Κεφάλαιο

1 Το μοντέλο διεργασιών του Win32

Τα λειτουργικά συστήματα που υλοποιούν όλο ή τμήμα του μοντέλου του Win32 (αυτή τη στιγμή Windows NT, Windows 95/98) υποστηρίζουν *προεκτοπιστική πολυνηματική επεξεργασία* (preemptinve multithreading). Αυτό σημαίνει ότι ο πυρήνας του λειτουργικού συστήματος θεωρεί τα νήματα ως κύρια μονάδα εκτέλεσης, τα οποία εκτελούνται εκ περιτροπής για ένα καθορισμένο χρονικό διάστημα. Μεταξύ των λειτουργικών συστημάτων αυτών υπάρχουν διαφορές ως προς την εκτέλεση παλαιότερων (16-bit) εφαρμογών, αλλά εδώ θα ασχοληθούμε με τις 32-bit εφαρμογές για τις οποίες η αντιμετώπιση είναι κοινή.

Νήματα και διεργασίες

Το Win32 διαχειρίζεται *διεργασίες* (processes), κάθε μία από τις οποίες μπορεί να αποτελείται από πολλά *νήματα εκτέλεσης* (threads of execution). Μια διεργασία είναι το στιγμιότυπο μίας εκτελούμενης εφαρμογής και για το ΛΣ αποτελείται από:

- Ένα χώρο εικονικών διευθύνσεων μεγέθους 4 GB (διευθύνσεις 32 bit). Από τα 4GB αυτά, ένα τμήμα είναι διαθέσιμο στη διεργασία, ενώ το υπόλοιπο είναι διευθύνσεις του πυρήνα.
- Τμήματα φυσικής μνήμης αντιστοιχισμένα σε εικονικές διευθύνσεις της διεργασίας.
- Τμήματα του swap file αντιστοιχισμένα σε εικονικές διευθύνσεις της διεργασίας.
- Τμήματα του εκτελέσιμου αρχείου που εκτελεί η διεργασία αντιστοιχισμένα σε εικονικές διευθύνσεις της διεργασίας.
- Δεσμευμένους πόρους του συστήματος όπως ανοιχτά αρχεία, αντικείμενα συγχρονισμού κλπ.
- Τουλάχιστον ένα νήμα εκτέλεσης.

Η βασική μονάδα εκτέλεσης στο Win32 είναι το *νήμα* (thread). Κάθε νήμα «περιέχεται» σε μία διεργασία και έχει πρόσβαση στο χώρο μνήμης αυτής. Εκτός από τη διεργασία στην οποία αντιστοιχεί, το ΛΣ χρειάζεται να γνωρίζει για κάθε νήμα:

- Τον κώδικα που εκτελεί, δηλαδή ουσιαστικά την τρέχουσα τιμή του PC για το νήμα.
- Τη στοίβα εκτέλεσής του.

Τα νήματα είναι οι βασικές οντότητες στις οποίες το λειτουργικό σύστημα αποδίδει χρόνο ΚΜΕ. Για κάθε νήμα το ΛΣ διατηρεί ένα σύνολο δεδομένων όπου κρατείται το *περιεχόμενό* του (context) όσο το νήμα περιμένει να δρομολογηθεί για επεξεργασία. Το περιεχόμενο ενός νήματος αποτελείται από τις τιμές των καταχωρητών, τη *στοίβα του πυρήνα* (kernel stack), μια εγγραφή για το *περιβάλλον* (environment) του νήματος, και τη στοίβα του ίδιου του νήματος που βρίσκεται στο χώρο διευθύνσεων της διεργασίας.

Όλα τα νήματα μιας διεργασίας μοιράζονται τον εικονικό χώρο διευθύνσεών της και μπορούν να προσπελάσουν τις παγκόσμιες μεταβλητές της όπως και τους πόρους συστήματος που κατέχει η διεργασία.

Όταν δημιουργείται μια διεργασία, τα Windows NT αυτόματα δημιουργούν το πρώτο της νήμα, το οποίο λέγεται και το *κύριο* (primary) νήμα. Αυτό μπορεί μετά να δημιουργήσει επιπλέον νήματα της διεργασίας αυτής και εκείνα με τη σειρά τους άλλα. Αν ένα οποιοδήποτε νήμα μιας διεργασίας δεν έχει τερματίσει, η διεργασία δεν πεθαίνει · μια διεργασία τερματίζεται όταν όλα της τα νήματα έχουν τερματίσει.

Τα νήματα μίας διεργασίας μπορούν να δημιουργήσουν άλλες διεργασίες, οι οποίες εκτελούνται με διαφορετικό χώρο εικονικών διευθύνσεων από την αρχική. Μεταξύ των διεργασιών δεν υπάρχει σχέση γονέα-παιδιού όπως σε άλλα ΛΣ.

ΑΝΑΦΟΡΑ ΣΕ ΔΙΕΡΓΑΣΙΕΣ ΚΑΙ ΝΗΜΑΤΑ

Μπορούμε μέσα σε ένα πρόγραμμα να αναφερθούμε σε μία διεργασία ή ένα νήμα με δύο τρόπους:

(α) Με το αναγνωριστικό της/του (process id ή thread id).

(β) Με ένα handle σε αυτήν/αυτό. Τα handles είναι ειδικά αναγνωριστικά που διαθέτει το Win32 στον προγραμματιστή για να μπορεί να αναφέρεται στα αντικείμενα του ΛΣ: διεργασίες/ νήματα, αρχεία, σηματοφορείς κλπ.

Για να πάρει ένα νήμα το handle της διεργασίας του, καλεί τη ρουτίνα GetCurrentProcess():

```
HANDLE GetCurrentProcess(VOID);
```

Η ρουτίνα αυτή επιστρέφει ένα ψευδο-handle της τρέχουσας διεργασίας. Το ψευδο-handle είναι μια συγκεκριμένη σταθερά η οποία ερμηνεύεται από το ΛΣ ως το handle της τρέχουσας διεργασίας. Για να δημιουργήσει το νήμα ένα κανονικό handle για τη διεργασία πρέπει να φτιάξει ένα αντίγραφο του ψευδο-handle με την DuplicateHandle().

Για να πάρει ένα νήμα το αναγνωριστικό της διεργασίας του καλεί την GetCurrentProcessID():

```
DWORD GetCurrentProcessId(VOID);
```

Το αναγνωριστικό αυτό είναι μοναδικό στο σύστημα.

Το πρόγραμμα που ακολουθεί τυπώνει το handle και το αναγνωριστικό της διεργασίας και του κύριου νήματός του.

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

main(void)
{
  HANDLE hPseudoHandle, hRealHandle;
  DWORD dwProcessId, dwThreadId;

  hPseudoHandle = GetCurrentProcess();
  dwProcessId = GetCurrentProcessId();

  DuplicateHandle(GetCurrentProcess(), hPseudoHandle,
                  GetCurrentProcess(), &hRealHandle,
                  0, FALSE, DUPLICATE_SAME_ACCESS);

  printf("Process pseudo-handle is 0x%lx\n",
         (unsigned long)hPseudoHandle);

  printf("Process real handle is 0x%lx\n",
         (unsigned long)hRealHandle);

  printf("Process id is %lu\n",
         (unsigned long)dwProcessId);

  CloseHandle(hRealHandle);

  hPseudoHandle = GetCurrentThread();
  dwThreadId = GetCurrentThreadId();

  DuplicateHandle(GetCurrentProcess(), hPseudoHandle,
                  GetCurrentProcess(), &hRealHandle,
                  0, FALSE, DUPLICATE_SAME_ACCESS);
```

```
  printf("Thread pseudo-handle is 0x%lx\n",
         (unsigned long)hPseudoHandle);

  printf("Thread real handle is 0x%lx\n",
         (unsigned long)hRealHandle);

  printf("Thread id is %lu\n",
         (unsigned long)dwThreadId);

  CloseHandle(hRealHandle);

  return 0;
}
```

Αν το εκτελέσουμε[1] θα δούμε:

```
Process pseudo-handle is 0xffffffff
Process real handle is 0x4c
Process id is 252
Thread pseudo-handle is 0xfffffffe
Thread real handle is 0x50
Thread id is 204
```

ΔΗΜΙΟΥΡΓΙΑ ΝΗΜΑΤΩΝ

Ένα νήμα μπορεί να δημιουργήσει κάποιο άλλο νήμα της ίδιας διεργασίας, το οποίο εκτελεί μία ρουτίνα του προγράμματος. Αυτό γίνεται με την κλήση CreateThread:

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

Οι παράμετροι της ρουτίνας είναι:

- lpThreadAttributes: παράμετρος σχετική με ασφάλεια που χρησιμοποιείται μόνο στα Windows NT. Δίνουμε τιμή NULL.
- dwStackSize: το αρχικό μέγεθος της στοίβας του νέου νήματος, 0 αν δε θέλουμε να το προσδιορίσουμε (οπότε παίρνει μια προκαθορισμένη τιμή)
- lpStartAddress: η ρουτίνα την οποία θα εκτελέσει το νήμα, η οποία παίρνει μια 32-bit παράμετρο και επιστρέφει μια 32-bit τιμή.
- lpParameter: η τιμή της παραμέτρου που θα περαστεί στη ρουτίνα
- lpThreadId: δείκτης σε μια μεταβλητή τύπου DWORD όπου θα αποθηκευθεί το αναγνωριστικό του νήματος

Αν η ρουτίνα επιτύχει, επιστρέφει το handle του νήματος που δημιουργήθηκε, αλλιώς επιστρέφει NULL. Στη δεύτερη περίπτωση ο κωδικός του λάθους δίνεται από την GetLastError().

Στο πρόγραμμα που ακολουθεί, το κύριο νήμα κατασκευάζει ένα άλλο νήμα που εκτελεί τη ρουτίνα MyRoutine() με παράμετρο 0 και μετά το ίδιο εκτελεί την ίδια ρουτίνα με παράμετρο 1[2].

[1] Όλα τα προγράμματα των παραδειγμάτων έχουν εκτελεστεί σε Windows NT.

```
/* Make with cl /MT sample2.c */

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

DWORD MyRoutine(DWORD dwNum)
{
  int i;

  for (i=0; i<8; i++) {
    printf("Thread %d iteration %d\n", (int)dwNum, i);

    Sleep(10); /* Delay for 10 ms */
  }

  return 0;
}

main(void)
{
  HANDLE hThread;
  DWORD dwThreadId, dwError;

  hThread = CreateThread(NULL, 0,
                         (LPTHREAD_START_ROUTINE)MyRoutine,
                         (LPVOID)0, 0, &dwThreadId);

  if (NULL == hThread) {
    dwError = GetLastError();
    fprintf(stderr, "Error %ld creating thread.\n", (long)dwError);
    return 1;
  }

  MyRoutine(1);

  CloseHandle(hThread);

  return 0;
}
```

Αν τρέξουμε το πρόγραμμα αυτό θα δούμε κάτι σαν αυτό:

```
Thread 1 iteration 0
Thread 0 iteration 0
Thread 1 iteration 1
Thread 0 iteration 1
Thread 1 iteration 2
Thread 0 iteration 2
Thread 1 iteration 3
Thread 0 iteration 3
Thread 1 iteration 4
Thread 0 iteration 4
Thread 1 iteration 5
Thread 0 iteration 5
Thread 1 iteration 6
Thread 0 iteration 6
Thread 1 iteration 7
Thread 0 iteration 7
```

[2] Το πρόγραμμα αυτό ακολουθεί μία «κακή» πρακτική: χρησιμοποιεί ρουτίνες της C library (printf κλπ.) σε συνδυασμό με την CreateThread, κάτι το οποίο προκαλεί πολύ μικρές απώλειες μνήμης. Κανονικά θα έπρεπε να χρησιμοποιήσουμε την beginthreadex που έχει σχεδόν τις ίδιες παραμέτρους με την CreateThread.

ΔΗΜΙΟΥΡΓΙΑ ΔΙΕΡΓΑΣΙΩΝ

Μια διεργασία και το κύριο νήμα της δημιουργείται με τη ρουτίνα CreateProcess, η οποία εκτελεί το προσδιοριζόμενο εκτελέσιμο αρχείο.

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

Οι βασικές παράμετροι της ρουτίνας είναι:

- lpApplicationName: το όνομα (πλήρες ή σχετικό με το τρέχον μονοπάτι) του εκτελέσιμου αρχείου (.exe).
- lpCommandLine: περιέχει τις παραμέτρους που θα περαστούν στο πρόγραμμα, NULL αν δε θέλουμε να περάσουμε παραμέτρους. Αν το lpApplicationName είναι NULL, τότε το όνομα του εκτελέσιμου αρχείου είναι το πρώτο token του lpCommandLine και τα υπόλοιπα δίνουν τις παραμέτρους.
- lpProcessAttributes και lpThreadAttributes: παράμετροι σχετικές με ασφάλεια, οι οποίες έχουν νόημα μόνο στα Windows NT. Συνήθως έχουν τιμή NULL.
- bInheritHandles: καθορίζει αν η νέα διεργασία θα κληρονομήσει τα handles των αρχείων και των άλλων αντικειμένων αυτής που τη δημιουργεί.
- dwCreationFlags: καθορίζει παραμέτρους για τη δημιουργία της διεργασίας,όπως η κατηγορία προτεραιότητάς της και το αν θα δημιουργηθεί σε κατάσταση suspended. Οι παράμετροι συνδυάζονται με τον δυαδικό OR τελεστή (|) για να δώσουν την τιμή του dwCreationFlags. Στην απλούστερη περίπτωση η τιμή του είναι 0, οπότε η διεργασία δημιουργείται και αρχίζει να εκτελείται.
- lpEnvironment και lpCurrentDirectory: χρησιμοποιούνται για να αλλάξουμε τις μεταβλητές περιβάλλοντος και τον τρέχοντα κατάλογο για τη νέα διεργασία. Γενικά έχουν την τιμή NULL.
- lpStartupInfo: δείκτης σε μια δομή τύπου STARTUPINFO η οποία καθορίζει πώς θα εμφανιστεί το παράθυρο της νέας διεργασίας. Η δομή αυτή μπορεί να χρησιμοποιηθεί για την ανακατεύθυνση των standard input, output και error stream της διεργασίας. Πριν από την κλήση της CreateProcess πρέπει να αρχικοποιηθεί το πεδίο cb της δομής με το μέγεθός της (βλ. παράδειγμα).
- lpProcessInformation: δείκτης σε μια δομή τύπου PROCESS_INFORMATION στην οποία μετά την επιστροφή της CreateProcess θα είναι αποθηκευμένα τα handles της διεργασίας και του κύριου νήματός της (πεδία hProcess και hThread) και τα αναγνωριστικά τους (πεδία dwProcessId και dwThreadId).

Η ρουτίνα επιστρέφει TRUE αν επιτύχει και FALSE αν αποτύχει. Στη δεύτερη περίπτωση ο κωδικός του λάθους δίνεται από τη ρουτίνα GetLastError().

Το παρακάτω πρόγραμμα δημιουργεί μία νέα διεργασία με το πρόγραμμα Notepad.

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

```
#include <stdio.h>

main(void)
{
  STARTUPINFO si;
  PROCESS_INFORMATION pi;
  DWORD dwError;
  HANDLE hProcess;

  ZeroMemory(&si, sizeof(STARTUPINFO));
  si.cb = sizeof(STARTUPINFO);

  if (!CreateProcess(NULL, "notepad.exe",
                     NULL, NULL,
                     FALSE, 0, NULL, NULL, &si, &pi)) {
    dwError = GetLastError();
    fprintf(stderr, "Error %ld in process creation.\n", (long)dwError);
    return 1;
  }

  printf("Process created successfully.\n"
         "Process handle is 0x%lx, process id is %ld.\n"
         "Main thread handle is 0x%lx, main thread id is %ld.\n",
         (unsigned long)pi.hProcess, (long)pi.dwProcessId,
         (unsigned long)pi.hThread, (long)pi.dwThreadId);

  return 0;
}
```

Και το αποτέλεσμά του είναι:

```
Process created successfully.
Process handle is 0x78, process id is 263.
Main thread handle is 0x10, main thread id is 161.
```

Τερματισμός διεργασιών και νημάτων

Μια διεργασία μπορεί να τερματιστεί με δυο τρόπους: είτε καλώντας τη ρουτίνα ExitProcess (αυτή είναι η συνηθισμένη μέθοδος) είτε καλώντας τη ρουτίνα TerminateProcess · αυτή η μέθοδος πρέπει να χρησιμοποιείται μόνο σαν έσχατη λύση.

Μια διεργασία τερματίζει όταν ένα από τα νήματά της καλεί τη ρουτίνα Exit Process:

```
VOID ExitProcess(UINT fuExitCode);
```

Αυτή η ρουτίνα προκαλεί τον τερματισμό της διεργασίας και θέτει τον κωδικό εξόδου (exit code) της διεργασίας στην τιμή uExitCode. Η ExitProcess δεν επιστρέφει τιμή γιατί η διεργασία έχει τερματιστεί. Κώδικας που ακολουθεί την κλήση στην ExitProcess δε θα εκτελεστεί ποτέ. Αυτή η μέθοδος τερματισμού είναι η πιο κοινή γιατί η ExitProcess καλείται όταν η main ή η WinMain (αντίστοιχη της main για παραθυρικές εφαρμογές) επιστρέψει στον κώδικα αρχικοποίησης της C Runtime βιβλιοθήκης. Ο κώδικας αυτός καλεί την ExitProcess περνώντας της την τιμή που έχει επιστρέψει η WinMain. Τα υπόλοιπα νήματα της διεργασίας τότε τερματίζονται μαζί με το κύριο.

Η κλήση της TerminateProcess τερματίζει επίσης μια διεργασία:

```
BOOL TerminateProcess(HANDLE hProcess, UINT fuExitCode);
```

Αυτή η ρουτίνα είναι διαφορετικά από την ExitProcess σε ένα πολύ βασικό σημείο: Μια διεργασία μπορεί να καλέσει την TerminateProcess για να τερματίσει τον εαυτό της ή και μια άλλη διεργασία. Η παράμετρος hProcess καθορίζει το handle της διεργασίας που θα τερματιστεί. Κατά τον τερματισμό της διεργασίας ο κωδικός εξόδου της καθορίζεται από την παράμετρο fuExitCode.

Πρέπει να σημειωθεί ότι η χρήση της TerminateProcess γενικώς αντενδείκνυται · πρέπει να χρησιμοποιείται μόνο αν μια διεργασία δεν μπορεί να τερματιστεί με κανένα άλλο, λιγότερο «βίαιο» τρόπο.

Όταν μια διεργασία τερματίζεται ενεργοποιούνται οι ακόλουθες διαδικασίες:

1. Ολα τα νήματα της διεργασίας τερματίζουν και αυτά την εκτέλεσή τους.
2. Ολα τα handles αντικειμένων του συστήματος (π.χ. πόροι του συστήματος) που κατείχε η διεργασία αποδεσμεύονται.
3. Ενεργοποιούνται τα νήματα του συστήματος που περίμεναν τη διεργασία να τερματίσει.
4. Η κατάσταση τερματισμού της διεργασίας από STILL_ACTIVE παίρνει την τιμή του κωδικού εξόδου.

Όταν η διεργασία τερματίζεται, το αντίστοιχο αντικείμενο στο σύστημα δεν απελευθερώνεται αυτόματα μέχρις ότου όλες οι αναφορές στο αντικείμενο πάψουν να υφίστανται. Επίσης ο τερματισμός μιας διεργασίας δεν προκαλεί τον τερματισμό των άλλων διεργασιών που αυτή έχει δημιουργήσει.

Ο εκτελέσιμος κώδικας της διεργασίας και οι πόροι του συστήματος που κατείχε απομακρύνονται από τη μνήμη. Όμως το τμήμα της μνήμης που κρατά το σύστημα για το αντικείμενο-διεργασία (δηλαδή το PCB) δεν ελευθερώνεται μέχρι ο μετρητής αναφορών της διεργασίας πάρει την τιμή 0. Αυτό θα γίνει όταν όλες οι άλλες διεργασίες που κρατούν handles για το αντικείμενο-διεργασία ενημερώσουν το σύστημα με κληση στην CloseHandle ότι δε χρειάζονται πλέον να αναφερθούν σε αυτό. Αφού η διεργασία έχει τερματίσει, το μόνο που μπορεί να κάνει κανείς με το handle της είναι να πάρει τον κωδικό εξόδου της με την GetExitCodeProcess.

```
BOOL GetExitCodeProcess(HANDLE hProcess, LPDWORD lpdwExitCode);
```

Ο κωδικός εξόδου επιστρέφεται στην παράμετρο τύπου DWORD που δείχνει η lpdwExitCode. Αν η διεργασία δεν έχει τερματιστεί όταν κληθεί η GetExitCodeProcess η θέση που δείχνει η lpdwExitCode παίρνει την τιμή STILL_ACTIVE (0x103). Αν η κλήση της ρουτίνας είναι επιτυχής επιστρέφεται TRUE.

Ένα νήμα μπορεί να τερματιστεί με τρεις τρόπους, τους οποίους θα ονομάσουμε «φυσικό θάνατο», «αυτοκτονία» και «φόνο».

Ένα νήμα πεθαίνει από «φυσικά αίτια» όταν η ρουτίνα του επιστρέψει. Η τιμή που επιστρέφει η ρουτίνα είναι και ο κωδικός εξόδου του νήματος, παρόμοιος με αυτόν των διεργασιών. Ο κωδικός εξόδου ενός νήματος δίνεται από τη ρουτίνα GetExitCodeThread:

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

Η παράμετρος hThread καθορίζει το handle του νήματος και η lpExitCode δείχνει στη θέση που θα αποθηκευθεί ο κωδικός.

Αν η ρουτίνα επιτύχει επιστρέφει TRUE, αλλιώς επιστρέφεται FALSE και ο κωδικός του λάθους δίνεται από την GetLastError. Αν το νήμα δεν έχει τελειώσει, τότε στη μεταβλητή που δείχνει η lpExitCode δίνεται η τιμή STILL_ACTIVE (0x103).

Στο πρόγραμμα που ακολουθεί, το κύριο νήμα δημιουργεί ένα δεύτερο νήμα και κάνει busy-wait μέχρι να τελειώσει:

```
/* Make with cl /MT sample5.c */

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

DWORD MyRoutine(DWORD dwNum)
{
  int i;

  for (i=0; i<8; i++) {
    printf("Thread %d iteration %d\n", (int)dwNum, i);

    Sleep(10); /* Delay for 10 ms */
  }

  return 0;
}

main(void)
{
  HANDLE hThread;
  DWORD dwThreadId, dwExitCode, dwError;

  hThread = CreateThread(NULL, 0,
                         (LPTHREAD_START_ROUTINE)MyRoutine,
                         (LPVOID)0, 0, &dwThreadId);

  if (NULL == hThread) {
    dwError = GetLastError();
    fprintf(stderr, "Error %ld creating thread.\n", (long)dwError);
    return 1;
  }

  do {
    if (!GetExitCodeThread(hThread, &dwExitCode)) {
      dwError = GetLastError();
      fprintf(stderr, "Error %ld in GetLastError().\n", (long)dwError);
      return 1;
    }

    printf("Exit code is 0x%lx.\n", (long)dwExitCode);
    Sleep(20);
  } while (STILL_ACTIVE == dwExitCode);

  CloseHandle(hThread);

  return 0;
}
```

Αν το τρέξουμε θα δούμε:

```
Exit code is 0x103.
Thread 0 iteration 0
Exit code is 0x103.
Thread 0 iteration 1
Thread 0 iteration 2
Exit code is 0x103.
Thread 0 iteration 3
Exit code is 0x103.
Thread 0 iteration 4
Thread 0 iteration 5
Exit code is 0x103.
Thread 0 iteration 6
Exit code is 0x103.
Thread 0 iteration 7
Exit code is 0x0.
```

Ένα νήμα «αυτοκτονεί» καλώντας τη ρουτίνα ExitThread().

```
VOID ExitThread(DWORD dwExitCode);
```

Η παράμετρος dwExitCode καθορίζει τον κωδικό εξόδου του νήματος που μπορεί να ληφθεί με τη ρουτίνα GetExitCodeThread().

Η ρουτίνα ExitThread καλείται αυτόματα και όταν ένα νήμα επιστρέψει από τη ρουτίνα του με παράμετρο την τιμή επιστροφής. Έτσι η κλήση return 0 μέσα στη ρουτίνα ενός νήματος είναι ισοδύναμη με την ExitThread(0).

Ο πιο βίαιος τρόπος τερματισμού ενός νήματος είναι να «δολοφονηθεί» με τη ρουτίνα TerminateThread(), η οποία μπορεί να κληθεί από ένα νήμα της ίδιας ή άλλης διεργασίας που γνωρίζει το handle του νήματος (και έχει δικαιώματα πρόσβασης σε αυτό για τα Windows NT):

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

Η παράμετρος hThread δίνει το handle του νήματος που θα δολοφονηθεί και η τιμή dwExitCode θα πάρει τη θέση του κωδικού εξόδου του. Η ρουτίνα επιστρέφει TRUE αν επιτύχει, αλλιώς FALSE, με τον κωδικό λάθους να δίνεται από τη ρουτίνα GetLastError().

Η TerminateThread είναι μια επικίνδυνη ρουτίνα που πρέπει να χρησιμοποιείται μόνο σε έσχατη ανάγκη. Πρέπει να καλείται μόνο αν ο καλών έχει πλήρη γνώση του κώδικα που εκτελεί το νήμα. Μερικά από τα προβλήματα που μπορεί να προκαλέσει η TerminateThread είναι:

- Αν το «θύμα» βρίσκεται μέσα σε ένα κρίσιμο τμήμα, αυτό δε θα ελευθερωθεί.
- Αν το νήμα εκτελεί ορισμένες κλήσεις του πυρήνα, η κατάσταση του πυρήνα για τη διεργασία του μπορεί να είναι ασυνεπής.
- Αν το νήμα διαχειρίζεται μια μοιραζόμενη δυναμική βιβλιοθήκη (DLL), η κατάστασή της μπορεί να καταστραφεί επηρεάζοντας τους άλλους χρήστες της.

Αν το τερματιζόμενο νήμα είναι το τελευταίο της διεργασίας του, αυτή τερματίζεται επίσης.

Όταν ένα νήμα πεθαίνει από φυσικά αίτια ή αυτοκτονεί, η μνήμη για τη στοίβα του απελευθερώνεται. Όταν όμως το νήμα δολοφονείται, η στοίβα διατηρείται μέχρι τον τερματισμό της διεργασίας γιατί άλλα νήματά της μπορεί ακόμα να χρησιμοποιούν δείκτες προς τη στοίβα αυτή.

Οι ρουτίνες ExitProcess() και TerminateProcess() δολοφονούν όλα τα νήματα μιας διεργασίας, αλλά οι στοίβες τους δεν διατηρούνται γιατί αμέσως μετά ελευθερώνεται όλος ο χώρος διευθύνσεων της διεργασίας.

Όταν ένα νήμα πεθαίνει, ενεργοποιούνται οι ακόλουθες διαδικασίες:

1. Ελευθερώνονται όλα τα αντικείμενα και οι πόροι του συστήματος που κατείχε το νήμα.
2. Ενεργοποιούνται τα νήματα που περίμεναν το νήμα να τερματίσει.
3. Η κατάσταση τερματισμού του νήματος από την τιμή STILL_ACTIVE παίρνει την τιμή του κωδικού εξόδου για το νήμα, ο οποίος καθορίζεται διαφορετικά για κάθε τρόπο θανάτου.
4. Αν το νήμα ήταν το τελευταίο ενεργό της διεργασίας, αυτή τερματίζεται επίσης.

Όταν τερματίζεται ένα νήμα, το αντίστοιχο thread control block του ΛΣ δεν ελευθερώνεται μέχρι να κλείσουν όλες οι αναφορές προς αυτό.

Κεφάλαιο

2 Χρονοδρομολόγηση νημάτων

Η χρονοδρομολόγηση των νημάτων γίνεται με αλγόριθμο κυκλικής επαναφοράς (round-robin) με δυναμικές προτεραιότητες. Υποψήφια για δρομολόγηση είναι όλα τα ενεργά νήματα. Ένα νήμα μπορεί να είναι ανενεργό αν:

- Το νήμα έχει δημιουργηθεί με την CreateThread() και στην παράμετρο dwCreationFlags έχει δοθεί και το flag CREATE_SUSPENDED.
- Έχουν εκτελεστεί μια ή περισσότερες κλήσεις της SuspendThread() για το νήμα αυτό.
- Το νήμα βρίσκεται σε κατάσταση αναμονής σε κάποιο αντικείμενο συγχρονισμού (με μια από τις ρουτίνες WaitXXX()) ή «κοιμάται» μετά από κλήση της Sleep().
- Το νήμα έχει καλέσει την GetMessage() για να διαβάσει το επόμενο μήνυμα από την ουρά μηνυμάτων του αλλά δεν είναι κάποιο διαθέσιμο εκείνη τη στιγμή.

Η αντιστοίχιση προτεραιότητας στα νήματα γίνεται σε τρία επίπεδα:

- Κάθε διεργασία ανήκει σε μια κλάση προτεραιότητας.
- Το νήμα έχει μια σχετική προτεραιότητα ως προς τη διεργασία.
- Το ΛΣ αυξάνει και μειώνει προσωρινά την προτεραιότητα ενός νήματος (priority boost).

Η τελική τιμή της προτεραιότητας ενός νήματος είναι μεταξύ 0 και 31.

ΚΛΆΣΕΙΣ ΠΡΟΤΕΡΑΙΟΤΉΤΩΝ

Οι κλάσεις προτεραιοτήτων στο Win32 είναι τέσσερις: ανενεργή (idle), κανονική (normal), υψηλή (high) και πραγματικού χρόνου (realtime). Για να προσδιορίσουμε την κλάση προτεραιότητας μιας διεργασίας χρησιμοποιούμε την παράμετρο dwCreationFlags της CreateProcess(), κάνοντας OR (|) τις υπόλοιπες τιμές που θέλουμε να της δώσουμε με μια από τις:

Κλάση	**Τιμή**	**Επίπεδο προτεραιότητας**
Ανενεργή	IDLE_PRIORITY_CLASS	4
Κανονική	NORMAL_PRIORITY_CLASS	9 / 7
Υψηλή	HIGH_PRIORITY_CLASS	13
Πραγματικού χρόνου	REALTIME_PRIORITY_CLASS	24

Αν η τιμή δεν προσδιοριστεί, τότε η διεργασία ανήκει στην κανονική κλάση. Οι διεργασίες της κανονικής κλάσης έχουν επίπεδο προτεραιότητας 9 αν βρίσκονται στο προσκήνιο (foreground) και 7 αν τρέχουν στο παρασκήνιο (background).

Διεργασίες που ανήκουν στην ανενεργή κλάση διακόπτονται για χάρη διεργασιών υψηλότερης κλάσης. Το επίπεδο αυτό είναι κατάλληλο για εφαρμογές όπως screen savers και μετρητές των πόρων του συστήματος (system resources monitors).

Η υψηλή κλάση προτεραιότητας είναι κατάλληλη για διεργασίες που χρειάζονται την «κυριαρχία» της ΚΜΕ για μικρά χρονικά διαστήματα αλλά είναι ανενεργές τον περισσότερο χρόνο. Ένα παράδειγμα τέτοιας εφαρμογής είναι ένας διαχειριστής διεργασιών (task manager) ο οποίος δεν

χρησιμοποιείται συχνά αλλά θέλουμε να έχει υψηλή προτεραιότητα όταν ενεργοποιείται (π.χ. για να σκοτώσει μια διεργασία που έχει κολλήσει).

Η κλάση προτεραιοτήτων πραγματικού χρόνου γενικά δεν πρέπει να χρησιμοποιείται, γιατί διεργασίες πραγματικού χρόνου που εκτελούνται για περισσότερο από ελάχιστο χρόνο εμποδίζουν την αποκρισιμότητα του συστήματος.

Προτεραιοτητα των νημάτων

Οι προτεραιότητες των νημάτων παίρνουν τιμές από 0 (χαμηλή) έως 31 (υψηλή). Η χρονοδρομολόγησή τους γίνεται με κυκλική επαναφορά μεταξύ των ενεργών νημάτων με την υψηλότερη προτεραιότητα. Αν ένα νήμα που έχει προτεραιότητα 10 δημιουργήσει ένα νήμα με προτεραιότητα 11, τότε διακόπτεται αμέσως και η ΚΜΕ παραχωρείται στο νέο νήμα (αν βέβαια αυτό είναι έτοιμο για εκτέλεση).

Η προτεραιότητα του νήματος είναι συνδυασμός της κλάσης προτεραιότητας της διεργασίας του και της δικής του σχετικής προτεραιότητας. Η σχετική προτεραιότητα δεν αλλάζει όταν αλλάζει η κλάση της διεργασίας και μπορεί να πάρει μια από τις τιμές:

Σχετική προτεραιότητα	**Τιμή προτεραιότητας**
THREAD_PRIORITY_IDLE	Αν η κλάση είναι REALTIME_PRIORITY_CLASS τότε το νήμα έχει προτεραιότητα 16, αλλιώς έχει προτεραιότητα 0.
THREAD_PRIORITY_LOWEST	(κλάση)-2
THREAD_PRIORITY_BELOW_NORMAL	(κλάση)-1
THREAD_PRIORITY_NORMAL	(κλάση)
THREAD_PRIORITY_ABOVE_NORMAL	(κλάση)+1
THREAD_PRIORITY_HIGHEST	(κλάση)+2
THREAD_PRIORITY_TIME_CRITICAL	Αν η κλάση είναι REALTIME_PRIORITY_CLASS τότε το νήμα έχει προτεραιότητα 31, αλλιώς έχει προτεραιότητα 15.

Η σχετική προτεραιότητα αυτή είναι δυνατόν να αλλάξει, αν π.χ. η διεργασία του νήματος βρεθεί στο προσκήνιο. Η τελική προτεραιότητα των νημάτων δίνεται από τον ακόλουθο πίνακα:

	Κλάση προτεραιότητας διεργασίας				
Σχετική προτεραιότητα νήματος	**Ανενεργή**	**Κανονική, στο παρασκήνιο**	**Κανονική, στο προσκήνιο**	**Υψηλή**	**Πραγματικού χρόνου**
Time critical	15	15	15	15	31
Highest	6	9	11	15	26
Above normal	5	8	10	14	25
Normal	4	7	9	13	24
Below normal	3	6	8	12	23
Lowest	2	5	7	11	22
Idle	1	1	1	1	16

Οι ρουτίνες που χειρίζονται τη σχετική προτεραιότητα ενός νήματος είναι οι SetThreadPriority(), GetThreadPriority(), SetThreadPriorityBoost() και GetThreadPriorityBoost().

ΑΛΛΑΓΗ ΠΡΟΤΕΡΑΙΟΤΗΤΩΝ

Αλλαγή της κλάσης προτεραιότητας μιας διεργασίας

Μια διεργασία μπορεί να αλλάξει την κλάση προτεραιότητάς της με την SetPriorityClass().

```
BOOL SetPriorityClass(
    HANDLE hProcess,       // handle to the process
    DWORD dwPriorityClass  // priority class value
 );
```

Η παράμετρος hProcess δίνει τη διεργασία και η dwPriorityClass δίνει τη νέα κλάση προτεραιότητας (βλ. «Κλάσεις προτεραιοτήτων»).

Για να πάρουμε την κλάση προτεραιότητας μιας διεργασίας καλούμε την GetPriorityClass().

```
DWORD GetPriorityClass(HANDLE hProcess);
```

Για να ξεκινήσουμε μια διεργασία από τη γραμμή εντολών (command line) με δεδομένη κλάση προτεραιότητας χρησιμοποιούμε μια από τις επιλογές /LOW, /NORMAL, /HIGH, ή /REALTIME με την εντολή start. Π.χ.

```
C:\> start /LOW calc.exe
```

Τα νήματα της διεργασίας που βρίσκεται ανά πάσα στιγμή στο προσκήνιο έχουν ενισχυμένη (boosted) προτεραιότητα. Αυτό μπορεί να γίνει επίσης (ή να ακυρωθεί) με την SetProcessPriorityBoost():

```
BOOL SetProcessPriorityBoost(
    HANDLE hProcess,           // handle to process
    BOOL DisablePriorityBoost  // priority boost control state
);
```

Αν η παράμετρος DisablePriorityBoost είναι FALSE τότε τα νήματα της διεργασίας έχουν ενισχυμένη προτεραιότητα.

Για να διαπιστώσουμε αν η προτεραιότητα της διεργασίας είναι όντως αυξημένη καλούμε την GetProcessPriorityBoost():

```
BOOL GetProcessPriorityBoost(
    HANDLE hProcess,             // handle to process
    PBOOL pDisablePriorityBoost  // indicates priority boost control state
);
```

Αλλαγή της προτεραιότητας ενός νήματος

Η σχετική προτεραιότητα για ένα νήμα αλλάζει με τη ρουτίνα SetThreadPriority().

```
BOOL SetThreadPriority(
    HANDLE hThread,  // handle to the thread
    int nPriority    // thread priority level
);
```

Η παράμετος hThread καθορίζει το νήμα του οποίου θα αλλάξει η προτεραιότητα και nPriority είναι η νέα του σχετική προτεραιότητα, μια από τις τιμές THREAD_PRIORITY_IDLE, THREAD_PRIORITY_LOWEST, THREAD_PRIORITY_BELOW_NORMAL, THREAD_PRIORITY_NORMAL, THREAD_PRIORITY_ABOVE_NORMAL, THREAD_PRIORITY_HIGHEST, THREAD_PRIORITY_TIME_CRITICAL.

Η προτεραιότητα του νήματος, ανάλογα με την κλάση της διεργασίας δίνεται στην παράγραφο «Προτεραιότητα των νημάτων».

Για να πάρουμε τη σχετική προτεραιότητα ενός νήματος καλούμε την GetThreadPriority().

```
int GetThreadPriority(
    HANDLE hThread       // handle to thread
);
```

Η ρουτίνα GetThreadPriorityBoost() προσδιορίζει την προσωρινή αύξηση της προτεραιότητας ενός νήματος όταν αυτό βρίσκεται στο προσκήνιο.

```
BOOL GetThreadPriorityBoost(
    HANDLE hThread,              // handle to thread
    PBOOL pDisablePriorityBoost  // indicates priority boost control state
);
```

Η τιμή της *pDisablePriorityBoost δείχνει αν η προτεραιότητας του νήματος hThread είναι αυξημένη ή όχι.

Για να αλλάξουμε το priority boost ενός νήματος καλούμε την SetThreadPriorityBoost().

```
BOOL SetThreadPriorityBoost(
    HANDLE hThread,           // handle to thread
    BOOL DisablePriorityBoost // priority boost control state
);
```

Αν η παράμετρος DisablePriorityBoost είναι FALSE, τότε το νήμα αποκτά «ενισχυμένη» προτεραιότητα, αλλιώς έχει την συνήθη προτεραιότητα.

Αν τρέξουμε το πρόγραμμα:

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

typedef struct {
  long Value;
  char *String;
} PAIR;

PAIR PriorityClasses[] = {
  { HIGH_PRIORITY_CLASS, "HIGH_PRIORITY_CLASS" },
  { IDLE_PRIORITY_CLASS, "IDLE_PRIORITY_CLASS" },
  { NORMAL_PRIORITY_CLASS, "NORMAL_PRIORITY_CLASS" },
  { REALTIME_PRIORITY_CLASS, "REALTIME_PRIORITY_CLASS" },
  { 0, NULL }
};

PAIR ThreadPriorities[] = {
  { THREAD_PRIORITY_ABOVE_NORMAL, "THREAD_PRIORITY_ABOVE_NORMAL" },
  { THREAD_PRIORITY_BELOW_NORMAL, "THREAD_PRIORITY_BELOW_NORMAL" },
  { THREAD_PRIORITY_HIGHEST, "THREAD_PRIORITY_HIGHEST" },
  { THREAD_PRIORITY_IDLE, "THREAD_PRIORITY_IDLE" },
  { THREAD_PRIORITY_LOWEST, "THREAD_PRIORITY_LOWEST" },
  { THREAD_PRIORITY_NORMAL, "THREAD_PRIORITY_NORMAL" },
  { THREAD_PRIORITY_TIME_CRITICAL, "THREAD_PRIORITY_TIME_CRITICAL" },
  { 0, NULL }
};

char *String4Value(DWORD dwValue, PAIR *Lookup)
{
  int I;
  for (I=0; Lookup[I].Value &&
            dwValue != Lookup[I].Value;
       I++)
    ;
  return Lookup[I].String;
}
```

```
main(void)
{
  HANDLE hProcess, hThread;
  BOOL bIsBoosted;

  hProcess = GetCurrentProcess();
  hThread = GetCurrentThread();

  printf("Process priority class is %s.\n",
         String4Value(GetPriorityClass(hProcess), PriorityClasses));

  GetProcessPriorityBoost(hProcess, &bIsBoosted);
  printf("The process is %spriority boosted.\n", (bIsBoosted ? "" : "not "));

  printf("Thread priority is %s\n",
         String4Value(GetThreadPriority(hThread), ThreadPriorities));

  GetThreadPriorityBoost(hProcess, &bIsBoosted);
  printf("The thread is %spriority boosted.\n", (bIsBoosted ? "" : "not "));

  return 0;
}
```

θα δούμε:

```
Process priority class is NORMAL_PRIORITY_CLASS.
The process is not priority boosted.
Thread priority is THREAD_PRIORITY_NORMAL
The thread is not priority boosted.
```

Αν το τρέξουμε με:

```
start /LOW /B sample6.exe
```

θα δούμε:

```
Process priority class is IDLE_PRIORITY_CLASS.
The process is not priority boosted.
Thread priority is THREAD_PRIORITY_NORMAL
The thread is not priority boosted.
```

Αν το τρέξουμε με:

```
start /HIGH /B sample6.exe
```

θα δούμε:

```
Process priority class is HIGH_PRIORITY_CLASS.
The process is not priority boosted.
Thread priority is THREAD_PRIORITY_NORMAL
The thread is not priority boosted.
```

Και αν το τρέξουμε με:

```
start /REALTIME /B sample6.exe
```

θα δούμε:

```
Process priority class is REALTIME_PRIORITY_CLASS.
The process is not priority boosted.
Thread priority is THREAD_PRIORITY_NORMAL
The thread is not priority boosted.
```

Το παρακάτω πρόγραμμα θέτει όλες τις δυνατές τιμές στην προτεραιότητά του:

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

typedef struct {
  long Value;
  char *String;
} PAIR;

PAIR PriorityClasses[] = {
  { HIGH_PRIORITY_CLASS, "HIGH_PRIORITY_CLASS" },
  { IDLE_PRIORITY_CLASS, "IDLE_PRIORITY_CLASS" },
  { NORMAL_PRIORITY_CLASS, "NORMAL_PRIORITY_CLASS" },
  { REALTIME_PRIORITY_CLASS, "REALTIME_PRIORITY_CLASS" },
  { 0, NULL }
};

PAIR ThreadPriorities[] = {
  { THREAD_PRIORITY_ABOVE_NORMAL, "THREAD_PRIORITY_ABOVE_NORMAL" },
  { THREAD_PRIORITY_BELOW_NORMAL, "THREAD_PRIORITY_BELOW_NORMAL" },
  { THREAD_PRIORITY_HIGHEST, "THREAD_PRIORITY_HIGHEST" },
  { THREAD_PRIORITY_IDLE, "THREAD_PRIORITY_IDLE" },
  { THREAD_PRIORITY_LOWEST, "THREAD_PRIORITY_LOWEST" },
  { THREAD_PRIORITY_NORMAL, "THREAD_PRIORITY_NORMAL" },
  { THREAD_PRIORITY_TIME_CRITICAL, "THREAD_PRIORITY_TIME_CRITICAL" },
  { 0, NULL }
};

char *String4Value(DWORD dwValue, PAIR *Lookup)
{
  int I;
  for (I=0; Lookup[I].Value &&
            dwValue != Lookup[I].Value;
       I++)
    ;
  return Lookup[I].String;
}

main(void)
{
  HANDLE hProcess, hThread;

  int I,J;

  hProcess = GetCurrentProcess();
  hThread = GetCurrentThread();

  for (I=0; I<(sizeof(PriorityClasses)/sizeof(PAIR)-1); I++) {
    DWORD dwClass, dwPriority;
    SetPriorityClass(hProcess, PriorityClasses[I].Value);
    dwClass = GetPriorityClass(hProcess);

    for (J=0; J<(sizeof(ThreadPriorities)/sizeof(PAIR)-1); J++) {
      SetThreadPriority(hThread, ThreadPriorities[J].Value);
      dwPriority = GetThreadPriority(hThread);

      printf("Priority is %s+%s\n",
             String4Value(dwClass, PriorityClasses),
             String4Value(dwPriority, ThreadPriorities));
    }
  }

  return 0;
}
```

Αν το τρέξουμε θα δούμε:

```
Priority is HIGH_PRIORITY_CLASS+THREAD_PRIORITY_ABOVE_NORMAL
```

```
Priority is HIGH_PRIORITY_CLASS+THREAD_PRIORITY_BELOW_NORMAL
Priority is HIGH_PRIORITY_CLASS+THREAD_PRIORITY_HIGHEST
Priority is HIGH_PRIORITY_CLASS+THREAD_PRIORITY_IDLE
Priority is HIGH_PRIORITY_CLASS+THREAD_PRIORITY_LOWEST
Priority is HIGH_PRIORITY_CLASS+THREAD_PRIORITY_NORMAL
Priority is HIGH_PRIORITY_CLASS+THREAD_PRIORITY_IDLE
Priority is IDLE_PRIORITY_CLASS+THREAD_PRIORITY_ABOVE_NORMAL
Priority is IDLE_PRIORITY_CLASS+THREAD_PRIORITY_BELOW_NORMAL
Priority is IDLE_PRIORITY_CLASS+THREAD_PRIORITY_HIGHEST
Priority is IDLE_PRIORITY_CLASS+THREAD_PRIORITY_IDLE
Priority is IDLE_PRIORITY_CLASS+THREAD_PRIORITY_LOWEST
Priority is IDLE_PRIORITY_CLASS+THREAD_PRIORITY_NORMAL
Priority is IDLE_PRIORITY_CLASS+THREAD_PRIORITY_NORMAL
Priority is NORMAL_PRIORITY_CLASS+THREAD_PRIORITY_ABOVE_NORMAL
Priority is NORMAL_PRIORITY_CLASS+THREAD_PRIORITY_BELOW_NORMAL
Priority is NORMAL_PRIORITY_CLASS+THREAD_PRIORITY_HIGHEST
Priority is NORMAL_PRIORITY_CLASS+THREAD_PRIORITY_IDLE
Priority is NORMAL_PRIORITY_CLASS+THREAD_PRIORITY_LOWEST
Priority is NORMAL_PRIORITY_CLASS+THREAD_PRIORITY_NORMAL
Priority is NORMAL_PRIORITY_CLASS+THREAD_PRIORITY_IDLE
Priority is REALTIME_PRIORITY_CLASS+THREAD_PRIORITY_ABOVE_NORMAL
Priority is REALTIME_PRIORITY_CLASS+THREAD_PRIORITY_BELOW_NORMAL
Priority is REALTIME_PRIORITY_CLASS+THREAD_PRIORITY_HIGHEST
Priority is REALTIME_PRIORITY_CLASS+THREAD_PRIORITY_IDLE
Priority is REALTIME_PRIORITY_CLASS+THREAD_PRIORITY_LOWEST
Priority is REALTIME_PRIORITY_CLASS+THREAD_PRIORITY_NORMAL
Priority is REALTIME_PRIORITY_CLASS+THREAD_PRIORITY_NORMAL
```

Κεφάλαιο 3 Αντικείμενα συγχρονισμού

Τα *αντικείμενα συγχρονισμού* (synchronization objects) του Win32 είναι οι mutexes, οι σηματοφορείς, και τα γεγονότα. Η κατάσταση ενός αντικειμένου συγχρονισμού μπορεί να είναι σηματοδοτημένη (signaled) ή όχι (non-signaled). Για παράδειγμα, ένας σηματοφορέας είναι σηματοδοτημένος όταν έχει μη αρνητική τιμή (τότε ένα νήμα δεν μπλοκάρεται σε αυτόν).

Για να χρησιμοποιούνται τα αντικείμενα συγχρονισμού και μεταξύ νημάτων διαφορετικών διεργασιών υπάρχει η δυνατότητα να πάρουν ένα όνομα, μια συμβολοσειρά που καθορίζεται κατά τη δημιουργία του αντικειμένου. Για να έχουν πρόσβαση άλλες διεργασίες στο ίδιο αντικείμενο αναφέρονται σε αυτό με το όνομά του. Πρέπει να σημειωθεί ότι ο χώρος ονομάτων (namespace) είναι κοινός για τα αντικείμενα αυτά, δηλαδή δεν είναι δυνατόν να υπάρχει ένας mutex και ένα γεγονός με το ίδιο όνομα.

Ρουτίνες αναμονής

Για όλα τα αντικείμενα συγχρονισμού υπάρχει κοινός τρόπος χρήσης: για να περιμένει ένα νήμα ώστε η κατάσταση ενός ή πολλών αντικειμένων να γίνει σηματοδοτημένη, χρησιμοποιεί μια από τις ρουτίνες αναμονής. Οι βασικές ρουτίνες αναμονής είναι οι WaitForSingleObject, WaitForMultipleObjects και SignalObjectAndWait.

Η ρουτίνα WaitForSingleObject

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

Η ρουτίνα αυτή μπλοκάρει το νήμα που την καλεί μέχρι είτε το αντικείμενο συγχρονισμού hHandle να σηματοδοτηθεί, είτε να εκπνεύσει το διάστημα dwMilliseconds. Η δεύτερη αυτή παράμετρος

μπορεί να πάρει την τιμή INFINITE, οπότε το νήμα ξεμπλοκάρεται μόνο όταν σηματοδοτηθεί το αντικείμενο, ή και την τιμή 0 οπότε η ρουτίνα ελέγχει την κατάσταση του αντικειμένου και επιστρέφει αμέσως.

Η ρουτίνα επιστρέφει μια από τις ακόλουθες τιμές:

- WAIT_OBJECT_0: το αντικείμενο σηματοδοτήθηκε
- WAIT_TIMEOUT: το διάστημα αναμονής εξέπνευσε χωρίς να σηματοδοτηθεί το αντικείμενο
- WAIT_FAILED: η κλήση απέτυχε
- WAIT_ABANDONED: το αντικείμενο είναι ένας mutex τον οποίο δεν ελευθέρωσε κάποιο νήμα πριν τερματίσει

Στο παρακάτω πρόγραμμα, το κύριο νήμα δημιουργεί ένα άλλο νήμα και περιμένει την ολοκλήρωσή του.

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

DWORD MyRoutine(DWORD dwNum)
{
  int i;

  for (i=0; i<8; i++)
    printf("Thread %d iteration %d\n", (int)GetCurrentThreadId(), i);

  return 0;
}

main(void)
{
  HANDLE hThread;
  DWORD dwThreadId, dwError;

  hThread = CreateThread(NULL, 0,
                         (LPTHREAD_START_ROUTINE)MyRoutine,
                         (LPVOID)0, 0, &dwThreadId);

  if (NULL == hThread) {
    dwError = GetLastError();
    fprintf(stderr, "Error %ld creating thread.\n", (long)dwError);
    return 1;
  }

  WaitForSingleObject(hThread, INFINITE);
  printf("Thread %ld finished... exiting.\n", (long)dwThreadId);

  CloseHandle(hThread);

  return 0;
}
```

Η έξοδος του προγράμματος είναι:

```
Thread 159 iteration 0
Thread 159 iteration 1
Thread 159 iteration 2
Thread 159 iteration 3
Thread 159 iteration 4
Thread 159 iteration 5
Thread 159 iteration 6
Thread 159 iteration 7
Thread 159 finished... exiting.
```

Η ρουτίνα WaitForMultipleObjects

```
DWORD WaitForMultipleObjects(
  DWORD nCount,
  CONST HANDLE *lpHandles,
  BOOL bWaitAll,
  DWORD dwMilliseconds
);
```

Η ρουτίνα αυτή μπλοκάρει το νήμα που την καλεί μέχρι είτε ένα ή όλα τα αντικείμενα συγχρονισμού του πίνακα lpHandles να σηματοδοτηθούν, είτε να εκπνεύσει το διάστημα dwMilliseconds. Η παράμετρος nCount (με τιμή το πολύ MAXIMUM_WAIT_OBJECTS) δίνει το πλήθος των αντικειμένων των οποίων τα handles βρίσκονται στον πίνακα lpHandles. Αν η bWaitAll έχει την τιμή TRUE, τότε η ρουτίνα περιμένει να σηματοδοτηθούν όλα τα αντικείμενα, αλλιώς αρκεί να σηματοδοτηθεί ένα από αυτά.

Όταν το νήμα που κάλεσε τη WaitForMultipleObjects περιμένει για όλα τα αντικείμενα, και μόνο μερικά από αυτά είναι σηματοδοτημένα, το νήμα δεν τα κρατά περιμένοντας τα υπόλοιπα, εμποδίζοντας άλλα νήματα να τα πάρουν· αντίθετα περιμένει για μια χρονική στιγμή όπου *όλα συγχρόνως* είναι σηματοδοτημένα.

Η ρουτίνα επιστρέφει μια από τις ακόλουθες τιμές:

- WAIT_OBJECT_0 έως (WAIT_OBJECT_0+nCount-1): Αν η παράμετρος bWaitAll είναι TRUE, δηλώνεται ότι όλα τα αντικείμενα έχουν σηματοδοτηθεί. Αλλιώς, η τιμή WAIT_OBJECT_*i* δηλώνει ότι το i αντικείμενο του πίνακα σηματοδοτήθηκε. Αν περισσότερα από ένα έχουν σημτοδοτηθεί ταυτόχρονα, επιστρέφεται το μικρότερο i.
- WAIT_TIMEOUT: το διάστημα αναμονής εξέπνευσε χωρίς να σηματοδοτηθούν τα αντικείμενα που ορίζει η bWaitAll
- WAIT_FAILED: η κλήση απέτυχε
- WAIT_ABANDONED_0 έως (WAIT_ABANDONED_0+nCount-1): Αν η παράμετρος bWaitAll είναι TRUE, δηλώνεται ότι όλα τα αντικείμενα έχουν σηματοδοτηθεί και τουλάχιστον ένα από τα αντικείμενα είναι ένας mutex τον οποίο δεν ελευθέρωσε κάποιο νήμα πριν τερματίσει. Αλλιώς, η τιμή WAIT_ABANDONED_*i* δηλώνει ότι το i αντικείμενο του πίνακα σηματοδοτήθηκε αλλά είναι ένας «εγκαταλελειμένος» mutex.

Η λειτουργία της WaitForMultipleObjects δε βρίσκεται σε κλήσεις συστήματος άλλων ΛΣ· για να προσομοιωθεί χρειάζεται ιδιαίτερη προσπάθεια και πολλοί πόροι συστήματος.

Το πρόγραμμα που ακολουθεί δημιουργεί τέσσερα νήματα και περιμένει την ολοκλήρωσή τους:

```
/* Make with cl /MT sample8.c */

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

DWORD MyRoutine(DWORD dwNum)
{
  int i;

  for (i=0; i<4; i++) {
    printf("Thread %d iteration %d\n", (int)dwNum, i);

    Sleep(10+3*dwNum); /* Delay for 10+3*dwNum ms */
  }

  printf("Thread %d finished\n", (int)dwNum);
```

```
  return 0;
}

main(void)
{
  HANDLE hThreads[4];
  DWORD dwThreadId, dwError;
  int I;

  for (I=0; I<sizeof(hThreads)/sizeof(HANDLE); I++) {
    hThreads[I] = CreateThread(NULL, 0,
                          (LPTHREAD_START_ROUTINE)MyRoutine,
                          (LPVOID)I, 0, &dwThreadId);

    if (NULL == hThreads[I]) {
      dwError = GetLastError();
      fprintf(stderr, "Error %ld creating thread.\n", (long)dwError);
      return 1;
    }
  }

  WaitForMultipleObjects(sizeof(hThreads)/sizeof(HANDLE),
                         hThreads, TRUE, INFINITE);
  printf("All threads finished... exiting.\n");

  for (I=0; I<sizeof(hThreads)/sizeof(HANDLE); I++)
    CloseHandle(hThreads[I]);

  return 0;
}
```

Αν εκτελέσουμε το πρόγραμμα, θα δούμε:

```
Thread 0 iteration 0
Thread 1 iteration 0
Thread 2 iteration 0
Thread 0 iteration 1
Thread 3 iteration 0
Thread 1 iteration 1
Thread 2 iteration 1
Thread 0 iteration 2
Thread 3 iteration 1
Thread 1 iteration 2
Thread 2 iteration 2
Thread 0 iteration 3
Thread 3 iteration 2
Thread 1 iteration 3
Thread 0 finished
Thread 2 iteration 3
Thread 3 iteration 3
Thread 1 finished
Thread 2 finished
Thread 3 finished
All threads finished... exiting.
```

Η ρουτίνα SignalObjectAndWait

```
BOOL SignalObjectAndWait(
  HANDLE hObjectToSignal,
  HANDLE hObjectToWaitOn,
  DWORD dwMilliseconds,
  BOOL bAlertable
);
```

Η ρουτίνα αυτή ατομικά σηματοδοτεί το αντικείμενο hObjectToSignal και περιμένει το hObjectToWaitOn. Μετά τη σηματοδότηση του hObjectToSignal η συμπεριφορά της και η τιμή επιστροφής της είναι ίδια με την WaitForSingleObject.

Η ύπαρξή της οφείλεται σε λόγους επίδοσης, γιατί κάθε λειτουργία πάνω σε αντικείμενο συγχρονισμού απαιτεί κλείδωμα της βάσης δεδομένων του συστήματος όπου κρατούνται τα αντικείμενα αυτά · με την SignalObjectAndWait εξοικονομείται ένα κλείδωμα (και η αναμονή που αυτό απαιτεί).

Η παράμετρος bAlertable αφορά ασύγχρονο I/O και για απλή χρήση της ρουτίνας δίνουμε τιμή FALSE.

MUTEXES

Η ονομασία mutex προέρχεται από τη συντόμευση του mutual exclusion (αμοιβαίος αποκλεισμός). Ο mutex είναι ένα αντικείμενο συγχρονισμού που μπορεί να ανήκει σε ένα μόνο νήμα κάθε φορά. Όταν κάποιο νήμα έχει αποκτήσει τον mutex, η κατάστασή του είναι μη σηματοδοτημένη (non signaled).

Ο mutex είναι το *μόνο* αντικείμενο συγχρονισμού το οποίο έχει ιδιοκτήτη. Όταν ένα νήμα έχει την ιδιοκτησία ενός mutex πρέπει το ίδιο να τον ελευθερώσει, σε αντίθεση π.χ. με τους σηματοφορείς όπου το ένα νήμα μπορεί να εκτελέσει μια λειτουργία P αλλά η λειτουργία V να εκτελεστεί από κάποιο άλλο νήμα. Για το λόγο αυτό έχει ληφθεί μέριμνα για την περίπτωση που το νήμα-ιδιοκτήτης ενός mutex τερματίσει χωρίς να τον ελευθερώσει. Τότε θεωρείται ότι ο mutex έχει «εγκαταλειφθεί» (abandoned mutex), και το επόμενο νήμα που θα τον πάρει στην ιδιοκτησία του θα ενημερωθεί για το γεγονός αυτό.

Η δημιουργία ενός mutex γίνεται με τη ρουτίνα CreateMutex():

```
HANDLE CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,
  BOOL bInitialOwner,
  LPCTSTR lpName
);
```

Η παράμετρος bInitialOwner έχει την τιμή TRUE εάν θέλουμε το νήμα που δημιουργεί τον mutex να πάρει απευθείας και την ιδιοκτησία του. Η παράμετρος lpName είναι το όνομα του mutex και μπορεί να είναι NULL. Αν το όνομα αυτό ανήκει σε έναν ήδη υπάρχοντα mutex, τότε επιστρέφεται το handle του και η GetLastError() επιστρέφει τον κωδικό ERROR_ALREADY_EXISTS. Η τιμή επιστροφής της ρουτίνας είναι το handle του mutex που δημιουργήθηκε · αν αποτύχει επιστρέφεται NULL και ο κωδικός λάθους δίνεται από την GetLastError().

Η χρήση του ονόματος δεν είναι απαραίτητη αν ο mutex θα χρησιμοποιηθεί μόνο από νήματα της ίδιας διεργασίας.

Νήματα *άλλων διεργασιών* μπορούν να πάρουν το handle του mutex καλώντας τη ρουτίνα OpenMutex() και δίνοντας το όνομά του στην παράμετρο lpName.

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

Για να πάρει ένα νήμα τον mutex στην ιδιοκτησία του χρησιμοποιεί κάποια από τις ρουτίνες αναμονής, π.χ. τη ρουτίνα WaitForSingleObject().

Για να την απελευθερώσει χρησιμοποιεί τη ReleaseMutex():

```
BOOL ReleaseMutex(HANDLE hMutex);
```

Ένα νήμα μπορεί να καλέσει πολλές φορές την WaitForSingleObject() για τον ίδιο mutex χωρίς αυτή να προκαλεί αδιέξοδο, αλλά μετά πρέπει να καλέσει ίσο αριθμό φορών την ReleaseMutex().

Όταν ο mutex δε χρειάζεται πλέον απελευθερώνεται με την CloseHandle():

```
BOOL CloseHandle(HANDLE hObject);
```

Το πρόγραμμα που ακολουθεί χρησιμοποιεί ένα mutex για να τυπώνουν τα νήματά του συγχρονισμένα μηνύματα στην οθόνη.

```
/* Make with cl /MT sample9.c user32.lib */

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

HANDLE hMutex;

DWORD MyRoutine(int nThread)
{
  int i;
  char szMessage[32];

  wsprintf(szMessage, "This is thread %d.\n", nThread);

  for (i=0; i<4; i++) {
    WaitForSingleObject(hMutex, INFINITE);
    printf("%s", szMessage);
    ReleaseMutex(hMutex);
    Sleep(50);
}

  return 0;
}

main(void)
{
  HANDLE hThreads[4];
  DWORD dwThreadId, dwError;
  int I;

  hMutex = CreateMutex(NULL, FALSE, NULL);
  if (NULL == hMutex) {
      dwError = GetLastError();
      fprintf(stderr, "Error %ld creating mutex.\n", (long)dwError);
      return 1;
  }

  for (I=0; I<sizeof(hThreads)/sizeof(HANDLE); I++) {
    hThreads[I] = CreateThread(NULL, 0,
                           (LPTHREAD_START_ROUTINE)MyRoutine,
                           (LPVOID)(I+1), 0, &dwThreadId);

    if (NULL == hThreads[I]) {
      dwError = GetLastError();
      fprintf(stderr, "Error %ld creating thread.\n", (long)dwError);
      return 1;
    }
  }

  MyRoutine(0);

  WaitForMultipleObjects(sizeof(hThreads)/sizeof(HANDLE),
                         hThreads, TRUE, INFINITE);
```

```
  for (I=0; I<sizeof(hThreads)/sizeof(HANDLE); I++)
    CloseHandle(hThreads[I]);

  CloseHandle(hMutex);

  return 0;
}
```

Αν το εκτελέσουμε, θα δούμε:

```
This is thread 0.
This is thread 1.
This is thread 2.
This is thread 3.
This is thread 4.
This is thread 0.
This is thread 1.
This is thread 2.
This is thread 3.
This is thread 4.
This is thread 0.
This is thread 1.
This is thread 2.
This is thread 3.
This is thread 4.
This is thread 0.
This is thread 1.
This is thread 2.
This is thread 3.
This is thread 4.
```

Σημαφορεις

Ο σηματοφορέας είναι ένα αντικείμενο που έχει τιμή μεταξύ 0 και μιας μέγιστης τιμής η οποία ορίζεται κατά τη δημιουργία του. Είναι σηματοδοτημένος όταν έχει μη μηδενική τιμή. Κάθε φορά που μια κλήση ρουτίνας αναμονής επιτύχει και επιστρέψει για το σηματοφορέα (λειτουργία P) η τιμή του μειώνεται κατά 1, και αυξάνεται ξανά όταν ελευθερωθεί (λειτουργία V).

Σε αντίθεση με τους mutexes, οι σηματοφορείς δεν έχουν ιδιοκτήτη, είναι δηλαδή δυνατόν να εκτελεστεί η λειτουργία P από ένα νήμα και η αντίστοιχη λειτουργία V από ένα άλλο. Επιπλέον αν ένα νήμα τερματιστεί κρατώντας ένα σηματοφορέα μπορεί να προκληθεί αδιέξοδο στο σύστημα γιατί αυτός δεν ελευθερώνεται.

Η δημιουργία ενός σηματοφορέα γίνεται με τη ρουτίνα CreateSemaphore():

```
HANDLE CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
  LONG lInitialCount,
  LONG lMaximumCount,
  LPCTSTR lpName
);
```

Η παράμετρος lMaximumCount ορίζει τη μέγιστη τιμή του σηματοφορέα και η lInitialCount ορίζει την αρχική τιμή του (μεταξύ 0 και lMaximumCount). Ο σηματοφορέας μπορεί να πάρει ένα όνομα με την lpName έτσι ώστε να είναι προσβάσιμος και σε άλλες διεργασίες. Αν το όνομα αυτό ανήκει σε έναν ήδη υπάρχοντα σηματοφορέα, τότε επιστρέφεται το handle του και η GetLastError() επιστρέφει τον κωδικό ERROR_ALREADY_EXISTS. Η τιμή επιστροφής της ρουτίνας είναι το handle του σηματοφορέα που δημιουργήθηκε· αν αποτύχει επιστρέφεται NULL και ο κωδικός λάθους δίνεται από την GetLastError().

Η χρήση του ονόματος δεν είναι απαραίτητη αν ο σηματοφορέας θα χρησιμοποιηθεί μόνο από νήματα της ίδιας διεργασίας.

Νήματα άλλων διεργασιών μπορούν να πάρουν το handle του σηματοφορέα καλώντας τη ρουτίνα OpenSemaphore() και δίνοντας το όνομά του στην παράμετρο lpName.

```
HANDLE OpenSemaphore(
  DWORD dwDesiredAccess,
  BOOL bInheritHandle,
  LPCTSTR lpName
);
```

Για να πάρει ένα νήμα το σηματοφορέα στην ιδιοκτησία του χρησιμοποιεί κάποια από τις ρουτίνες αναμονής, π.χ. τη ρουτίνα WaitForSingleObject().

Για να την απελευθερώσει χρησιμοποιεί τη ReleaseSemaphore() (λειτουργία V):

```
BOOL ReleaseSemaphore(
  HANDLE hSemaphore,
  LONG lReleaseCount,
  LPLONG lpPreviousCount
);
```

Στην παράμετρο lReleaseCount δίνουμε το πόσο πρέπει να αυξηθεί η τιμή του σηματοφορέα, ενώ στην lpPreviousCount περνάμε τη διεύθυνση μίας μεταβλητής όπου θα γραφεί η τιμή του σηματοφορέα πριν από την αύξηση (μπορεί να είναι NULL).

Όταν ο σηματοφορέας δε χρειάζεται πλέον απελευθερώνεται με την CloseHandle().

Στο πρόγραμμα που ακολουθεί, χρησιμοποιούμε ένα σηματοφορέα με αρχική τιμή 3 για να επιτρέψουμε σε 3 από τα νήματα να έχουν πρόσβαση σε κάποια κοινά δεδομένα.

```
/* Make with cl /MT sample10.c */

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

HANDLE hSemaphore;

DWORD MyRoutine(int nThread)
{
  int i, j;

  for (i=0; i<2; i++) {
    WaitForSingleObject(hSemaphore, INFINITE);

    printf("Thread %d entered\n", nThread);

    ReleaseSemaphore(hSemaphore, 1, NULL);
    Sleep(50);

    printf("Thread %d exited\n", nThread);
  }

  return 0;
}

main(void)
{
  HANDLE hThreads[5];
  DWORD dwThreadId, dwError;
  int I;

  hSemaphore = CreateSemaphore(NULL, 3, 3, NULL);
  if (NULL == hSemaphore) {
      dwError = GetLastError();
      fprintf(stderr, "Error %ld creating semaphore.\n", (long)dwError);
      return 1;
  }
```

```
  for (I=0; I<sizeof(hThreads)/sizeof(HANDLE); I++) {
    hThreads[I] = CreateThread(NULL, 0,
                          (LPTHREAD_START_ROUTINE)MyRoutine,
                          (LPVOID)(I+1), 0, &dwThreadId);

    if (NULL == hThreads[I]) {
      dwError = GetLastError();
      fprintf(stderr, "Error %ld creating thread.\n", (long)dwError);
      return 1;
    }
  }

  MyRoutine(0);

  WaitForMultipleObjects(sizeof(hThreads)/sizeof(HANDLE),
                         hThreads, TRUE, INFINITE);

  for (I=0; I<sizeof(hThreads)/sizeof(HANDLE); I++)
    CloseHandle(hThreads[I]);

  CloseHandle(hSemaphore);

  return 0;
}
```

Αν το εκτελέσουμε, θα δούμε:

```
Thread 0 entered
Thread 1 entered
Thread 2 entered
Thread 3 entered
Thread 4 entered
Thread 5 entered
Thread 0 exited
Thread 0 entered
Thread 1 exited
Thread 1 entered
Thread 2 exited
Thread 2 entered
Thread 3 exited
Thread 3 entered
Thread 4 exited
Thread 4 entered
Thread 5 exited
Thread 5 entered
Thread 0 exited
Thread 1 exited
Thread 2 exited
Thread 3 exited
Thread 4 exited
Thread 5 exited
```

Γεγονότα

Το γεγονός είναι ένα αντικείμενο συγχρονισμού το οποίο σηματοδοτείται «χειρωνακτικά» από ένα νήμα. Υπάρχουν δυο είδη γεγονότων: αυτά που επανέρχονται στη μη σηματοδοτημένη κατάσταση αυτόματα, και εκείνα που πρέπει να τα επαναφέρει ένα νήμα. Τα πρώτα αναφέρονται και σαν γεγονότα συγχρονισμού (synchronization events) ενώ τα δεύτερα αναφέρονται και σαν γεγονότα ενημέρωσης (notification events).

Σε αντίθεση με τους mutexes, τα γεγονότα δεν έχουν ιδιοκτήτη, δηλαδή οποιοδήποτε νήμα μπορεί να τα σηματοδοτήσει ή να τα επαναφέρει. Για το λόγο αυτό δεν είναι δυνατόν το ΛΣ να γνωρίζει

ποιο νήμα είναι ο «λογικός» ιδιοκτήτης ενός γεγονότος και να φροντίσει να προλάβει το αδιέξοδο αν αυτός τερματίσει χωρίς να σηματοδοτήσει το γεγονός.

Η δημιουργία ενός γεγονότος γίνεται με τη ρουτίνα CreateEvent():

```
HANDLE CreateEvent(
  LPSECURITY_ATTRIBUTES lpEventAttributes,
  BOOL bManualReset,
  BOOL bInitialState,
  LPCTSTR lpName
);
```

Η παράμετρος bManualReset καθορίζει το πώς θα «αποσηματοδοτείται» το γεγονός (TRUE για χειρωνακτική αποσηματοδότηση και FALSE για αυτόματη). Η bInitialState ορίζει αν αρχικά το γεγονός θα είναι σηματοδοτημένο. Το γεγονός μπορεί προαιρετικά να πάρει ένα όνομα με την παράμετρο lpName. Αν το όνομα αυτό ανήκει σε ένα ήδη υπάρχον γεγονός, τότε επιστρέφεται το handle του και η GetLastError() επιστρέφει τον κωδικό ERROR_ALREADY_EXISTS.

Η χρήση του ονόματος δεν είναι απαραίτητη αν το γεγονός θα χρησιμοποιηθεί μόνο από νήματα της ίδιας διεργασίας.

Νήματα άλλων διεργασιών μπορούν να πάρουν το handle του γεγονότος καλώντας τη ρουτίνα OpenEvent() και δίνοντας το όνομά του στην παράμετρο lpName.

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

Ένα νήμα περιμένει να ενεργοποιηθεί το γεγονός με κάποια από τις ρουτίνες αναμονής, π.χ. τη ρουτίνα WaitForSingleObject(). Τα γεγονότα αυτόματης αποσηματοδότησης, μόλις η ρουτίνα αναμονής επιστρέψει δηλώνοντας επιτυχία, (δηλαδή ότι το γεγονός σηματοδοτήθηκε), επανέρχονται στην κατάσταση μη σηματοδότησης. Αντίθετα, τα γεγονότα χειρωνακτικής αποσηματοδότησης μένουν σηματοδοτημένα έως ότου το αναλάβει αυτό κάποιο νήμα.

Η κατάσταση ενός γεγονότος γίνεται σηματοδοτημένη όταν ένα νήμα καλέσει μια από τις ρουτίνες SetEvent() και PulseEvent():

```
BOOL SetEvent(HANDLE hEvent);
BOOL PulseEvent(HANDLE hEvent);
```

Η λειτουργία των ρουτινών εξαρτάται από τον τύπο του γεγονότος και το αν ήδη κάποια νήματα περιμένουν το γεγονός και συνοψίζεται στον ακόλουθο πίνακα:

	Λειτουργία της SetEvent	
	Δεν περιμένει κάποιο νήμα	**Ένα ή περισσότερα νήματα περιμένουν**
Γεγονός αυτόματης επαναφοράς	Το γεγονός παραμένει σηματοδοτημένο μέχρι να εκτελεστεί κάποια ρουτίνα αναμονής γι' αυτό, οπότε επανέρχεται στη μη σηματοδοτημένη κατάσταση	Ένα από τα νήματα απελευθερώνεται από την αναμονή και το γεγονός επανέρχεται στη μη σηματοδοτημένη κατάσταση
Γεγονός χειρωνακτικής επαναφοράς	Το γεγονός παραμένει σηματοδοτημένο μέχρι να εκτελεστεί κάποια ρουτίνα αναμονής γι' αυτό, και μετά παραμένει σηματοδοτημένο	Όλα τα νήματα απελευθερώνονται από την αναμονή και το γεγονός παραμένει σηματοδοτημένο
	Λειτουργία της PulseEvent	
	Δεν περιμένει κάποιο νήμα	**Ένα ή περισσότερα νήματα περιμένουν**
Γεγονός αυτόματης επαναφοράς	Η κατάσταση του γεγονότος σηματοδοτείται στιγμιαία και μετά επανέρχεται στη μη σηματοδοτημένη κατάσταση	Ένα από τα νήματα απελευθερώνεται από την αναμονή και το γεγονός επανέρχεται στη μη σηματοδοτημένη κατάσταση
Γεγονός χειρωνακτικής επαναφοράς	Η κατάσταση του γεγονότος σηματοδοτείται στιγμιαία και μετά επανέρχεται στη μη σηματοδοτημένη κατάσταση	Όλα τα νήματα απελευθερώνονται από την αναμονή και το γεγονός επανέρχεται στη μη σηματοδοτημένη κατάσταση

Για να επανέλθει η κατάσταση του γεγονότος στη μη σηματοδοτημένη καλούμε την ResetEvent():

```
BOOL ResetEvent(HANDLE hEvent);
```

Αυτή η ρουτίνα χρησιμοποιείται για γεγονότα χειρωνακτικής επαναφοράς.

Όταν το γεγονός δε χρειάζεται πλέον απελευθερώνεται με την CloseHandle().

Στο πρόγραμμα που ακολουθεί, τα νήματα τυπώνουν από έναν αριθμό στην οθόνη και «παραχωρούν» με τυχαίο τρόπο τη σειρά το ένα στο άλλο.

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

HANDLE hEvent;

DWORD MyRoutine(int nThread)
{
  int I;

  for (I=0; I<50; I++) {
    WaitForSingleObject(hEvent, INFINITE);
    printf("%d", nThread);
    SetEvent(hEvent);
  }

  return 0;
}

main(void)
{
  HANDLE hThreads[5];
  DWORD dwThreadId, dwError;
  int I;

  hEvent = CreateEvent(NULL, FALSE, TRUE, NULL);
```

```
  if (NULL == hEvent) {
    fprintf(stderr, "Cannot create event. Error %ld.\n", (long)GetLastError());
    exit(1);
  }

  for (I=0; I<sizeof(hThreads)/sizeof(HANDLE); I++) {
    hThreads[I] = CreateThread(NULL, 0,
                           (LPTHREAD_START_ROUTINE)MyRoutine,
                           (LPVOID)(I+1), 0, &dwThreadId);

    if (NULL == hThreads[I]) {
      dwError = GetLastError();
      fprintf(stderr, "Error %ld creating thread.\n", (long)dwError);
      return 1;
    }
  }

  MyRoutine(0);

  WaitForMultipleObjects(sizeof(hThreads)/sizeof(HANDLE),
                         hThreads, TRUE, INFINITE);

  for (I=0; I<sizeof(hThreads)/sizeof(HANDLE); I++)
    CloseHandle(hThreads[I]);

  CloseHandle(hEvent);

  return 0;
}
```

Αν το εκτελέσουμε, θα δούμε:

```
00000001234512341234123412341234123412341234123412341234501234501234012345012340
12340123540123401234012340123401234012340123405123041230512340123501234012350123401 23
50123401523014230152304152301523015234015230152301523015230152301523015423015230
15230415230152305405050505555555555454545454545454545454544444
```

Κεφάλαιο 4
Μέθοδοι συγχρονισμού και επικοινωνίας μεταξύ νημάτων

Τα αντικείμενα συγχρονισμού, δηλαδή οι mutexes, οι σηματοφορείς και τα γεγονότα, μπορούν να χρησιμοποιηθούν από διαφορετικά νήματα της ίδιας διεργασίας (χωρίς να έχουν όνομα) ή και από διαφορετικές διεργασίες (αν το αντικείμενο έχει ένα κοινώς συμφωνημένο όνομα). Μεταξύ των νημάτων της ίδιας διεργασίας μπορούν να χρησιμοποιηθούν και άλλες μέθοδοι αμοιβαίου αποκλεισμού ή συγχρονισμού. Αυτές είναι τα *κρίσιμα τμήματα*, οι ρουτίνες *Interlocked* και η *αναστολή* και *επανενεργοποίηση* των νημάτων.

ΚΡΙΣΙΜΑ ΤΜΗΜΑΤΑ

Τα κρίσιμα τμήματα προσφέρουν υπηρεσίες αμοιβαίου αποκλεισμού όπως οι mutexes με τη βασική διαφορά ότι τα κρίσιμα τμήματα μπορούν να χρησιμοποιηθούν μόνο από νήματα της ίδιας διεργασίας. Είναι πιο γρήγορα και αποδοτικά από τους mutexes.

Όπως και στους mutexes, μόνο ένα νήμα μπορεί ανά πάσα στιγμή να έχει την ιδιοκτησία του ένα κρίσιμο τμήμα, να εκτελεί δηλαδή τον κώδικα για τον οποίο απαιτείται αμοιβαίος αποκλεισμός.

Η δημιουργία και αρχικοποίηση ενός κρίσιμου τμήματος γίνεται με τη ρουτίνα InitializeCriticalSection:

```
VOID InitializeCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

Η παράμετρος lpCriticalSection είναι δείκτης προς μια δομή CRITICAL_SECTION η οποία χρησιμοποιείται από το σύστημα για να κρατήσει στοιχεία για το κρίσιμο τμήμα. Αφού η αρχικοποίηση έχει γίνει, τα νήματα της διεργασίας μπορούν να μπουν στο κρίσιμο τμήμα με τις ρουτίνες EnterCriticalSection και TryEnterCriticalSection:

```
VOID EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
BOOL TryEnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

Η EnterCriticalSection μπλοκάρει μέχρι το κρίσιμο τμήμα να είναι ελεύθερο, οπότε το παίρνει και επιστρέφει. Η TryEnterCriticalSection επιστρέφει αμέσως, είτε πήρε το κρίσιμο τμήμα είτε όχι. Αν το έχει πάρει επιστρέφει την τιμή TRUE, αλλιώς επιστρέφει FALSE.

Για να ελευθερώσει το νήμα το κρίσιμο τμήμα καλεί την LeaveCriticalSection

```
VOID LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

Για κάθε κλήση της EnterCriticalSection και για κάθε επιτυχή κλήση της TryEnterCriticalSection (που επέστρεψε TRUE) πρέπει να γίνεται και η αντίστοιχη κλήση της LeaveCriticalSection. Η LeaveCriticalSection δεν πρέπει να καλείται από ένα νήμα που δεν έχει την ιδιοκτησία του κρίσιμου τμήματος.

Όπως και στους mutexes, το νήμα που κρατά το κρίσιμο τμήμα μπορεί να καλέσει επανειλημμένες φορές την EnterCriticalSection ή την TryEnterCriticalSection χωρίς να προκληθεί αδιέξοδο, πρέπει όμως να καλέσει την LeaveCriticalSection ισάριθμο πλήθος φορών.

Όταν το κρίσιμο τμήμα δε χρειάζεται πλέον, διαγράφεται με την DeleteCriticalSection:

```
VOID DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

Στο πρόγραμμα που ακολουθεί, τα νήματα της διεργασίας συγχρονίζονται με ένα critical section για να τυπώσουν μηνύματα στην οθόνη.

```
/* Make with cl sample12.c user32.lib */

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

CRITICAL_SECTION cs;

DWORD MyRoutine(int nThread)
{
  int i;
  char szMessage[32];

  wsprintf(szMessage, "This is thread %d.\n", nThread);

  for (i=0; i<4; i++) {
    EnterCriticalSection(&cs);
    printf("%s", szMessage);
    LeaveCriticalSection(&cs);
    Sleep(50);

  }

  return 0;
}

main(void)
{
  HANDLE hThreads[4];
  DWORD dwThreadId, dwError;
  int I;

  InitializeCriticalSection(&cs);

  for (I=0; I<sizeof(hThreads)/sizeof(HANDLE); I++) {
    hThreads[I] = CreateThread(NULL, 0,
                           (LPTHREAD_START_ROUTINE)MyRoutine,
                           (LPVOID)(I+1), 0, &dwThreadId);

    if (NULL == hThreads[I]) {
      dwError = GetLastError();
      fprintf(stderr, "Error %ld creating thread.\n", (long)dwError);
      return 1;
    }
  }

  MyRoutine(0);

  WaitForMultipleObjects(sizeof(hThreads)/sizeof(HANDLE),
                         hThreads, TRUE, INFINITE);

  for (I=0; I<sizeof(hThreads)/sizeof(HANDLE); I++)
    CloseHandle(hThreads[I]);

  DeleteCriticalSection(&cs);

  return 0;
}
```

Αν εκτελέσουμε το πρόγραμμα θα δούμε:

```
This is thread 0.
This is thread 1.
This is thread 2.
This is thread 3.
This is thread 4.
This is thread 0.
This is thread 1.
This is thread 2.
This is thread 3.
This is thread 4.
This is thread 0.
This is thread 1.
This is thread 2.
This is thread 3.
This is thread 4.
This is thread 0.
This is thread 1.
This is thread 2.
This is thread 3.
This is thread 4.
```

Οι ρουτινες Interlocked

Η οικογένεια ρουτινών Interlocked προσφέρει απλές υπηρεσίες ασφαλούς πρόσβασης σε μοιραζόμενες μεταβλητές από νήματα της ίδιας διεργασίας. Όλες εκτελούν τη λειτουργία τους (ανάγνωση και ενημέρωση μιας 32-bit μεταβλητής) ατομικά και επιστρέφουν μια 32-bit τιμή. Οι ρουτίνες αυτές είναι:

InterlockedIncrement

```
LONG InterlockedIncrement(LPLONG lpAddend);
```

Η ρουτίνα InterlockedIncrement αυξάνει κατά 1 την τιμή της 32-bit μεταβλητής με διεύθυνση lpAddend. Αν μετά την αύξηση η τιμή της είναι θετική, επιστρέφει μια θετική τιμή, αν η τιμή της μεταβλητής είναι αρνητική επιστρέφει μια αρνητική τιμή, αλλιώς επιστρέφει 0. Η τιμή επιστροφής δεν είναι υποχρεωτικά ίδια με το αποτέλεσμα της αύξησης.

Το πρόγραμμα που ακολουθεί «μετρά» το πλήθος των επαναλήψεων που έκαναν όλα τα νήματά του.

```
/* Make with cl /MT sample13.c */

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

LONG lTotalLoops = 0;

DWORD MyRoutine(int nThread)
{
  int nLoops = GetTickCount() % ((nThread+1)*13);

  printf("Thread %d will execute %d loops\n", nThread, nLoops);

  for (; nLoops; nLoops--)
    InterlockedIncrement(&lTotalLoops);

  return 0;
}

main(void)
{
  HANDLE hThreads[4];
  DWORD dwThreadId, dwError;
  int I;
```

```
  for (I=0; I<sizeof(hThreads)/sizeof(HANDLE); I++) {
    hThreads[I] = CreateThread(NULL, 0,
                           (LPTHREAD_START_ROUTINE)MyRoutine,
                           (LPVOID)(I+1), 0, &dwThreadId);

    if (NULL == hThreads[I]) {
      dwError = GetLastError();
      fprintf(stderr, "Error %ld creating thread.\n", (long)dwError);
      return 1;
    }
  }

  MyRoutine(0);

  WaitForMultipleObjects(sizeof(hThreads)/sizeof(HANDLE),
                         hThreads, TRUE, INFINITE);

  for (I=0; I<sizeof(hThreads)/sizeof(HANDLE); I++)
    CloseHandle(hThreads[I]);

  printf("The threads executed %ld loops in total.\n", lTotalLoops);

  return 0;
}
```

Αν το εκτελέσουμε θα δούμε:

```
Thread 0 will execute 4 loops
Thread 1 will execute 17 loops
Thread 2 will execute 27 loops
Thread 3 will execute 11 loops
Thread 4 will execute 21 loops
The threads executed 80 loops in total.
```

InterlockedDecrement

```
LONG InterlockedDecrement(LPLONG lpAddend);
```

Η ρουτίνα InterlockedDecrement μειώνει κατά 1 την τιμή της 32-bit μεταβλητής με διεύθυνση lpAddend. Αν μετά τη μείωση η τιμή της είναι θετική, επιστρέφει μια θετική τιμή, αν η τιμή της μεταβλητής είναι αρνητική επιστρέφει μια αρνητική τιμή, αλλιώς επιστρέφει 0. Η τιμή επιστροφής δεν είναι υποχρεωτικά ίδια με το αποτέλεσμα της μείωσης.

Το παρακάτω πρόγραμμα αποφασίζει με «τυχαίο» τρόπο το συνολικό πλήθος επαναλήψεων που θα κάνουν τα νήματά του. Η κλήση Sleep(0) γίνεται για να παραχωρήσουν τα νήματα το υπόλοιπο του κβάντου χρόνου τους (αλλιώς, επειδή οι επαναλήψεις είναι λίγες, θα γίνονταν όλες από ένα νήμα).

```
/* Make with cl /MT sample14.c */

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

LONG lTotalLoops = 0;

DWORD MyRoutine(int nThread)
{
  int nLoops;

  for (nLoops = 0; ; nLoops++) {
    if (0 > InterlockedDecrement(&lTotalLoops)) {
      break;
    }

    Sleep(0);
  }
```

```
  printf("Thread %d executed %d loops\n", nThread, nLoops);

  return 0;
}

main(void)
{
  HANDLE hThreads[4];
  DWORD dwThreadId, dwError;
  int I;

  lTotalLoops = (GetTickCount() % 50) + 50;
  printf("The threads will execute %ld loops in total.\n", lTotalLoops);

  for (I=0; I<sizeof(hThreads)/sizeof(HANDLE); I++) {
    hThreads[I] = CreateThread(NULL, 0,
                           (LPTHREAD_START_ROUTINE)MyRoutine,
                           (LPVOID)(I+1), 0, &dwThreadId);

    if (NULL == hThreads[I]) {
      dwError = GetLastError();
      fprintf(stderr, "Error %ld creating thread.\n", (long)dwError);
      return 1;
    }
  }

  MyRoutine(0);

  WaitForMultipleObjects(sizeof(hThreads)/sizeof(HANDLE),
                         hThreads, TRUE, INFINITE);

  for (I=0; I<sizeof(hThreads)/sizeof(HANDLE); I++)
    CloseHandle(hThreads[I]);

  return 0;
}
```

Αν το εκτελέσουμε θα δούμε:

```
The threads will execute 96 loops in total.
Thread 1 executed 19 loops
Thread 2 executed 19 loops
Thread 3 executed 19 loops
Thread 4 executed 19 loops
Thread 0 executed 20 loops
```

InterlockedExchange

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

Η ρουτίνα αυτή θέτει την τιμή της μεταβλητής που δείχνει η παράμετρος Target με την Value και επιστρέφει την παλιά της τιμή. Έτσι για να ανταλλάξουμε τις τιμές δυο 32-bit μεταβλητών ατομικά θα κάνουμε την κλήση:

```
LONG x, y;
x = InterlockedExchange(&y, x);
```

InterlockedExchangeAdd

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

Η ρουτίνα αυτή είναι παρόμοια με την InterlockedExchange, με τη διαφορά ότι δεν θέτει την τιμή της μεταβλητής που δείχνει η παράμετρος Target, αλλά προσθέτει την τιμή Increment σε αυτήν, και επιστρέφει την παλιά της τιμή.

Έτσι τα τμήματα κώδικα (α) και (β) είναι ισοδύναμα αν το (β) εκτελεστεί ατομικά:

```
LONG x, y;
y = InterlockedExchangeAdd(&x, 10);
```

(α)

```
LONG x, y;
y=x;
x += 10;
```

(β)

InterlockedCompareExchange

```
PVOID InterlockedCompareExchange(
  PVOID *Destination,
  PVOID Exchange,
  PVOID Comperand
);
```

Η InterlockedCompareExchange συγκρίνει την τιμή της 32-bit μεταβλητής που δείχνει η παράμετρος Destination με την τιμή Comperand. Αν είναι ίσες, τότε αναθέτει στην Destination την τιμή Exchange. Η ρουτίνα επιστρέφει την αρχική τιμή της Destination. Τα πρόσημα των τιμών αγνοούνται στη σύγκριση.

Με τη ρουτίνα αυτή μπορούμε π.χ. να φτιάξουμε έναν απλό mutex, χωρίς ιδιοκτήτη, ο οποίος κάνει busy-wait. Το ρόλο του mutex θα παίζει μία μεταβλητή, που θα έχει την τιμή 1 όταν ο mutex είναι ελεύθερος και 0 όταν είναι κατειλημμένος. Η ρουτίνα MyGetMutex, που παίρνει το mutex θα είναι:

```
void GetMyMutex(LONG *mutex)
{
    while (0 == InterlockedCompareExchange(mutex, 0, 1))
      ;
}
```

και η ρουτίνα ReleaseMyMutex που τον απελευθερώνει είναι η

```
void ReleaseMyMutex(LONG *mutex)
{
    *mutex = 1;
}
```

Αναστολή και επανενεργοποίηση λειτουργίας νημάτων

Για να αναστείλουμε τη λειτουργία ενός νήματος καλούμε τη ρουτίνα SuspendThread δίνοντας για παράμετρο το handle του νήματος και για να την ενεργοποιήσουμε ξανά καλούμε την ResumeThread:

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

Το σύστημα κρατά ένα μετρητή (με μέγιστη τιμή MAXIMUM_SUSPEND_COUNT) που καταγράφει το πλήθος των κλήσεων SuspendThread που έχουν γίνει ανά πάσα στιγμή για κάθε νήμα. Ο μετρητής αυτός αυξάνεται με κάθε κλήση της SuspendThread και μειώνεται με κάθε κλήση της ResumeThread. Το νήμα ενεργοποιείται μετά από την κλήση της ResumeThread μόνο αν ο μετρητής του έχει φθάσει στο 0.

Και οι δυο ρουτίνες επιστρέφουν την προηγούμενη τιμή του μετρητή, ενώ σε περίπτωση λάθους επιστρέφουν 0xFFFFFFFF.

Πρέπει να σημειωθεί ότι κλήση της ResumeThread σε νήμα που εκτελείται κανονικά (μετρητής = 0) δεν αλλάζει την τιμή του μετρητή.

Στο παρακάτω πρόγραμμα, τα νήματα τυπώνουν έναν αριθμό «κυκλικά». Κάθε νήμα, αφού τυπώσει τον αριθμό του, επανενεργοποιεί το επόμενό του, και μετά αναστέλει τον εαυτό του. Παρατηρούμε ότι αρχικά όλα τα νήματα εκτός από ένα δημιουργούνται ανενεργά.

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

#define THREAD_COUNT 5

HANDLE hThreads[THREAD_COUNT];

DWORD MyRoutine(int nThread)
{
  int I;

  for (I=0;I<20; I++) {
    printf("%d", nThread);
    ResumeThread(hThreads[(nThread+1)%THREAD_COUNT]);
    SuspendThread(hThreads[nThread]);
  }
  ResumeThread(hThreads[(nThread+1)%THREAD_COUNT]);

  return 0;
}

main(void)
{
  DWORD dwThreadId, dwError;
  int I;

  for (I=0; I<THREAD_COUNT; I++) {
    hThreads[I] = CreateThread(NULL, 0,
                           (LPTHREAD_START_ROUTINE)MyRoutine,
                           (LPVOID)I,
                           (0 != I)*CREATE_SUSPENDED,
                           &dwThreadId);

    if (NULL == hThreads[I]) {
      dwError = GetLastError();
      fprintf(stderr, "Error %ld creating thread.\n", (long)dwError);
      return 1;
    }
  }

  WaitForMultipleObjects(THREAD_COUNT, hThreads, TRUE, INFINITE);

  for (I=0; I<THREAD_COUNT; I++)
    CloseHandle(hThreads[I]);

  return 0;
}
```

Αν το εκτελέσουμε θα δούμε:

```
01234012340123401234012340123401234012340123401234012340123401234012340123401234
01234012340123401234
```

Επικοινωνία νημάτων

Τα νήματα μπορούν να επικοινωνήσουν μεταξύ τους με δύο τρόπους: με κοινή μνήμη και με μηνύματα.

Κοινή μνήμη

Τα νήματα μιας διεργασίας μοιράζονται τον ίδιο χώρο διευθύνσεων, αυτόν της διεργασίας. Ο πιο απλός τρόπος επικοινωνίας μεταξύ τους λοιπόν είναι να γράφουν και να διαβάζουν στην ίδια προσυμφωνημένη θέση μνήμης (π.χ. παγκόσμια μεταβλητή), προφανώς όμως είναι απαραίτητος ο συγχρονισμός όταν χρησιμοποιούν τη μνήμη αυτή.

Επικοινωνία των νημάτων με μηνύματα

Ο μηχανισμός των μηνυμάτων για επικοινωνία των νημάτων είναι επέκταση αυτού που χρησιμοποιείται στις παραθυρικές εφαρμογές.

Ένα μήνυμα είναι μία ακέραια τιμή, η οποία συνοδεύεται από δύο παραμέτρους των 32 bit. Π.χ. στις παραθυρικές εφαρμογές, η σταθερά WM_MOVE (=0x0003) δηλώνει ότι ένα παράθυρο μετακινήθηκε, και στις παραμέτρους του μηνύματος καταγράφεται η νέα του θέση. Στις παραμέτρους αυτές, που έχουν συνήθως τα ονόματα wParam και lParam, μπορεί να κωδικοποιηθεί μία τιμή ή ένας δείκτης.

Τα προγράμματα μπορούν να χρησιμοποιούν τα προκαθορισμένα μηνύματα των Windows, αλλά και να ορίσουν δικά τους, που ξεκινούν από την τιμή WM_USER (+1, +2, κλπ.).

Η αποστολή ενός μηνύματος γίνεται με τη ρουτίνα PostThreadMessage(), ενώ η λήψη μηνύματος από το νήμα γίνεται με τις GetMessage() ή PeekMessage().

Η ρουτίνα PostThreadMessage στέλνει το μήνυμα Msg με παραμέτρους wParam και lParam στο νήμα idThread και επιστρέφει αμέσως, προτού το νήμα να επεξεργαστεί το μήνυμα.

```
BOOL PostThreadMessage(
  DWORD idThread,
  UINT Msg,
  WPARAM wParam,
  LPARAM lParam
);
```

Αν πέτυχε η αποστολή του μηνύματος, η PostThreadMessage επιστρέφει μια μη μηδενική τιμή. Για να επιτύχει η αποστολή του μηνύματος πρέπει το νήμα-παραλήπτης να έχει δημιουργήσει την ουρά μηνυμάτων του με μια οποιαδήποτε κλήση συστήματος σε ρουτίνα της βιβλιοθήκης USER ή της GDI. Ο πιο απλός τρόπος δημιουργίας της ουράς είναι να επιχειρήσει το νήμα να ελέγξει αν του έχουν σταλεί μηνύματα με την PeekMessage().

Οι ρουτίνες GetMessage() και PeekMessage() παίρνουν σαν παράμετρο (μεταξύ άλλων) δείκτη σε ένα structure με τύπο MSG και συμπληρώνουν τα πεδία του. Το structure αυτό είναι δηλωμένο ως εξής:

```
typedef struct tagMSG {
    HWND   hwnd;
    UINT   message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD  time;
    POINT  pt;
} MSG;
```

Επειδή τα μηνύματα εδώ δεν αφορούν παράθυρα αλλά νήματα, τα πεδία hwnd και pt δεν έχουν νόημα. Τα υπόλοιπα πεδία είναι το μήνυμα (message), οι δυο του παράμετροι (wParam και lParam) και ο χρόνος κατά τον οποίο στάλθηκε το μήνυμα (time).

Η αφαίρεση ενός μηνύματος από την ουρά του νήματος γίνεται με την κλήση GetMessage().

```
BOOL GetMessage(
  LPMSG lpMsg,
  HWND hWnd,
  UINT wMsgFilterMin,
  UINT wMsgFilterMax
);
```

Η παράμετρος lpMsg είναι δείκτης σε ένα struct τύπου MSG που θα λάβει τις πληροφορίες για το μήνυμα. Το hWnd είναι το παράθυρο που αφορά το μήνυμα και για επικοινωνία μεταξύ νημάτων παίρνει την τιμή NULL. Τα wMsgFilterMin, wMsgFilterMax είναι η ελάχιστη και μέγιστη αντίστοιχα αριθμητική τιμή για τα μηνύματα που μπορούν να αφαιρεθούν από την ουρά. Με τον τρόπο αυτό το νήμα μπορεί να αφαιρέσει ένα μήνυμα που δε βρίσκεται στην κεφαλή της ουράς μηνυμάτων. Αν και τα δύο έχουν την τιμή 0, τότε αφαιρείται το πρώτο μήνυμα της ουράς.

Αν η ουρά του νήματος δεν περιέχει κάποιο μήνυμα, η GetMessage() μπλοκάρει μέχρι να σταλεί κάποιο μήνυμα. Επιστρέφει -1 σε περίπτωση λάθους, και 1 για όλα τα μηνύματα εκτός από το WM_QUIT για το οποίο επιστρέφει 0.

Για να ελέγξει ένα νήμα αν η ουρά του περιέχει κάποιο μήνυμα χωρίς όμως να μπλοκάρει, καλεί την PeekMessage:

```
BOOL PeekMessage(
  LPMSG lpMsg,
  HWND hWnd,
  UINT wMsgFilterMin,
  UINT wMsgFilterMax,
  UINT wRemoveMsg
);
```

Οι τέσσερεις πρώτες παράμετροι είναι ίδιες με αυτές της GetMessage(). Η τελευταία, με το όνομα wRemoveMsg, μπορεί να πάρει ως τιμή μια από τις ακόλουθες σταθερές:

- PM_NOREMOVE: Το μήνυμα δεν αφαιρείται από την ουρά, αλλά αντιγράφεται στο *lpMsg.
- PM_REMOVE: Το μήνυμα αντιγράφεται στο *lpMsg και αφαιρείται από την ουρά μηνυμάτων.

Η PeekMessage() επιστρέφει αμέσως είτε υπάρχει κατάλληλο μήνυμα στην ουρά του νήματος είτε όχι. Στην πρώτη περίπτωση επιστρέφει μη μηδενική τιμή, ενώ στη δεύτερη επιστρέφει 0.

Στο παρακάτω πρόγραμμα, το κύριο νήμα στέλνει μήνυμα στα υπόλοιπα νήματα για να τερματιστούν.

```
/* Make with cl /MT sample16.c user32.lib */

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

#define THREAD_COUNT 5
#define MY_MESSAGE   (WM_USER+1)

DWORD WINAPI MyRoutine(DWORD nThread)
{
  MSG msg;
```

```
  GetMessage(&msg, NULL, MY_MESSAGE, MY_MESSAGE);

  printf("Thread %d received message %u, wParam = %ld, lParam = %ld\n",
         (int)nThread, msg.message, msg.wParam, msg.lParam);

  return 0;
}

main(void)
{
  DWORD dwThreadIds[THREAD_COUNT], dwError;
  int I;
  HANDLE hThreads[THREAD_COUNT];

  for (I=0; I<THREAD_COUNT; I++) {
    hThreads[I] = CreateThread(NULL, 0,
                           (LPTHREAD_START_ROUTINE)MyRoutine,
                           (LPVOID)I,
                           0, dwThreadIds+I);

    if (NULL == hThreads[I]) {
      dwError = GetLastError();
      fprintf(stderr, "Error %ld creating thread.\n", (long)dwError);
      return 1;
    }
  }

  Sleep(100);

  for (I=0; I<THREAD_COUNT; I++)
    PostThreadMessage(dwThreadIds[I], MY_MESSAGE, I+2, 3*I+5);

  WaitForMultipleObjects(THREAD_COUNT, hThreads, TRUE, INFINITE);

  for (I=0; I<THREAD_COUNT; I++)
    CloseHandle(hThreads[I]);

  return 0;
}
```

Αν το εκτελέσουμε, θα δούμε:

```
Thread 0 received message 1025, wParam = 2, lParam = 5
Thread 1 received message 1025, wParam = 3, lParam = 8
Thread 2 received message 1025, wParam = 4, lParam = 11
Thread 3 received message 1025, wParam = 5, lParam = 14
Thread 4 received message 1025, wParam = 6, lParam = 17
```

Κεφάλαιο 5

Μέθοδοι επικοινωνίας μεταξύ διεργασιών

Ο συγχρονισμός μεταξύ διεργασιών μπορεί να γίνει όπως και στα νήματα με αντικείμενα συγχρονισμού (χρησιμοποιώντας κοινώς συμφωνημένα ονόματα). Η επικοινωνία όμως μεταξύ τους δεν μπορεί να γίνει με τους ίδιους τρόπους όπως στα νήματα: κάθε διεργασία έχει το δικό της χώρο μνήμης και δεν είναι πάντα δυνατό να στείλουμε μηνύματα από μία διεργασία στην άλλη.

Το Win32 διαθέτει μεθόδους επικοινωνίας διεργασιών οι οποίες αφορούν έναν μόνο υπολογιστή, ή μπορούν να λειτουργήσουν και σε διαφορετικούς υπολογιστές ενός δικτύου. Οι μέθοδοι επικοινωνίας και ανταλλαγής δεδομένων στον ίδιο υπολογιστή είναι:

- «Μοιραζόμενη μνήμη» μέσω memory mapped files.
- Ανώνυμες σωληνώσεις.
- Η ενσωμάτωση αρχείων με το OLE (Object Linking and Embedding). Το πρωτόκολλο αυτό παρέχει τη δυνατότητα της δημιουργίας σύνθετων αρχείων (compound documents). Για κάθε τμήμα του αρχείου μπορεί να είναι υπεύθυνη μία διαφορετική διεργασία, η οποία καλείται να το διαχειριστεί όποτε αυτό είναι απαραίτητο.
- Το μήνυμα WM_COPYDATA. Με αυτό το μήνυμα μπορούν να αποσταλούν δεδομένα που βρίσκονται στο χώρο διευθύνσεων της μίας διεργασίας και το ΛΣ αναλαμβάνει να τα «μεταφέρει» στο χώρο διευθύνσεων της άλλης.

Οι δικτυακές μέθοδοι επικοινωνίας και ανταλλαγής δεδομένων μεταξύ διεργασιών που διαθέτει το Win32 είναι:

- Ο clipboard. Διεργασίες στον ίδιο ή σε διαφορετικούς υπολογιστές μπορούν να ανταλλάξουν δεδομένα μέσω του clipboard, συμφωνώντας μόνο για την κωδικοποίηση των δεδομένων.
- Το πρωτόκολλο DDE (Dynamic Data Exchange). Πρόκειται για μία επέκταση του clipboard, η οποία παρέχει υπηρεσίες συνεχούς ανταλλαγής δεδομένων.
- Τα mailslots, που είναι παρόμοια με τις ουρές μηνυμάτων των νημάτων. Μία διεργασία δημιουργεί ένα mailslot, στο οποίο οι υπόλοιπες αποστέλλουν μηνύματα. Τα mailslots δίνουν τη δυνατότητα για broadcasting μικρών μηνυμάτων.
- Οι σωληνώσεις (named pipes).
- Οι κλήσεις RPC (Remote Procedure Call), οι οποίες είναι συμβατές με το πρότυπο DCE (Distributed Computing Environment) του Open Software Foundation, και παρέχουν τη δυνατότητα επικοινωνίας με οποιοδήποτε άλλο λειτουργικό σύστημα υποστηρίζει το DCE.
- Τα Windows sockets, που βασίζονται στα BSD sockets. Και αυτά παρέχουν τη δυνατότητα επικοινωνίας με οποιοδήποτε λειτουργικό σύστημα υλοποιεί τα sockets.

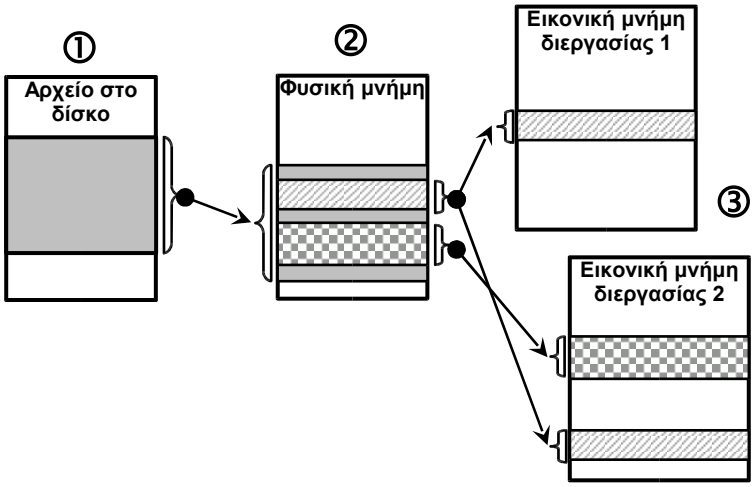
Στο κεφάλαιο αυτό θα ασχοληθούμε με τη μοιραζόμενη μνήμη (file mapping) και τις σωληνώσεις (pipes).

File mapping

Μία διεργασία μπορεί να απεικονίσει ένα αρχείο στο χώρο μνήμης της και να διαβάζει ή να γράφει σε αυτό σαν να ήταν ένα τμήμα μνήμης (πιο ειδικά ένας πίνακας). Αυτή η διαδικασία είναι χρήσιμη

όταν μία διεργασία χρειάζεται συχνή τυχαία προσπέλαση σε ένα αρχείο, αλλά μπορεί να χρησιμοποιηθεί και για την υλοποίηση μοιραζόμενης μνήμης, όταν δύο ή περισσότερες διεργασίες απεικονίζουν στο χώρο μνήμης τους το ίδιο αρχείο ή το ίδιο τμήμα του swap file. Η πρόσβαση βέβαια στα κοινά δεδομένα πρέπει να ελέγχεται με κάποιο αντικείμενο συγχρονισμού.

Η διεργασία που θέλει να απεικονίσει ένα αρχείο στο χώρο διευθύνσεών της πρέπει να ακολουθήσει τα παρακάτω βήματα:



① Να ανοίξει το αρχείο με την κλήση CreateFile, που θα δούμε αναλυτικά στο Κεφ. 6. Αν η απεικόνιση αφορά το swap file, αυτό το βήμα δεν είναι απαραίτητο. Η CreateFile επιστρέφει ένα handle για το ανοικτό αρχείο, το οποίο χρησιμοποιεί στη συνέχεια η διεργασία.

② Να δημιουργήσει το αντικείμενο του file mapping με την κλήση CreateFileMapping ή να το ανοίξει με την OpenFileMapping.

③ Να απεικονίσει κάποιο τμήμα του αρχείου στο χώρο διευθύνσεών της με την κλήση MapViewOfFile. Η κλήση αυτή επιστρέφει την αρχική εικονική διεύθυνση του τμήματος του αρχείου.

Στη συνέχεια η διεργασία μπορεί να χρησιμοποιεί τη διεύθυνση αυτή σαν ένα κοινό δείκτη, διαβάζοντας και γράφοντας στο αρχείο. Όταν ολοκληρώσει τη χρήση του αρχείου, πρέπει να ακολουθήσει την αντίστροφη πορεία:

④ Να ακυρώσει την απεικόνιση του αρχείου στη μνήμη της διεργασίας με την κλήση UnmapViewOfFile.

⑤ Να καταστρέψει το αντικείμενο του file mapping με την κλήση CloseHandle.

⑥ Να κλείσει το αρχείο που είχε ανοίξει αρχικά με την κλήση CloseHandle.

CreateFileMapping

Με αυτή την κλήση δημιουργείται ένα αντικείμενο για το file mapping, στο οποίο μπορούν να «συνδεθούν» στη συνέχεια άλλες διεργασίες. Στο αντικείμενο πρέπει να δοθεί ένα όνομα.

```
HANDLE CreateFileMapping(
  HANDLE hFile,
  LPSECURITY_ATTRIBUTES lpFileMappingAttributes,
  DWORD flProtect,
  DWORD dwMaximumSizeHigh,
  DWORD dwMaximumSizeLow,
  LPCTSTR lpName
);
```

Η παράμετρος hFile είναι το handle ενός ανοικτού αρχείου, ή η τιμή (HANDLE)0xFFFFFFFF που υποδηλώνει το swap file. Η επόμενη παράμετρος αφορά τα χαρακτηριστικά ασφαλείας του αντικειμένου και συνήθως είναι NULL. Στην παράμετρο flProtect δίνουμε τους τρόπους πρόσβασης στο αντικείμενο, που μπορούν να είναι PAGE_READONLY (μόνο για ανάγνωση), PAGE_READWRITE (για ανάγνωση και εγγραφή) ή PAGE_WRITECOPY (για copy on write). Η πρόσβαση στο αντικείμενο πρέπει να είναι συμβατή με τον τρόπο που έχουμε ανοίξει το αρχείο με την CreateFile. Στις dwMaximumSizeHigh και dwMaximumSizeLow δίνουμε το μέγιστο μέγεθος του mapping· αν είναι 0, τότε το μέγιστο μέγεθός του είναι ίδιο με το μέγεθος του αρχείου. Τέλος, στην παράμετρο lpName δίνουμε το όνομα του αντικειμένου. Αν η κλήση επιτύχει, επιστρέφει το handle του αντικειμένου. Αν το αντικείμενο υπάρχει ήδη, μία κλήση της GetLastError αμέσως μετά θα επιστρέψει ERROR_ALREADY_EXISTS.

OpenFileMapping

Με αυτή την κλήση ανοίγουμε ένα αντικείμενο mapping που υπάρχει ήδη, μέσω του ονόματός του.

```
HANDLE OpenFileMapping(
  DWORD dwDesiredAccess,
  BOOL bInheritHandle,
  LPCTSTR lpName
);
```

Η παράμετρος dwDesiredAccess δηλώνει την επιθυμητή πρόσβαση στο αντικείμενο και μπορεί να είναι FILE_MAP_READ (για ανάγνωση), FILE_MAP_WRITE (για ανάγνωση και εγγραφή), ή FILE_MAP_COPY (για copy on write). Η πρόσβαση αυτή πρέπει να είναι συμβατή με εκείνη που προσδιορίστηκε όταν δημιουργήθηκε το αντικείμενο. Με την παράμετρο bInheritHandle καθορίζουμε αν επιθυμούμε να κληροδοτήσουμε το handle σε τυχόν άλλες διεργασίες που θα δημιουργήσουμε και lpName είναι το όνομα του αντικειμένου. Αν η κλήση επιτύχει, επιστρέφει το handle του αντικειμένου.

MapViewOfFile

Με την κλήση αυτή «απεικονίζουμε» ένα τμήμα ενός mapping στον χώρο διευθύνσεων της διεργασίας.

```
LPVOID MapViewOfFile(
  HANDLE hFileMappingObject,
  DWORD dwDesiredAccess,
  DWORD dwFileOffsetHigh,
  DWORD dwFileOffsetLow,
  DWORD dwNumberOfBytesToMap
);
```

Το handle του αντικειμένου του mapping (που έχουμε πάρει από την CreateFileMapping ή την OpenFileMapping) δίνεται στην παράμετρο hFileMappingObject. Η πρόσβαση στο τμήμα του αρχείου που απεικονίζουμε δίνεται από το dwDesiredAccess και μπορεί να είναι PAGE_READONLY (μόνο για ανάγνωση), PAGE_READWRITE (για ανάγνωση και εγγραφή) ή PAGE_WRITECOPY (για copy on write). Οι επόμενες δύο παράμετροι, dwFileOffsetHigh και dwFileOffsetLow συνδυάζονται για να δώσουν μία τιμή 64 bit, την θέση μέσα στο αρχείο από την οποία θα γίνει η απεικόνιση. Το πλήθος των bytes που θα απεικονιστούν δίνονται από την

παράμετρο dwNumberOfBytesToMap · αν αυτή είναι 0, η απεικόνιση φθάνει μέχρι το τέλος του αρχείου.

Αν η κλήση επιτύχει, επιστρέφει την αρχική διεύθυνση του τμήματος όπου έγινε η απεικόνιση, αλλιώς επιστρέφει NULL.

UnmapViewOfFile

Με την κλήση αυτή ακυρώνεται η αντιστοίχιση τμήματος από ένα αντικείμενο file mapping στη μνήμη μίας διεργασίας.

```
BOOL UnmapViewOfFile(LPCVOID lpBaseAddress);
```

Σαν παράμετρο της δίνουμε τη διεύθυνση που μας είχε επιστρέψει η MapViewOfFile.

Το πρόγραμμα που ακολουθεί, δημιουργεί μία απεικόνιση του κώδικά του και τυπώνει επαναληπτικά ένα τμήμα του στην οθόνη, με μικρή καθυστέρηση. Αν δύο ή περισσότερα στιγμιότυπα του προγράμματος εκτελούνται, τυπώνουν συγχρόνως τα ίδια περιεχόμενα.

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

#define MAPPING_NAME    "Test mapping"

main(void)
{
  HANDLE hFile, hMapping;
  PVOID pFileMap;
  int I, J;

   hFile = CreateFile("mapping1.c", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0,
NULL);
  if (INVALID_HANDLE_VALUE == hFile)
    hMapping = OpenFileMapping(FILE_MAP_READ, FALSE, MAPPING_NAME);
  else
         hMapping = CreateFileMapping(hFile, NULL, PAGE_READONLY, 0, 0,
MAPPING_NAME);

  if (NULL == hMapping) {
     fprintf(stderr, "Cannot create or open mapping. Error %ld\n", GetLastError
());
    CloseHandle(hFile);
    exit(1);
  }

  pFileMap = MapViewOfFile(hMapping, FILE_MAP_READ, 0, 0, 0);
  if (NULL == pFileMap) {
    fprintf(stderr, "Cannot map view. Error %ld\n", GetLastError());
    CloseHandle(hMapping);
    CloseHandle(hFile);
    exit(1);
  }

  for (I=0; I<1000; I++) {
    printf("%c", ((char *)pFileMap)[I%200]);
    Sleep(10);
  }

  UnmapViewOfFile(pFileMap);
  CloseHandle(hMapping);
  CloseHandle(hFile);

  return 0;
}
```

Το παρακάτω πρόγραμμα, δημιουργεί μία απεικόνιση 20 bytes από το swap file με δικαιώματα ανάγνωσης και εγγραφής. Αν από την CreateFileMapping διαπιστώσει ότι είναι το πρώτο στιγμιότυπο του προγράμματος που εκτελείται, γράφει επαναληπτικά και στα 20 bytes της απεικόνισης την ίδια τιμή, αλλά διαφορετική τιμή σε κάθε επανάληψη. Τα άλλα στιγμιότυπα του προγράμματος τυπώνουν τα περιεχόμενα της μοιραζόμενης μνήμης.

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

#define MAPPING_NAME    "Test mapping"
#define MAPPING_BYTES   20

main(void)
{
  HANDLE hMapping;
  PVOID pFileMap;
  int I, J;
  BOOL bFirst = TRUE;

  hMapping = CreateFileMapping((HANDLE)0xffffffff, NULL,
                               PAGE_READWRITE, 0, MAPPING_BYTES, MAPPING_NAME);

  if (NULL == hMapping) {
     fprintf(stderr, "Cannot create or open mapping. Error %ld\n", GetLastError
());
    exit(1);
  }

  if (ERROR_ALREADY_EXISTS == GetLastError())
    bFirst = FALSE;

  pFileMap = MapViewOfFile(hMapping, FILE_MAP_READ|FILE_MAP_WRITE, 0, 0, 0);
  if (NULL == pFileMap) {
    fprintf(stderr, "Cannot map view. Error %ld\n", GetLastError());
    CloseHandle(hMapping);
    exit(1);
  }

  if (bFirst) {
    for (I=0; I<1000; I++) {
      ((char *)pFileMap)[I%MAPPING_BYTES] = 'A'+(I/MAPPING_BYTES);
      Sleep(10);
    }
  }
  else {
    for (I=0; I<1000; I++) {
      printf("%c", ((char *)pFileMap)[I%MAPPING_BYTES]);
      Sleep(10);
    }
  }

  UnmapViewOfFile(pFileMap);
  CloseHandle(hMapping);

  return 0;
}
```

Αν εκτελέσουμε το πρόγραμμα αυτό σε πολλές διεργασίες, η μία διεργασία (η πρώτη) δεν θα τυπώνει τίποτα, ενώ οι υπόλοιπες θα τυπώνουν:

```
AAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCCCCDDDDDDDDDDDDDDDDDDDD
EEEEEEEEEEEEEEEEEEEFFFFFFFFFFFFFFFFFFFGGGGGGGGGGGGGGGGGGGGHHHHHHHHHHHHHHHHHHH
IIIIIIIIIIIIIIIIIIIJJJJJJJJJJJJJJJJJJJKKKKKKKKKKKKKKKKKKKLLLLLLLLLLLLLLLLLLLL
MMMMMMMMMMMMMMMMMMMNNNNNNNNNNNNNNNNNNNOOOOOOOOOOOOOOOOOOOPPPPPPPPPPPPPPPPPPP
QQQQQQQQQQQQQQQQQQQQRRRRRRRRRRRRRRRRRRRSSSSSSSSSSSSSSSSSSSTTTTTTTTTTTTTTTTTTT
UUUUUUUUUUUUUUUUUUUVVVVVVVVVVVVVVVVVVVWWWWWWWWWWWWWWWWWWWXXXXXXXXXXXXXXXXXXX
```

```
κλπ.
```

Σωληνώσεις

Δύο διεργασίες μπορούν να επικοινωνήσουν μεταξύ τους μέσω μίας σωλήνωσης, η οποία μπορεί να είναι ανώνυμη ή να έχει όνομα.

Ανώνυμες σωληνώσεις

Οι ανώνυμες σωληνώσεις έχουν περιορισμένη εφαρμογή: μόνο δύο διεργασίες μπορούν να επικοινωνήσουν από αυτές και έχουν μία κατεύθυνση μόνο. Οι δύο διεργασίες πρέπει να τρέχουν στον ίδιο υπολογιστή.

Μία διεργασία δημιουργεί μία ανώνυμη σωλήνωση με την κλήση CreatePipe.

```
BOOL CreatePipe(
  PHANDLE hReadPipe,
  PHANDLE hWritePipe,
  LPSECURITY_ATTRIBUTES lpPipeAttributes,
  DWORD nSize
);
```

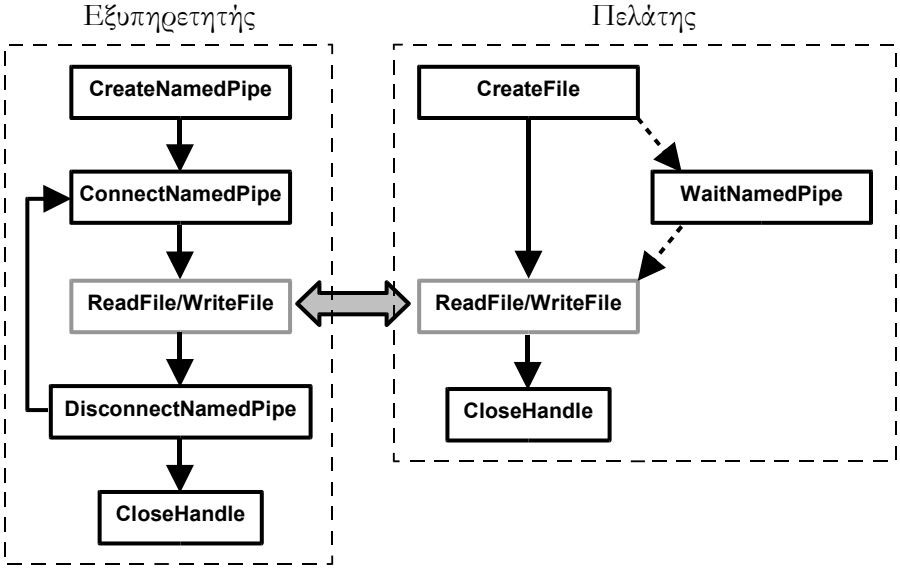
Οι δύο πρώτες παράμετροι είναι διευθύνσεις μεταβλητών που θα δεχθούν τα άκρα ανάγνωσης και εγγραφής της σωλήνωσης. Η τρίτη παράμετρος αφορά την ασφάλεια (συνήθως NULL) και η τελευταία υποδεικνύει στο σύστημα μία τιμή για το μέγεθος των buffers της σωλήνωσης· αν nSize==0, τότε επιλέγεται μία προκαθορισμένη τιμή.

Στη συνέχεια, η διεργασία μπορεί να «στείλει» την τιμή του handle για ένα από τα δύο άκρα της σωλήνωσης για να αρχίσει η επικοινωνία, που γίνεται με κλήσεις ReadFile και WriteFile (βλ. Κεφ. 6). Όταν η ανταλλαγή δεδομένων ολοκληρωθεί, οι διεργασίες κλείνουν τα δύο άκρα της σωλήνωσης με την CloseHandle.

Σωληνώσεις με όνομα

Μία σωλήνωση με όνομα (στο εξής απλά σωλήνωση) είναι ένας απλός τρόπος μονόδρομης ή αμφίδρομης επικοινωνίας μεταξύ διεργασιών. Κάθε σωλήνωση διαθέτει έναν *εξυπηρετητή*, τη διεργασία που την δημιούργησε και έναν ή περισσότερους *πελάτες*, οι οποίοι συνδέονται στη σωλήνωση. Για κάθε πελάτη δημιουργείται ένα ιδιαίτερο στιγμιότυπο της σωλήνωσης, με ξεχωριστούς buffers.

Η διαδικασία επικοινωνίας μεταξύ εξυπηρετητή και πελάτη φαίνεται στο σχήμα.



CreateNamedPipe

Αρχικά η διεργασία-εξυπηρετητής δημιουργεί τη σωλήνωση με την κλήση CreateNamedPipe. Η κλήση αυτή μπορεί να επαναληφθεί για τη δημιουργία πολλών στιγμιοτύπων της σωλήνωσης, που θα εξυπηρετούν πολλούς πελάτες παράλληλα.

```
HANDLE CreateNamedPipe(
  LPCTSTR lpName,
  DWORD dwOpenMode,
  DWORD dwPipeMode,
  DWORD nMaxInstances,
  DWORD nOutBufferSize,
  DWORD nInBufferSize,
  DWORD nDefaultTimeOut,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

Οι παράμετροι της κλήσης είναι:

- lpName: Το όνομα της σωλήνωσης. Έχει τη μορφή **\\.\pipe**Όνομα-Σωλήνωσης. Το όνομα «Όνομα-Σωλήνωσης» είναι case insensitive.
- dwOpenMode: Ο τρόπος δημιουργίας της σωλήνωσης, με κυριότερες τιμές PIPE_ACCESS_DUPLEX (διπλής κατευθύνσεως), PIPE_ACCESS_INBOUND (ο πελάτης μόνο στέλνει δεδομένα στον εξυπηρετητή) ή PIPE_ACCESS_OUTBOUND (ο εξυπηρετητής μόνο στέλνει δεδομένα στον πελάτη).
- dwPipeMode: Τρόπος λειτουργίας της σωλήνωσης (τα δεδομένα γράφονται σαν ακολουθία bytes ή μηνυμάτων · τα δεδομένα διαβάζονται από τη σωλήνωση σαν ακολουθία bytes ή μηνυμάτων · οι λειτουργίες της σωλήνωσης μπλοκάρουν μέχρι να ολοκληρωθούν ή όχι). Πρέπει να έχει την ίδια τιμή σε κάθε κλήση της CreateNamedPipe για την ίδια σωλήνωση.
- nMaxInstances: Το μέγιστο πλήθος από στιγμιότυπα της σωλήνωσης που μπορούν να είναι ανοικτά ταυτόχρονα. Πρέπει να έχει την ίδια τιμή σε κάθε κλήση της CreateNamedPipe για την ίδια σωλήνωση.
- nOutBufferSize, nInBufferSize: Το μέγεθος των buffers εισόδου και εξόδου για τη σωλήνωση. Αυτές είναι «συνιστώμενες» τιμές από τη διεργασία και δεν είναι δεσμευτικές για το σύστημα.
- nDefaultTimeOut: Ο προκαθορισμένος χρόνος που θα περιμένει ένας πελάτης για να συνδεθεί με την WaitNamedPipe. Πρέπει να έχει την ίδια τιμή σε κάθε κλήση της CreateNamedPipe για την ίδια σωλήνωση.
- lpSecurityAttributes: Χαρακτηριστικά ασφαλείας, συνήθως NULL.

Αν η κλήση επιτύχει, επιστρέφει ένα handle για τη σωλήνωση.

ConnectNamedPipe

Ένας εξυπηρετητής περιμένει σύνδεση από ένα πελάτη σε μία σωλήνωση με την κλήση ConnectNamedPipe.

```
BOOL ConnectNamedPipe(
  HANDLE hNamedPipe,
  LPOVERLAPPED lpOverlapped
);
```

Στην παράμετρο hNamedPipe δίνουμε το handle που μας έχει επιστρέψει η CreateNamedPipe. Η παράμετρος lpOverlapped χρησιμοποιείται αν θέλουμε η σύνδεση να γίνει ασύγχρονα και η ConnectNamedPipe να επιστρέψει ανεξάρτητα από το αν έγινε η σύνδεση. Αν το lpOverlapped

είναι NULL, η κλήση κολλά έως ότου ένας πελάτης συνδεθεί με την CreateFile ή την CallNamedPipe.

Αν ο εξυπηρετητής θέλει να έχει πολλά στιγμιότυπα της ίδιας σωλήνωσης ταυτόχρονα ανοικτά, πρέπει μόλις επιστρέψει η ConnectNamedPipe να δημιουργήσει ένα νέο στιγμιότυπο (με την CreateNamedPipe πάλι) και να κάνει σε ένα άλλο νήμα κλήση στην ConnectNamedPipe.

Οι αναγνώσεις και εγγραφές σε μία σωλήνωση που έχει συνδεθεί με πελάτη γίνονται με τις κλήσεις ReadFile και WriteFile που περιγράφονται στο Κεφ.6.

DisconnectNamedPipe

Όταν ολοκληρωθεί η επικοινωνία μεταξύ πελάτη και εξυπηρετητή, ο εξυπηρετητής καλεί την DisconnectNamedPipe για να αποσυνδέσει τη σωλήνωση. Μετά από αυτή την κλήση, ο εξυπηρετητής μπορεί να καλέσει ξανά την ConnectNamedPipe για να αναμείνει την επόμενη σύνδεση.

```
BOOL DisconnectNamedPipe(HANDLE hNamedPipe);
```

Ο πελάτης

Για να συνδεθεί με μία σωλήνωση, η διεργασία-πελάτης πρέπει να χρησιμοποιήσει την κλήση CreateFile.

```
HANDLE CreateFile(
  LPCTSTR lpFileName,
  DWORD dwDesiredAccess,
  DWORD dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD dwCreationDistribution,
  DWORD dwFlagsAndAttributes,
  HANDLE hTemplateFile
);
```

Στην παράμετρο lpFileName δίνουμε το όνομα της σωλήνωσης με τη μορφή

\\Όνομα-Υπολογιστή**pipe**\Όνομα-Σωλήνωσης

Αν ο εξυπηρετητής εκτελείται στον ίδιο υπολογιστή με τον πελάτη, τότε το «Όνομα-Υπολογιστή» μπορεί να είναι «.». Οι υπόλοιπες παράμετροι περιγράφονται στη σελ. 48.

Η CreateFile μπορεί να αποτύχει, γιατί ο εξυπηρετητής δεν έχει καλέσει ακόμα την ConnectNamedPipe ή γιατί όλα τα στιγμιότυπα της σωλήνωσης είναι συνδεδεμένα με άλλους πελάτης. Στην περίπτωση αυτή θα επιστρέψει INVALID_HANDLE_VALUE. Ο πελάτης μπορεί να μπει σε κατάσταση αναμονής μέχρι να γίνει διαθέσιμο κάποιο στιγμιότυπο με την WaitNamedPipe.

```
BOOL WaitNamedPipe(LPCTSTR lpNamedPipeName, DWORD nTimeOut);
```

Στην πρώτη παράμετρο δίνουμε το όνομα της σωλήνωσης (ίδιο με αυτό που δώσαμε στην CreateFile) και στη δεύτερη δίνουμε το χρόνο για τον οποίο θα περιμένει ο πελάτης, σε ms. Όταν αυτό το διάστημα εκπνεύσει, η WaitNamedPipe επιστρέφει, ανεξάρτητα από το αν είναι διαθέσιμο κάποιο στιγμιότυπο της σωλήνωσης. Αντί για μία αριθμητική τιμή, μπορούμε να δώσουμε μία από τις σταθερές:

- NMPWAIT_USE_DEFAULT_WAIT: Ο πελάτης περιμένει για το προκαθορισμένο διάστημα της σωλήνωσης (την παράμετρο nDefaultTimeOut της CreateNamedPipe).
- NMPWAIT_WAIT_FOREVER: Ο πελάτης επιστρέφει μόνο αν είναι διαθέσιμο ένα στιγμιότυπο της σωλήνωσης.

Τα δεδομένα γράφονται και διαβάζονται από τη σωλήνωση με τις ReadFile και WriteFile. Όταν ο πελάτης τελειώσει, κλείνει το handle της σωλήνωσης με την CloseHandle.

Τα δύο προγράμματα που ακολουθούν είναι ένας απλός εξυπηρετητής που δέχεται μία σύνδεση κάθε φορά και στέλνει σε αυτή 10 φορές την τρέχουσα τιμή του χρονομέτρου του συστήματος (tick count) και ο αντίστοιχος πελάτης.

Εξυπηρετητής

```
/* Make with cl pipes1.c user32.lib */

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

main(int argc, char *argv[])
{
  char *szPipeName = "TestPipe", szFullName[256];
  HANDLE hPipe;
  int nClients, nReps;

  if (1 < argc)
    szPipeName = argv[1];

  wsprintf(szFullName, "\\\\.\\pipe\\%s", szPipeName);
  hPipe = CreateNamedPipe(szFullName, PIPE_ACCESS_OUTBOUND,
                          PIPE_TYPE_MESSAGE|PIPE_READMODE_MESSAGE,
                          1, 256, 256, 1000, NULL);
  if (NULL == hPipe) {
    fprintf(stderr, "Cannot create named pipe. Error %ld.\n", GetLastError());
    exit(1);
  }

  for (nClients=0; nClients<10; nClients++) {
    ConnectNamedPipe(hPipe, NULL);
    printf("Connected to client.\n");

    for (nReps=0; nReps<10; nReps++) {
      DWORD dwTime = GetTickCount(), dwWritten;

      if (!WriteFile(hPipe, &dwTime, sizeof(dwTime), &dwWritten, NULL)) {
        fprintf(stderr, "Error %ld writing data to pipe... exiting.\n",
                GetLastError());
        DisconnectNamedPipe(hPipe);
        CloseHandle(hPipe);
        exit(2);
      }

      Sleep(200);
    }

    DisconnectNamedPipe(hPipe);
  }

  CloseHandle(hPipe);

  return 0;
}
```

Πελάτης

```
/* Make with cl pipes2.c user32.lib */

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

main(int argc, char *argv[])
```

```
{
  char *szPipeName = "TestPipe",
       *szServer = ".",
       szFullName[256];

  HANDLE hPipe;
  DWORD dwTime, dwRead;

  if (1 < argc)
    szServer = argv[1];

  if (2 < argc)
    szPipeName = argv[2];

  wsprintf(szFullName, "\\\\%s\\pipe\\%s", szServer, szPipeName);

  hPipe = CreateFile(szFullName, GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);

  if (INVALID_HANDLE_VALUE == hPipe &&
      !WaitNamedPipe(szFullName, NMPWAIT_USE_DEFAULT_WAIT)) {
     fprintf(stderr, "Cannot connect to named pipe. Error %ld.\n", GetLastError
());
    exit(1);
  }

  while (ReadFile(hPipe, &dwTime, sizeof(dwTime), &dwRead, NULL))
    printf("Time read: %lu\n", dwTime);

  CloseHandle(hPipe);

  return 0;
}
```

Κεφάλαιο

6 Το σύστημα αρχείων

Για να διαβάσει ή να γράψει κανείς σε αρχεία, μπορεί κατ' αρχή να χρησιμοποιήσει τη standard βιβλιοθήκη της C (fopen, fread, κλπ.). Η βιβλιοθήκη αυτή όμως δεν παρέχει όλες τις δυνατές υπηρεσίες του συστήματος αρχείων (προσωρινά αρχεία, memory mapped files) και οι περιγραφητές αρχείων που περιέχει δεν μπορούν να χρησιμοποιηθούν σαν παράμετροι σε κλήσεις συστήματος. Για το λόγο αυτό, σε προγράμματα που γράφονται για Win32 είναι προτιμότερο να χρησιμοποιούμε τις κλήσεις συστήματος για τα αρχεία.

Άνοιγμα ή δημιουργία αρχείου

Για να ανοίξουμε ή να δημιουργήσουμε ένα αρχείο χρησιμοποιούμε την κλήση CreateFile. Με την κλήση αυτή, όπως έχουμε δει, μπορούμε να «ανοίξουμε» και άλλα αντικείμενα του λειτουργικού συστήματος, όπως τα pipes.

```
HANDLE CreateFile(
  LPCTSTR lpFileName,
  DWORD dwDesiredAccess,
  DWORD dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD dwCreationDistribution,
  DWORD dwFlagsAndAttributes,
  HANDLE hTemplateFile
);
```

Στην παράμετρο lpFileName δίνουμε το όνομα του αρχείου, με ή χωρίς πλήρες μονοπάτι. Η παράμετρος dwDesiredAccess δίνει τον τρόπο πρόσβασης που θέλουμε να έχουμε στο αρχείο: GENERIC_READ για ανάγνωση, GENERIC_WRITE για εγγραφή, GENERIC_READ | GENERIC_WRITE και για τα δύο. Η παράμετρος dwShareMode δίνει το αν και πώς το αρχείο μπορεί να μοιραστεί όσο είναι ανοικτό: με FILE_SHARE_READ δίνουμε πρόσβαση σε όποιον θέλει να ανοίξει το αρχείο για ανάγνωση, με FILE_SHARE_WRITE READ δίνουμε πρόσβαση σε όποιον θέλει να ανοίξει το αρχείο για εγγραφή. Η παράμετρος lpSecurityAttributes αφορά την ασφάλεια (συνήθως είναι NULL).

Με την παράμετρο dwCreationDistribution ελέγχουμε τον τρόπο δημιουργίας ή ανοίγματος του αρχείου:

CREATE_NEW	Δημιουργεί ένα νέο αρχείο και αποτυγχάνει αν το αρχείο υπάρχει.
CREATE_ALWAYS	Δημιουργεί ένα νέο αρχείο · αν το αρχείο υπάρχει, διαγράφει τα περιεχόμενά του.
OPEN_EXISTING	Ανοίγει το αρχείο, μόνο αν υπάρχει.
OPEN_ALWAYS	Ανοίγει το αρχείο και αν δεν υπάρχει το δημιουργεί.

TRUNCATE_EXISTING	Ανοίγει το αρχείο, το οποίο πρέπει να υπάρχει και διαγράφει όλα τα περιεχόμενά του. Πρέπει να έχει ζητηθεί πρόσβαση εγγραφής για το αρχείο (GENERIC_WRITE).

Η παράμετρος dwFlagsAndAttributes καθορίζει διάφορα άλλα χαρακτηριστικά του αρχείου. Π.χ. με την τιμή FILE_FLAG_HIDDEN καθορίζεται ότι το αρχείο είναι «κρυμμένο», με την τιμή FILE_ATTRIBUTE_TEMPORARY ορίζουμε ένα προσωρινό αρχείο κλπ.

Η παράμετρος hTemplateFile είναι το handle ενός template αρχείου που περιέχει πληροφορίες για τα χαρακτηριστικά του αρχείου που ανοίγουμε ή δημιουργούμε και συνήθως είναι NULL (δεν υποστηρίζεται στα Win95).

Η κλήση επιστρέφει το handle στο ανοικτό αρχείο αν επιτύχει, ή την τιμή INVALID_HANDLE_-VALUE αν αποτύχει. Ο κωδικός λάθους βρίσκεται αν καλέσουμε την GetLastError.

Κλείσιμο αρχείου

Κλείνουμε ένα αρχείο με τη ρουτίνα CloseHandle.

Ανάγνωση και εγγραφή σε αρχείο

Η ανάγνωση από ένα αρχείο (το οποίο έχει ανοιχθεί με δικαιώματα που περιέχουν το GENERIC_READ) γίνεται με τη ρουτίνα ReadFile:

```
BOOL ReadFile(
  HANDLE hFile,
  LPVOID lpBuffer,
  DWORD nNumberOfBytesToRead,
  LPDWORD lpNumberOfBytesRead,
  LPOVERLAPPED lpOverlapped
);
```

Για πρώτη παράμετρο, δίνουμε στη ReadFile το handle ενός ανοικτού αρχείου. Η παράμετρος lpBuffer είναι η διεύθυνση μνήμης όπου θα αποθηκευθούν τα δεδομένα, τα οποία θα έχουν μέγεθος nNumberOfBytesToRead. Η παράμετρος lpNumberOfBytesRead είναι η διεύθυνση μίας μεταβλητής όπου γράφεται το πλήθος των bytes που διαβάστηκαν τελικά. Το lpOverlapped αφορά ασύγχρονη ανάγνωση και για απλή (σύγχρονη) ανάγνωση είναι NULL.

Η ReadFile επιστρέφει TRUE αν η ανάγνωση πέτυχε, αλλιώς επιστρέφει FALSE.

Η εγγραφή σε ένα αρχείο (το οποίο έχει ανοιχθεί με δικαιώματα που περιέχουν το GENERIC_WRITE) γίνεται με τη ρουτίνα WriteFile:

```
BOOL WriteFile(
  HANDLE hFile,
  LPCVOID lpBuffer,
  DWORD nNumberOfBytesToWrite,
  LPDWORD lpNumberOfBytesWritten,
  LPOVERLAPPED lpOverlapped
);
```

Η παράμετρος lpBuffer είναι η διεύθυνση των δεδομένων που θα γραφούν στο αρχείο, τα οποία έχουν μέγεθος nNumberOfBytesToWrite. Η παράμετρος lpNumberOfBytesWritten είναι η διεύθυνση μίας μεταβλητής όπου γράφεται το πλήθος των bytes που γράφτηκαν τελικά.

Οι υπόλοιπες παράμετροι και η τιμή επιστροφής είναι όπως και στη ReadFile.

Το πρόγραμμα που ακολουθεί αντιγράφει ένα αρχείο σε ένα άλλο (αυτό γίνεται και με την κλήση συστήματος CopyFile).

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

main(int argc, char *argv[])
{
  HANDLE hReadFile, hWriteFile;
  DWORD dwError, dwBytesRead;

  if (3 > argc) {
    fprintf(stderr, "Usage: %s source-file destination-file\n", argv[0]);
    exit(1);
  }

  hReadFile = CreateFile(argv[1], GENERIC_READ,
                         FILE_SHARE_READ, NULL,
                         OPEN_EXISTING, 0, NULL);

  if (INVALID_HANDLE_VALUE == hReadFile) {
    fprintf(stderr, "Cannot open file %s for reading. Error %ld\n",
            argv[1], GetLastError());
    exit(2);
  }

  hWriteFile = CreateFile(argv[2], GENERIC_WRITE,
                          0, NULL,
                          CREATE_ALWAYS, 0, NULL);

  if (INVALID_HANDLE_VALUE == hWriteFile) {
    fprintf(stderr, "Cannot open file %s for writing. Error %ld\n",
            argv[2], GetLastError());
    CloseHandle(hReadFile);
    exit(2);
  }

  do {
    DWORD dwBytesWritten;
    BYTE pBuffer[1024]; /* Read 1KB at a time */

    if (!ReadFile(hReadFile, pBuffer, sizeof(pBuffer), &dwBytesRead, NULL)) {
      fprintf(stderr, "Error %ld reading file.\n", GetLastError());
      break;
    }

    if (!WriteFile(hWriteFile, pBuffer, dwBytesRead, &dwBytesWritten, NULL)) {
      fprintf(stderr, "Error %ld writing file.\n", GetLastError());
      break;
    }

  } while (0 != dwBytesRead);


  CloseHandle(hReadFile);
  CloseHandle(hWriteFile);

  return 0;
}
```

Μετακινηση μεσα στο αρχειο

Κάθε ανάγνωση και εγγραφή από ένα αρχείο γίνεται από το «τρέχον» σημείο του αρχείου, το οποίο αλλάζει αν αυτή επιτύχει. Όταν η εφαρμογή ανοίγει το αρχείο, το τρέχον σημείο του βρίσκεται στην αρχή του.

Εκτός από την έμμεση μετακίνηση μέσα στο αρχείο με αναγνώσεις και εγγραφές, μία εφαρμογή μπορεί να θέσει το τρέχον σημείο απευθείας με την κλήση SetFilePointer.

```
DWORD SetFilePointer(
  HANDLE hFile,
  LONG lDistanceToMove,
  PLONG lpDistanceToMoveHigh,
  DWORD dwMoveMethod
);
```

Στην πρώτη παράμετρο δίνουμε το handle του αρχείου και στη δεύτερη το πόσο θα μετακινηθεί η τρέχουσα θέση. Η τρίτη παράμετρος μας δίνει τη δυνατότητα να ορίσουμε την απόσταση μετακίνησης σαν μία τιμή των 64 bits και δέχεται τα υψηλότερης τάξης 32 bit της νέας τρέχουσας θέσης. Στην παράμετρο dwMoveMethod δίνουμε το σημείο από το οποίο θα γίνει η μετακίνηση: με FILE_BEGIN μετριέται από την αρχή του αρχείου, με FILE_CURRENT μετριέται από την τρέχουσα θέση και με FILE_END μετριέται από το τέλος του αρχείου.

Αν η κλήση επιτύχει, επιστρέφει τη νέα τιμή της τρέχουσας θέσης (τα «κάτω» 32 bit, ενώ τα υπόλοιπα βρίσκονται στην *lpDistanceToMoveHigh). Αν αποτύχει επιστρέφει 0xFFFFFFFF.

Μία κλήση της SetFilePointer θα επηρεάσει όλα τα νήματα μίας διεργασίας ή και όλες τις διεργασίες που χρησιμοποιούν το ίδιο handle, ή ακόμα και νέα handles που έχουν δημιουργηθεί με την DuplicateHandle. Αν θέλουμε πολλά νήματα να διαβάζουν και να γράφουν από διαφορετικά σημεία του ίδιου αρχείου, είτε πρέπει να χρησιμοποιήσουμε αμοιβαίο αποκλεισμό (με πτώση στην απόδοση) ή κάθε νήμα να ανοίγει το αρχείο ξεχωριστά.

Κλείδωμα αρχείου

Για να έχουμε αποκλειστική πρόσβαση σε ένα τμήμα ενός αρχείου, το οποίο μοιράζονται πολλές διεργασίες, μπορούμε να χρησιμοποιήσουμε τις κλήσεις συστήματος LockFile και LockFileEx, UnlockFile και UnlockFileEx.

```
BOOL LockFile(
  HANDLE hFile,
  DWORD dwFileOffsetLow,
  DWORD dwFileOffsetHigh,
  DWORD nNumberOfBytesToLockLow,
  DWORD nNumberOfBytesToLockHigh
);
```

Η παράμετρος hFile είναι το handle του αρχείου το οποίο πρέπει να έχει ανοιχθεί με ένα τουλάχιστον από τα δικαιώματα GENERIC_READ και GENERIC_WRITE. Οι τιμές dwFileOffsetLow και dwFileOffsetHigh συνδυάζονται για να δώσουν μία τιμή 64 bits, την αρχική διεύθυνση της περιοχής που θα κλειδωθεί. Οι τιμές nNumberOfBytesToLockLow και nNumberOfBytesToLockHigh συνδυάζονται για να δώσουν μία τιμή 64 bits, το πλήθος των bytes που θα κλειδωθούν. Αν η κλήση επιτύχει, επιστρέφει μη μηδενική τιμή, αλλιώς επιστρέφει 0.

Αν η περιοχή που θέλουμε να κλειδώσουμε (ή τμήμα της) είναι ήδη κλειδωμένη, η LockFile επιστρέφει αμέσως την τιμή 0, δεν κολλά δηλαδή.

Για να ξεκλειδώσουμε την περιοχή που έχουμε κλειδώσει με LockFile χρησιμοποιούμε την UnlockFile.

```
BOOL UnlockFile(
  HANDLE hFile,
  DWORD dwFileOffsetLow,
  DWORD dwFileOffsetHigh,
  DWORD nNumberOfBytesToLockLow,
  DWORD nNumberOfBytesToLockHigh
);
```

Οι παράμετροι της UnlockFile είναι ταυτόσημες με της LockFile και πρέπει να παρέχουν ακριβώς τις ίδιες τιμές που δόθηκαν κατά το κλείδωμα.

Η LockFileEx μπορεί να περιμένει μία περιοχή του αρχείου να απελευθερωθεί αν είναι κλειδωμένη:

```
BOOL LockFileEx(
  HANDLE hFile,
  DWORD dwFlags,
  DWORD dwReserved,
  DWORD nNumberOfBytesToLockLow,
  DWORD nNumberOfBytesToLockHigh,
  LPOVERLAPPED lpOverlapped
);
```

Η παράμετρος hFlags μπορεί να πάρει τουλάχιστον μία από τις τιμές LOCKFILE_FAIL_IMMEDIATELY (επιστρέφει αμέσως αν δεν μπορεί να κλειδωθεί η περιοχή) και LOCKFILE_EXCLUSIVE_LOCK (το κλείδωμα είναι αποκλειστικό). Αν καμία από τις δύο δεν δοθεί (dwFlags == 0), η κλήση κολλά μέχρι να επιτύχει το κλείδωμα, το οποίο είναι μοιραζόμενο (μπορούν να γίνουν στην ίδια περιοχή άλλα μοιραζόμενα κλειδώματα αλλά όχι αποκλειστικά).

Η παράμετρος dwReserved παίρνει την τιμή 0. Το πλήθος των bytes που θα κλειδωθούν είναι ο 64 bit συνδυασμός των nNumberOfBytesToLockLow και nNumberOfBytesToLockHigh. Η αρχική διεύθυνση από όπου αρχίζει το κλείδωμα δίνεται από το συνδυασμό των τιμών lpOverlapped->Offset και lpOverlapped->OffsetHigh · η παράμετρος lpOverlapped δεν μπορεί να πάρει την τιμή NULL.

Η τιμή επιστροφής της κλήσης δηλώνει αν αυτή πέτυχε.

Η περιοχή του αρχείου που κλειδώθηκε με την LockFileEx ξεκλειδώνεται με την κλήση UnlockFileEx.

```
BOOL UnlockFileEx(
  HANDLE hFile,
  DWORD dwReserved,
  DWORD nNumberOfBytesToLockLow,
  DWORD nNumberOfBytesToLockHigh,
  LPOVERLAPPED lpOverlapped
);
```

Οι παράμετροι και η τιμή επιστροφής είναι ίδιες με τις αντίστοιχες της LockFileEx.

Το πρόγραμμα που ακολουθεί αποτελείται από 5 νήματα, κάθε ένα από τα οποία κλειδώνει και ξεκλειδώνει το ίδιο τμήμα ενός αρχείου, άλλοτε αποκλειστικά και άλλοτε μοιραζόμενα.

```
/* Make with cl /MT files3.c */

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

int nReps = 0;

DWORD MyRoutine(DWORD nThread)
{
```

```
  HANDLE hFile;
  int I;

  hFile = CreateFile("test.locking", GENERIC_READ|GENERIC_WRITE,
                     FILE_SHARE_READ|FILE_SHARE_WRITE, NULL,
                     OPEN_EXISTING, 0, NULL);

  if (INVALID_HANDLE_VALUE == hFile) {
     fprintf(stderr, "Cannot open file test.locking. Error %ld\n", GetLastError
());
    exit(1);
  }

  for (I=0; I<nReps; I++) {
    OVERLAPPED ov;
    ZeroMemory(&ov, sizeof(ov));
    ov.Offset = 0;
    ov.OffsetHigh = 0;

    if (GetTickCount() % 2) {
      /* Shared ("read") lock */

      LockFileEx(hFile, 0,
                 0, 4, 0,
                 &ov);

      printf("Thread %ld created a read lock.\n", nThread);
    }
    else {
      /* Exclusive ("write") lock */

      LockFileEx(hFile, LOCKFILE_EXCLUSIVE_LOCK,
                 0, 4, 0,
                 &ov);

      printf("Thread %ld created a write lock.\n", nThread);
    }

    Sleep(201);

    UnlockFileEx(hFile, 0, 4, 0, &ov);
    printf("Thread %ld unlocked the file.\n", nThread);
  }

  CloseHandle(hFile);

  return 0;
}

main(int argc, char *argv[])
{
  HANDLE hThreads[4];
  DWORD dwThreadId;
  int I;

  if (1 <argc)
    nReps = atoi(argv[1]);
  if (0 == nReps)
    nReps = 3;

  for (I=0; I<4; I++) {
    hThreads[I] = CreateThread(NULL, 0,
                               (LPTHREAD_START_ROUTINE)MyRoutine,
                               (LPVOID)I, 0, &dwThreadId);
    if (NULL == hThreads[I]) {
        fprintf(stderr, "Cannot create thread %d. Error %ld\n", I, GetLastError
());
```

```
      exit(2);
    }
  }

  MyRoutine(4);

  WaitForMultipleObjects(4, hThreads, TRUE, INFINITE);

  for (I=0; I<4; I++)
    CloseHandle(hThreads[I]);

  return 0;
}
```

Με την εκτέλεσή του θα δούμε κάτι σαν αυτό:

```
Thread 4 created a write lock.
Thread 4 unlocked the file.
Thread 0 created a write lock.
Thread 0 unlocked the file.
Thread 1 created a write lock.
Thread 1 unlocked the file.
Thread 2 created a write lock.
Thread 2 unlocked the file.
Thread 3 created a write lock.
Thread 3 unlocked the file.
Thread 4 created a write lock.
Thread 4 unlocked the file.
Thread 0 created a read lock.
Thread 1 created a read lock.
Thread 2 created a read lock.
Thread 0 unlocked the file.
Thread 1 unlocked the file.
Thread 2 unlocked the file.
Thread 3 created a write lock.
Thread 3 unlocked the file.
Thread 4 created a write lock.
Thread 4 unlocked the file.
Thread 0 created a write lock.
Thread 0 unlocked the file.
Thread 1 created a write lock.
Thread 1 unlocked the file.
Thread 2 created a write lock.
Thread 2 unlocked the file.
Thread 3 created a read lock.
Thread 3 unlocked the file.
```

Περιεχόμενα

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
& ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Εισαγωγή στον πολυπρογραμματισμό σε περιβάλλον Win32

ΔΕΚΕΜΒΡΙΟΣ 2004

Γ. ΠΑΠΑΚΩΝΣΤΑΝΤΙΝΟΥ

Π. ΤΣΑΝΑΚΑΣ