7. API Unix vs API Windows

Contents

7.1. Deosebiri formale Unix - Windows

Principalele deosebiri formale între SO Unix şi SO Windows sunt prezentate în tabelul următor:

		Unix	Windows
1	Specificare absoluta fisier	`/dir1/dir2/.../dirn/fisier`	`d:\dir1\dir2\...\dirn\fisier`
2	Separator directoare PATH	`dir1:dir2:...:dirn`	`dir1;dir2;...;dirn`
3	Specificare optiune	`com -opt`	`com /opt`
4	Separtor linii in fisier text (Mac OS linie\rlinie CR)	`linie\nlinie (LF = 0A)`	`linie\r\nlinie (CR LF = 0D 0A)`
5	Parametrii linie comanda: com arg1 arg2 ... argn	`$0 $1 ... $9`	`%0 %1 ... %9`
6	Valoarea unei variabile shell	`${nume}`	`%nume%`

7.2. Fişiere de comenzi de tip bat (MS-Windows)

Detalii in cap. 6, Boian Fl. s.a. Sisteme de operare, Risoprint, 2006

Fisierele de comenzi DOS / Windows sunt, intr-o oarecare masura, similare fisierelor de comenzi Shell de sub Unix. Ele sunt mult mai sarace in facilitati si directive decat cele oferite de sistemele Shell. Un fisier de comenzi DOS / Windows contine in interiorul lui comenzi DOS si un numar limitat de directive. Din punct de vedere formal, numele unui fisier de comenzi trebuie sa se termine (sa fie de tipul / extensia) .bat

Continutul unui fisier de comenzi bat:

- comenzi DOS;
- etichete (nume la inceput de linie precedat de :);
- caracterele speciale | > < @ % ;
- parametri formali (%n);
- variabile globale (%nume%);
- variabile locale (numai in FOR);
- directive.

Directivele principale:

`FOR %%variabilalocala IN ( multime ) DO comanda` **variabilalocala** parcurge **multime** si pentru fiecare valoare executa **comanda**

`CALL fisiercomenzi [ parametri ]` fişierul de comenzi cheama **fisierdecomenzi** şi după execuţie revine.

`IF [NOT] ERRORLEVEL n comanda` daca codul de retur al comenzii precedente este mai mare sau egal decât **n** (sau strict mai mic în cazul NOT), atunci se executa **comanda**.

`IF [NOT] sir1 == sir2 comanda` dacă cele două şiruri sunt egale (sau diferite in cazul NOT), atunci se executa **comanda**.

`IF [NOT] EXIST fisier comanda` dacă **fisier** există (sau nu există în cazul NOT), atunci se executa **comanda**.

`GOTO eticheta` urmatoarea linie de executat va fi cea marcata cu **eticheta**.

`SHIFT` mută spre stânga cu o poziţie argumentele liniei de comandă: %0 se pierde, %1 devine %0, %2 devine %1 s.a.m.d.

`SET nume=valoare` defineste variabila de mediu **nume** careia îi atribuie **valoare**; utilizarea ei (obtinerea valorii) se face prin %**nume**%

`ECHO [ ON | OFF | mesaj ]` permite sau interzice afisarea la executie a liniilor fisierului de comenzi, sau afiseaza **mesaj** pe iesirea standard.

`PAUSE [ mesaj ]` afişează **mesaj** pe iesirea standard şi asteapta apăsarea unei taste.

`REM comentariu` definirea unei linii comentariu

Comenzi DOS mai des folosite:

- de lucru cu discul: `diskcopy, sys, format, fdisk, chkdsk`
- de lucru cu directoare: `mkdir(md), chdir(cd), rmdir(rd), dir, path, subst`
- de lucru cu fisiere: `more, attrib, del, erase, deltree, fc, find, move, rename, sort, xcopy, copy, type, print`
- eticheta de volum: `label, vol`
- alte comenzi: `choice, edit, keyb, mode, cls, date, time, ver, echo, rem`

In unele situatii, directivele bat si comenzile DOS nu sunt suficiente pentru rezolvarea unor probleme cu fisiere bat. Din aceasta cauză, utilizatorul se vede nevoit să mai scrie o serie de mici programe (de exemplu în C sau C++), care să se termine cu diverse coduri de retur şi să fie integrate în fisierele de comenzi.

7.3. Exemplu de fişier bat: concatenare fişiere

Sa se scrie un fisier de comenzi care primeste cel puţin doi parametrii: primul este numele, eventual specificat absolut, al unui fişier text în care se concatenează fişierele ale căror nume urmează în lista de parametrii. Pentru rezolvare vom reţine in variabila **dest** primul parametru. Apoi facem un SHIFT pentru a trece la următorul parametru. In continuare, într-o structură repetitivă vom parcurgem restul parametrilor. Pentru fiecare testam existenta fişierului şi în caz afirmativ il vom adăuga continutul la **dest**.

Sursa, in fisierul **concat.bat** este:

```
@echo off
REM verificam numarul de parametrii
if "%2"=="" goto err1
set dest=%1
shift
REM parcurgem ceilalti parametrii din linia de comanda
:loop
if "%1"=="" goto end
REM verificam daca exista %1 si in caz afirmativ il concatenam
if EXIST %1 type %1 >>%dest%
shift
goto loop

:err1
echo Trebuie minim doi parametrii!
goto end
:end
REM afisam continutul fisierului toate.txt
type %dest%
```

Apelul se face:

```
- - ->concat.bat dir1\toate.txt a.txt b.txt c ex1.pas tpc.cpp
```

7.4. Particularităţi C / C++ pentru MS-Windows

Limbajul nativ de dezvoltare a aplicaţiilor Windows este C++. Din această cauză, în cele ce urmează vom descrie principiile programării folosind construcţii C şi C++. Headerul `<windows.h>` conţine principalele construcţii de limbaj folosite în interfaţa Windows.

Constante. In `<windows.h>` se definesc o serie de constante. Numele acestora este compus din două părţi: o primă parte indică grupul din care face parte constanta, apoi caracterul "_" şi în final numele specific, de regulă suficient de lung încât să sugereze ce reprezintă. Pentru detalii se poate consulta MSDN.

Tipuri de date. Windows foloseşte o serie de tipuri de date prin care s-a urmărit creşterea portabilităţii aplicaţiilor în cazul unor noi arhitecturi de calculatoare. Astfel, avem tipurile BOOL, BYTE, DWORD

(32 biţi), FARPROC (pointer spre funcţie), LPSTR (pointer către string), LPMSG (pointer către o structură MSG) etc.

Pentru desemnarea obiectelor sunt definite nişte tipuri de date speciale: *descriptor* sau *handle*. Acestea sunt întregi pe 16 biţi prin intermediul cărora se pot referi obiecte: fişiere, procese, threaduri, evenimente, timere etc.
Nume de variabile. Atribuirea de nume pentru variabile se face respectându-se anumite convenţii, provenite din experienţa programatorilor. Este vorba de notaţia ungară de denumire a variabilelor. Ele nu sunt restricţii impuse de sistem, dar este de preferat să fie respectate. De regulă numele atribuite sunt lungi, încep cu literă mică, iar în cadrul numelor apar litere mari la începuturile cuvintelor care le compun. De multe ori, când este vorba de o singură variabilă de un anumit tip, numele ei este numele tipului, scris cu literă mică.

Tot ca şi convenţii, începuturile (prefixele) numelor de variabile au semnificaţie: `b` pentru BOOL, `by` pentru BYTE, `c` pentru char, `dw` pentru DWORD, `fn` pentru funcţie `h` pentru handle, `i` pentru int, `lp` pentru pointer lung, `w` pentru WORD etc.

7.5. Aplicaţii consolă; un filtru

Cele mai simple aplicaţii care se pot scrie sub Windows sunt aplicaţiile consolă. Acestea sunt, în fapt, aplicaţii cu intrare şi ieşire standard în mod text, la fel ca şi la programele simple sub Unix. Spre exemplu, un program extrem de simplu este un filtru. Acesta citeşte linie cu linie de la intrarea standard şi dă la ieşire aceleaşi linii, scurtate la primele 10 caractere. Sursa `Filtru.cpp` este este prezentată în continuare.

```
#include <stdio.h>
#include <string.h>
   int main(int c, char* a[]) {
      char l[128];
      for (;;) {
         if (gets(l)==NULL)
            break;
         if (strlen(l) > 10)
            l[10]= 0;
         printf("%s\r\n", l);
      }
      return 0;
   }
```

Lansarea unui astfel de filtru se face, de asemenea, dintr-o fereastră `Cmd`, putându-se, la fel ca în Unix sau Dos, să se redirecteze intrarea şi ieşirea lui standard, astfel:

```
Filtru.exe  <FisierIntrare  >FisierIesire
```

7.6. Acces la fişiere şi aşteptarea unor evenimente

Tabelul care urmează prezintă numele unor funcţii de lucru cu fişiere sub Windows. Pentru mai multe informatii legate de prototipurile acestor funcţii, se recomandă consultarea documentaţiei MSDN.

Nume funcţie	**Rol**
`CreateFile`	Crearea unui fişier cu anumite atribute
`OpenFile`	Deschide un fişier deja creat
`WriteFile`	Scrie binar într-un fişier

`ReadFile`	Citeşte binar dintr-un fişier
`CloseHandle`	Inchide fişier, eveniment, timer etc.

Win32 API oferă un set de funcţii de aşteptare pentru a permite unui program să îşi suspende temporar execuţia în aşteptarea unui eveniment. Funcţiile de aşteptare blochează execuţia programului până când criteriul specificat a fost îndeplinit. Tipul funcţiei de aşteptare determină criteriul utilizat. În timpul aşteptării procesul consumă foarte puţine resurse sistem, fiind vorba de o aşteptare pasivă – intrare în sleep. Tabelul de mai jos prezintă pe scurt rolurile principalelor funcţii de aşteptare.

Funcţie wait	Descriere
`WaitForSingleObject()`	Aşteaptă după un anumit obiect ca acesta să ajungă în starea setat (de exemplu terminarea unui proces sau valoarea pozitiva a unui semafor).
`WaitForSingleObjectEx()`	Ca şi precedentul, plus aşteptarea a altor două evenimente: terminarea unei operaţii de intrare ieşire, sau sosirea unui apel asincron în threadul curent.
`WaitForMultipleObjects()`	Aşteaptă după o mulţime de obiecte. Ieşirea din aşteptare se poate face fie când unul dintre obiecte este setat, fie când toate obiectele ajung în starea setat.
`WaitForMultipleObjectsEx()`	Ca şi precedentul, plus aşteptarea celor două evenimente specificate în cazul funcţiei WaitForSingleObjectEx.

Cea mai simplă dintre aceste funcţii şi cea mai des utilizată este `WaitForSingleObject`.

7.7. Procese Windows

Crearea unui proces în Windows se face prin apelul funcţiei `CreateProcess` dintr-un alt proces. Funcţia are următorul prototip:

```
BOOL CreateProcess (LPCTSTR               lpszImageName,
                    LPCTSTR               lpszCommandLine,
                    LPSECURITY_ATTRIBUTES lpsaProcess,
                    LPSECURITY_ATTRIBUTES lpsaThread,
                    BOOL                  fInheritHandles,
                    DWORD                 fdwCreate,
                    LPVOID                lpvEnvironment,
                    LPTSTR                lpszCurDir,
                    LPSTARTUPINFO         lpsiStartInfo,
                    LPPROCESS_INFORMATION lppiProcInfo);
```

Atunci când se apelează funcţia `CreateProcess`, sistemul creează un spaţiu de adresare şi încarcă noul proces în acest spaţiu. După această operaţie, sistemul creează threadul primar pentru noul proces şi-l lansează în execuţie. Să vedem semnificaţia parametrilor funcţiei `CreateProcess`:

Pentru semnificaţia parametrilor se poate consulta MSDN.

Terminarea unui proces. Un proces poate fi terminat pe două căi: apelând din interior funcţia `ExitProcess` sau apelând din exterior funcţia `TerminateProcess`. Este preferabilă prima cale, cea de-a doua trebuie folosită doar pentru situaţii extreme. Prototipul celor două funcţii sunt:

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

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

Pentru detalii se poate consulta MSDN.

7.8. Exemplu: rezumatul directorului curent

Pentru utilizare sub Windows, vom folosi ca intermediar un fisier de comenzi `ls.bat`, care contine o singura linie:

```
dir %1
```

Sursa `execWin.cpp` a programului este:

```
#include <windows.h>
#include <stdio.h>
int main() {
    STARTUPINFO si = { sizeof(si) };
    PROCESS_INFORMATION pi;
//  ls.bat contine linia: dir %1
    printf("Procesul parinte %d va creea un fiu\n", GetCurrentProcessId());
//  Rulati alternativ cu una dintre urmatoarele doua linii comentata:
    BOOL b = CreateProcess("ls.bat", NULL, NULL, NULL,
//    BOOL b = CreateProcess("ls.bat", "ls.bat *.cpp", NULL, NULL,
                           FALSE, 0, NULL, NULL, &si, &pi);
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Terminat procesul fiu %d creeat de parintele %d\n",
    pi.dwProcessId, GetCurrentProcessId());
    return 0;
}
```

7.9. Exemplu: câte perechi de numere nenule au suma un număr par?

Problema, rezolvată şi sub Unix, este trivial de simplă, dar potrivită pentru a exemplifica utilizarea `CreateProcess`, `WaitForSingleObject` si `ExitProcess`.

Enuntul problemei: Se dau la linia de comanda n perechi de numere intregi. Programul va crea **n** procese fii, fiecare primind doua argumente consecutive din linia de comanda. Oricare dintre fii intoarce codul de retur:

- 0 daca perechea are suma para,
- 1 daca suma este impara,
- 2 daca unul dintre argumente este nul sau nenumeric.

Părintele aşteaptă terminarea fiilor şi va afişa rezultatul. In continuare vom implementa un program separat pentru procesul fiu. Sursa lui, `paritateFiu.cpp` este:

```
#include <stdio.h>
#include <windows.h>
main(int argc, char* argv[]) {
    int n1, n2;
    n1 = atoi(argv[1]);   // atoi intoarce 0
    n2 = atoi(argv[2]); // si la nenumeric
    if (n1 == 0 || n2 == 0) ExitProcess(2);
    if ((n1 + n2) % 2 == 0) ExitProcess(0);
    else                    ExitProcess(1);
```

```
}
```

Acesta va fi compilat cu:

```
gcc -o paritateFiu paritateFiu.cpp
```

Sursa `paritate.cpp` este:

```
#include <stdio.h>
#include <windows.h>
main(int argc, char* argv[]) {
    STARTUPINFO si = { sizeof(si) };
    PROCESS_INFORMATION pi[100];
    char linieCom[1000];
    int pare = 0, impare = 0, nenum = 0, i;
    DWORD n1;
    for (i = 1; i < argc-1; i += 2) {
        strcpy(linieCom, "paritateFiu.exe ");
        strcat(linieCom, argv[i]);
        strcat(linieCom, " ");
        strcat(linieCom, argv[i+1]);
        BOOL b = CreateProcess("paritateFiu.exe", linieCom, NULL, NULL,
                        FALSE, 0, NULL, NULL, &si, &pi[i]);
    }
    // Parintele asteapta terminarile fiilor
    for (i = 1; i < argc-1; i += 2) {
        WaitForSingleObject(pi[i].hProcess, INFINITE);
        GetExitCodeThread(pi[i].hThread, &n1);
        switch (n1) {
            case 0: pare++;break;
            case 1: impare++;break;
            default: nenum++;
        }
    }
    printf("Pare %d, impare %d, nenumerice %d\n",pare, impare, nenum);
}
```

7.10. Exemplu: capitalizarea mai multor fişiere text

Dorim să transformăm un fişier text într-un alt fisier text, cu acelaşi conţinut, dar în care toate cuvintele din el sa înceapa cu literă mare. Un astfel de program va fi apelat:

```
capitalizare fisierintrare fisieriesire
```

Ne propunem sa prelucram simultan mai multe astfel de fisiere. De aceea vom creea un proces master, care primeste la linia de comanda numele fisierelor al caror continut va fi capitalizat:

```
master fisier1 fisier2 - - - fisiern
```

Rezultatul va consta din fisierele:
`fisier1.CAPIT, fisier2.CAPIT, - - - fisiern.CAPIT`

Procesul master va crea **n** procese fii, iar fiecare fiu i va lansa prin `CreateProcess` programul:

```
capitalizare fisi fisi.CAPIT
```

Sursa `capitalizare.cpp` este:

```
#include <stdio.h>
#include <windows.h>
#include<ctype.h>
#define MAXLINIE 100
main(int argc, char* argv[]) {
    FILE *fi, *fo;
    char linie[MAXLINIE], *p;
    fi = fopen(argv[1], "r");
    fo = fopen(argv[2], "w");
    if (fi == NULL && fo == NULL) ExitProcess(1);
    for ( ; ; ) {
        p = fgets(linie, MAXLINIE, fi);
        linie[MAXLINIE-1] = '\0';
        if (p == NULL) break;
        if (strlen(linie) == 0) continue;
        linie[0] = toupper(linie[0]);
        for (p = linie; ; ) {
            p = strstr(p, " ");
            if (p == NULL) break;
            p++;
            if (*p == '\n') break;
            *p = toupper(*p);
        }
        fprintf(fo, "%s", linie);
    }
    fclose(fo);
    fclose(fi);
}
```

Programul primeste la linia de comanda numele celor doua fisiere. Se deschid aceste fisiere si se citeste fisierul de intrare linie cu linie. Cu ajutorul pointerului p, se parcurge linia curenta si se cauta pe rand cate un spatiu, dar care sa nu fie ultimul caracter din linie. Urmatorul caracter este apoi transformat in litera mare (`toupper` face aceasta transformare numai daca caracterul este efectiv o litera mica).

Sursa `master.cpp` este:

```
#include <stdio.h>
#include <windows.h>
main(int argc, char* argv[]) {
    STARTUPINFO si = { sizeof(si) };
    PROCESS_INFORMATION pi[100];
    int i;
    char nume[200];
    for (i=1; argv[i]; i++) {
        strcpy(nume, "capitalizare ");
        strcat(nume, argv[i]);
        strcat(nume, " ");
        strcat(nume, argv[i]);
        strcat(nume, ".CAPIT"); // fabricat numele iesirii
        // incarcat programul de capitalizare
        BOOL b = CreateProcess("capitalizare.exe", nume, NULL, NULL,
                               FALSE, 0, NULL, NULL, &si, &pi[i]);
    }
    printf("Lansat simultan %d procese de capitalizare\n",i-1);
}
```

Se parcurg argumentele liniei de comanda si pentru fiecare dintre ele se creeaza un proces fiu. In tabloul nume se construieste numele fisierului de iesire. Apoi se incarca programul capitalizare cu cele doua nume de fisiere date "la linia de comanda".

Cele doua programe se compileaza:

```
gcc -o capitalizare capitalizare.c
gcc -o master master.c
```

Lansarea se face:

```
master fis1 fis2 - - - fisn
```

7.11. Pipe sub Windows

În Windows, ca şi în Unix, există două posibilităţi de a folosi pipe în IPC. O primă variantă este *pipe anonime,* care se pot folosi numai pentru comunicarea între procese de pe aceeaşi maşină. A doua variantă este *pipe cu nume,* folosite pentru comunicarea între procese ce operează nu neapărat pe aceeaşi maşină Windows.

Un **pipe anonim** poate fi folosit, ca şi pipe-ul de sub Unix, pentru comunicarea între procese descendente din creatorul pipe-ului. In urma creării, procesul creator obţine doi descriptori - handle - unul de citire şi altul de scriere. Procesul creator poate trimite fiilor (nepoţilor etc.) handle-urile pipe-ului, în momentul creării proceselor fii prin apeluri ale funcţiei `CreateProces`. Pentru ca fiul să moştenească handle-ul la pipe, părintele trebuie să seteze parametrul `fInheritedHandle` din apelul `CreateProces`, la valoarea TRUE.

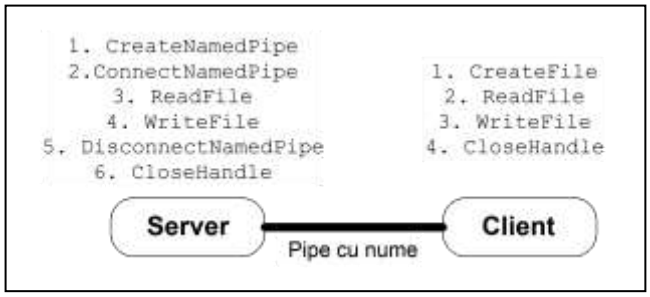
Un pipe fără nume se creează:

```
BOOL CreatePipe (PHANDLE phRead,PHANDLE phWrite,
       LPSECURITY_ATTRIBUTES lpsa, DWORD cbPipe);
```

Funcţia întoarce TRUE în caz de succes sau FALSE la eşec.

Pipe cu nume este un mecanism de comunicare între două sisteme diferite, ambele fiind operaţionale pe platforme Windows

In figura următoare sunt prezentate succesiunile apelurilor sistem, atât pentru server, cât şi pentru client. Cititorul poate uşor observa particularizările necesare pentru comunicarea prin pipe anonim.



Crearea unui pipe cu nume se face prin apelul sistem `CreateNamedPipe`, cu prototipul:

```
HANDLE CreateNamedPipe(LPSTR numePipe,
                       DWORD  optiuniModOpen,
                       DWORD  optiuniModPipe,
                       DWORD  nMaxInstances,
                       DWORD  lungBufOut,
                       DWORD  lungBufIn,
                       DWORD  timeOut,
                       LPSECURITY_ATTRIBUTES  lpsa)
```

`numePipe` este un string prin care se indică numele pipe-ului. Convenţiile Microsoft de specificare a acestor nume impun două sintaxe, una pentru pipe local şi alta pentru pipe de pe o altă maşină. Aceste specificări sunt:

```
\\.\PIPE\numePipePeMAsina
\\adresaMasina\PIPE\numePipePeMasina
```

Pentru alte detalii, vezi MSDN.

După crearea unui pipe cu nume, serverul apelează:

```
ConnectNamedPipe(HANDLE hNamedPipe, LPOVERLAPPED lpo)
```

Primul handle este cel întors de crearea pipe. Al doilea parametru, de regulă NULL, indică faptul că se aşteaptă la conectare până când un client se conectează efectiv la pipe. (A se compara această regulă cu cea similară de la FIFO de sub Unix).

La fel ca şi la pipe anonime, se folosesc apelurile `ReadFile` şi `WriteFile` pentru schimbul cu pipe.

Serverul îşi încheie activitatea apelând:

```
DisconnectNamedPipe(HANDLE hNamedPipe);
CloseHandle (HANDLE hNamedPipe);
```

Pentru client, conectarea la un pipe cu nume presupune un apel sistem `CreateFile`:

In cazul creării unui pipe cu nume, `numeFisier` reprezintă numele pipe-ului, cu sintaxa specificată mai sus, la apelul `CreateNamedPipe`.

7.12. Threaduri MS-Windows; generalităţi

Tabelul de mai jos prezintă comparativ, tipurile de date, variabilele şi principalele funcţii care operează cu threaduri:

API elems. \OS	**Linux**	**MS Windows**

Headers	`#include<stdio.h>`
	`#include<pthread.h>`
	`#include<stdlib.h>`
	`#include <semaphore.h>`
		`#include <windows.h>`
		`#include <stdlib.h>`
		`#include <stdio.h>`
		`#include <math.h>`

Libraries	`-pthread -lm`	

Data Types	`pthread_t`
	`pthread_mutex_t`
	`pthread_cond_t`
	`pthread_rwlock_t`
	`sem_t`
		`HANDLE`
		`CRITICAL_SECTION`
		`CONDITION_VARIABLE`
		`SRWLOCK`
		`HANDLE`

Threads	`pthread_create` `pthread_join`	`CreateThread` `WaitForSingleObject`
Function Decl	`void* worker(void* a)`	`DWORD WINAPI worker(LPVOID a)`

Mutexes	pthread_mutex_init pthread_mutex_lock pthread_mutex_unlock pthread_mutex_destroy	InitializeCriticalSection EnterCriticalSection LeaveCriticalSection DeleteCriticalSection
Conditional Variables	pthread_cond_init pthread_cond_wait pthread_cond_signal pthread_cond_destroy	InitializeConditionVariable SleepConditionVariableCS WakeConditionVariable !Trebuie compilate cu Visual Studio incepand cu Vista, Windows 7 si mai recente!
Read/Write Locks	pthread_rwlock_init pthread_rwlock_wrlock pthread_rwlock_rdlock pthread_rwlock_unlock pthread_rwlock_destroy	InitializeSRWLock AcquireSRWLockExclusive AcquireSRWLockShared ReleaseSRWLockExclusive AcquireSRWLockShared !Trebuie compilate cu Visual Studio incepand cu Vista, Windows 7 si mai recente!
Semaphores	sem_init sem_wait sem_post sem_destroy	CreateSemaphore WaitForSingleObject ReleaseSemaphore CloseHandle

Sub Windows NT, *threadul* este cea mai mică entitate executabilă la nivel nucleu. Fiecare proces conţine unul sau mai multe thread-uri. In momentul creării procesului, odată cu el se crează *threadul primar* al acestuia. Threadul primar poate crea la rândul său alte thread-uri cu care va partaja spaţiul de adrese al procesului comun. De asemenea, ele mai partajează şi alte resurse sistem: descriptori de fişiere, etc.

7.13. Operaţii asupra threadurilor: creare, terminare

Prototipul funcţiei de **creare** este:

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

`lpStartAddress` - pointer la funcţia ce dirijează threadul
`lpParameter` - argumentul funcţiei
`lpThreadId` - pointer la identificatorul threadului

La crearea threadului, este generat un descriptor care identifică în mod unic threadul în sistem. După creare, se lansează în execuţie funcţia specificată prin parametrul `lpStartAddress`. Această funcţie are parametrii specificaţi prin `lpParameter` şi întoarce o valoare de tip DWORD. Pentru a determina valoarea întoarsă de această funcţie, se poate folosi funcţia `GetExistCodeThread`().

Terminarea unui thread. Un thread îşi încheie execuţia în următoarele condiţii:

- la ieşirea din procedura asociată threadului.
- la apelul funcţiilor `ExitProcess()`, `ExitThread()` apelate din threadul curent.
- dacă se apelează `ExitProcess()` sau `TerminateThread()` din alte procese, cu argument handler-ul threadului care urmează a fi distrus sau din alte thread-uri, folosind, de asemenea, funcţia `TerminateThread()`.

Prototipurile unora dintre funcţiile de terminare a unui threa sunt:

```
void ExitThread(UINT exitcode);
BOOL TerminateThread(HANDLE hThread, DWORD exitcode);
BOOL GetExitCodeThread(HANDLE hThread, LPDWORD exitcode);
```

Parametrul `hThread` identifică threadul care se va termina.

Apelul `ExitThread` provoacă terminarea threadului curent cu întoarcerea codului de ieşire `exitcode`. După apelul funcţiei, stiva asociată threadului este eliberată, iar starea obiectului thread, privit ca eveniment, devine semnalată.

7.14. Instrumente standard de sincronizare

Mecanismele de comunicare şi sincronizare între thread-uri sunt furnizate de interfaţa Win32API, care furnizează primitive de lucru cu *evenimente, semafoare, variabile mutex, secţiuni critice*. Aceste obiecte de sincronizare au: *semnalat* şi *nesemnalat*. Starea semnalat presupune de obicei îndeplinirea unei condiţii şi semnalarea acestui fapt unor thread-uri interesate. Pentru aşteptarea semnalării (starea semnalat) se poate folosi funcţia `WaitForSingleObject`

Variabilele mutexError! Bookmark not defined. permit implementarea accesului exclusiv la o resursă partajată între mai multe thread-uri. Semantica obiectelor de sincronizare mutex este similară cu cea întâlnită la implementarea thread-urilor de pe platformele Unix. ***Crearea*** unei astfel de variabile se face cu funcţia:

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

Ocuparea unei variabile mutex se face prin funcţiile de aşteptare, de exemplu apelul `WaitForSingleObject(g_hMutex, INFINITE)` se va termina doar când starea variabilei mutex indentificată prin handler-ul `g_hMutex` devine semnalată.

Eliberarea unei variabile mutex se realizeaza cu funcţia:
`BOOL ReleaseMutex(HANDLE hMutex);`

Distrugerea unei variabile mutex se face fie prin invocând `CloseHandle()`. Dacă acest apel lipseşte, variabilele mutex sunt eliminate de sistem.

Semafoare fără nume. Pentru sincronizarea thread**Error! Bookmark not defined.**-urilor din cadrul aceluiaşi proces, este de preferată utilizarea semafoarelor fără nume, deoarece nucleul sistemului este scutit de gestiunea lor. Crearea unui astfel de semafor trebuie făcută folosind apelul `CreateSemaphore`, (vezi MSDN). Pentru a fi semafor anonim trebuie ca ultimul parametru, pointer la numele semaforului, să aibă valoarea NULL.

Valoarea semaforului poate fi mărită cu o cantitate pozitivă apelând funcţia `ReleaseSemaphore()` Ea are ca prim argument handle-ul semaforului şi cantitatea cu care se măreşte valoarea.

Aşteptarea la semafor se face folosind funcţiile de aşteptare descrise mai sus.

Secţiuni critice. O variabilă de tip secţiune critică se declară astfel:

```
CRITICAL_SECTION  numeSectiuneCritica;
```

Utilizarea secţiunii critice se face astfel:

```
EnterCriticalSection(&numeSectiuneCritica);
 - - - Corpul sectiunii critice - - -
LeaveCriticalSection(&numeSectiuneCritica);
```

7.15. Exemplu: m trenuri trec pe n linii între A şi B

TrenuriMutexCond Windows

```
#include <windows.h>
#include <winbase.h>
#include <stdlib.h>
#include <stdio.h>
#define N 5
#define M 13
#define SLEEP 4
CRITICAL_SECTION mut, mutcond;
CONDITION_VARIABLE cond; // A fost adaugat la Vista si
da eroare de compilare la windows mai vechi!
int linie[N], tren[M];
DWORD tid[M];
HANDLE doneEv[M];
int liniilibere;


//rutina unui thread
DWORD WINAPI trece(LPVOID tren) {
    int i, s, t, l;
    t = *(int*)tren;
    s = 1 + rand() % SLEEP; // Modificati timpii de
stationare

    EnterCriticalSection(&mutcond);
    for ( ; liniilibere == 0; ) {
        SleepConditionVariableCS(&cond,        &mutcond,
INFINITE);
    }
    LeaveCriticalSection(&mutcond);

    EnterCriticalSection(&mut);
    for (l = 0; l < N; l++)
        if (linie[l] == -1) break;
    linie[l] = t;
    liniilibere--;
    printf("Trenul %d pe linia %d pentru %d secunde.
Trenuri in gara:", t, l, s);
    for (i=0; i< N; i++)
        if (linie[i] != -1)
            printf(" %d",linie[i]);
    printf(".\n");
    fflush(stdout);
    LeaveCriticalSection(&mut);

    Sleep(s); // Modificati timpii de sleep

    EnterCriticalSection(&mut);
    linie[l] = -1;
    liniilibere++;
    LeaveCriticalSection(&mut);

    EnterCriticalSection(&mutcond);
    WakeConditionVariable(&cond);
```

```
    LeaveCriticalSection(&mutcond);
    SetEvent(doneEv[t]);
}


//main
main(int argc, char* argv[]) {
    int i;
    InitializeCriticalSection(&mut);
    InitializeCriticalSection(&mutcond);
    InitializeConditionVariable(&cond);
    liniilibere = N;
    for (i = 0; i < N; linie[i] = -1, i++);
    for (i=0; i < M; tren[i] = i, i++);
    for (i=0; i < M; i++) doneEv[i] = CreateEvent(0,
FALSE, FALSE, 0);

    // ce credeti despre ultimul parametru &i?
    for (i=0; i < M; i++) CreateThread(NULL, 16384,
(LPTHREAD_START_ROUTINE)trece,
                          (LPVOID)&tren[i],          0,
&tid[i]);
    for       (i=0;       i       <       M;       i++)
WaitForSingleObject(doneEv[i],INFINITE);

    DeleteCriticalSection(&mut);
    DeleteCriticalSection(&mutcond);
    DeleteConditionVariable(&cond);
}
```