatile UINT count = 0;

void CountThread(DWORD iter) {
	for (DWORD i = 0; i < iter; i++) {
		cout << "I'm alive!\n";
		Sleep(1000);
	}
}

int main() {
	HANDLE countHandle;
	DWORD threadID;
	DWORD iterations = 3;
	int count=0;

	countHandle=CreateThread(0, 0, (LPTHREAD_START_ROUTINE) CountThread, (VOID *) iterations, 0, &threadID);
	if (countHandle==0)
		cout << "Cannot create thread: " << GetLastError() << endl;

	while ( WaitForSingleObject(countHandle, 200) == WAIT_TIMEOUT)
		cout << "waiting for the thread to finish " << count++ << endl;

}
```

```
I'm alive!
waiting for the thread to finish 0
waiting for the thread to finish 1
waiting for the thread to finish 2
waiting for the thread to finish 3
Iw'ami tailnigv ef!o
r the thread to finish 4
waiting for the thread to finish 5
waiting for the thread to finish 6
waiting for the thread to finish 7
waiting for the thread to finish 8
I'm alive!
waiting for the thread to finish 9
waiting for the thread to finish 10
waiting for the thread to finish 11
waiting for the thread to finish 12
waiting for the thread to finish 13
Press any key to continue
```

Νήματα με Παραμέτρους

- Για περισσότερες από μία παραμέτρους, μπορούμε να περάσουμε δείκτη σε μια **σταθερή** δομή. Σταθερές δομές είναι αυτές που είναι είτε παγκόσμιες μεταβλητές, ή δηλωμένες τοπικά ως static, ή είναι δεσμευμένες στο heap. Τοπικές μεταβλητές σε συναρτήσεις που εξαφανίζονται όταν βγούμε από το scope δεν είναι σταθερές.

Αναμονή τερματισμού πολλαπλών νημάτων

- Όταν έχουμε πολλά νήματα, η συνάρτηση WaitForMultipleObjects() μπορεί να χρησιμοποιηθεί για να τα περιμένει όλα, ή κάποιο από αυτά.

DWORD WaitForMultipleObjects(DWORD nCount, const HANDLE* lpHandles, BOOL bWaitAll, DWORD dwMilliseconds)

  – Παίρνει ώς παράμετρο το μέγεθος του πίνακα νημάτων, τον πίνακα νημάτων, μια τιμή boolean που καθορίζει εάν θέλουμε να επιστρέψει όταν τελειώσουν όλα τα νήματα ή όταν τελειώσει οποιοδήποτε νήμα, και τον μέγιστο χρόνο που θέλουμε να περιμένει.

Παράδειγμα 3

```
#include <windows.h>
#include <iostream>
#pragma comment(lib,"kernel32.lib")

using namespace std;

struct params {
	DWORD from;
	DWORD to;
};

void CountThread(params *p) {
	for (DWORD i = p->from; i < p->to; i++) {
		cout << "Thread counter is " << i << endl;
		Sleep(1000);
	}
	delete p;
}
```

Παράδειγμα 3 (συνέχεια)

```
int main() {
	HANDLE countHandles[3];
	DWORD threadID;

	for (int i = 0; i < 3; i++) {
		params *p = new params;
		p->from = 10;
		p->to = p->from + i + 1;
		countHandles[i] = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) CountThread, (VOID *) p, 0,
	&threadID);
		if (countHandles[i]==0) cout << "Cannot create thread: " << GetLastError() << endl;
		Sleep(100);
	}

	WaitForMultipleObjects(3, countHandles, TRUE, INFINITE);
}
```

```
Thread counter is 10
Thread counter is 10
Thread counter is 10
Thread counter is 11
Thread counter is 11
Thread counter is 12
Press any key to continue
```

Χρήσιμες Συναρτήσεις

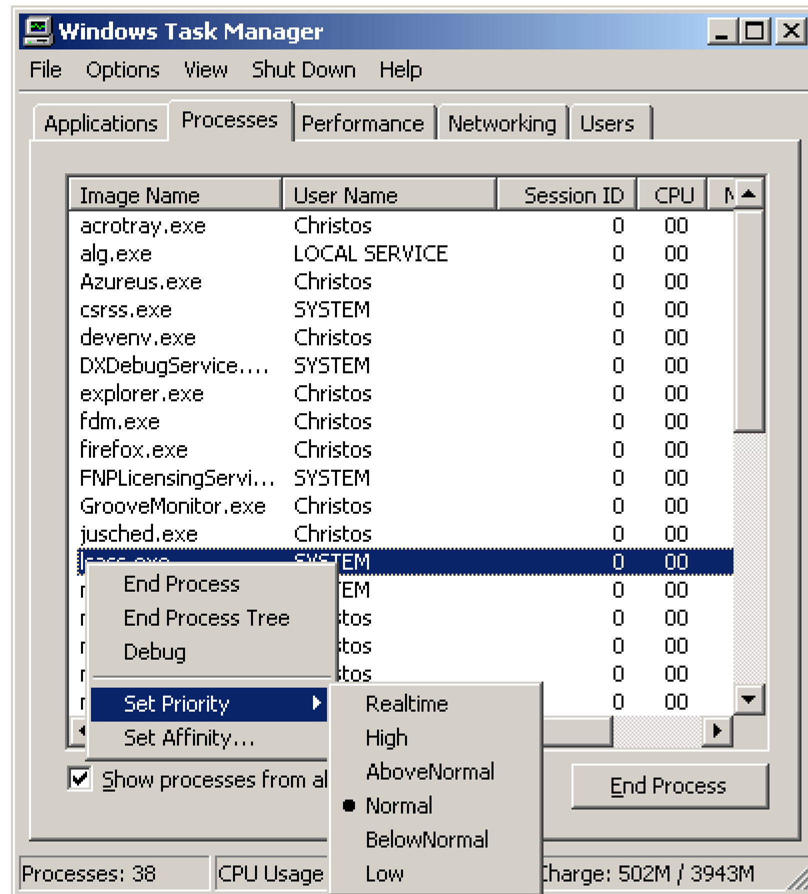
- Μπορούμε να πάρουμε τον αριθμό επεξεργαστών και άλλες σχετικές πληροφορίες με την συνάρτηση GetSystemInfo()
- Η συνάρτηση GetCurrentThreadId() επιστρέφει τον μοναδικό αριθμό του νήματος στο σύστημα.
- Όταν θέλουμε να τερματίσουμε κάποιο συγκεκριμένο νήμα, μπορούμε να ελέξουμε την εάν δεν έχει τερματίσει με την συνάρτηση GetExitCodeThread(), και να το τερματίσουμε με την συνάρτηση TerminateThread(), παρόλο που η συνάρτηση WaitForSingleObject() θα μπορούσε και εδώ να χρησιμοποιηθεί όπως στα προηγούμενα παραδείγματα.
- Όταν τα νήματα χρησιμοποιούν πόρους τους οποίους θέλουμε να απελευθερώσουν πριν τερματίσουν, ένας εναλλακτικός καλύτερος τρόπος τερματισμού θα ήταν να θέσουμε μια boolean μεταβλητή quit[], την οποία το κάθε νήμα θα χρησιμοποιά μέσα σε ένα βρόχο και μόλις γίνει true, το νήμα θα ελευθερώνει τους πόρους πριν τερματίσει.

Παράδειγμα 4 (Πολλαπλά CPU)

```
#include <windows.h>
#include <iostream>
#pragma comment(lib,"kernel32.lib“)
using namespace std;
void CountThread() {
        cout << "I am thread " << GetCurrentThreadId() << endl;
        Sleep(INFINITE);
}
int main() {
        HANDLE *countHandles;
        DWORD *threadID;
        SYSTEM_INFO sysInfo;
        DWORD threadStatus;
        GetSystemInfo(&sysInfo);
        countHandles = new HANDLE[sysInfo.dwNumberOfProcessors];
        threadID = new DWORD[sysInfo.dwNumberOfProcessors];
        for (DWORD i = 0; i < sysInfo.dwNumberOfProcessors; i++) {
                countHandles[i] = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) CountThread, 0, 0, &threadID[i]);
                if (countHandles[i]==0) cout << "Cannot create thread: " << GetLastError() << endl;
                Sleep(100);
        }
        for (DWORD i = 0; i < sysInfo.dwNumberOfProcessors; i++) {
                GetExitCodeThread(countHandles[i], &threadStatus);
                if (threadStatus == STILL_ACTIVE) {
                                cout << "Terminating thread " << i << endl;
                                TerminateThread(countHandles[i], 0);
                                CloseHandle(countHandles[i]);
                }}}
```

```
I am thread 3564
I am thread 1304
Terminating thread 0
Terminating thread 1
Press any key to continue
```

Προτεραιότητες Νημάτων



Προτεραιότητες Διεργασιών

1. IDLE_PRIORITY_CLASS (4)
2. BELOW_NORMAL_PRIORITY_CLASS (5)
3. NORMAL_PRIORITY_CLASS (9 foreground, 7 background)
4. ABOVE_NORMAL_PRIORITY_CLASS (11)
5. HIGH_PRIORITY_CLASS (13)
6. REALTIME_PRIORITY_CLASS (24)

Προτεραιότητες Νημάτων

1. THREAD_PRIORITY_TIME_CRITICAL (+3)
2. THREAD_PRIORITY_HIGHEST (+2)
3. THREAD_PRIORITY_ABOVE_NORMAL (+1)
4. THREAD_PRIORITY_NORMAL (0)
5. THREAD_PRIORITY_BELOW_NORMAL (-1)
6. THREAD_PRIORITY_LOWEST (-2)
7. THREAD_PRIORITY_ABOVE_IDLE (-3)
8. THREAD_PRIORITY_IDLE (-4)

Προτεραιότητες Νημάτων

- Η προτεραιότητα ενός νήματος μπορεί να ανατεθεί εν σχέση με την προτεραιότητα του πατέρα. Η προτεραιότητα που αναθέτουμε σε κάποιο νήμα, προσθέτετε πάνω στην προτεραιότητα του πατέρα, και το αποτέλεσμα είναι η προτεραιότητα που έχει το νήμα **σε ολόκληρο** το σύστημα.
- Κάθε καινούρια διεργασία ξεκινά με κανονική προτεραιότητα. Τα νήματα μιας διεργασίας μπορούν να αλλάξουν προτεραιότητα τους με την συνάρτηση BOOL SetThreadPriority (HANDLE hThread, int nPriority)

Άλλες Συναρτήσεις Νημάτων

- DWORD SuspendThread (HANDLE hThread) – Σταματά την εκτέλεση ενός νήματος
- DWORD ResumeThread (HANDLE hThread) – Συνεχίζει την εκτέλεση ενός νήματος
- VOID ExitThread (DWORD dwExitCode) – Αναγκάζει ένα νήμα να τερματίσει με συγκεκριμένο κωδικό
- HANDLE GetCurrentThread (VOID) – Επιστρέφει το handle του τρέχοντος νήματος

Συγχρονισμός Νημάτων

- Συναρτήσεις Interlocked
- Αντικείμενα CRITICAL_SECTION
- Mutexes
- Semaphores (Σηματοφόροι)
- Events (Γεγονότα)

Συναρτήσεις Interlocked

- Συγχρονίζουν την πρόσβαση σε μεταβλητές που μοιράζονται πολλά νήματα
- Εμποδίζουν το νήμα από να γίνει preempt από το CPU ενώ αλλάζει την τιμή της μεταβλητής
- Είναι αποδοτικές επειδή είναι υλοποιημένες σε user space με λίγες εντολές μηχανής

Χρήση Συναρτήσεων Interlocked

- Για αύξηση και μείωση τις τιμής μιας μεταβλητής κατά 1

```
LONG InterlockedIncrement (LONG volatile* Addend);
LONG InterlockedDecrement (LONG volatile* Addend);

oldValue = InterlockedIncrement(&value)
```

- Για να αυξήσουμε ή να μειώσουμε την τιμή μιας μεταβλητής περισσότερο από 1

```
LONG InterlockedExchangeAdd (PLONG Addend,  LONG Increment)

oldValue = InterlockedExchangeAdd(&value,-3)
```

Χρήση Συναρτήσεων Interlocked

- Για να αλλάξουμε την τιμή μιας μεταβλητής με μια άλλη

```
LONG InterlockedExchange (LPLONG Target, LONG Value)

oldValue = InterlockedExchange(&value,10)
```

- Για να αλλάξουμε την τιμή μιας μεταβλητής με μια άλλη υπό συνθήκη

```
LONG InterlockedCompareExchange( LONG volatile* Destination,
  LONG Exchange, LONG Comparand );
```

Εάν *Destination == Comparand τότε αλλάζετε η τιμή

```
oldValue = InterlockedCompareExchange (&value, newValue, testValue)
```

Αντικείμενα CRITICAL_SECTION

- Απλός μηχανισμός για να λύσουμε το πρόβλημα των critical section
- Υλοποιούνται σε user space άρα είναι γρήγορα
- Ένα νήμα που έχει τον έλεγχο ενός CRITICAL_SECTION μπορεί να ξαναμπεί όσες φορές θέλει χωρίς να μπλοκαριστεί αλλά πρέπει να βγει τον ίδιο αριθμό φορών

Χρήση CRITICAL_SECTION

- Για να αρχικοποιήσουμε και να διαγράψουμε τα CRITICAL_SECTIONs

VOID InitializeCriticalSection (LPCRITICAL_SECTION lpCriticalSection)

VOID DeleteCriticalSection (LPCRITICAL_SECTION lpCriticalSection)

- Για να μπούμε και να βγούμε από critical sections

VOID EnterCriticalSection (LPCRITICAL_SECTION lpCriticalSection)

BOOL TryEnterCriticalSection (LPCRITICAL_SECTION lpCriticalSection)

VOID LeaveCriticalSection (LPCRITICAL_SECTION lpCriticalSection)

Παράδειγμα CRITICAL_SECTION

```
CRITICAL_SECTION cs1;
volatile DWORD N = 0, M;
/* N is a global variable, shared by all threads. */
InitializeCriticalSection (&cs1);
   ...
EnterCriticalSection (&cs1);
if (N < N_MAX) { M = N; M += 1; N = M; }
LeaveCriticalSection (&cs1);
   ...
DeleteCriticalSection (&cs1);
```

Μειονεκτήματα CRITICAL_SECTION

- Δεν μοιράζονται μεταξύ διεργασιών
- Δεν έχουν ώρα λήξης. Δηλαδή αν ένα νήμα δεν καλέσει το LeaveCriticalSection για οποιοδήποτε λόγω τότε τα υπόλοιπα νήματα θα μείνουν μπλοκάρισμα για πάντα εκτός και αν χρησιμοποιούν το TryEnterCriticalSection

Mutexes

- Υλοποιούνται στον πυρήνα
- Μοιράζονται μεταξύ διεργασιών
- Έχουν ώρα λήξης
- Όταν ένα νήμα που κρατά ένα mutex τερματίσει τότε το mutex ελευθερώνεται

Χρήση Mutexes

- Για να δημιουργήσουμε ένα mutex

HANDLE CreateMutex (LPSECURITY_ATTRIBUTES lpsa,
BOOL bInitialOwner, LPCTSTR lpMutexName)

  - *lpsa: Παράμετροι ασφάλειας mutex*
  - bInitialOwner: TRUE εάν θέλουμε να αποκτήσουμε τον έλεγχο του mutex μόλις δημιουργηθεί
  - lpMutexName: Προαιρετικό όνομα του mutex

- Για να πάρουμε το handle ενός mutex

HANDLE OpenMutex (DWORD dwDesiredAccess,
BOOL bInheritHandle, LPCTSTR lpName)

  - dwDesiredAccess: Δικαιώματα που θέλουμε για το mutex
  - bInheritHandle: TRUE εάν θέλουμε το handle να μπορεί να κληρονομηθεί από άλλα process
  - lpName: Το όνομα του mutex που θέλουμε

- Για να αποκτήσουμε τον έλεγχο σε ένα Mutex χρησιμοποιούμε τα WaitForSingleObject και WaitForMultipleObjects που είδαμε πιο πριν

- Για να ελευθερώσουμε ένα mutex

BOOL ReleaseMutex (HANDLE hMutex)

Παράδειγμα Mutexes

```
volatile INT count;
HANDLE mutex;

void CountThread(INT iterations)
{
  for (int i=0; i<iterations; i++)
  {
    WaitForSingleObject(mutex, INFINITE);
    count++;
    ReleaseMutex(mutex);
  }
}

int main()
{
  HANDLE handles[4];
  DWORD threadID;

  mutex = CreateMutex(0, FALSE, 0);

  for (int i=0; i<4; i++)
    handles[i]=CreateThread(0, 0,(LPTHREAD_START_ROUTINE) CountThread, (VOID *) 25000, 0, &threadID);

  WaitForMultipleObjects(4, handles, TRUE, INFINITE);

  CloseHandle(mutex);

  cout << "Global count = " << count << endl;
}
```

```
Global count = 100000
```

Semaphores

- Υλοποιούνται στον πυρήνα
- Ο πυρήνας κρατά ένα counter που όταν είναι > 0 σημαίνει ότι είναι το semaphore είναι ελεύθερο και όταν είναι 0 τότε πρέπει να περιμένουμε για να το χρησιμοποιήσουμε

Χρήση Semaphores

- Για δημιουργία semaphore

HANDLE CreateSemaphore (LPSECURITY_ATTRIBUTES lpsa,
LONG lSemInitial, LONG lSemMax, LPCTSTR lpSemName)

  - *lpsa: Παράμετροι ασφάλειας semaphore*
  - lSemInitial: Αρχική τιμή του semaphore
  - lSemMax: Μέγιστη τιμή του semaphore
  - lpSemName : Προαιρετικό όνομα του semaphore

- Για να αποκτήσουμε τον έλεγχο σε ένα Semaphore χρησιμοποιούμε τα WaitForSingleObject και WaitForMultipleObjects που είδαμε πιο πριν

- Για απελευθέρωση

BOOL ReleaseSemaphore (HANDLE hSemaphore,
LONG cReleaseCount, LPLONG lpPreviousCount)

  - hSemaphore: Το semaphore που θέλουμε να ελευθερώσουμε
  - cReleaseCount: Αριθμό semaphore που θέλουμε να ελευθερώσουμε (συνήθως 1)
  - lpPreviousCount: Μας επιστρέφει το προηγούμενο αριθμό του semaphore εάν θέλουμε

Περιορισμοί Semaphores

- Όταν θέλουμε να μειώσουμε το count του semaphore περισσότερο από 1, δεν μπορούμε με το WaitForMultipleObjects γιατί όταν στο array υπάρχει το ίδιο handle περισσότερο από μια φορά παίρνουμε error. Έτσι, είμαστε αναγκασμένοι να χρησιμοποιήσουμε διαδοχικές κλήσεις στην συνάρτηση WaitForSingleObject. Όμως τότε υπάρχει το πρόβλημα ότι το νήμα μπορεί να γίνει preempt, πριν προλάβει να δεσμεύσει τον σωστό αριθμό.

- Πιθανή λύση: Να χρησιμοποιήσουμε συνδυασμό CRITICAL_SECTION με Semaphores.

```
EnterCriticalSection (&csSem);
WaitForSingleObject (hSem, INFINITE);
WaitForSingleObject (hSem, INFINITE);
LeaveCriticalSection (&csSem);
```

Παράδειγμα Semaphores

```
volatile INT count;
HANDLE semaphore;

void CountThread(INT iterations)
{
  LONG semaCount;
  for (int i=0; i<iterations; i++)
  {
    WaitForSingleObject(semaphore, INFINITE);
    count++;
    ReleaseSemaphore(semaphore, 1, &semaCount);
  }
}

const INT numThreads=4;

int main()
{
  HANDLE handles[numThreads];
  DWORD threadID;
  semaphore = CreateSemaphore(0, 1, 1, 0);

  for (int i=0; i<numThreads; i++)
    handles[i]=CreateThread(0, 0, (LPTHREAD_START_ROUTINE) CountThread,(VOID *) 25000, 0, &threadID);

  WaitForMultipleObjects(numThreads, handles,TRUE, INFINITE);
  CloseHandle(semaphore);
  cout << "Global count = " << count << endl;
  return 0;
}
```

```
Global count = 100000
```

Events

- Υλοποιούνται στον πυρήνα
- Χρησιμοποιούνται για να ενημερώσουν κάποιο νήμα που περιμένει για ένα γεγονός
- Δυο είδη. manual-reset και auto-reset. Τα manual-reset μπορούν να ειδοποιήσουν πολλά νήματα ταυτόχρονα και μένουν ενεργά μέχρι να απενεργοποιηθούν με κλήση συστήματος, ενώ τα auto-reset ειδοποιούν ένα νήμα κάθε φορά και απενεργοποιούνται αυτόματα

Χρήση Events

- Για δημιουργία event

HANDLE CreateEvent (LPSECURITY_ATTRIBUTES lpsa,
BOOL bManualReset, BOOL bInitialState,
LPTCSTR lpEventName)

  - lpsa: Παράμετροι ασφάλειας event
  - bManualReset : TRUE εάν θέλουμε το event να είναι manual-reset
  - bInitialState: TRUE εάν θέλουμε το event να είναι ενεργοποιημένο
  - lpEventName : Προαιρετικό όνομα του event

- Για να πάρουμε το handle ενός event

HANDLE OpenEvent (DWORD dwDesiredAccess,
BOOL bInheritHandle, LPCTSTR lpName)

  - dwDesiredAccess: Δικαιώματα που θέλουμε για το event
  - bInheritHandle: TRUE εάν θέλουμε το handle να μπορεί να κληρονομηθεί από άλλα process
  - lpName: Το όνομα του event που θέλουμε

Χρήση Events

- Για να ενεργοποιήσουμε ένα event

BOOL SetEvent (HANDLE hEvent)

- Για να απενεργοποιήσουμε ένα event

BOOL ResetEvent (HANDLE hEvent)

- Για να ενεργοποιήσουμε ένα event και να το απενεργοποιήσουμε ταυτόχρονα εάν είναι manual-reset. Εάν είναι auto-reset έχει το ίδιο αποτέλεσμα με το SetEvent

BOOL PulseEvent (HANDLE hEvent)

Παράδειγμα Events

```
volatile INT count;
HANDLE event;

void CountThread(INT iterations)
{
for (int i=0; i<iterations; i++)
  {
    WaitForSingleObject(event, INFINITE);
    count++;
    SetEvent(event);
  }
}

const INT numThreads=4;

int main()
{
  HANDLE handles[numThreads];
  DWORD threadID;

  event = CreateEvent(0, FALSE, TRUE, 0);

  for (int i=0; i<numThreads; i++)
    handles[i]=CreateThread(0, 0, (LPTHREAD_START_ROUTINE) CountThread, (VOID *) 25000, 0, &threadID);

WaitForMultipleObjects(numThreads, handles, TRUE, INFINITE);

  CloseHandle(event);

  cout << "Global count = " << count << endl;
}
```

```
Global count = 100000
```

Βιβλιογραφία

- Johnson, M. H. (2004). Windows System Programming Third Edition. Addison Wesley Professional

- Marshall, B. & Ron, R. (2000). Win32 System Services: The Heart of Windows 98 and Windows 2000, Third Edition. Prentice Hall PTR

- Microsoft Co., (2007). MSDN Documentation. http://msdn2.microsoft.com/en-us/default.aspx