

# Threaduri în WIN32 API

---

CURS NR. 5

# Procese și threaduri

Threadul descrie o cale de execuție independentă în cadrul procesului

Procesul conține unul sau mai multe threaduri (fire de execuție)

- Procesul este unitatea de grupare a resurselor și container de threaduri
- Threadul este unitatea de execuție

Threadurile din același proces sunt **concurente** și **partajează resursele procesului** (spațiul de adrese, tabela de handle-uri, etc.) dar sunt **planificate spre execuție independent**, deci au nevoie de

- Parte de stivă pentru apeluri sistem, variabile automate, întreruperi
- Variabile de context – regiștrii speciali – Stack Pointer (Instruction Pointer), etc.

La crearea unui nou proces se creează implicit un thread principal în proces – threadul care execută funcția principală (main-ul)

## Analogie

- Procesul este un program în execuție
- Threadul este o funcție în execuție

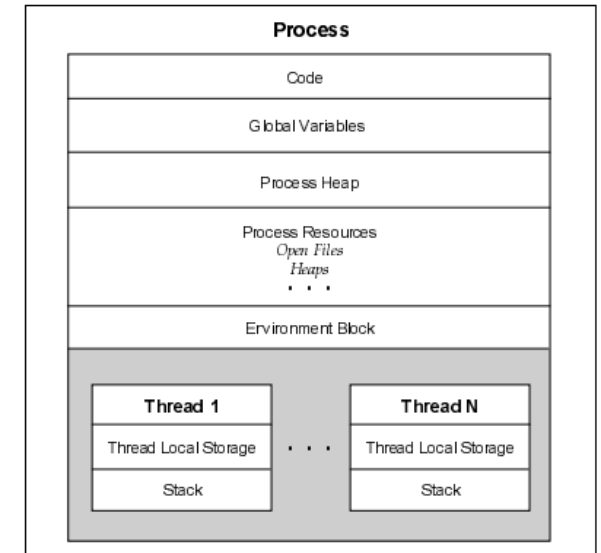


Figure 6-1 A Process and Its Threads

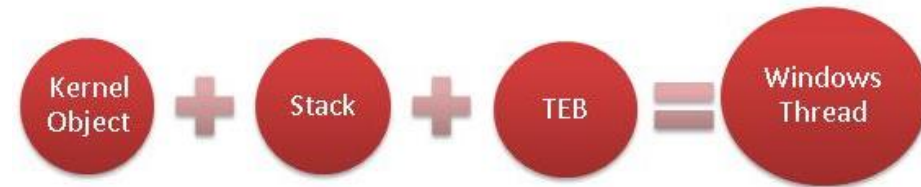
# Procese single-threaded vs. multithreaded

## Procesul constă din două componente

- Obiect kernel al procesului
- Spațiu de adrese (include cod, date, stiva, heap, etc.)
- PEB – Process Environment Block

## Thread-ul constă din două componente

- Obiect kernel al threadului
- Stiva threadului
- TEB – Thread Environment Block



## Relația dintre threaduri

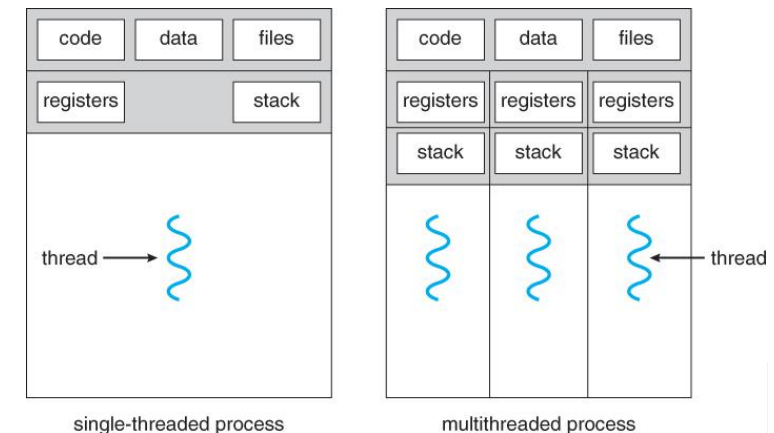
- Threadurile din același proces **sunt egale** – doar threadul principal este mai special – el ține în viață procesul (execută funcția principală)
- Nu există relația părinte – fiu, deși ne referim la aceste concepte pentru a simplifica identificarea threadurilor

## Aplicații multithreaded

- Exemple: Visual Studio compilează automat codul când programatorul nu mai tastează
  - Editor de text – verifică în fundal (background) automat sintaxa și greșelile gramaticale din text
  - Copierea fișierelor în background
  - Browsere de web care comunica cu serverul în fundal

## SO multithreaded

- Exemple și avantaje



# Multithreading

**Provocare:** organizarea și coordonarea firelor de execuție astfel încât să simplifice programul și să beneficieze de paralelismul inherent al programului și al sistemului de calcul

- **AVANTAJE:** permite aplicațiilor să fie scalabile, interactive, eficiente, etc.
  - **Cost mai redus** la crearea și gestiunea threadurilor față de gestiunea proceselor
  - **Comunicare facilă între threaduri**
    - Threadurile concurente ale procesului partajează majoritatea resurselor procesului - datele și codul
    - Au și un spațiu de stocare individual, dar datele unui thread pot fi accesate și modificate de alte threaduri
      - Threadurile nu sunt protejate unele de altele – cum e în cazul proceselor
    - (Procesele au spații de adrese distincte → comunicarea și partajarea resurselor este mai dificilă)
  - **Execuție paralelă** pe sisteme multiprocesor / multicore
- În general se preferă aplicațiile multithreaded – un proces cu mai multe threaduri – pentru eficiență
- **DEZAVANTAJE:** poate introduce o serie de noi probleme – de consistența datelor – rezolvarea cărora necesită mecanisme de sincronizare
  - Threadurile din cadrul aceluiași proces partajează cele mai multe resurse ale procesului
    - Pot modifica date partajate, pot modifica datele altui thread, etc.
  - Accesul concurent la resurse partajate creează probleme de sincronizare (condiție de cursă, interblocare, etc.)
    - Poate degrada performanța

# Kernel și obiecte kernel (să ne reamintim...)

---

Kernelul este componenta centrală – nucleul sistemului de operare

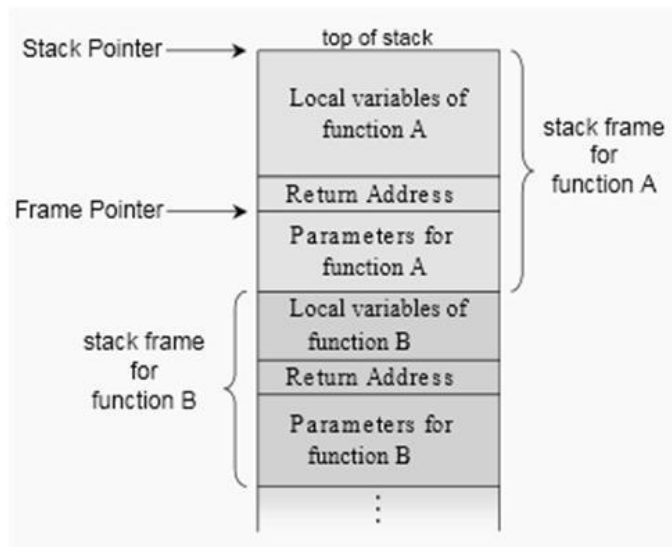
- Furnizează un nivel de abstractizare – gestionează și facilitează interacțiunea aplicațiilor cu resursele hardware ale sistemului. Permite rularea programelor și utilizarea resurselor

## Obiecte kernel

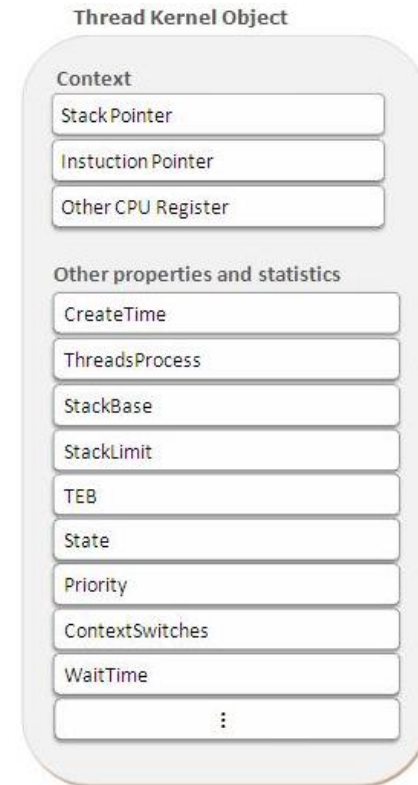
- Pentru gestiunea diverselor resurse sistem – procese, threaduri, fișiere, joburi, evenimente, mutexuri, mapare de fișier, etc.
- Obiect kernel - structură de date alocată, menținută și gestionată de kernel – accesibilă doar kernelului
  - Câmpurile structurii stochează informații despre obiectul kernel
    - Câmpuri generale, ca de ex. attribute de securitate, contor de utilizare, etc.
    - Câmpuri specifice tipului de obiect

# Obiectul kernel de tip thread

- Fiecare thread din sistem are asociat un obiect kernel de tip thread
  - Permite gestiunea și execuția threadului de către sistem
- Conține
  - **Context**: starea regiștrilor procesor la ultima execuție a threadului
    - SP- stack pointer, IP – instruction pointer, etc.
  - Permite schimbarea de context (salvarea informațiilor de context pentru reluarea la un moment ulterior a execuției din aceeași stare de unde s-a rămas)
- **Stiva**: spațiul de memorie unde threadul își stochează parametri, datele locale și adresele de retur



- SO alocă două tipuri de stivă pentru thread
  - Stiva utilizator
    - Pentru variabile locale, argumente, adrese de retur
    - Implicit se alocă 1 MB stivă pentru mod utilizator
  - Stiva kernel
    - Pentru argumente la apelul funcțiilor din kernel
    - Pentru variabile locale și adrese de retur la apelul unor funcții kernel din alte funcții kernel
    - Implicit se alocă 12 KB (pe sisteme de 32 biți) sau 24 KB (pe sisteme de 64 biți) pentru stiva kernel

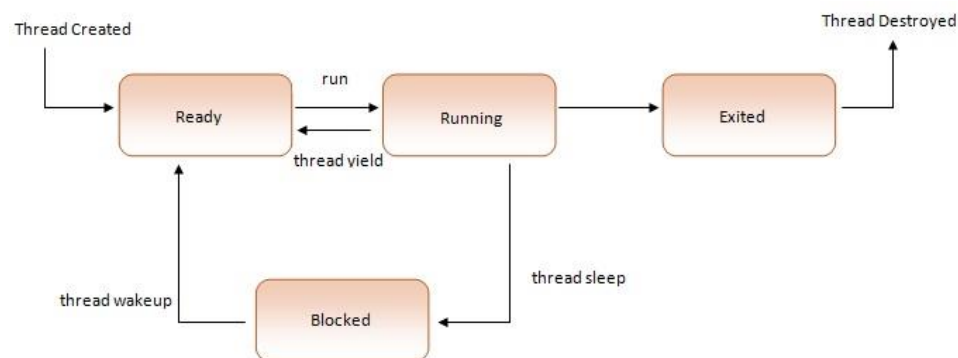


# Stările threadului

Threadul este unitatea de execuție – este o entitate dinamică

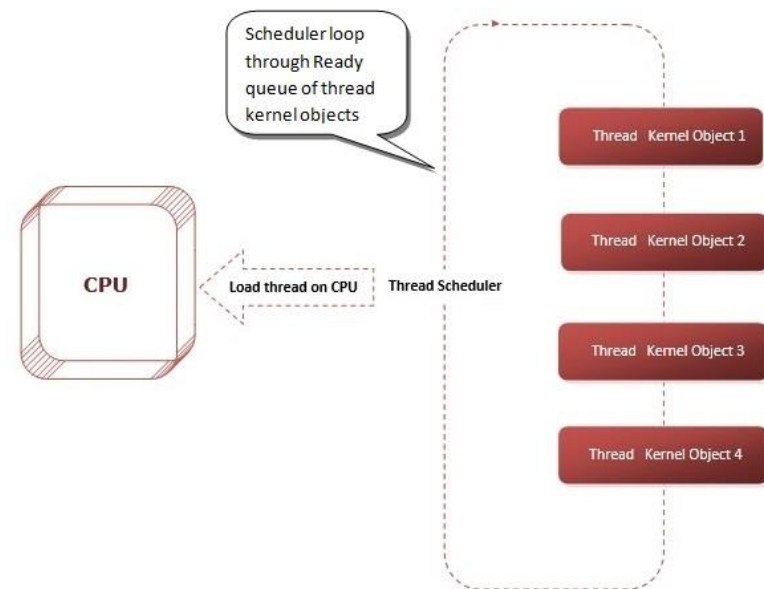
## Stările

- Obiectele kernel ale threadurilor care se află în aceeași stare sunt plasate în cozi – implementate ca liste dublu înlănțuite



## Planificatorul

- Alege următorul thread spre execuție pe baza unei politici de planificare
  - din cele aflate în coada Ready
- Când?
  - La expirarea cuantei de timp sau la recepționarea unei întreruperi
- Schimbare de context
  - Salvează informațiile de context ale threadului care a rulat pe procesor
  - Încarcă în regiștrii procesorului informațiile de context ale threadului ales
  - Lansează în execuție threadul
    - Acesta își continuă execuția de unde a rămas la ultima execuție



# Funcția executată de thread

La crearea unui nou thread se specifică funcția care va fi executată de thread și argumentele acesteia

Funcția are o formă generală – pentru a corespunde scopurilor variate

```
DWORD ThreadFunc(PVOID pvParam) {  
    DWORD dwResult = 0;  
    ///...  
    return(dwResult);  
}
```

## Threadul

- Își începe execuția cu prima instrucțiune din funcția specificată și
- se termină după ultima instrucțiune din funcție
  - Se eliberează partea de stivă
  - Se decrementează contorul de referiri la obiectul thread (la 0 se elimină obiectul din sistem)

## Observații

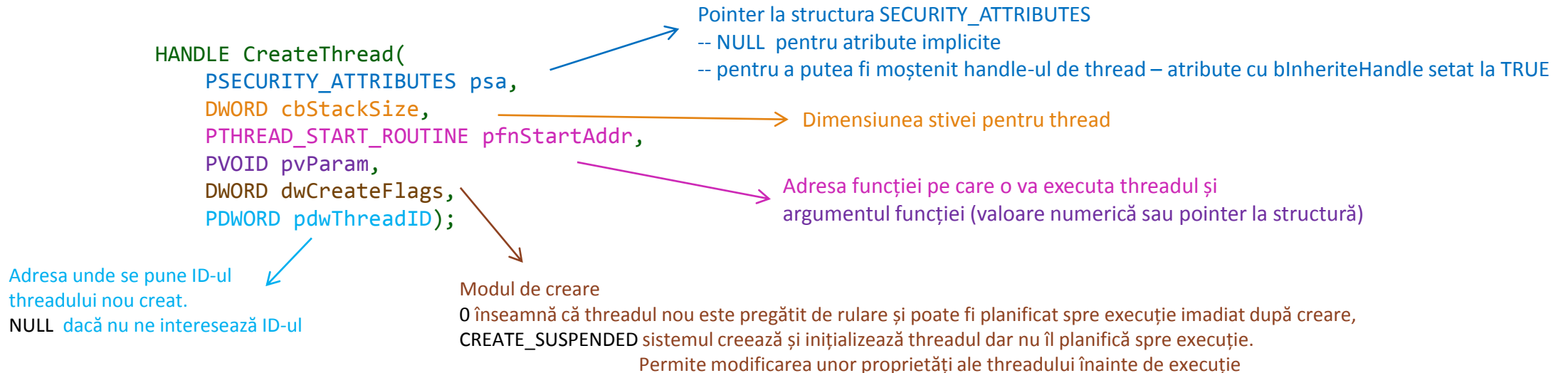
- Threadul principal execută funcția principală (main-ul) – numele funcției este impus
  - Cunoaștem convenția pentru numirea argumentelor funcției principale
- Celelalte threaduri execută funcții cu formă generică - și pot avea nume arbitrare
  - Funcția are un singur parametru – de tipul definit de programator
  - Funcția returnează o singură valoare
- Importanța scrierii programelor care folosesc pe cât posibil doar variabile locale și funcții care comunică prin parametri transmiși
  - Variabilele globale și statice sunt partajate de toate threadurile + concurență → posibile probleme inconsistente a datelor → necesită mecanisme explicite de sincronizare



# Crearea unui nou thread

WinAPI furnizează funcția `CreateThread` pentru crearea unui nou thread (secundar)

- Prototipul funcției: 6 parametri



- Returnează handle-ul threadului nou creat (și ID-ul în `pdwThreadId`)

La apelul funcției `CreateThread` sistemul efectuează

- Creează un nou obiect kernel thread – adică structura de date prin care SO gestionează threadul
- Alocă memorie pentru stiva threadului din spațiul de adrese al procesului
- Threadurile din același proces sunt concurente și partajează resursele procesului – comunicare facilă

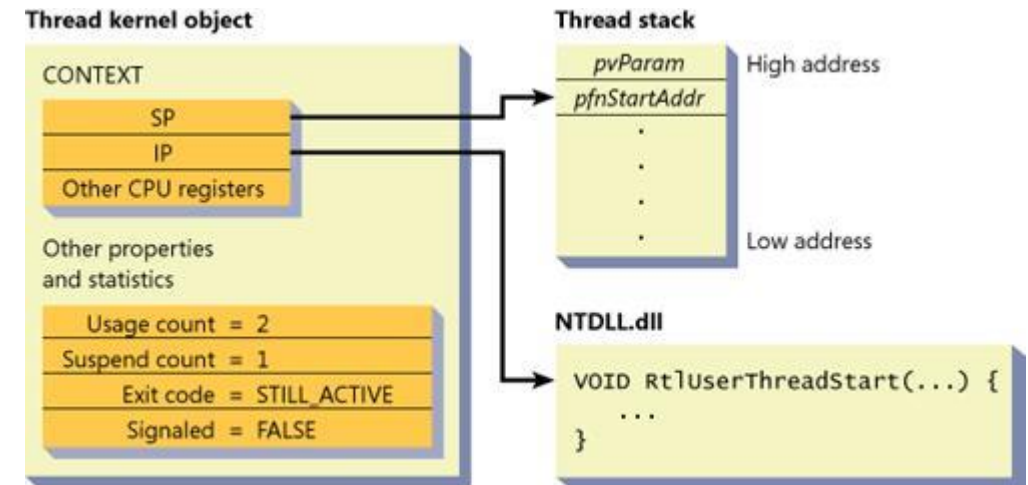
# Crearea unui nou thread

## 1. Crearea unui nou obiect kernel

- Popularea informațiilor de stare:
  - Contor de referiri: 2
    - Threadul creat + handle returnat threadului apelant
  - Contor de suspendare: 1
    - Dacă `CREATE_SUSPENDED` nu apare în flag-uri, atunci contorul se decrementează la 0 și threadul poate fi planificat spre execuție pe procesor
  - Cod de exit: `STILL_ACTIVE` (0x103)
  - Stare: nesemnalizat

## 2. Alocarea memoriei pentru stiva threadului

- din spațiul de adrese al procesului
- Stiva crește de la adrese mari către adrese mici
- Popularea stivei cu
  - Argumentul: `pvParam`
  - Adresa funcției de executat: `pfnStartAddr`



## 3. Contextul threadului (TEB – Thread Environment Block)

- Contextul este dat de starea setului de regiștri de pe procesor proprii threadului – care descriu execuția threadului
  - Valorile regiștrilor se salvează în structura **CONTEXT** – definită în **WinNT.h** – care face parte din obiectul kernel al thread-ului
- Cei mai importanți regiștrii
  - IP** (Instruction Pointer), **SP** (Stack Pointer)
- La inițializarea obiectului kernel
  - SP** indică adresa din stiva threadului unde s-a plasat adresa funcției `pfnStartAddr`
  - IP** indică adresa funcției `RtlUserThreadStart` (din modulul **NTDLL.dll**) – aici își începe execuția threadul

# Crearea unui nou thread

Când threadul este lansat în execuție pentru prima dată – începe execuția funcției RtlUserThreadStart

- IP – conține adresa funcției RtlUserThreadStart – funcție nedocumentată
- RtlUserThreadStart găsește argumentele pe stivă – sistemul a inițializat primele elemente din stivă imediat după alocarea spațiului
- Utilitatea funcției RtlUserThreadStart – funcție wrapper
  - Crează un cadru pentru tratarea structurată a excepțiilor (structured exception handling - SEH) pe durata executării de către thread a funcției specificate la crearea threadului
    - Excepțiile care apar beneficiază de tratare implicită din partea sistemului
  - Se apelează funcția threadului cu argumentul specificat la crearea threadului
  - La revenirea din funcția threadului,
    - se apelează ExitThread cu parametru valoarea returnată de funcția threadului
    - Se decrementează contorul de referiri ale threadului
    - Threadul își termină execuția
  - Dacă apar excepții care nu sunt tratate explicit – SEH tratează aceste excepții implicit
    - Mesaj de eroare
    - Se apelează ExitProcess - și se termină întregul proces, nu doar threadul care a declanșat excepția

## Observații

- Funcția RtlUserThreadStart pune pe stivă adresa de retur a funcției threadului – astfel funcția poate reveni și returna o valoare
- Threadul se termină întotdeauna în cadrul funcției RtlUserThreadStart – prin ExitThread sau ExitProcess
  - RtlUserThreadStart nu revine niciodată – moare cu terminarea threadului
- O funcție similară este apelată și la crearea unui nou proces – inițializarea threadului principal al procesului (funcția BaseProcessStart)

# CreateThread – observații

- Parametrul `cbStackSize`

- Indică dimensiunea stivei proprii a threadului nou creat

- Dacă este setat la 0 →

se iau în considerare valorile setate în opțiunea `/STACK` a linkeditorului

`/STACK: [reserve] [, commit]`

- `reserve` – setează dimensiunea spațiului **rezervat** pentru stivă din spațiul de adrese al procesului – implicit este 1MB
- `commit` – specifică volumul de spațiu fizic **alocat** inițial spațiului rezervat – implicit o pagină
- Pe măsura execuției threadului, dacă threadul își depășește limitele stivei – apare o excepție și sistemul alocă încă o pagină (sau cât este specificat la `commit`) la spațiul rezervat
  - Permite ca stiva threadului să își modifice dimensiunea în mod dinamic
- Dacă este o valoare pozitivă → se rezervă tot spațiul necesar threadului încă de la creare
  - Spațiul rezervat este valoarea mai mare dintre
    - Parametrul specificat, sau
    - Valoarea setată în opțiunea `/STACK` plasat de linkeditor în fișierul `.exe`
  - Spațiul alocat fizic este egal cu parametrul specificat
- Spațiul rezervat este o limită maximă pentru stivă – permite captarea erorilor date de recursivitate infinită

```
HANDLE CreateThread(  
    PSECURITY_ATTRIBUTES psa,  
    DWORD cbStackSize,  
    PTHREAD_START_ROUTINE pfnStartAddr,  
    PVOID pvParam,  
    DWORD dwCreateFlags,  
    PDWORD pdwThreadId);
```

# CreateThread – observații

- Parametrii `pfnStartAddr` și `pvParam`
  - Indică adresa funcției executate de thread și argumentul funcției
- Mai multe threaduri pot executa concurrent aceeași funcție sau funcții diferite
  - Atenție la accesul și modificarea valorilor partajate!

```
HANDLE CreateThread(  
    PSECURITY_ATTRIBUTES psa,  
    DWORD cbStackSize,  
    PTHREAD_START_ROUTINE pfnStartAddr,  
    PVOID pvParam,  
    DWORD dwCreateFlags,  
    PDWORD pdwThreadId);
```

```
DWORD WINAPI FirstThread(PVOID pvParam) {  
    // Initialize a stack-based variable  
    int x = 0;  
    DWORD dwThreadId;  
    // Create a new thread.  
    HANDLE hThread = CreateThread(NULL, 0, SecondThread, (PVOID)&x, 0, &dwThreadId);  
    // We don't reference the new thread anymore, so close our handle to it.  
    CloseHandle(hThread);  
    // Our thread is done.  
    // BUG: our stack will be destroyed, but SecondThread might try to access it.  
    return(0);  
}  
  
DWORD WINAPI SecondThread(PVOID pvParam) {  
    // Do some lengthy processing here. ...  
    // Attempt to access the variable on FirstThread's stack.  
    // NOTE: This may cause an access violation - it depends on timing!  
    * ((int *)pvParam) = 5; /*...*/ return(0);  
}
```

Threadul care execută funcția `FirstThread` și threadul nou creat sunt concurente

Threadul care execută funcția `SecondThread` primește ca și argument adresa unei variabile locale funcției `FirstThread`

Dacă `SecondThread` se execută după ce `FirstThread` deja și-a terminat execuția, și stiva aferentă threadului a fost dealocată, accesul și modificarea argumentului poate rezulta în acces invalid

Soluția? ... Folosim variabile statice?

→ SINCRONIZARE

# CreateThread – observații

## Specificarea atributelor de securitate pentru proces și thread

- Procesul părinte poate specifica atributele de securitate pt procesul fiu și threadul principal din acesta
  - Proprietatea de moștenibilitate
  - NULL – pentru attribute implicite

```
// Prepare a STARTUPINFO structure for spawning processes.
STARTUPINFO si = { sizeof(si) };
SECURITY_ATTRIBUTES saProcess, saThread;
PROCESS_INFORMATION piProcessB, piProcessC;
TCHAR szPath[MAX_PATH];

// Prepare to spawn Process B from Process A.
// Process B's handle set to inheritable.
saProcess.nLength = sizeof(saProcess);
saProcess.lpSecurityDescriptor = NULL;
saProcess.bInheritHandle = TRUE;
// The thread's handle set to NOT inheritable.
saThread.nLength = sizeof(saThread);
saThread.lpSecurityDescriptor = NULL;
saThread.bInheritHandle = FALSE;

// Spawn Process B.
_tcscpy_s(szPath, _countof(szPath), TEXT("ProcessB"));
CreateProcess(NULL, szPath, &saProcess, &saThread, FALSE, 0, NULL,
             NULL, &si, &piProcessB);
```

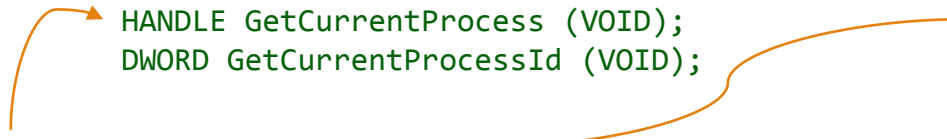
```
HANDLE CreateThread(
    PSECURITY_ATTRIBUTES psa,
    DWORD cbStackSize,
    PTHREAD_START_ROUTINE pfnStartAddr,
    PVOID pvParam,
    DWORD dwCreateFlags,
    PDWORD pdwThreadId);
```

- Argumentul bInheritHandles indică dacă handle-urile moștenibile din părinte vor fi accesibile fiului (fiul primește copii ale handle-urilor moștenite)

```
// Prepare to spawn Process C from Process A.
// Since NULL, the handles to Process C's process and
// primary thread objects default to "noninheritable."
// If Process A were to spawn another process, this new
// process would NOT inherit handles to Process C's process
// and thread objects.
// Because TRUE is passed for the bInheritHandles parameter,
// Process C will inherit the handle that identifies Process
// B's process object but will not inherit a handle to
// Process B's primary thread object.
_tcscpy_s(szPath, _countof(szPath), TEXT("ProcessC"));
CreateProcess(NULL, szPath, NULL, NULL, TRUE, 0, NULL,
             NULL, &si, &piProcessC);
```

# Informații de identificare

## Funcții pentru obținerea handle-ului pentru procesul / threadul apelant



```
HANDLE GetCurrentProcess (VOID);  
DWORD GetCurrentProcessId (VOID);
```

```
HANDLE GetCurrentThread (VOID);  
DWORD GetCurrentThreadId (VOID);
```

- Se returnează un pseudo-handle – nu este moștenibil
- Exemplu: interogarea timpilor de execuție pentru procesul în care se execută threadul și pentru thread

```
FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;  
GetProcessTimes( GetCurrentProcess(),  
&ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
```

```
FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;  
GetThreadTimes(GetCurrentThread(),  
&ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
```

## Funcții pentru obținerea id-ului și al handle-ului pentru procesul / threadul apelant

- Funcții pentru obținerea id-ului de proces / thread unic per sistem – aceste funcții nu sunt atât de utile ca cele care returnează handle

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

```
DWORD GetCurrentThreadId (VOID);
```

## Funcții pentru obținerea id-ului pentru un proces / thread cu handle specificat

```
DWORD GetProcessId (HANDLE Process);
```

```
DWORD GetThreadId (HANDLE Thread);
```

# Informații de identificare

## Handle real și pseudo-handle

- Un handle real poate identifica în mod unic și fără ambiguități un thread sau proces
- Funcțiile GetCurrentProcess și GetCurrentThread returnează pseudo-handle-uri
  - Nu afectează contorul de referiri, nu identifică în mod unic obiectul kernel
- Exemplu cu probleme

```
DWORD WINAPI ParentThread(PVOID pvParam) {  
    HANDLE hThreadParent = GetCurrentThread();  
    CreateThread(NULL, 0, ChildThread, (PVOID)hThreadParent, 0, NULL);  
    // Function continues...  
}  
  
DWORD WINAPI ChildThread(PVOID pvParam) {  
    HANDLE hThreadParent = (HANDLE)pvParam;  
    FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;  
    GetThreadTimes(hThreadParent,  
                   &ftCreationTime, &ftExitTime, &ftKernelTime,  
                   &ftUserTime);  
    // Function continues...  
}
```

Threadul “părinte” dorește să transmită threadului “fiu” un handle care să identifice threadul “părinte”

Se transmite un pseudo-handle !!!  
- Adică un handle către threadul curent

Threadul “fiu” interoghează informațiile proprii, și nu cele referitoare la threadul “părinte” – cum ne-am putea aștepta



# Informații de identificare

## Obținerea handle-ului real

- Un handle real poate identifica în mod unic și fără ambiguități un thread sau proces
- Funcția DuplicateHandle permite obținerea unui handle relativ la proces pe baza unui obiect kernel relativ la un alt proces

```
BOOL DuplicateHandle( HANDLE hSourceProcess, HANDLE hSource,
                     HANDLE hTargetProcess, PHANDLE phTarget,
                     DWORD dwDesiredAccess, BOOL bInheritHandle,
                     DWORD dwOptions);
```

```
HANDLE hProcess;
DuplicateHandle(
    GetCurrentProcess(), // Handle of process that the process
                        // pseudohandle is relative to
    GetCurrentProcess(), // Process' pseudohandle
    GetCurrentProcess(), // Handle of process that the new, real,
                        // process handle is relative to
    &hProcess,           // Will receive the new, real
                        // handle identifying the process
    0,                   // Ignored because of DUPLICATE_SAME_ACCESS
    FALSE,               // New process handle is not inheritable
    DUPLICATE_SAME_ACCESS); // New process handle has same
                        // access as pseudohandle
```

**Obținerea unui handle de proces real pe baza unui pseudo-handle**

# Informații de identificare

## Obținerea handle-ului real

- Un handle real poate identifica în mod unic și fără ambiguități un thread sau proces
- Funcția DuplicateHandle permite obținerea unui handle relativ la proces pe baza unui obiect kernel relativ la un alt proces

```
BOOL DuplicateHandle( HANDLE hSourceProcess, HANDLE hSource,
                    HANDLE hTargetProcess, PHANDLE phTarget,
                    DWORD dwDesiredAccess, BOOL bInheritHandle,
                    DWORD dwOptions);
```

```
DWORD WINAPI ParentThread(PVOID pvParam) {
    HANDLE hThreadParent;
    DuplicateHandle(
        GetCurrentProcess(), // Handle of process that thread pseudohandle is relative to
        GetCurrentThread(),   // Parent thread's pseudohandle
        GetCurrentProcess(),  // Handle of process that the new, real, thread handle is relative to
        &hThreadParent,       // Will receive the new, real, handle identifying the parent thread
        0,                   // Ignored due to DUPLICATE_SAME_ACCESS
        FALSE,               // New thread handle is not inheritable
        DUPLICATE_SAME_ACCESS); // New thread handle has same
    // access as pseudohandle
    CreateThread(NULL, 0, ChildThread, (PVOID)hThreadParent, 0, NULL);
    // Function continues...
}

DWORD WINAPI ChildThread(PVOID pvParam) {
    HANDLE hThreadParent = (HANDLE)pvParam;
    FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;
    GetThreadTimes(hThreadParent, &ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
    CloseHandle(hThreadParent);
    // Function continues...
}
```

hThreadParent este setat la valoarea handle-ului real al threadului “părinte”

DuplicateHandle crește numărul de referiri → trebuie închis explicit handle-ul

# Terminarea threadului

---

## Patru moduri

- **Se revine din funcția threadului** (ex. Fluxul de control a ajuns la finalul instrucțiunilor din funcția threadului)
  - Singurul mod de terminare care garantează eliberarea tuturor resurselor deținute de thread
    - SO eliberează memoria folosită de stiva threadului
    - Codul de terminare al threadului este setat la valoarea de retur a funcției executate
    - Sistemul decrementează contorul de referințe al threadului (obiect kernel) – dacă ajunge la 0, obiectul kernel este distrus
- **Threadul apelează funcția ExitThread** `VOID ExitThread(DWORD dwExitCode);`
  - Funcția ExitThread termină threadul apelant și sistemul eliberează resursele sistemului deținute de thread
    - Este posibil ca resursele să nu fie complet eliberate (unele resurse C/C++)
- **Un thread apelează funcția TerminateThread** `BOOL TerminateThread( HANDLE hThread, DWORD dwExitCode);`
  - Orice thread poate apela funcția TerminateThread pentru a termina un alt thread – funcție asincronă
  - Stiva threadului nu este dealocată decât la terminarea procesului din care a făcut parte threadul
  - Threadul victimă nu este notificat – unele date pot fi în stare de inconsistență
- **La terminarea procesului – la terminarea threadului principal**
  - Când procesul se termină / threadul principal se termină, toate threadurile din cadrul procesului sunt terminate și stiva threadului este eliberată
    - Este posibil ca resursele să nu fie complet eliberate (unele resurse C/C++), datele să nu fie scrise din buffere pe disc, etc.

# Terminarea threadului

---

## La terminarea unui thread

- Handle-urile obiectelor utilizator sunt eliberate (ferestre, hook-uri)
- Cele mai multe obiecte sunt deținute de proces – for fi eliberate la terminarea procesului
- Codul de exit al threadului se schimbă de la STILL\_ACTIVE la codul de return al funcției sau codul specificat de funcțiile ExitThread sau TerminateThread
- Starea obiectului kernel este semnalizată
- Dacă threadul care se termină este ultimul thread activ din proces, atunci se termină și procesul
- Se decrementează contorul de referiri la thread

După terminarea threadului, obiectul thread asociat nu este șters decât dacă contorul de referiri la obiectul thread ajunge la zero

- Alte procese pot apela GetExitCodeThread pentru a obține informații legate de terminarea threadului – codul de exit
  - Dacă threadul cu handle-ul hThread încă nu s-a terminat, atunci la adresa pdwExitCode se pune codul pentru starea STILL\_ACTIVE (0x103)
  - În caz de succes se returnează TRUE

```
BOOL GetExitCodeThread( HANDLE hThread, PDWORD pdwExitCode);
```

# Suspendarea și reluarea threadului

Fiecare thread are un contor de suspendare (în cadrul obiectului kernel)

- Se poate executa – poate fi planificat spre execuție – numai dacă acest contor este 0

Threadul poate fi creat în starea de suspendare (contor = 1), folosind flagul `CREATE_SUSPENDED`

- Permite modificarea unor proprietăți ale threadului înainte de prima execuție

Operații asupra contorului de suspendare

- **ResumeThread** și **SuspendThread**
  - Funcțiile iau ca și parametru handle-ul threadului și returnează valoarea precedentă a contorului de suspendare (sau -1 resp. 0xFFFFFFFF)

```
// Spawn the process that is to be in the job.
// Note: You must first spawn the process and then place the process in
// the job. This means that the process' thread must be initially
// suspended so that it can't execute any code outside of the job's
// restrictions.
STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;
TCHAR szCmdLine[8];
_tcscpy_s(szCmdLine, _countof(szCmdLine), TEXT("CMD"));

BOOL bResult =
    CreateProcess( NULL, szCmdLine, NULL, NULL, FALSE, CREATE_SUSPENDED | CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi);

// Place the process in the job.
// Note: If this process spawns any children, the children are
// automatically part of the same job.
AssignProcessToJobObject(hjob, pi.hProcess);

// Now we can allow the child process' thread to execute code.
ResumeThread(pi.hThread);
CloseHandle(pi.hThread);
```

## Exemplu reluat – vezi joburi

Orice thread poate apela `SuspendThread` pentru a suspenda un alt thread (dacă cunoaște handle-ul).

Threadul se poate autosuspenda, dar nu se poate reveni singur din suspendare.

# Suspendarea execuției pentru un interval de timp

Threadul poate indica sistemului că nu dorește să fie planificat spre execuție o anumită durată de timp

```
VOID WINAPI Sleep(  
    DWORD dwMilliseconds );
```

Durata de suspendare



- Threadul se suspendă și cedează restul timpului din time-slice-ul alocat
- După durata de timp specificată threadul va intra în coada threadurilor pregătite de execuție pentru a fi planificată
- Valori speciale ale duratei
  - INFINITE – threadul va fi suspendat și nu va mai fi planificat spre execuție (este mai potrivit să se termine și să elibereze resursele)
  - 0 – threadul cedează restul time-slice-ului și planificatorul alege un thread spre execuție (este posibil să fie tot acest thread ales)

```
#include <windows.h>  
#include <stdio.h>  
  
int _tmain() {  
    printf("starting to sleep...\n");  
    Sleep(5000); // sleep five seconds  
    printf("sleep ended\n");  
}
```

## Predarea procesorului pentru a executa un thread mai puțin prioritar

- Similar cu Sleep(0) – dar permite unui thread de prioritate mai mică să se execute o cantitate de timp
  - Utilizat pentru a grăbi eliberarea unor resurse deținute de threadul mai puțin prioritar și care sunt necesare pentru threadul care apelează funcția

```
BOOL WINAPI SwitchToThread(void);
```

# Așteptarea terminării unui thread

Cea mai simplă metodă de sincronizare – similar cu utilizarea în cadrul proceselor

```
DWORD WaitForSingleObject(HANDLE hObject, DWORD dwTimeout);
```

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

## Funcțiile de așteptare

- Suspendă execuția threadului apelant până când threadul specificat se termină
- Pot aștepta după diferite obiecte kernel
  - inclusiv threaduri, procese
- Aștepta pentru ca obiectul specificat să fie semnalat
  - ExitThread și TerminateThread semnalează obiectul thread și eliberează threadurile care așteptau după acele obiecte
- Au o perioadă de time-out

Așteaptă până când threadul creat își termină execuția

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

DWORD WINAPI ThreadFunction(LPVOID lpParam) {
    DWORD count = (DWORD)lpParam;
    // increment counter
    for (int i = 0; i < 100; i++)
        count++;
    _tprintf(_T("In second thread: counter is %d\n"), count);
    return count;
}

int _tmain(void) {
    DWORD dwThreadID, exThread;
    DWORD counter = 0;

    // CreateThread
    HANDLE hThread = CreateThread(NULL, (DWORD)NULL, (LPTHREAD_START_ROUTINE)ThreadFunction,
                                  (LPVOID)counter, (DWORD)NULL, &dwThreadID);

    // suspend main thread until new thread completes
    WaitForSingleObject(hThread, INFINITE);

    // examine create thread response
    if (!GetExitCodeThread(hThread, &exThread)) {
        _tprintf(_T("GetExitCodeThread: %0X\n"), GetLastError()); return;
    }
    _tprintf(_T("Second thread returned: %d\n"), exThread);

    // close thread handle
    CloseHandle(hThread);
    return 0;
}
```

# Exemplu – lucrul cu threaduri (1)

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

```
#define MAX_THREADS 3
#define BUF_SIZE 255
```

Se definesc constantele simbolice,  
se declară prototipurile funcțiilor

```
DWORD WINAPI MyThreadFunction(LPVOID lpParam);
void ErrorHandler(LPTSTR lpszFunction);
```

```
// Sample custom data structure for threads to use.
// This is passed by void pointer so it can be any data type
// that can be passed using a single void pointer (LPVOID).
```

```
typedef struct MyData {
    int val1;
    int val2;
} MYDATA, *PMYDATA;
```

Se definește structura de date  
care va fi transmisă threadurilor



# Exemplu – lucrul cu threaduri

```
int _tmain() {
    PMYDATA pDataArray[MAX_THREADS];
    DWORD   dwThreadIdArray[MAX_THREADS];
    HANDLE   hThreadArray[MAX_THREADS];

    // Create MAX_THREADS worker threads.

    for (int i = 0; i < MAX_THREADS; i++) {
        // Allocate memory for thread data.

        pDataArray[i] = (PMYDATA)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, sizeof(MYDATA));
        if (pDataArray[i] == NULL) {
            // If the array allocation fails, the system is out of memory
            // so there is no point in trying to print an error message.
            // Just terminate execution.
            ExitProcess(2);
        }

        // Generate unique data for each thread to work with.
        pDataArray[i]->val1 = i;
        pDataArray[i]->val2 = i + 100;
    }
}
```

Alocarea dinamică a memoriei

Popularea datelor pentru fiecare thread care urmează să fie creat

Transmite pointer la variabilă locală. Threadul principal trebuie să rămână în viață – altfel pointerul devine invalid !!!

```
// Create the thread to begin execution on its own.
hThreadArray[i] = CreateThread( NULL, // default security attributes
                                0, // use default stack size
                                MyThreadFunction, // thread function name
                                pDataArray[i], // argument to thread function
                                0, // use default creation flags
                                &dwThreadIdArray[i]); // returns the thread identifier

// Check the return value for success.
// If CreateThread fails, terminate execution.
// This will automatically clean up threads and memory.
if (hThreadArray[i] == NULL) {
    tprintf(_T("CreateThread failed with: %0X\n"), GetLastError());
    ExitProcess(3);
}
} // End of main thread creation loop.
```

# Exemplu – lucrul cu threaduri

```
// main function continues....
// Wait until all threads have terminated.
WaitForMultipleObjects(MAX_THREADS, hThreadArray, TRUE, INFINITE);

// Close all thread handles and free memory allocations.

for (int i = 0; i < MAX_THREADS; i++) {
    CloseHandle(hThreadArray[i]);
    if (pDataArray[i] != NULL) {
        HeapFree(GetProcessHeap(), 0, pDataArray[i]);
        pDataArray[i] = NULL; // Ensure address is not reused.
    }
}

return 0;
}
```

Eliberarea memoriei alocate dinamic



```
DWORD WINAPI MyThreadFunction(LPVOID lpParam) {
    HANDLE hStdout;
    PMYDATA pDataArray;

    TCHAR msgBuf[BUF_SIZE];
    size_t cchStringSize;
    DWORD dwChars;

    // Make sure there is a console to receive output results.
    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    if (hStdout == INVALID_HANDLE_VALUE)
        return 1;

    // Cast the parameter to the correct data type.
    // The pointer is known to be valid because
    // it was checked for NULL before the thread was created.
    pDataArray = (PMYDATA)lpParam;

    // Print the parameter values using thread-safe functions.
    StringCchPrintf(msgBuf, BUF_SIZE, TEXT("Parameters = %d, %d\n"),
        pDataArray->val1, pDataArray->val2);
    StringCchLength(msgBuf, BUF_SIZE, &cchStringSize);
    WriteConsole(hStdout, msgBuf, (DWORD)cchStringSize, &dwChars, NULL);

    return 0;
}
```

# Multithreading și bibliotecile din CRT

Bibliotecile din CRT au fost dezvoltate având la bază modelul de execuție single threaded (1 thread/proces)

- Multe funcții din biblioteci nu sunt potrivite pentru utilizare în mediu multithreaded – nu sunt **thread-safe**
- Motiv: păstrează rezultate intermediare în date stocate static sau global
  - Două threaduri concurente pot accesa simultan biblioteca rezultând în modificare simultană a datelor globale ale bibliotecii
- Exemple: errno, strtok, tmpfile, asctime, etc.
  - Funcția strtok caută în iterații succesive următoarea apariție a unui token
    - Între apeluri succesive păstrează starea persistentă folosind stocare statică – toate threadurile care accesează funcția strtok au acces la starea intermediară !!

## Soluție

- Furnizarea unei implementări thread-safe a bibliotecii C (LIBCMT.LIB) și
- Crearea și asocierea unei structuri de date la threadul nou creat astfel încât funcțiile din bibliotecile C să își păstreze datele intermediare în acele spații de stocare individuale fiecărui thread
- → pentru crearea threadului folosim funcția **\_beginthreadex** (în loc de CreateThread)
- → pentru terminarea threadului folosim funcția **\_endthreadex** (în loc de ExitThread)

# \_beginthreadex și \_endthreadex

## Prototipul funcției

- Parametrii sunt aceleași ca și în cazul CreateThread – cu excepția tipurilor și numelor parametrilor
- Returnează handle-ul către threadul nou creat sau 0

```
uintptr_t _beginthreadex( // NATIVE CODE
    void *security,
    unsigned stack_size,
    unsigned ( __stdcall *start_address )( void * ),
    void *arglist,
    unsigned initflag,
    unsigned *thrdaddr );
```

## Funcționalitate

- Fiecărui thread i se alocă un bloc de memorie **\_tiddata** din heap-ul CRT
  - Acolo se stochează inclusiv adresa funcției de efectuat de thread și parametrii acesteia
- \_beginthreadex apelează intern funcția CreateThread dar cu argumentele care s-au stocat în \_tiddata
- Threadul își începe execuția cu funcția RtlUserThreadStart și apoi trece la execuția funcției specificate
- Asocierea dintre structura de date \_tiddata și threadul creat se realizează prin TlsSetValue (TLS – Thread-Local Storage)
- Se formează un cadru SEH (structured exception handling) pentru tratarea excepțiilor în jurul funcției apelate de thread
- Valoarea returnată de funcția threadului este setată ca și cod de retur al threadului prin funcția \_endthreadex
  - Funcția \_endthreadex obține adresa structurii \_tiddata al threadului folosind TlsGetValue, eliberează blocul de memorie și apelează ExitThread cu codul setat la valoarea returnată de funcția threadului

# Exemplu – lucrul cu threaduri și CRT

```
// crt_begthrdex.cpp
// compile with: /MT
#include <windows.h>
#include <stdio.h>
#include <process.h>

unsigned Counter;
unsigned __stdcall SecondThreadFunc(void* pArguments) {
    printf("In second thread...\n");
    while (Counter < 1000000)
        Counter++;
    _endthreadex(0);
    return 0;
}

int main() {
    HANDLE hThread;
    unsigned threadID;

    printf("Creating second thread...\n");
    // Create the second thread.
    hThread = (HANDLE)_beginthreadex(NULL, 0, &SecondThreadFunc, NULL, 0, &threadID);

    // Wait until second thread terminates. If you comment out the line below, Counter will not be correct because
    // the thread has not terminated, and Counter most likely has not been incremented to 1000000 yet.
    WaitForSingleObject(hThread, INFINITE);
    printf("Counter should be 1000000; it is-> %d\n", Counter);
    // Destroy the thread object.
    CloseHandle(hThread);
}
```

# Timpii de execuție ai threadului

Determinarea timpului de execuție consumat de thread pentru efectuarea sarcinii asignate

- Durata de timp în care threadul a rulat pe procesor

```
BOOL WINAPI GetThreadTimes(  
    HANDLE      hThread,   
    LPFILETIME  lpCreationTime,   
    LPFILETIME  lpExitTime,   
    LPFILETIME  lpKernelTime,   
    LPFILETIME  lpUserTime );
```

Diagram illustrating the parameters of the `GetThreadTimes` function:

- `hThread`: Handle-ul threadului. Este necesar dreptul de acces **THREAD\_QUERY\_INFORMATION** sau **THREAD\_QUERY\_LIMITED\_INFORMATION**
- `lpCreationTime`: Structură FILETIME care primește timpul de creare a threadului
- `lpExitTime`: Structură FILETIME care primește timpul de terminare a threadului
- `lpKernelTime`: Structură FILETIME care primește timpul de execuție în mod kernel al threadului
- `lpUserTime`: Structură FILETIME care primește timpul de execuție în mod utilizator al threadului

- Returnează patru valori de timp diferite
  - Timpul de create: valoare absolută exprimată în intervale de 100 ns care s-au scurs din 1 ianuarie 1601
  - Timpul de terminare: valoare absolută exprimată în intervale de 100 ns care s-au scurs din 1 ianuarie 1601
    - Valoare nedefinită, dacă threadul încă nu s-a terminat
  - Timpul kernel: valoarea relativă exprimată în intervale de 100 ns cât timp threadul a executat codul sistemului de operare în modul kernel
  - Timpul utilizator: valoarea relativă exprimată în intervale de 100 ns cât timp threadul a executat codul aplicației

Timpii de execuție a procesului pot fi obținuți cu funcția `GetProcessTimes`

- Se aplică tuturor threadurilor din proces – chiar și threadurilor care s-au terminat deja
  - Ex. Timpul kernel al procesului este suma tuturor duratelor de timp în care threaduri din proces au rulat în mod kernel

# Exemplu: căutare paralelă de șabloane

---

Aplicația grepMP (vezi procese) a folosit procese paralele pentru căutarea șabloanelor în fișierele de intrare

Aplicația grepMT folosește threaduri concurente din cadrul unui singur proces

- se apelează astfel:

`grepMT pattern F1 F2 ... FN`

- Fiecare fișier de intrare F1 ... FN este căutat de câte un thread separat
  - fiecare rulează funcția ThGrep(pattern, Fk)
- WaitForMultipleObjects așteaptă terminarea unui singur thread de căutare
  - Diferență față de grepMP – unde se așteptau toate procesele
- După terminarea unui thread de căutare, rezultatele sale sunt afișate
  - Se afișează numele fișierului și linia completă în care s-a găsit șablonul
  - Ordinea de terminare a threadurilor variază de la o execuție la alta
- Codul de exit de la programele grep este folosit pentru determinarea dacă s-a detectat șablonul

# Exemplu: căutare paralelă de șabloane

## Codul sursă al aplicației grepMT

```
/* Chapter 7. grepMT. */
/* Parallel grep - Multiple thread version. */

/* grep pattern files.
   Search one or more files for the pattern.
   The results are listed in the order in which the threads complete,
   not in the order the files are on the command line.
   This is primarily to illustrate the non-determinism of thread completion.
   To obtain ordered output, use the technique in Program 8-1. */

/* Be certain to define _MT in Environment.h or use the
   build...settings...C/C++...CodeGeneration...Multithreaded library. */

#include "Everything.h"

typedef struct { /* grep thread's data structure. */
    int argc;
    TCHAR targv[4][MAX_COMMAND_LINE];
} GREP_THREAD_ARG;

typedef GREP_THREAD_ARG *PGR_ARGS;

/* Source code for grep is omitted from text. */
static DWORD WINAPI ThGrep(PGR_ARGS pArgs);

VOID _tmain(int argc, LPTSTR argv[])
/* Create a separate THREAD to search each file on the command line.
   Report the results as they come in. Each thread is given a temporary file,
   in the current directory, to receive the results.
   This program modifies Program 8-1, which used processes. */
{
    PGR_ARGS gArg; /* Points to array of thread args. */
    HANDLE * tHandle; /* Points to array of thread handles. */
    TCHAR commandLine[MAX_COMMAND_LINE];
    BOOL ok;
    DWORD threadIndex, exitCode;
    int iThrd, threadCount;
    STARTUPINFO startUp;
    PROCESS_INFORMATION processInfo;
```

```
/* Start up info for each new process. */

GetStartupInfo(&startUp);

if (argc < 3)
    ReportError(_T("No file names."), 1, TRUE);

/* Create a separate "grep" thread for each file on the command line.
   Each thread also gets a temporary file name for the results.
   argv[1] is the search pattern. */

tHandle = (PHANDLE) malloc((argc - 2) * sizeof(HANDLE));
gArg = (PGR_ARGS) malloc((argc - 2) * sizeof(GREP_THREAD_ARG));

for (iThrd = 0; iThrd < argc - 2; iThrd++) {

    /* Set: targv[1] to the pattern
       targv[2] to the input file
       targv[3] to the output file. */

    _tcscpy(gArg[iThrd].targv[1], argv[1]); /* Pattern. */
    _tcscpy(gArg[iThrd].targv[2], argv[iThrd + 2]); /* Search file. */

    if (GetTempFileName /* Temp file name */
        (_T("."), _T("Gre"), 0, gArg[iThrd].targv[3]) == 0)
        ReportError(_T("Temp file failure."), 3, TRUE);

    /* Output file. */

    gArg[iThrd].argc = 4;

    /* Create a thread to execute the command line. */

    tHandle[iThrd] = (HANDLE) _beginthreadex(
        NULL, 0, ThGrep, &gArg[iThrd], 0, NULL);

    if (tHandle[iThrd] == 0)
        ReportError(_T("ThreadCreate failed."), 4, TRUE);
}
```



# Exemplu: căutare paralelă de șabloane

## Codul sursă al aplicației grepMT (continuare)

```
/* Threads are all running. Wait for them to complete
   one at a time and put out the results. */
/* Redirect output for "cat" process listing results. */
```

```
startUp.dwFlags = STARTF_USESTDHANDLES;
startUp.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);
startUp.hStdError = GetStdHandle(STD_ERROR_HANDLE);
```

```
threadCount = argc - 2;
```

```
while (threadCount > 0) {
    threadIndex = WaitForMultipleObjects(threadCount, tHandle, FALSE, INFINITE);
    iThrd = (int)threadIndex - (int)WAIT_OBJECT_0;
    if (iThrd < 0 || iThrd >= threadCount)
        ReportError(_T("Thread wait error."), 5, TRUE);
    GetExitCodeThread(tHandle[iThrd], &exitCode);
    CloseHandle(tHandle[iThrd]);
```

```
/* List file contents (if a pattern match was found)
   and wait for the next thread to terminate. */
```

```
if (exitCode == 0) {
    if (argc > 3) { /* Print file name if more than one. */
        _tprintf(_T("%s\n"),
            gArg[iThrd].targv[2]);
        fflush(stdout);
    }
    _stprintf(commandLine, _T("cat \"%s\""), gArg[iThrd].targv[3]);
```

```
    ok = CreateProcess(NULL, commandLine, NULL, NULL,
        TRUE, 0, NULL, NULL, &startUp, &processInfo);
```

```
    if (!ok) ReportError(_T("Failure executing cat."), 6, TRUE);
    WaitForSingleObject(processInfo.hProcess, INFINITE);
    CloseHandle(processInfo.hProcess);
    CloseHandle(processInfo.hThread);
}
```

```
if (!DeleteFile(gArg[iThrd].targv[3]))
    ReportError(_T("Cannot delete temp file."), 7, TRUE);
```

```
/* Move the handle of the last thread in the list
   to the slot occupied by thread that just completed
   and decrement the thread count. Do the same for
   the temp file names. */
```

```
tHandle[iThrd] = tHandle[threadCount - 1];
_tcscpy(gArg[iThrd].targv[3], gArg[threadCount - 1].targv[3]);
_tcscpy(gArg[iThrd].targv[2], gArg[threadCount - 1].targv[2]);
threadCount--;
```

```
}
free(tHandle);
free(gArg);
```

```
C:\Windows\System32\cmd.exe

C:\WSP4_Examples\Run8>timep.exe grepMP.exe carrot AfricanTales.txt CelticTales.t
xt ChineseTales.txt DutchTales.txt IndianTales.txt OrientalTales.txt PersianTales
.txt WelshTales.txt
DutchTales.txt:
between a carrot and a sweet potato. It was as thick as reeds in a swamp
PersianTales.txt:
carrots when pulled out of their native soil. Cadmus hardly knew whether
WelshTales.txt:
as a hogshhead, or a push-ball, or a market wagon loaded with carrots.
Real Time: 00:00:00.448
User Time: 00:00:00.015
Sys Time: 00:00:00.015

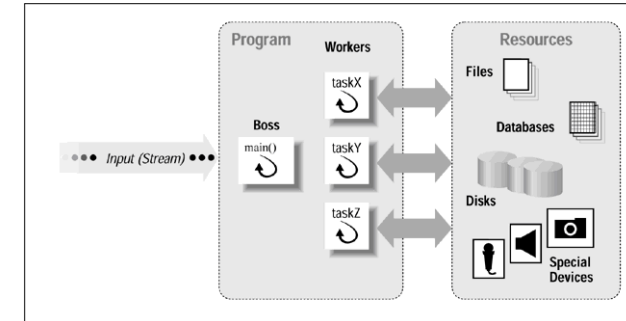
C:\WSP4_Examples\Run8>timep.exe grepMT.exe carrot AfricanTales.txt CelticTales.t
xt ChineseTales.txt DutchTales.txt IndianTales.txt OrientalTales.txt PersianTales
.txt WelshTales.txt
PersianTales.txt
carrots when pulled out of their native soil. Cadmus hardly knew whether
WelshTales.txt
as a hogshhead, or a push-ball, or a market wagon loaded with carrots.
DutchTales.txt
between a carrot and a sweet potato. It was as thick as reeds in a swamp
Real Time: 00:00:00.291
User Time: 00:00:00.640
Sys Time: 00:00:00.062

C:\WSP4_Examples\Run8>
```

# Modele de programare paralelă

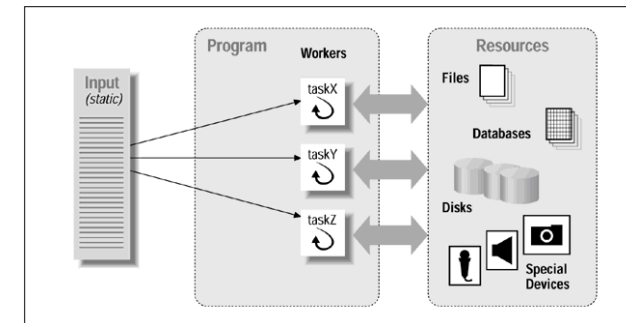
## Modelul Boss-Worker

- Exemplificat de aplicația grepMT
  - Threadul Boss (principal) preia comenzile, creează threadurile Worker și indică acestora sarcinile de executat



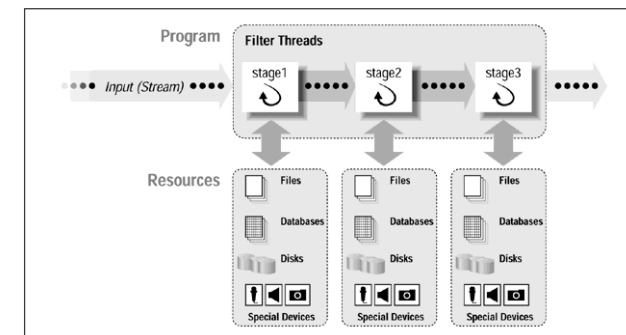
## Modelul Peer (Workcrew)

- Toate threadurile lucrează la sarcinile proprii fără un lider anume
- Fiecare thread este responsabil de propriile date de intrare și ieșire
  - Are o cale privată de a le obține



## Modelul Pipeline

- Sarcinile sunt alcătuite din mai multe suboperații (etape sau filtre)
- Fiecare intrare trebuie să treacă prin fiecare etapă
- Fiecare etapă poate procesa simultan câte o unitate distinctă de intrare



# Materiale de studiu

---

- Johnson HART, Windows System Programming, 4th edition, Addison Wesley, 2010
  - Capitolul 7
  - Exemplele incluse sunt din cartea de mai sus – vezi arhiva de pe moodle: WSP4\_Examples.zip