Obiecte kernel în WIN32 API

CURS NR. 3/2

Kernel și obiecte kernel

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 resurselor sistem – procese, threaduri, fișiere, joburi, evenimente, mutex-uri, 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
      - de ex. atribute de securitate, contor de utilizare, etc.
    - Câmpuri specifice tipului de obiect
      - Ex. Un proces are identificator de proces, prioritate, cod de terminare, etc.
        un fișier are mod de deschidere, pointer de fișier (poziție în cadrul fișierului), mod de partajare, etc.

Obiecte utilizator sau obiecte grafice - GDI (Graphical Device Interface)

- Nu sunt obiecte kernel
- Ex. Ferestre, meniuri, cursorul mousului, fonturi, etc.

Kernel și obiecte kernel

Obiectele kernel sunt deținute de kernel

- Doar kernelul poate accesa obiectul kernel
- Aplicațiile nu au acces direct la componentele obiectului kernel
- Windows furnizează un set de funcții prin care aplicația poate cere kernelului să creeze și să manipuleze obiecte kernel

- Exemple de prototipuri de funcții pentru crearea obiectelor kernel:

```
HANDLE CreateFile(
           PCTSTR pszFileName,
           DWORD dwDesiredAccess,
           DWORD dwShareMode,
           PSECURITY_ATTRIBUTES psa,
           DWORD dwCreationDisposition,
           DWORD dwFlagsAndAttributes,
           HANDLE hTemplateFile);
```

Exemplu:

```
//  Opens the existing file.
   hFile = CreateFile(argv[1], // file name
       GENERIC_READ,           // open for reading
       0,                      // do not share
       NULL,                   // default security
       OPEN_EXISTING,          // existing file only
       FILE_ATTRIBUTE_NORMAL,  // normal file
       NULL);                  // no template
```

Kernel și obiecte kernel

Obiectele kernel sunt deținute de kernel

- Doar kernelul poate accesa obiectul kernel
- Aplicațiile nu au acces direct la componentele obiectului kernel
- Windows furnizează un set de funcții prin care aplicația poate cere kernelului să creeze și să manipuleze obiecte kernel

- Exemple de prototipuri de funcții pentru crearea obiectelor kernel:

```
HANDLE CreateFileMapping(
            HANDLE hFile,
            PSECURITY_ATTRIBUTES psa,
            DWORD flProtect,
            DWORD dwMaximumSizeHigh,
            DWORD dwMaximumSizeLow,
            PCTSTR pszName);
```

Exemplu:

```
// Create a file mapping object for the file
hMapFile = CreateFileMapping(
    hFile,            // current file handle
    NULL,             // default security
    PAGE_READWRITE,   // read/write permission
    0,                // size of mapping object, high
    1024,             // size of mapping object, low
    TEXT("Mapare")); // name of mapping object
```

Kernel și obiecte kernel

Obiectele kernel sunt deținute de kernel

- Doar kernelul poate accesa obiectul kernel
- Aplicațiile nu au acces direct la componentele obiectului kernel
- Windows furnizează un set de funcții prin care aplicația poate cere kernelului să creeze și să manipuleze obiecte kernel

- Exemple de prototipuri de funcții pentru crearea obiectelor kernel:

```
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);
```

```
HANDLE CreateThread(
            PSECURITY_ATTRIBUTES psa,
            size_t dwStackSize,
            LPTHREAD_START_ROUTINE pfnStartAddress,
            PVOID pvParam,
            DWORD dwCreationFlags,
            PDWORD pdwThreadId);
```

```
HANDLE CreateSemaphore(
            PSECURITY_ATTRIBUTES psa,
            LONG lInitialCount,
            LONG lMaximumCount,
            PCTSTR pszName);
```

Obiecte kernel

Handle-ul obiectului kernel

- La crearea unui nou obiect kernel – funcția de creare returnează aplicației un handle
- Handle-ul identifică în mod unic obiectul kernel în cadrul procesului
- Funcțiile de gestiune a obiectului kernel primesc ca și argument handle-ul obiectului

- Handle-ul este relativ la proces
  - Fiecare proces menține un tabel de handle-uri – în care stochează referințele la obiectele kernel accesibile pentru proces
  - Handle-ul este de fapt un index în acest tabel

Contor de utilizare

- Componentă a structurii obiect kernel
- Obiectele kernel sunt proprietatea kernelului – nu sunt deținute de procesele care le creează
  - La crearea obiectului kernel contorul de utilizări este inițializat la 1
  - Dacă un alt proces referă același obiect kernel → se incrementează contorul de utilizări
  - Dacă un proces închide handle-ul prin care referă obiectul kernel → se decrementează contorul de utilizări
    - La terminarea unui proces toate obiectele kernel referite de proces – din tabela handle-urilor definite per proces – vor avea contorul de utilizări decrementate
  - Dacă contorul de utilizări ajunge la 0 (zero), atunci obiectul kernel este distrus și memoria alocată obiectului este eliberată

Obiecte kernel

Atribute de securitate

- Descriptorul de securitate are rolul de a proteja obiectul kernel
  - Specifică drepturile de acces la obiect pentru utilizatori și grupuri de utilizatori
- Marea majoritate a funcțiilor care creează obiecte kernel au ca și parametru un pointer către o structură care conține atribute de securitate pentru obiect (spre deosebire de obiecte utilizator - care nu au atribute de securitate)
  - Structura se numește **SECURITY_ATTRIBUTES**

```
typedef struct _SECURITY_ATTRIBUTES {
           DWORD  nLength;
           LPVOID lpSecurityDescriptor;
           BOOL   bInheritHandle;
} SECURITY_ATTRIBUTES;
```

Dimensiunea structurii

Proprietatea de moștenibilitate

- În multe cazuri se specifică NULL ca valoare a parametrului pentru atribute de securitate (`lpSecurityDescriptor`)
  - Obiectul se creează cu atribute de securitate implicite – pe baza token-ului de acces al procesului curent

- Exemplu de utilizare a atributelor de securitate

```
SECURITY_ATTRIBUTES sa;

sa.nLength = sizeof(sa); // Used for versioning
sa.lpSecurityDescriptor = pSD; // Address of an initialized SD
sa.bInheritHandle = FALSE; // Discussed later

HANDLE hFileMapping = CreateFileMapping(INVALID_HANDLE_VALUE, // use paging file
                                        &sa,
                                        PAGE_READWRITE, 0, 1024, TEXT("MyFileMapping"));
```

Obiecte kernel

Atribute de securitate

- Exemplu – crearea unui obiect kernel mutex accesibil tuturor

```
/ Example: SECURITY_ATTRIBUTES
// with security descriptor
#include <windows.h>
#include <stdio.h>

BOOL CreateNullDacl(SECURITY_ATTRIBUTES *sa) {
   PSECURITY_DESCRIPTOR pSD;

   if (sa == NULL) return FALSE;

   pSD = (PSECURITY_DESCRIPTOR)LocalAlloc(LPTR, SECURITY_DESCRIPTOR_MIN_LENGTH);
   if (pSD == NULL) return FALSE;

   if (!InitializeSecurityDescriptor(pSD, SECURITY_DESCRIPTOR_REVISION))
      return FALSE;

   // Add a null DACL to the security descriptor.
   if (!SetSecurityDescriptorDacl(pSD, TRUE, (PACL)NULL, FALSE))
      return TRUE;

   sa->nLength = sizeof(SECURITY_ATTRIBUTES);
   sa->lpSecurityDescriptor = pSD;
   sa->bInheritHandle = TRUE;

   return TRUE;
}
```

```
void FreeNullDacl(SECURITY_ATTRIBUTES *sa) {
   if (sa == NULL) return;

   if (sa->lpSecurityDescriptor) {
      LocalFree(sa->lpSecurityDescriptor);
      sa->lpSecurityDescriptor = NULL;
   }
}

int main() {
   SECURITY_ATTRIBUTES sa;
   HANDLE hMutex;

   CreateNullDacl(&sa);

   hMutex = CreateMutex(&sa, FALSE, TEXT("my.mutex"));
   // Everyone can use this mutex
}
```

Obiecte kernel

Tabela handle-urilor

- Fiecare proces menține o tabelă de handle-uri pentru obiectele kernel
  - Structura și gestiunea tabelei de handle-uri nu este documentată – se oferă o viziune de ansamblu...
  - Tablou de structuri în care fiecare element conține un pointer către obiectul kernel, o mască de acces și flag-uri

Crearea obiectelor kernel

- La inițializarea unui nou proces tabela de handle-uri al procesului este goală
- La crearea unui obiect kernel
  - Kernelul alocă memorie pentru obiectul kernel și inițializează obiectul
  - Kernelul caută o intrare liberă în tabela handle-urilor – la intrarea găsită efectuează
    - Inițializează componenta pointer a structurii la adresa de memorie a structurii obiectului kernel
    - Masca de acces este setată să permită acces deplin
    - Setează flag-ul în confirmitate cu valoarea componentei din structura care descrie atributele de securitate (`bInheritHandle`)
  - Funcțiile de creare returnează handleuri relative la proces (handle-ul este indexul din tabela handle-urilor)
    - Exemplu de valori: 4, 8, etc.
    - NULL înseamnă eșec în crearea obiectului (există și excepții: CreateFile returnează -1 (INVALID_HANDLE_VALUE) la eșec)

```
HANDLE hMutex = CreateMutex(…);
if (hMutex == INVALID_HANDLE_VALUE) {
   // We will never execute this code because
   // CreateMutex returns NULL if it fails.
}
```

```
HANDLE hFile = CreateFile(…);
if (hFile == NULL) {
   // We will never execute this code because CreateFile
   // returns INVALID_HANDLE_VALUE (-1) if it fails.
}
```

Obiecte kernel

Închiderea obiectelor kernel

```
BOOL CloseHandle(HANDLE hobject);
```

- Funcția CloseHandle
  - indică faptul că procesul nu mai dorește să refere obiectul kernel
  - Verifică în tabela handle-urilor dacă handle-ul este valid
    - Dacă handle-ul este invalid – CloseHandle returnează FALSE și GetLastError returnează ERROR_INVALID_HANDLE
  - Decrementează contorul de utilizări al obiectului
    - Dacă contorul scade la 0, obiectul este distrus și eliminat din memorie
  - Înainte de revenire din CloseHandle, funcția elimină referința la obiect din intrarea indicată de handle din tabela handle-urilor
    - Handle-ul devine invalid
  - Recomandare: variabila în care stocăm handle-ul returnat de funcția de creare a obiectului kernel să fie setat la NULL după închiderea handle-ului, pentru a evita situațiile de genul
    - Refolosirea variabilei într-o funcție care manipulează obiecte kernel
      - variabila referă un handle invalid!!
      - Variabila referă un handle realocat pentru alt obiect kernel !!
- La terminarea procesului toate obiectele kernel referite sunt închise – contorul de referiri decrementat
  - Resursele procesului dealocate

Partajarea obiectelor kernel

Threadurile din procese distincte pot avea nevoie de partajarea unor obiecte kernel

Motivație

- Partajarea unor date între procese distincte – prin fișiere mapate în memorie
- Partajarea unor date între procese de pe sisteme distincte conectate la rețea – prin fișiere pipe cu nume
- Sincronizare execuției thredurilor din procese distincte – folosind semafoare, mutex-uri, etc.

Problema

- Handle-urile sunt relative la proces
  - Pentru securitate și robustețe

Soluții

- Moștenirea handle-ului
- Obiecte cu nume
- Duplicarea handle-ului

Partajarea obiectelor kernel

Moștenirea handle-ului

- Doar în cazul proceselor aflate în relația părinte-fiu (procese înrudite)
- Procesul părinte trebuie să indice sistemului la crearea obiectului kernel că dorește ca handle-ul obiectului să fie moștenibil
  - Prin folosirea atributelor de securitate
    - Dacă **bInheritHandles** este setat la TRUE, atunci flagul asociat handle-ului din tabela de handle-uri este setat la 1, dacă e FALSE, flagul este 0
    - Dacă se folosesc atribute de securitate implicite, NULL pentru parametrul PSECURITY_ATTRIBUTES în funcția de creare a obiectului → flagul este implicit 0 și handle-ul nu este moștenibil

- La creare procesului fiu dacă parametrul **bInheritHandles** din funcția CreateProcess este
  - TRUE, atunci fiul moștenește handle-urile moștenibile din tabela de handle-uri al procesului părinte
    - adică sistemul copiază intrările asociate handle-urilor moștenibile din tabela de handle-uri al părintelui în tabela fiului
      - exact pe pozițiile la care se aflau în părinte
    - Sistemul incrementează contorul de utilizare pentru obiectele a căror handle-uri au fost moștenite
  - FALSE, atunci fiul este creat cu tabela handle-urilor goală – fiul nu moștenește de la părinte

```
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE; // Make the returned handle inheritable.
```

- Schimbarea și interogarea flagului asociat handle-ului

```
BOOL SetHandleInformation( HANDLE hObject, DWORD dwMask, DWORD dwFlags);

BOOL GetHandleInformation( HANDLE hObject, PDWORD pdwFlags);
```

Partajarea obiectelor kernel

Numirea obiectelor

- Metodă de partajare a obiectelor kernel între procese oarecare (chiar și între procese neînrudite)

Multe obiectele kernel pot fi create **cu nume** sau **anonime** (fără nume)

- Exemple de funcții de creare a obiectelor kernel care permit specificarea unui nume – prin parametrul pszName

```
HANDLE CreateFileMapping(
               HANDLE hFile,
               PSECURITY_ATTRIBUTES psa,
               DWORD flProtect,
               DWORD dwMaximumSizeHigh,
               DWORD dwMaximumSizeLow,
               PCTSTR pszName);
```

```
HANDLE CreateEvent(
               PSECURITY_ATTRIBUTES psa,
               BOOL bManualReset,
               BOOL bInitialState,
               PCTSTR pszName);
```

```
HANDLE CreateSemaphore(
               PSECURITY_ATTRIBUTES psa,
               LONG lInitialCount,
               LONG lMaximumCount,
               PCTSTR pszName);
```

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

- Dacă pszNume este NULL (cum e foarte des cazul) → obiectul kernel este anonim (fără nume)
- Dacă pszNume nu este NULL – este un string: numele obiectului kernel → obiectul kernel are nume
  - Spațiul de nume pentru obiecte kernel este partajat de toate tipurile de obiecte
    - Nu putem crea două obiecte kernel cu același nume – chiar dacă sunt de tipuri diferite
      - Exemplu de creare eșuată datorită faptului că un alt obiect cu același nume există deja

```
HANDLE hMutex = CreateMutex(NULL, FALSE, TEXT("JeffObj"));

HANDLE hSem = CreateSemaphore(NULL, 1, 1, TEXT("JeffObj"));
DWORD dwErrorCode = GetLastError();
```

Obiect cu numele “JeffObj” există deja

CreateSemaphore eșuează: returnează NULL
GetLastError: 6 (ERROR_INVALID_HANDLE)

Partajarea obiectelor kernel

Partajarea obiectelor pe baza numelui

- Exemplu:
  - Procesul A ceează cu succes un obiect kernel cu numele “JeffMutex”

```
HANDLE hMutexProcessA = CreateMutex(NULL, FALSE, TEXT("JeffMutex"));
```

Prin numirea obiectului partajarea este posibilă chiar dacă obiectul este creat nemoștenibil

  - Procesul B (independent de A - neînrudite) dorește ulterior să acceseze obiectul cu numele “JeffMutex”

```
HANDLE hMutexProcessB = CreateMutex(NULL, FALSE, TEXT("JeffMutex"));
```

Apelul nu va crea un nou obiect kernel, ci creează un handle relativ la procesul B către obiectul deja existent

- Funcționare
  - Sistemul caută în spațiul numelor un obiect cu numele specificat (în cazul nostru: “JeffMutex”)
    - Dacă găsește, verifică dacă este de tipul dorit (cazul nostru: Mutex)
      - Dacă da, se verifică drepturile procesului B asupra obiectului
    - Dacă oricare din aceste condiții nu este îndeplinită funcția Create* eșuează și returnează NULL
  - Sistemul caută o intrare liberă în tabela handle-urilor din procesul B, inițializează structura să refere obiectul kernel regăsit și incrementează contorul de referiri al obiectului kernel
    - Funcția returnează procesului B handle-ul către obiectului dorit – handle relativ la procesul B
- Observație
  - Valorile handle-urilor (`hMutexProcessA` și `hMutexProcessB`) sunt de regulă diferite (nu este necesar să fie aceleași)
  - Funcțiile Create* returnează handle către obiectele dorite având toate drepturile de acces setate
    - Dacă se dorește restricționarea accesului → folosim funcțiile Create*Ex (varianta extinsă)

Partajarea obiectelor kernel

Partajarea obiectelor pe baza numelui

- Exemplu:
  - Procesul A ceează cu succes un obiect kernel cu numele “JeffMutex”

```
HANDLE hMutexProcessA = CreateMutex(NULL, FALSE, TEXT("JeffMutex"));
```

  - Procesul B (independent de A - neînrudite) dorește ulterior să acceseze obiectul cu numele “JeffMutex”

```
HANDLE hMutexProcessB = CreateMutex(NULL, FALSE, TEXT("JeffMutex"));
```

- Funcțiile Create* creează un obiect kernel nou dacă acesta nu există deja
  - Altfel deschid un handle către obiectul deja existent
  - Aplicația poate testa dacă a creat obiectul sau doar a obținut un handle către obiect interogând GetLastError imediat după funcția de creare

Dacă obiectul referit există deja, atunci sistemul ignoră parametri referitori la atributele de securitate și moștenibilitate

```
HANDLE hMutex = CreateMutex(&sa, FALSE, TEXT("JeffObj"));
if (GetLastError() == ERROR_ALREADY_EXISTS) {
    // Opened a handle to an existing object.
    // sa.lpSecurityDescriptor and the second parameter
    // (FALSE) are ignored.
}
else {
    // Created a brand new object.
    // sa.lpSecurityDescriptor and the second parameter
    // (FALSE) are used to construct the object.
}
```

Partajarea obiectelor kernel

Partajarea obiectelor pe baza numelui (cont.)

- O metodă alternativă pentru partajare pe baza numelui este utilizarea funcțiilor Open* în loc de funcțiile Create*
  - Create* vs. Open*
    - Dacă obiectul nu există deja Create* îl creează, Open* eșuează
  - Toate funcțiile Open* au același prototip

```
HANDLE OpenMutex(
                DWORD dwDesiredAccess,
                BOOL bInheritHandle,
                PCTSTR pszName);
```

```
HANDLE OpenEvent(
                DWORD dwDesiredAccess,
                BOOL bInheritHandle,
                PCTSTR pszName);
```

```
HANDLE OpenSemaphore(
                DWORD dwDesiredAccess,
                BOOL bInheritHandle,
                PCTSTR pszName);
```

```
HANDLE OpenFileMapping(
                DWORD dwDesiredAccess,
                BOOL bInheritHandle,
                PCTSTR pszName);
```

```
HANDLE OpenJobObject(
                DWORD dwDesiredAccess,
                BOOL bInheritHandle,
                PCTSTR pszName);
```

    - Ultimul parametru este numele obiectului (pszName) și NU poate fi NULL

  - Funcționalitate
    - Sistemul caută în spațiul de nume obiectul cu numele specificat
      - Dacă nu găsește, funcția eșuează și returnează NULL iar GetLastError este 2 (ERROR_FILE_NOT_FOUND)
      - Dacă regăsește un obiect numele dat
        - De tip diferit, funcția returnează NULL iar GetLastError este 6 (ERROR_INVALID_HANDLE)
        - De același tip ca cel căutat
          - Se verifică drepturile de acces
          - Se actualizează tabela handle –urilor
          - Se incrementează contorul de utilzări al obiectului
          - Se returnează handle-ul către obiectul cerut

Partajarea obiectelor kernel

Partajarea obiectelor pe baza numelui (cont.)

- Cazul de utilizare cel mai des întâlnit pentru obiecte kernel cu nume este prevenirea rulării simultane a mai multor instanțe ale aceleiași aplicații
  - Se creează un obiect kernel (poate fi de orice tip) cu una din funcțiile Create*
    - La revenirea din funcție se apelează imediat GetLastError
      - dacă valoarea returnată este ERROR_ALREADY_EXISTS, atunci o altă instanță a aplicației este deja în rulare
      - și instanța de față se poate termina fără să-și continue execuția
  - Exemplu:

```
int _tmain(int argc, TCHAR *argv[]) {

    HANDLE h = CreateMutex(NULL, FALSE, TEXT("{FA531CC1-0497-11d3-A180-00105A276C3E}"));

    if (GetLastError() == ERROR_ALREADY_EXISTS) {
       // There is already an instance of this application running.
       // Close the object and immediately return.
       CloseHandle(h);
       return(0);
    }

    // This is the first instance of this application running.
    // ...
    // Before exiting, close the object.
    CloseHandle(h);
    return(0);
}
```

Partajarea obiectelor kernel

Dublarea handle-ului de obiect kernel

- Ideea de bază
  - Se copiază o intrare din tabela handle-urilor unui proces în tabela handle-urilor altui proces

```
BOOL DuplicateHandle(
                HANDLE hSourceProcessHandle,
                HANDLE hSourceHandle,
                HANDLE hTargetProcessHandle,
                PHANDLE phTargetHandle,
                DWORD dwDesiredAccess,
                BOOL bInheritHandle,
                DWORD dwOptions);
```

Handle către obiectul kernel **proces sursă**

Handle către obiectul kernel **care va fi partajat**

Handle către obiectul kernel **proces destinație**

Adresa variabilei handle unde se va stoca handle-ul către obiectul kernel copiat în tabela procesului destinație

Masca pentru drepturile de acces

mască identică cu cea folosită în procesul sursă
→ Se ignoră parametrul `dwDesiredAccess`

Proprietatea de moștenibilitate

Modul de dublare: 0 sau o combinație dintre **DUPLICATE_SAME_ACCESS** și **DUPLICATE_CLOSE_SOURCE**

După copiere, handle-ul din sursă se închide
→ Obiectul este predat altui proces.
→ Contorul de utilizări nu este afectat

  - Cazul general: trei procese implicate
    - **Procesul C** (care apelează funcția DuplicateHandle) copiază din **procesul S** (source) în **procesul T** (target) handle-ul obiectului kernel deținut de procesul S
      - Handle-urile `hSourceProcessHandle` și `hTargetProcessHandle` sunt relative la procesul C
      - Handle-ul obiectului kernel `hSourceHandle` este relativ la procesul sursă S
    - Problema: procesul destinație T nu primește nici o notificare că un nou obiect kernel i-a devenit accesibil
      - Procesul C trebuie să-l anunțe pe T prin intermediul unor mecanisme de comunicare între procese (IPC)
  - Cazul cel mai popular: două procese implicate
    - Procesul S care deține un obiectul kernel, dorește să partajeze obiectul cu procesul T, și deci S apelează DuplicateHandle

Partajarea obiectelor kernel

Dublarea handle-ului de obiect kernel

- Exemplul 1

```
// All of the following code is executed by Process S.
// Create a mutex object accessible by Process S.
HANDLE hObjInProcessS = CreateMutex(NULL, FALSE, NULL);
// Get a handle to Process T's kernel object.
HANDLE hProcessT = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dwProcessIdT);
HANDLE hObjInProcessT; // An uninitialized handle relative to Process T.

   // Give Process T access to our mutex object.
DuplicateHandle(GetCurrentProcess(), hObjInProcessS, hProcessT, &hObjInProcessT, 0, FALSE, DUPLICATE_SAME_ACCESS);

// Use some IPC mechanism to get the handle value of hObjInProcessS into Process T.
...
// We no longer need to communicate with Process T.
CloseHandle(hProcessT);
...
// When Process S no longer needs to use the mutex, it should close it.
CloseHandle(hObjInProcessS);
```

GetCurrentProcess returnează handle-ul procesului curent (procesul sursă)

La revenirea din funcție, `hObjInProcessT` conține handle-ul relativ la procesul T al obiectului partajat

Handle către obiectul mutex creat mai sus

Handle către procesul destinație obținut prin OpenProcess (pe baza id-ului `dwProcessIdT`)

- Observație: La revenirea din funcție, `hObjInProcessT` conține **handle-ul relativ la procesul T** al obiectului partajat
  - Atenție!!! Procesul S să nu închidă handle-ul `hObjInProcessT`, deoarece nu este un handle relativ la tabela sa de handle-uri
    - Procesul S poate avea în tabela handle-urilor pe poziția indicată de `hObjInProcessT` un handle invalid, sau un handle către un alt obiect

```
// Process S should never attempt to close the duplicated handle.
CloseHandle(hObjInProcessT);
```

Partajarea obiectelor kernel

Dublarea handle-ului de obiect kernel

- Exemplul 2:
  - Procesul S a creat un obiect kernel de tipul mapare de fișier cu drepturi de citire și scriere
  - Procesul S apelează o funcție care va efectua doar operații de citire asupra obiectului
    - Pentru a ne asigura că funcția nu va modifica în mod accidental obiectul → transmitem un handle către obiect doar cu drept de citire

```
int _tmain(int argc, TCHAR *argv[]) {

   // Create a file-mapping object; the handle has read/write access.
   HANDLE hFileMapRW = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0, 10240, NULL);

   // Create another handle to the file-mapping object;
   // the handle has read-only access.
   HANDLE hFileMapRO;
   DuplicateHandle(GetCurrentProcess(), hFileMapRW, GetCurrentProcess(), &hFileMapRO, FILE_MAP_READ, FALSE, 0);

   // Call the function that should only read from the file mapping.
   ReadFromTheFileMapping(hFileMapRO);

   // Close the read-only file-mapping object.
   CloseHandle(hFileMapRO);
   // We can still read/write the file-mapping object using hFileMapRW.
   ...
   // When the main code doesn't access the file mapping anymore, close it.
   CloseHandle(hFileMapRW);

   return(0);
}
```

Dublarea handle-ului în cadrul aceluiași proces

Dreptul de acces restricționat doar la citire

Materiale de studiu

- Jeffrey RICHTER & Christophe NASARRE, Windows via C, C++, 5th edition, Microsoft Press, 2008
  - Capitolul 3

- Kernel Objects, MSDN: https://msdn.microsoft.com/en-us/library/windows/desktop/ms724485(v=vs.85).aspx