Procese și Joburi în WIN32 API

CURS NR. 4

Procese și threaduri

Concept central în sisteme de operare

Abstracția unui program în execuție

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

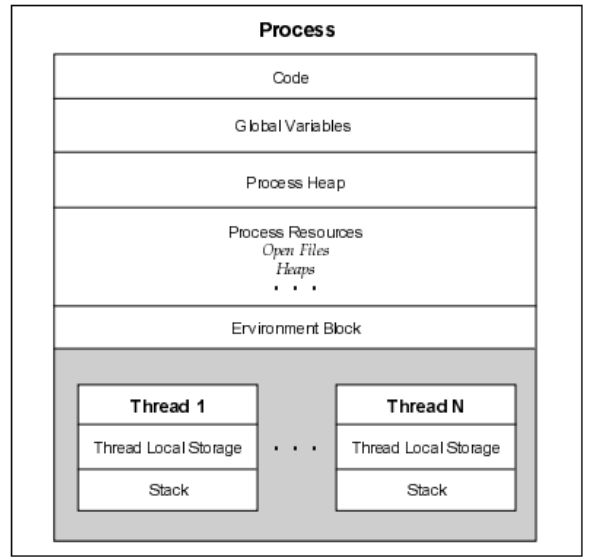


Figure 6-1 A Process and Its Threads

Sistemul poate rula aparent simultan zeci sau sute de procese / threaduri

- Pe unul sau un număr restrâns de procesoare
  - Virtualizarea procesorului: multiplexare în timp

Procesul este caracterizat de resursele pe care le accesează sau le modifică

- Spațiul de adrese (cu segmentul de cod, date, heap, stivă) – distinct și separat de spațiul de adrese al altor procese
- Variabile de mediu (ex. Calea curentă de căutare - path)
- Set (tabela) de handle-uri către obiecte kernel deschise sau alte heap-uri

Threadurile din același proces partajează resursele procesului dar sunt planificate spre execuție independent, deci au nevoie de

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

Crearea unui nou proces

WinAPI furnizează funcția CreateProcess pentru crearea unui nou proces cu un singur thread

- Prototipul funcției: 10 parametri

```
BOOL CreateProcess (
            LPCTSTR lpApplicationName,
            LPTSTR lpCommandLine,
            LPSECURITY_ATTRIBUTES lpsaProcess,
            LPSECURITY_ATTRIBUTES lpsaThread,
            BOOL bInheritHandles,
            DWORD dwCreationFlags,
            LPVOID lpEnvironment,
            LPCTSTR lpCurDir,
            LPSTARTUPINFO lpStartupInfo,
            LPPROCESS_INFORMATION lpProcInfo);
```

Numele executabilului și argumentele acestuia

Structura cu atributele de securitate ale procesului și ale threadului

NULL → atribute implicite

Moștenirea handle-urilor deschise (fișiere, mapări, etc.) din procesul părinte

Variabile de mediu
NULL → moștenire de la părinte

Directorul curent
NULL → moștenire de la părinte

Modul de creare:
CREATE_SUSPENDED: threadul pricipal este suspendat (va reveni doar cu ResumeThread)
DETACHED_PROCESS sau CREATE_NEW_CONSOLE: atașarea la o consolă nouă – altfel moștenește consola părintelui
CREATE_UNICODE_ENVIRONMENT: trebuie setat dacă UNICODE este definit
CREATE_NEW_PROCESS_GROUP: noul proces devine root-ul unui nou grup de procese
+ altele referitoare la prioritate – cel mai adesea folosim NORMAL_PRIORITY_CLASS.

Specifică handle-urile pentru dispozitivele standard de i/e; setările pentru fereastra principală (procese cu GUI), etc.
Informațiile din părinte pot fi accesate apelând în prealabil GetStartupInfo

Structura în care se returnează
-handle-ul procesului
-handle-ul threadului
-identificatorii

```
typedef struct _PROCESS_INFORMATION {
        HANDLE hProcess;
        HANDLE hThread;
        DWORD dwProcessId;
        DWORD dwThreadId;
  } PROCESS_INFORMATION;
```

De ce și handle-uri și id-uri?

- Returnează TRUE (valoare nenulă) dacă procesul și threadul principal s-au creat cu succes

Windows nu menține o ierarhie de procese

- Denumirile proces părinte – proces fiu sunt doar convenție
- După revenirea din funcția CreateProcess – cele două procese sunt complet independente

Închidem handle-ul de thread și de proces - când nu mai avem nevoie de proces

CreateProcess – observații

- Numele executabilului este indicat de unul din parametri `lpApplicationName`, sau `lpCommandLine`
  - De regulă `lpApplicationName` este **NULL** și doar `lpCommandLine` este specificat unde primul element este numele executabilului
    - Cale absolută sau relativă (caz în care se caută în directorul curent al procesului, directorul sistemului Windows, directoarele din PATH)
    - folosim ghilimele dacă numele conține spații (pt a indica unde se termină numele și unde încep parametrii)
    - specificăm și extensia (.exe, .bat, etc.)
  - Noul proces își poate obține argumentele folosind abordarea standard cu **`argv`**, sau poate invoca **`GetCommandLine`**

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

void _tmain(int argc, TCHAR *argv[])
{
   STARTUPINFO si;
   PROCESS_INFORMATION pi;

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

   if (argc != 2)
   {
      printf("Usage: %s [cmdline]\n", argv[0]);
      return;
   }
```

```
   // Start the child process.
   if (!CreateProcess(NULL,   // No module name (use command line)
            argv[1],        // Command line
            NULL,           // Process handle not inheritable
            NULL,           // Thread handle not inheritable
            FALSE,          // Set handle inheritance to FALSE
            0,              // No creation flags
            NULL,           // Use parent's environment block
            NULL,           // Use parent's starting directory
            &si,            // Pointer to STARTUPINFO structure
            &pi))           // Pointer to PROCESS_INFORMATION structure
   {
      printf("CreateProcess failed (%d).\n", GetLastError());
      return;
   }

   // Wait until child process exits.
   WaitForSingleObject(pi.hProcess, INFINITE);

   // Close process and thread handles.
   CloseHandle(pi.hProcess);
   CloseHandle(pi.hThread);
}
```

CreateProcess – observații

```
BOOL CreateProcess (
    LPCTSTR lpApplicationName,
    LPTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpsaProcess,
    LPSECURITY_ATTRIBUTES lpsaThread,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCTSTR lpCurDir,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcInfo);
```

- Parametrul `lpCommandLine` este un pointer la șir VARIABIL
  - Funcția modifică acest argument și apoi înainte de revenire resetează valoarea inițială

```
STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;
CreateProcess(NULL, TEXT("NOTEPAD"), NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi);
```

Șir constant → EROARE

```
STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;
TCHAR szCommandLine[] = TEXT("NOTEPAD");
CreateProcess(NULL, szCommandLine, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi);
```

Copiem șirul constant într-un buffer variabil

- La parsarea parametrului lpCommandLine primul element (cuvânt) este considerat numele executabilului
  - Dacă nu se specifică extensia – implicit se adaugă extensia .exe

lpApplicationName este în 99% din cazuri NULL – dacă nu, atunci conține numele executabilului cu tot cu extensie

- Dacă se specifică doar numele (nu calea întreagă), atunci se caută executabilul doar în directorul curent

```
// Make sure that the path is in a read/write section of memory.
TCHAR szPath[] = TEXT("README.TXT");
// Spawn the new process.
CreateProcess(TEXT("C:\\WINDOWS\\SYSTEM32\\NOTEPAD.EXE"), szPath, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi);
```

CreateProcess – observații

```
BOOL CreateProcess (
        LPCTSTR lpApplicationName,
        LPTSTR lpCommandLine,
        LPSECURITY_ATTRIBUTES lpsaProcess,
        LPSECURITY_ATTRIBUTES lpsaThread,
        BOOL bInheritHandles,
        DWORD dwCreationFlags,
        LPVOID lpEnvironment,
        LPCTSTR lpCurDir,
        LPSTARTUPINFO lpStartupInfo,
        LPPROCESS_INFORMATION lpProcInfo);
```

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 atribute 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);
```

- 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);
```

CreateProcess – observații

```
BOOL CreateProcess (
        LPCTSTR lpApplicationName,
        LPTSTR lpCommandLine,
        LPSECURITY_ATTRIBUTES lpsaProcess,
        LPSECURITY_ATTRIBUTES lpsaThread,
        BOOL bInheritHandles,
        DWORD dwCreationFlags,
        LPVOID lpEnvironment,
        LPCTSTR lpCurDir,
        LPSTARTUPINFO lpStartupInfo,
        LPPROCESS_INFORMATION lpProcInfo);
```

Parametrul lpStartupInfo

- Pointer către structura STARTUPINFO

```
typedef struct _STARTUPINFO {
    DWORD cb;
    PSTR lpReserved;
    PSTR lpDesktop;
    PSTR lpTitle;
    DWORD dwX;
    DWORD dwY;
    DWORD dwXSize;
    DWORD dwYSize;
    DWORD dwXCountChars;
    DWORD dwYCountChars;
    DWORD dwFillAttribute;
    DWORD dwFlags;
    WORD wShowWindow;
    WORD cbReserved2;
    PBYTE lpReserved2;
    HANDLE hStdInput;
    HANDLE hStdOutput;
    HANDLE hStdError;
} STARTUPINFO, *LPSTARTUPINFO;
```

Important pentru setarea ferestrei principale (aplicații cu GUI), a fișierelor de i/o, etc.

```
STARTUPINFO si = { sizeof(si) };
CreateProcess(..., &si, ...);
```

- Cele mai multe aplicții folosesc valori implicite
  - → Trebuie inițializată structura: toate componentele la zero, exceptie componenta cb – se setează la dimensiunea structurii
    - Dacă nu se inițializează, componentele conțin valori arbitrare !! Omiterea inițializării este o eroare foarte frecventă !!!
- Obținerea unei copii a structurii inițializate de părinte

```
VOID GetStartupInfo(LPSTARTUPINFO pStartupInfo);
```

CreateProcess – observații

```
BOOL CreateProcess (
        LPCTSTR lpApplicationName,
        LPTSTR lpCommandLine,
        LPSECURITY_ATTRIBUTES lpsaProcess,
        LPSECURITY_ATTRIBUTES lpsaThread,
        BOOL bInheritHandles,
        DWORD dwCreationFlags,
        LPVOID lpEnvironment,
        LPCTSTR lpCurDir,
        LPSTARTUPINFO lpStartupInfo,
        LPPROCESS_INFORMATION lpProcInfo);
```

Parametrul lpProcInfo

- Pointer către structura PROCESS_INFORMATION

```
typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess;
    HANDLE hThread;
    DWORD dwProcessId;
    DWORD dwThreadId;
} PROCESS_INFORMATION;
```

- La crearea unui obiect kernel (ex. proces, thread) contorul de referiri la obiect este setat la 1
  - La revenirea din CreateProcess contorul este 2 – pentru proces și pentru thread (referința și din părinte și din fiu)
- Pentru eliberarea în timpul rulării a obiectului kernel al procesului fiu (și analogic pentru threadul principal din procesul fiu) trebuie ca
  - Fiul să se termine (decrementează contorul cu 1)
  - Părintele să închidă handle-ul către fiu – folosind CloseHandle (decrementează contorul cu 1)

- **Închiderea handle-ului** nu înseamnă implicit terminarea procesului / threadului
  - Doar se decrementează contorul de referiri
  - Informațiile statistice nu mai sunt disponibile pentru procesul care a apelat funcția CloseHandle

- **Id**-ul și **handle**-ul -- de ce ambele?
  - Id-ul este unic la nivel de sistem - după eliberare valoarea este refolosită !
  - Id-ul este folosit de aplicațiile utilitare – celelalte aplicații de regulă ignoră id-ul

nici un obiect
nu are id-ul 0

Informații de identificare

Funcții pentru obținerea id-ului și al 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
- Se poate crea un **handle de proces real pe baza id-ulu**i de proces, folosind funcția OpenProcess, care returnează handle-ul, sau NULL

```
HANDLE OpenProcess (
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    DWORD dwProcessId);
```

Modul de acces:

- SYNCHRONIZE
- PROCESS_TERMINATE
- PROCESS_QUERY_INFORMATION
- PROCESS_SET_INFORMATION
- PROCESS_QUERY_INFORMATION

- PROCESS_ALL_ACCESS
- PROCESS_CREATE_PROCESS
- PROCESS_CREATE_THREAD
- PROCESS_DUP_HANDLE

și altele

Indică dacă handle-ul procesului este moștenibil

Id-ul procesului pentru care doim să obținem handle-ul

- Procesul părinte poate obține informații de identificare despre procesul fiu prin intermediul structurii `PROCESS_INFORMATION`
  - Deci cunoaște id-ul și handle-ul procesului fiu
    - Dacă părintele închide handle-ul de proces al fiului nu implică terminarea procesului fiu – ci doar ștergerea accesului la fiu din procesul părinte

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

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

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

Terminarea procesului – 4 moduri

- **Threadul principal se termină** (ex. Toate instrucțiunile din funcția principală s-au executat)
  - Chiar dacă alte threaduri încă mai rulează – terminarea procesului implică terminarea tuturor threadurilor
  - Singurul mod de terminare care garantează eliberarea tuturor resurselor deținute de threadul principal
    - SO eliberează memoria folosită de stiva threadului
    - Codul de terminare al procesului este returnat în valoarea de retur a funcției principale
    - Sistemul decrementează contorul de referințe al procesului (obiect kernel) – dacă ajunge la 0, obiectul kernel este distrus

- **Un thread al procesului apelează funcția `ExitProcess`** `VOID ExitProcess(UINT fuExitCode);`
  - Funcția `ExitProcess` termină procesul și setează valoarea de retur la valoarea parametrului fuExitCode
  - Observație: apelând funcția ExitThread din threadul principal va termina doar threadul – procesul va exista până când mai are alte threaduri în execuție
  - Este posibil ca resursele să nu fie complet eliberate

- **Un thread din alt proces apelează funcția `TerminateProcess`** `BOOL TerminateProcess( HANDLE hProcess, UINT fuExitCode);`
  - Orice thread poate apela funcția TerminateProcess pentru a termina un alt proces sau chiar procesul în care rulează
  - Procesul victimă nu este notificat – unele date pot fi în stare de inconsistență, eliberarea resurselor efectuată implicit de SO

- **Toate threadurile procesului se termină** `BOOL GetExitCodeProcess( HANDLE hProcess, PDWORD pdwExitCode);`
  - Când nu mai sunt thread-uri în execuție, procesul se termină și codul de exit este valoarea returnată de ultimul thread care s-a terminat

Așteptarea terminării unui proces

Cea mai simplă metodă de sincronizare

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

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

Funcțiile de așteptare

- Suspendă execuția threadului apelant din procesul părinte până când procesul fiu se termină

- Pot aștepta după diferite obiecte kernel
  - inclusiv procese
- Pot aștepta după
  - un singur proces,
  - primul proces sau
  - toate procesele dintr-un set
- Au o perioadă de time-out

```
PROCESS_INFORMATION pi;
DWORD dwExitCode;
// Spawn the child process.
BOOL fSuccess = CreateProcess(..., &pi);
if (fSuccess) {
// Close the thread handle as soon as it is no longer needed!
    CloseHandle(pi.hThread);

    // Suspend our execution until the child has terminated.
    WaitForSingleObject(pi.hProcess, INFINITE);

    // The child process terminated; get its exit code.
    GetExitCodeProcess(pi.hProcess, &dwExitCode);

    // Close the process handle as soon as it is no longer needed.
    CloseHandle(pi.hProcess);
}
```

Închiderea handle-ului threadului principal al procesul fiu – nu provoacă terminarea threadului – doar decrementează contorul de referiri

Așteaptă până obiectul `pi.hProcess` este semnalat (primește semnal la terminare)

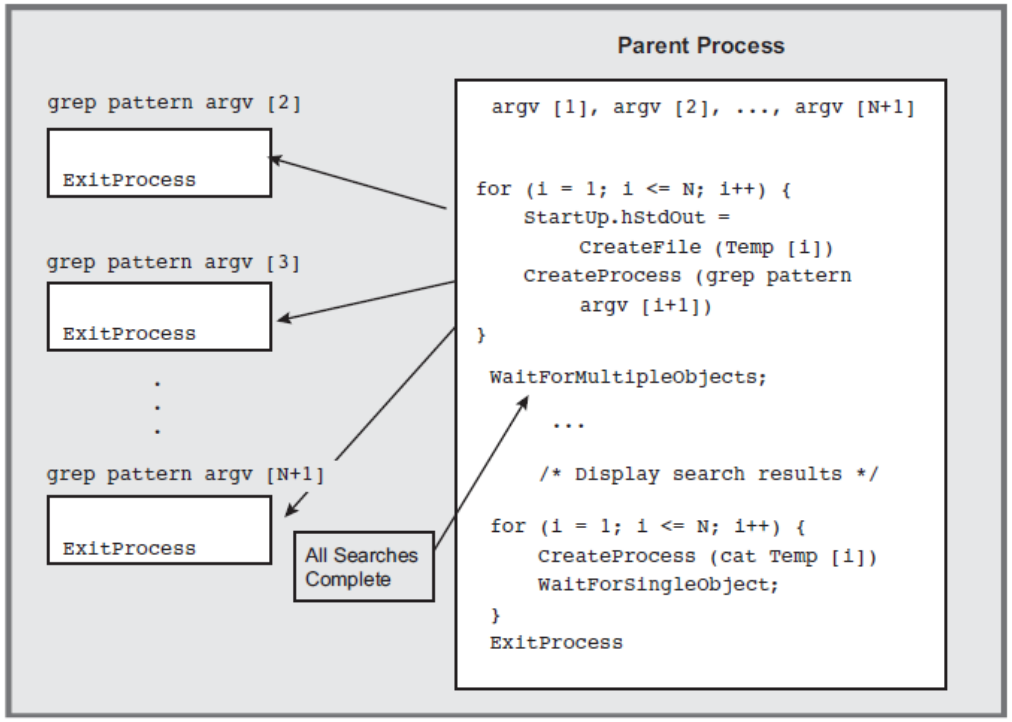
Dacă părintele închide handle-urile la procesul fiu și threadul acestuia, fiul rulează detașat de părinte

Exemplu: căutare paralelă de șabloane

Programul grepMP.c se apelează astfel:

```
grepMP pattern F1 F2 ... FN
```

- Fiecare fișier de intrare F1 ... FN este căutat de câte un proces separat
  - fiecare rulează programul: `grep pattern Fk`
- Handle-ul fișierului temporar este moștenibil
- WaitForMultipleObjects așteaptă terminarea fiecărui proces de căutare
  - WaitForMultipleObjects are o limită de MAXIMUM_WAIT_OBJECTS egal cu 64 de handle-uri după care poate aștepta → deci programul apelează funcția de mai multe ori
- După terminarea fiecărui proces de căutare, rezultatele (din fișierul temporar) sunt afișate în ordine
  - Se afișează numele fișierului și linia completă în care s-a găsit șablonul
- Codul de exit de la programele grep este folosit pentru a determina dacă s-a detectat șablonul



File Searching Using Multiple Processes

Exemplu: căutare paralelă de șabloane

Codul sursă al aplicației grepMP

```
/* Chapter 6. grepMP. */
/* Multiple process version of grep command. */

/* This program illustrates:
1. Creating processes.
2. Setting a child process standard I/O using the process start-up structure.
3. Specifying to the child that the parent's file handles are inheritable.
4. Synchronizing with process termination using WaitForMultipleObjects
and WaitForSingleObject.
5.Generating and using temporary files to hold the output of each process. */

#include "Everything.h"

int _tmain (int argc, LPTSTR argv[])

/*Create a separate process to search each file on the
command line. Each process is given a temporary file,
in the current directory, to receive the results. */
{
   HANDLE hTempFile;
   SECURITY_ATTRIBUTES stdOutSA = /* SA for inheritable handle. */
       { sizeof (SECURITY_ATTRIBUTES), NULL, TRUE};
   TCHAR commandLine[MAX_PATH + 100];
   STARTUPINFO startUpSearch, startUp;
   PROCESS_INFORMATION processInfo;
   DWORD exitCode, dwCreationFlags = 0;
   int iProc;
   HANDLE *hProc;  /* Pointer to an array of proc handles. */
   typedef struct {TCHAR tempFile[MAX_PATH];} PROCFILE;
   PROCFILE *procFile; /* Pointer to array of temp file names. */

#ifdef UNICODE
    dwCreationFlags = CREATE_UNICODE_ENVIRONMENT;
#endif
```

```
   if (argc < 3)
      ReportError (_T ("Usage: grepMP pattern files."), 1, FALSE);

   /* Startup info for each child search process as well as
      the child process that will display the results. */

   GetStartupInfo (&startUpSearch);
   GetStartupInfo (&startUp);

   /* Allocate storage for an array of process data structures,
      each containing a process handle and a temporary file name. */

   procFile = malloc ((argc - 2) * sizeof (PROCFILE));
   hProc = malloc ((argc - 2) * sizeof (HANDLE));

   /* Create a separate "grep" process for each file on the
      command line. Each process also gets a temporary file
      name for the results; the handle is communicated through
      the STARTUPINFO structure. argv[1] is the search pattern. */

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

      /* Create a command line of the form: grep argv[1] argv[iProc + 2] */
      /* Allow spaces in the file names. */
      _stprintf (commandLine, _T ("grep \"%s\" \"%s\""),
                 argv[1], argv[iProc + 2]);

      /* Create the temp file name for std output. */

      if (GetTempFileName (_T ("."), _T ("gtm"), 0, procFile[iProc].tempFile) == 0)
         ReportError (_T ("Temp file failure."), 2, TRUE);
```

Exemplu: căutare paralelă de șabloane

Codul sursă al aplicației grepMP (continuare)

```
      /* Set the std output for the search process. */

      hTempFile = /* This handle is inheritable */
      CreateFile (procFile[iProc].tempFile,
          /** GENERIC_READ | Read access not required **/ GENERIC_WRITE,
          FILE_SHARE_READ | FILE_SHARE_WRITE, &stdOutSA,
          CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
      if (hTempFile == INVALID_HANDLE_VALUE)
         ReportError (_T ("Failure opening temp file."), 3, TRUE);

      /* Specify that the new process takes its std output
         from the temporary file's handles. */

      startUpSearch.dwFlags = STARTF_USESTDHANDLES;
      startUpSearch.hStdOutput = hTempFile;
      startUpSearch.hStdError = hTempFile;
      startUpSearch.hStdInput = GetStdHandle (STD_INPUT_HANDLE);
      /* Create a process to execute the command line. */


      if (!CreateProcess (NULL, commandLine, NULL, NULL,
              TRUE, dwCreationFlags, NULL, NULL, &startUpSearch, &processInfo))
         ReportError (_T ("ProcCreate failed."), 4, TRUE);

      /* Close unwanted handles */
      CloseHandle (hTempFile); CloseHandle (processInfo.hThread);

      /* Save the process handle. */

      hProc[iProc] = processInfo.hProcess;
   }
```

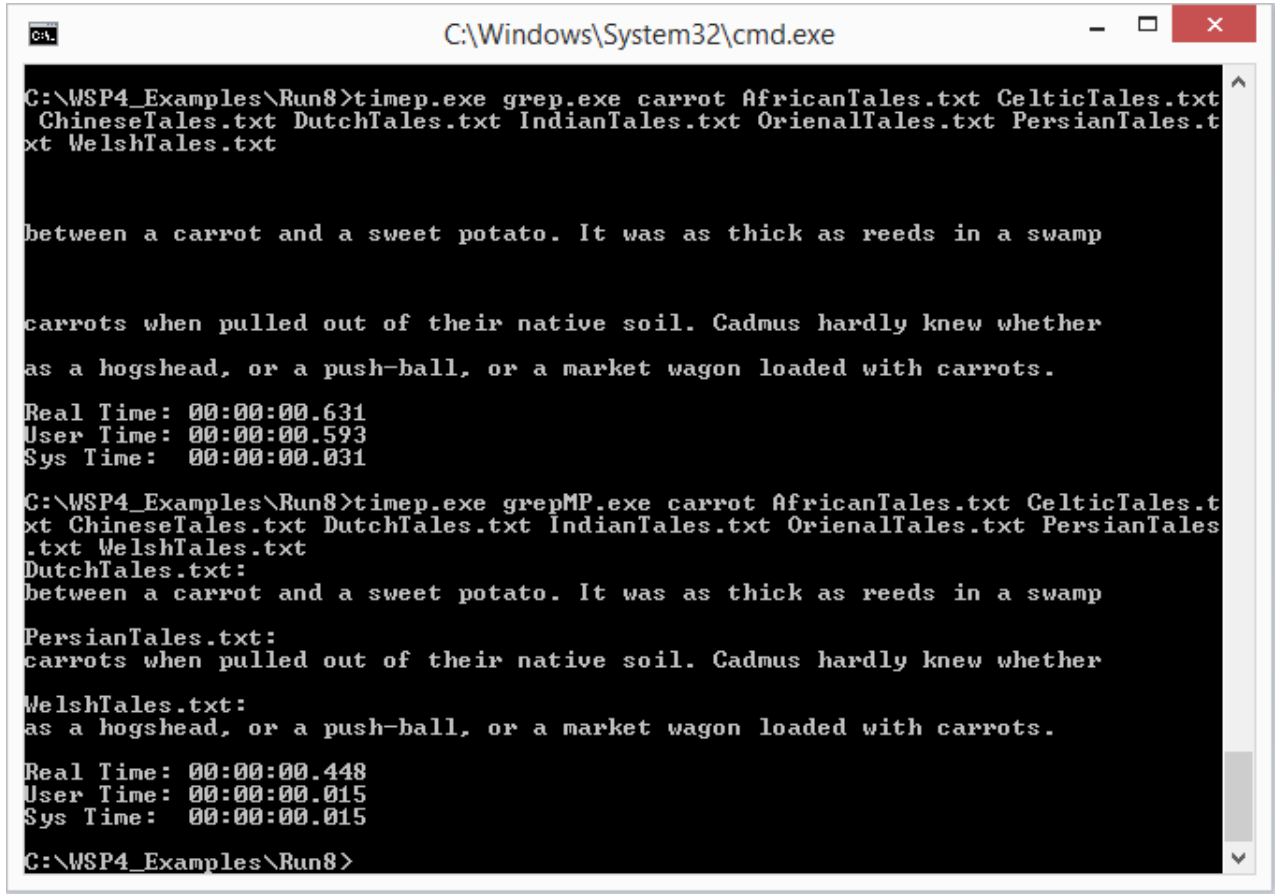
```
   /* Processes are all running. Wait for them to complete, then output
      the results - in the order of the command line file names. */
   for (iProc = 0; iProc < argc-2; iProc += MAXIMUM_WAIT_OBJECTS)
      WaitForMultipleObjects (min(MAXIMUM_WAIT_OBJECTS, argc - 2 - iProc),
                              &hProc[iProc], TRUE, INFINITE);

   /* Result files sent to std output using "cat".
      Delete each temporary file upon completion. */

   for (iProc = 0; iProc < argc - 2; iProc++) {
      if (GetExitCodeProcess (hProc[iProc], &exitCode) && exitCode == 0) {
         /* Pattern was detected - List results. */
         /* List the file name if there is more than one file to search */
         if (argc > 3) _tprintf (_T("%s:\n"), argv[iProc+2]);
            _stprintf (commandLine, _T ("cat \"%s\""), procFile[iProc].tempFile);
         if (!CreateProcess (NULL, commandLine, NULL, NULL,
                   TRUE, dwCreationFlags, NULL, NULL, &startUp, &processInfo))
            ReportError (_T ("Failure executing cat."), 0, TRUE);
         else {
            WaitForSingleObject (processInfo.hProcess, INFINITE);
            CloseHandle (processInfo.hProcess);
            CloseHandle (processInfo.hThread);
         }
      }

      CloseHandle (hProc[iProc]);
      if (!DeleteFile (procFile[iProc].tempFile))
         ReportError (_T ("Cannot delete temp file."), 6, TRUE);
   }
   free (procFile); free (hProc);
   return 0;
}
```

Exemplu: căutare paralelă de șabloane

Exemplu de rulare a aplicației grepMP

- Pentru fișiere de dimensiuni mai mari și mai mici
- Cu timpii de execuție pentru
  - varianta secvențială (grep.c) respectiv
  - Varianta concurentă cu mai multe procese
- Timpii de execuție depind și de platformă
- Procesele de căutare sunt complet independente
  - Threadurile principale ale proceselor pot fi planificate spre execuție pe procesoare (core-uri) separate
- Câștigul în performanță nu este liniar
  - Datorită încărcării suplimentare și a afișării secvențiale a rezultatelor

  dar este substanțial
  - Arată avantajul delegării sarcinilor computaționale către procese independente



Timpii de execuție ai procesului

Determinarea timpului de execuție consumat de proces considerând toate threadurile din proces

- Timpul de creare, timpul de terminare, timpul kernel, timpul utilizator

```
BOOL WINAPI GetProcessTimes(
          HANDLE     hProcess,
          LPFILETIME lpCreationTime,
          LPFILETIME lpExitTime,
          LPFILETIME lpKernelTime,
          LPFILETIME lpUserTime );
```

Handle-ul threadului. Este necesar dreptul de acces **PROCESS_QUERY_INFORMATION** sau **PROCESS _QUERY_LIMITED_INFORMATION**

Structură FILETIME care primește timpul de creare a threadului

Structură FILETIME care primește timpul de terminare a threadului

Structură FILETIME care primește timpul de execuție în mod kernel al threadului

Structură FILETIME care primește timpul de execuție în mod utilizator al threadului

- Se consideră timpii tuturor threadurilor din proces – chiar și threadurile 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

- Structura FILETIME are două componente de 32 de biți (exprimă timpul scurs de la data de 1 ianuarie 1601, Greenwich)

```
typedef struct _FILETIME {
            DWORD dwLowDateTime;
            DWORD dwHighDateTime;
} FILETIME, *PFILETIME;
```

  - Pentru a calcula timpul total scrus de la crearea threadului până la terminare se poate crea o uniune cu LONGLONG pentru a facilita calcularea diferenței de timp
    - Ex. Dacă procesul a petrecut 1 sec în modul kernel → componenta *lpKernelTime* va fi completată cu valoarea de 10 milioane pe 64 de biți (unități de timp de 100 de nanosecunde)

Timpii de execuție ai procesului

Programul `timep` afișează timpii reali și timpii sistem

```
/* timeprint: Execute a command line and display the time
   (elapsed, kernel, user) required. */

/* This program illustrates:
1. Creating processes.
2. Obtaining the elapsed times.
3. Converting file times to system times.
4. Displaying system times. */

#include "Everything.h"
```

```
int _tmain(int argc, LPTSTR argv[]) {
   STARTUPINFO startUp;
   PROCESS_INFORMATION procInfo;
   union {/* Structure required for file time arithmetic. */
      LONGLONG li;
      FILETIME ft;
   } createTime, exitTime, elapsedTime;

   FILETIME kernelTime, userTime;
   SYSTEMTIME elTiSys, keTiSys, usTiSys;
   LPTSTR targv, cLine = GetCommandLine();
   HANDLE hProc;

   targv = argv[1];
   /*  Skip past the first blank-space delimited token on the command line */
   if (argc <= 1 || NULL == targv)
      ReportError(_T("Usage: timep command ..."), 1, FALSE);

   GetStartupInfo(&startUp);
```

```
   /* Execute the command line and wait for the process to complete. */
   if (!CreateProcess(NULL, targv, NULL, NULL, TRUE,
            NORMAL_PRIORITY_CLASS, NULL, NULL, &startUp, &procInfo))
      ReportError(_T("\nError starting process. %d"), 3, TRUE);
   hProc = procInfo.hProcess;
   if (WaitForSingleObject(hProc, INFINITE) != WAIT_OBJECT_0)
      ReportError(_T("Failed waiting for process termination. %d"), 5, TRUE);

   if (!GetProcessTimes(hProc, &createTime.ft,
                        &exitTime.ft, &kernelTime, &userTime))
      ReportError(_T("Can not get process times. %d"), 6, TRUE);

   elapsedTime.li = exitTime.li - createTime.li;
   FileTimeToSystemTime(&elapsedTime.ft, &elTiSys);
   FileTimeToSystemTime(&kernelTime, &keTiSys);
   FileTimeToSystemTime(&userTime, &usTiSys);

   _tprintf(_T("Real Time: %02d:%02d:%02d.%03d\n"), elTiSys.wHour, elTiSys.wMinute,
                                          elTiSys.wSecond, elTiSys.wMilliseconds);
   _tprintf(_T("User Time: %02d:%02d:%02d.%03d\n"), usTiSys.wHour, usTiSys.wMinute,
                                          usTiSys.wSecond, usTiSys.wMilliseconds);
   _tprintf(_T("Sys Time:  %02d:%02d:%02d.%03d\n"), keTiSys.wHour, keTiSys.wMinute,
                                          keTiSys.wSecond, keTiSys.wMilliseconds);

   CloseHandle(procInfo.hThread);
   CloseHandle(procInfo.hProcess);

   return 0;
}
```

Job-uri

Job-ul este un container de procese

- Permite tratarea unui grup de procese ca o singură entitate

Utilitate

- Definirea unor restricții asupra setului de procese din job
  - Restricții de limite de bază: previn monopolizarea resurselor sistemului de către procese din job
  - Restricții de interfața: previn modificarea interfeței utilizator de către procese din job
  - Restricții de securitate: previn accesarea unor resurse

Procesul care face parte dintr-un job nu poate părăsi jobul

- Nu poate ieși de sub influența restricțiilor definite pentru toate procesele care fac parte din acel job
- Procesele fii create de joburile din cadrul unui proces fac parte implicit din job

Jobul este un obiect kernel

- se creează cu funcția CreateJobObject

```
HANDLE CreateJobObject( PSECURITY_ATTRIBUTES psa, PCTSTR pszName);
```

- Când nu mai este nevoie de job se închide handle-ul jobului
  - Nu implică terminarea proceselor din job
    - Jobul este marcat spre ștergere, dar va continua să existe până se termină toate procesele din job
  - După închidere jobul nu mai poate fi accesat / referit de nici un proces

- Terminarea tuturor proceselor din job: TerminateJobObject

```
BOOL TerminateJobObject( HANDLE hJob, UINT uExitCode);
```

Job-uri: Exemplu

Exemplu: crearea unui job și definirea unor restricții

- Testarea dacă procesul aparține deja de un job – dacă da, se afișează un mesaj de eroare și se revine

```
BOOL IsProcessInJob( HANDLE hProcess, HANDLE hJob, PBOOL pbInJob);
```

- Se creează un nou obiect kernel de tip job

```
HANDLE CreateJobObject( PSECURITY_ATTRIBUTES psa, PCTSTR pszName);
```

- Se definesc restricțiile asupra proceselor din job

```
BOOL SetInformationJobObject( HANDLE hJob, JOBOBJECTINFOCLASS  obObjectInformationClass,
                              PVOID  pJobObjectInformation, DWORD cbJobObjectInformationSize);
```

- Se creează procesul fiu care va face parte din job
  - Crearea procesului fiu – cu threadul principal suspendat pentru a pemite plasarea procesului fiu în job înainte de a se executa
  - Se plasează procesul fiu în job

```
BOOL AssignProcessToJobObject( HANDLE hJob, HANDLE hProcess);
```

  - Se rezumă threadul principal (se scoate din starea de suspendare)

- Se așteaptă terminarea procesului sau îndeplinirea unor condiții impuse la restricții

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

- Se afișează informații statistice legate de timpii de execuție a procesului din job

```
BOOL GetProcessTimes( HANDLE hProcess, LPFILETIME lpCreationTime, LPFILETIME lpExitTime,
                      LPFILETIME lpKernelTime, LPFILETIME lpUserTime );
```

- Se închid handle-urile și se eliberează resursele

Job-uri: Exemplu

```
void StartRestrictedProcess() {
   // Check if the process is already associated with a job.
   // If it is, there is no way to switch to another job.
   BOOL bInJob = FALSE;
   IsProcessInJob(GetCurrentProcess(), NULL, &bInJob);
   if (bInJob) {
      MessageBox(NULL, TEXT("Process already in a job"),
                 TEXT(""), MB_ICONINFORMATION | MB_OK);
      return;
   }

   // Create a job kernel object.
   HANDLE hjob = CreateJobObject(NULL, TEXT("RestrictedProcessJob"));



   // Place some restrictions on processes in the job.
   // First, set some basic restrictions.
   JOBOBJECT_BASIC_LIMIT_INFORMATION jobli = { 0 };
   // The process always runs in the idle priority class.
   jobli.PriorityClass = IDLE_PRIORITY_CLASS;
   // The job cannot use more than 1 second of CPU time.
   jobli.PerJobUserTimeLimit.QuadPart = 10000; // 1 sec in 100-ns intervals

   jobli.LimitFlags = JOB_OBJECT_LIMIT_PRIORITY_CLASS | JOB_OBJECT_LIMIT_JOB_TIME;
   SetInformationJobObject(hjob, JobObjectBasicLimitInformation, &jobli, sizeof(jobli));
```

Job-uri: Exemplu (2)

```
// 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);
```

Job-uri: Exemplu (3)

```
  // Wait for the process to terminate or
   // for all the job's allotted CPU time to be used.
   HANDLE h[2];
   h[0] = pi.hProcess;
   h[1] = hjob;
   DWORD dw = WaitForMultipleObjects(2, h, FALSE, INFINITE);

   switch (dw - WAIT_OBJECT_0) {
      case 0: // The process has terminated...
           break;
      case 1: // All of the job's allotted CPU time was used...
           break;
   }
   FILETIME CreationTime;
   FILETIME ExitTime;
   FILETIME KernelTime;
   FILETIME UserTime;
   TCHAR szInfo[MAX_PATH];
   GetProcessTimes(pi.hProcess, &CreationTime, &ExitTime, &KernelTime, &UserTime);

   StringCchPrintf(szInfo, _countof(szInfo), TEXT("Kernel = %u | User = %u\n"),
   KernelTime.dwLowDateTime / 10000, UserTime.dwLowDateTime / 10000);
   MessageBox(GetActiveWindow(), szInfo, TEXT("Restricted Process times"),
   MB_ICONINFORMATION | MB_OK);
   // Clean up properly.
   CloseHandle(pi.hProcess);
   CloseHandle(hjob);
}
```

Gestiunea proceselor

Corespondențele dintre funcțiile CRT și apelurile Win32 API – pentru gestiunea proceselor

Process and Environment Control Routines

CRT	Win32 API
abort	none
assert	none
atexit	none
_cexit	none
_c_exit	none
_exec functions	none
exit	ExitProcess
_exit	ExitProcess
getenv	GetEnvironmentVariable
_getpid	GetCurrentProcessId
longjmp	none
_onexit	none
perror	FormatMessage
_putenv	SetEnvironmentVariable
raise	RaiseException
setjmp	none
signal (ctrl-c only)	SetConsoleCtrlHandler
_spawn functions	CreateProcess
system	CreateProcess

Materiale de studiu

- Johnson HART, Windows System Programming, 4th edition, Addison Wesley, 2010
  - Capitolul 6
  - Exemplele incluse sunt din cartea de mai sus – vezi arhiva de pe moodle: WSP4_Examples.zip