Noyau d'un système d'exploitation
INF2610

Chapitre 8 : Cas de Windows

Département de génie informatique et génie logiciel

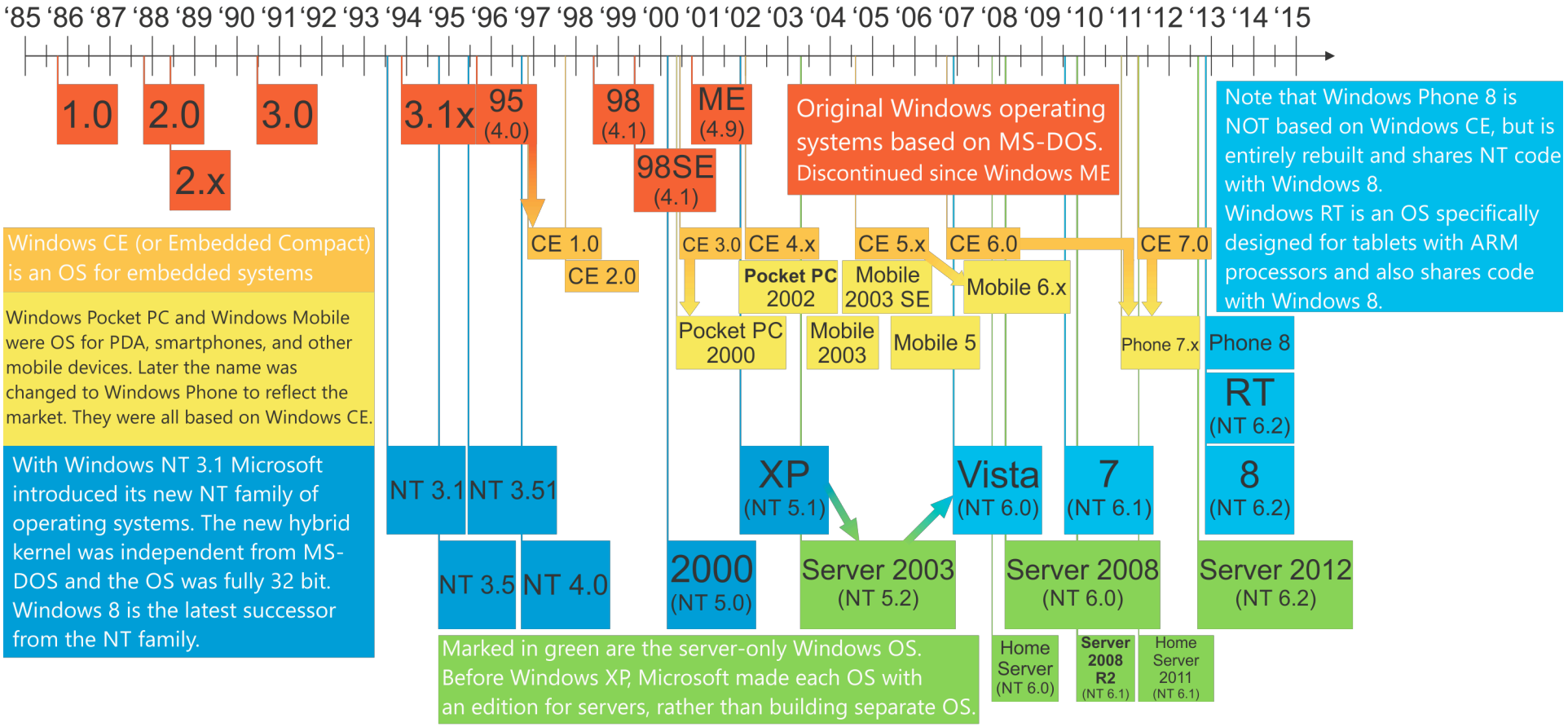
POLYTECHNIQUE MONTRÉAL
AFFILIÉE À L'UNIVERSITÉ DE MONTRÉAL

Automne 2016

Chapitre 8 - Cas de Windows (API WIN32)

- **Introduction**
- **Objets Windows**
- **Processus et threads**
  - **Création de processus et de threads**
  - **Terminaison de processus et récupération de la valeur de retour**
  - **Attente de la fin d'un processus ou d'un thread**
- **Sémaphores, mutex et événements**
- **Communication interprocessus**
  - **Héritage d'objets**
  - **Tubes anonymes et redirection des E/S standards**
- **Annexes**
  - **Threads – allocation dynamique**
  - **Mémoire partagée → mappage de fichiers**

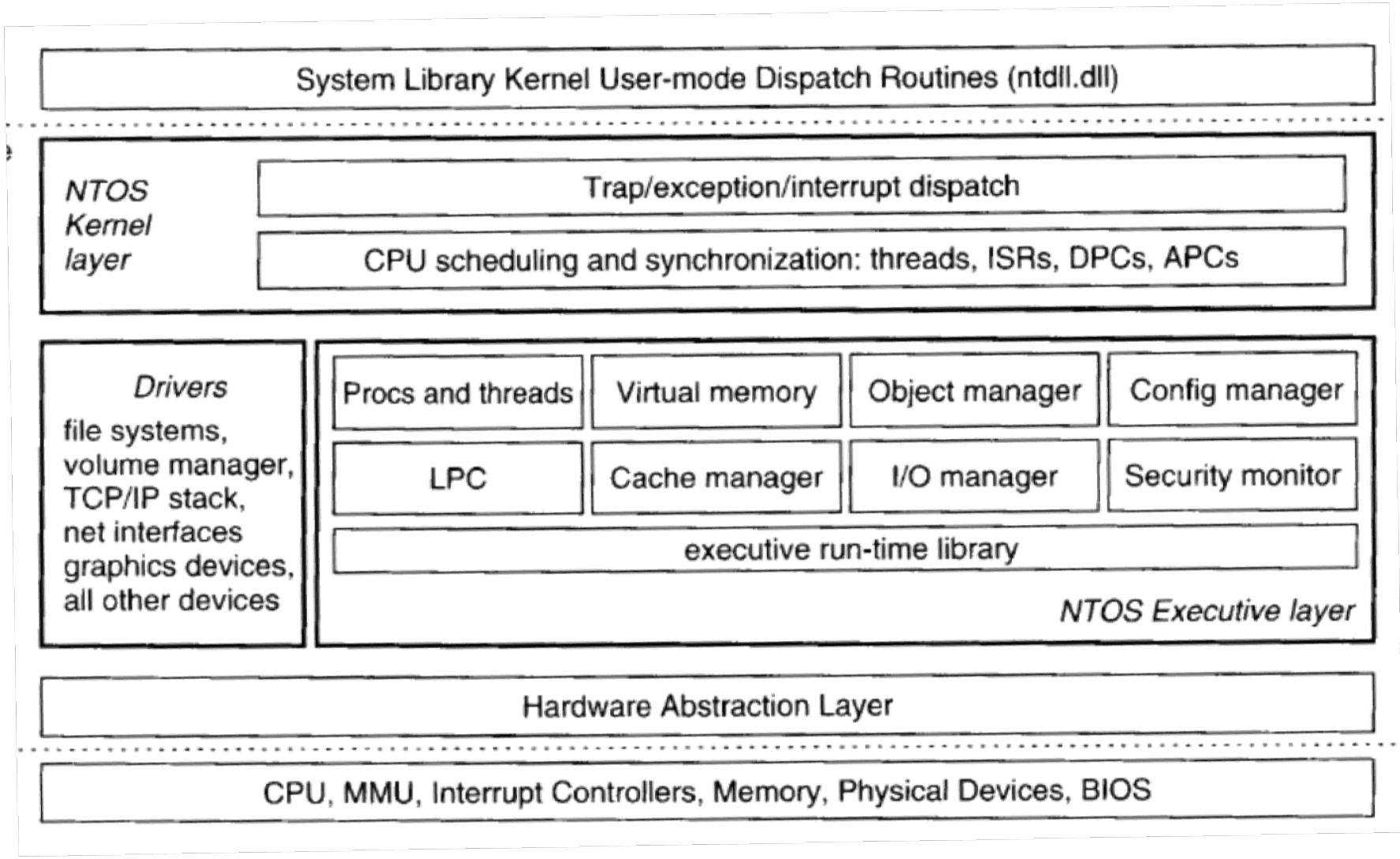
Introduction



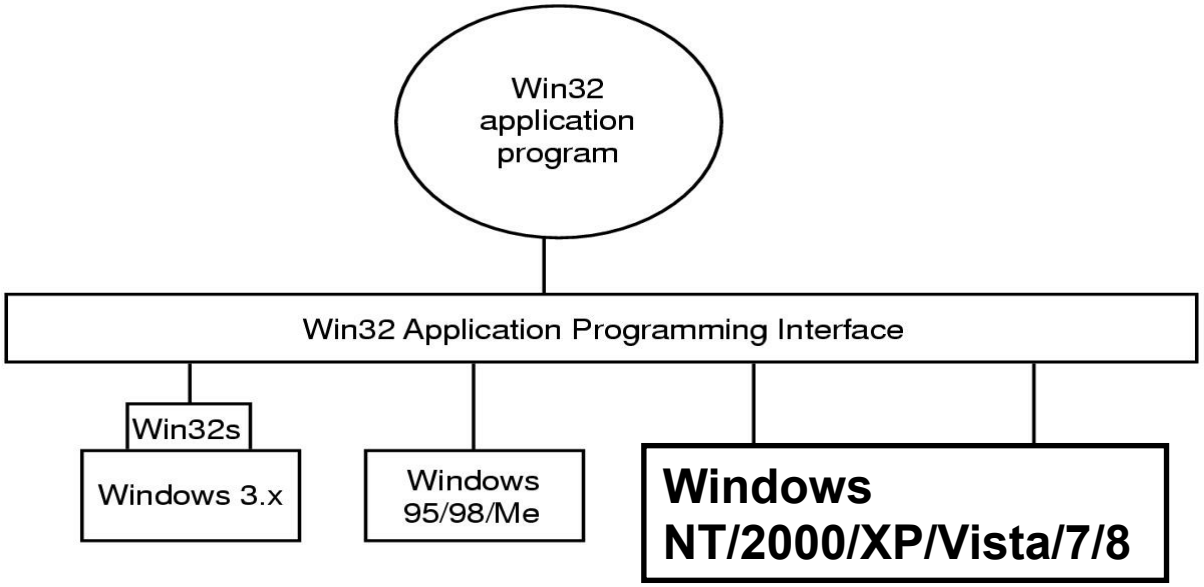
Explanation of arrows: I. Windows CE is based on code from Windows 95. II. Windows Pocket PC 2000 is based on Windows CE 3.0. III. Windows Mobile 6.x is based on Windows CE 5.x, rather than CE 6.0. IV. Windows Phone 7 is based on code from both Windows CE 6.0 and CE 7.0. V. Windows Vista was built on code from Windows Server 2003, rather than Windows XP.

http://upload.wikimedia.org/wikipedia/commons/6/6d/Windows_Updated_Family_Tree.png

Introduction (2)

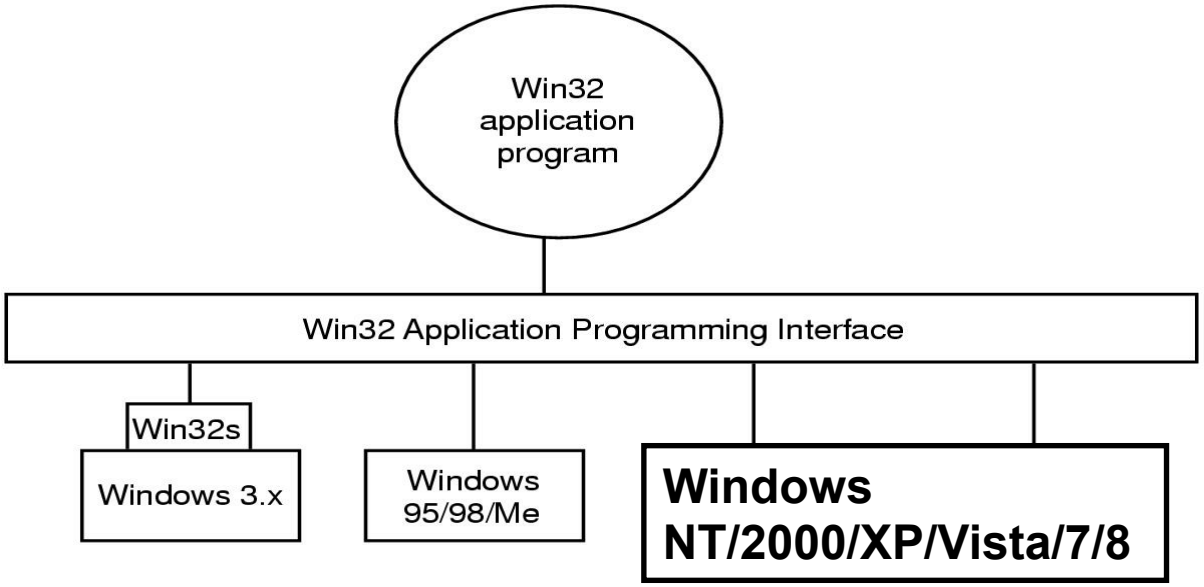


Introduction (3)



- Win32 permet de développer des programmes qui devraient fonctionner sur toutes les versions de Windows.
- .Net est construit au dessus Win32.
- Windows supporte POSIX avec Cygwin ou Windows Services for Unix.
- http://msdn.microsoft.com/en-us/library/ms682425.aspx

Introduction (4)



- Win32 est une interface de type utilisateur – système.
- API Win32 = { fonctions }.
- Win32 permet de créer, détruire, synchroniser, faire communiquer des processus, des threads, etc.
- Win32 permet aux processus de gérer leurs propres espaces d'adressage virtuels, de gérer des fichiers, de mapper des fichiers dans leurs espaces virtuels, etc.

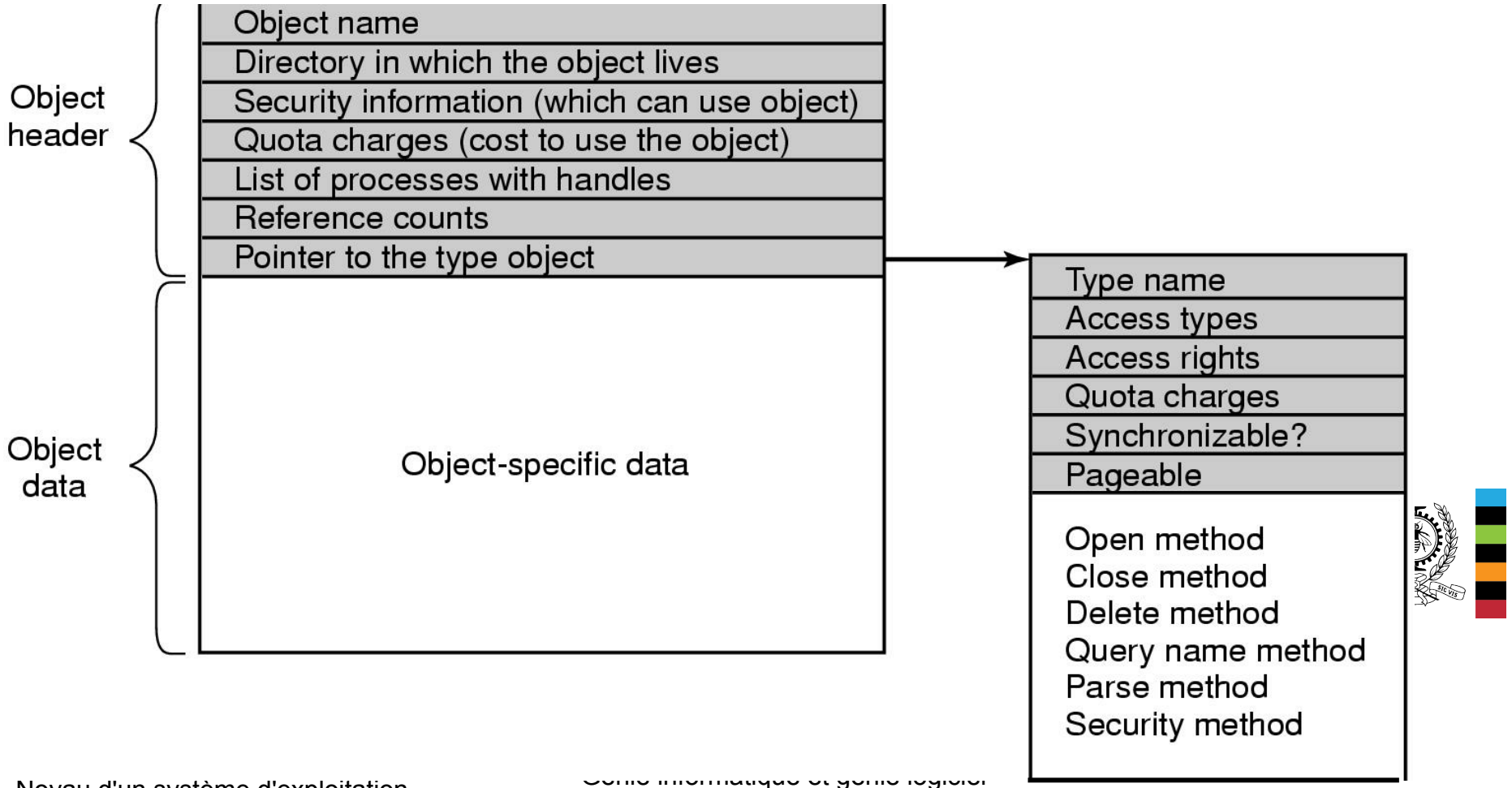
Objets Windows

- Sous Windows, un objet désigne une ressource du système ou une structure de données
- Tous ces objets sont gérés par le module *Object Manager*.
- Object Manager offre une interface uniforme et cohérente pour la gestion d'objets de différents types.

Type	Description
Process	User process
Thread	Thread within a process
Semaphore	Counting semaphore used for interprocess synchronization
Mutex	Binary semaphore used to enter a critical region
Event	Synchronization object with persistent state (signaled/not)
Port	Mechanism for interprocess message passing
Timer	Object allowing a thread to sleep for a fixed time interval
Queue	Object used for completion notification on asynchronous I/O
Open file	Object associated with an open file
Access token	Security descriptor for some object
Profile	Data structure used for profiling CPU usage
Section	Structure used for mapping files onto virtual address space
Key	Registry key
Object directory	Directory for grouping objects within the object manager
Symbolic link	Pointer to another object by name
Device	I/O device object
Device driver	Each loaded device driver has its own object

Objets Windows (2)

- Au niveau utilisateur, ces objets sont référencés par des « handles » (des numéros de 32 bits) qui sont convertis, par l'Object Manager, en des pointeurs vers les objets.



Objets windows (3)

- Win32 dispose d'un ensemble de fonctions (CreateXXXX, OpenXXXX) qui permettent à un utilisateur (processus) de créer ou d'ouvrir des objets gérés par le noyau (kernel-mode objects).

- L'appel à une de ces fonctions va réaliser un véritable appel système (NtCreateXXXX, NtOpenXXXX). Cet appel système retourne un handle de 64 bits dont seulement 32 bits sont retournés, par la fonction, au processus appelant.

- Ce handle de 32 bits est en fait l'identificateur de l'objet au niveau du processus. **Il correspond à une entrée (position) dans la table des handles du processus appelant ➔ chaque processus a sa propre table des handles qui est gérée par le noyau.**

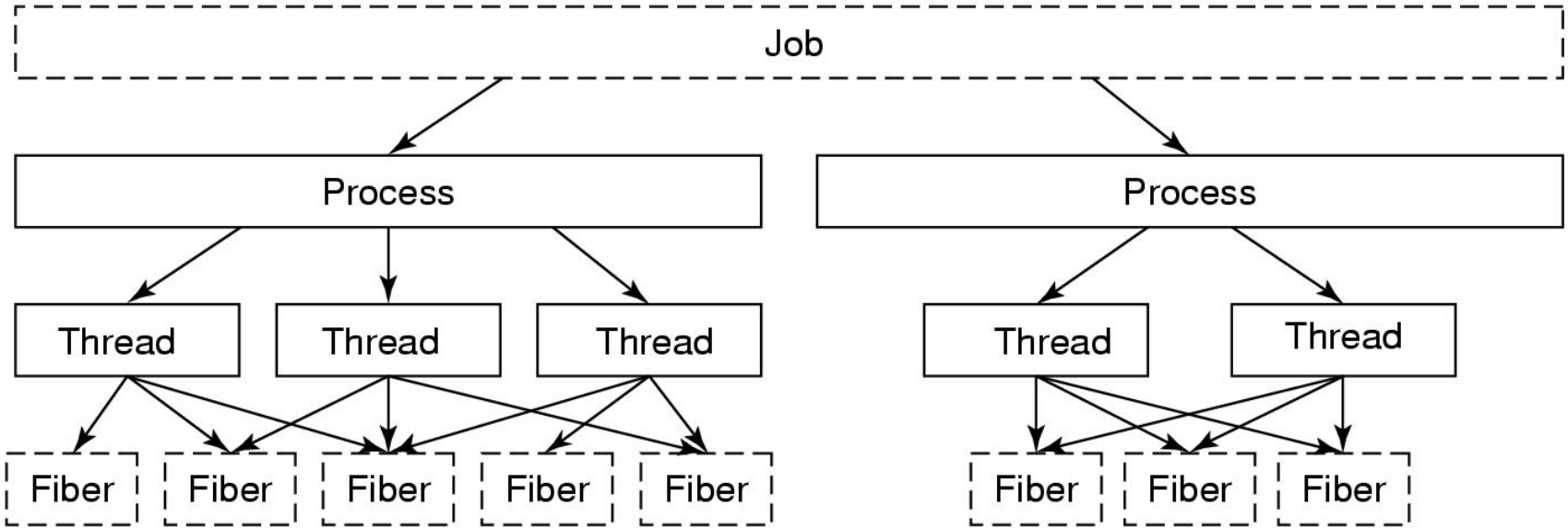
- Un **jeton de sécurité** est associé à chaque objet système indiquant ses droits d'accès (security token).

Jobs, processus, threads et fibres

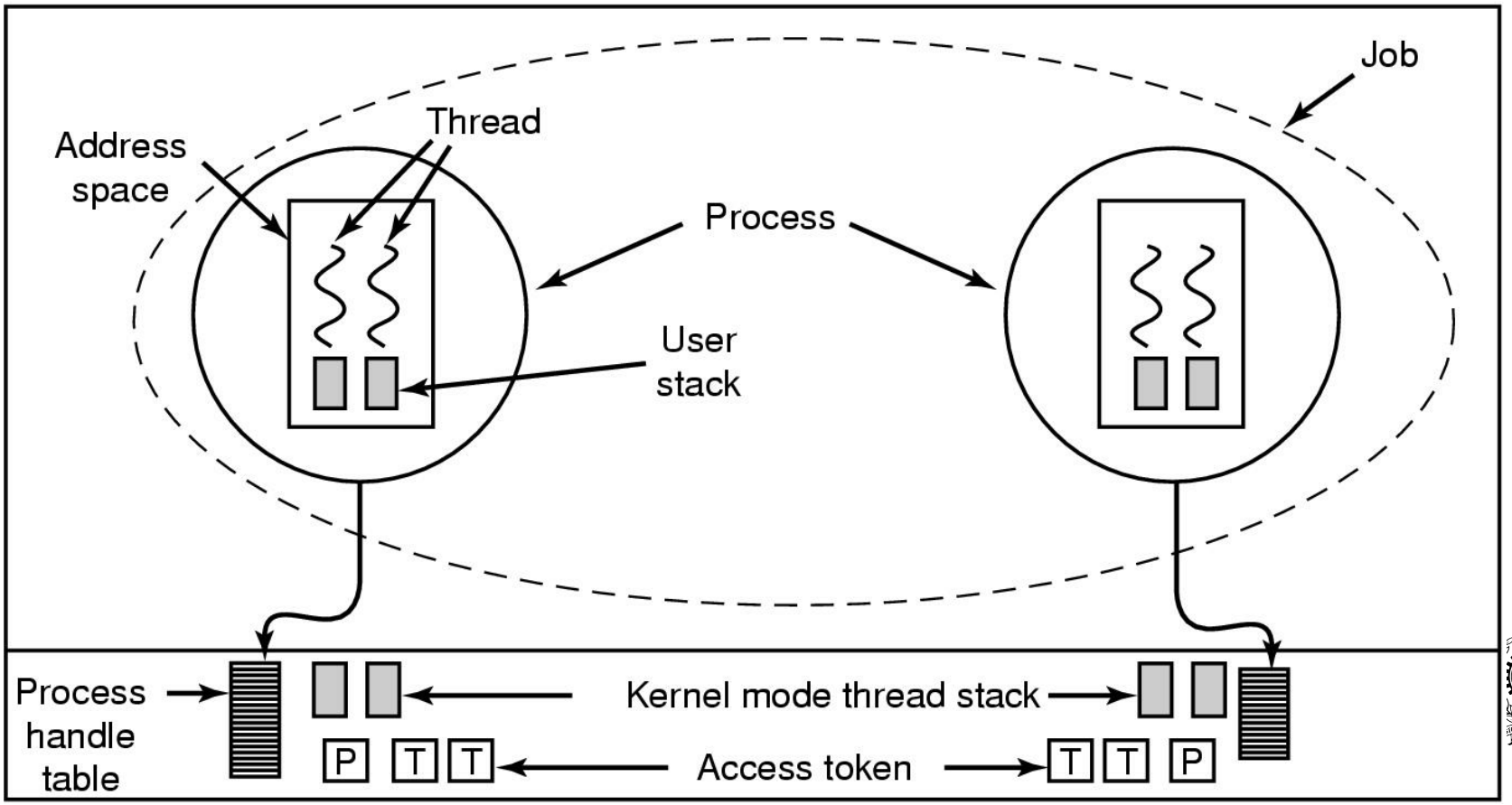
Name	Description
Job	Collection of processes that share quotas and limits
Process	Container for holding resources
Thread	Entity scheduled by the kernel
Fiber	Lightweight thread managed entirely in user space

- Une application Win32 est composée d'un ou plusieurs processus.
- Un ou plusieurs threads fonctionnent dans le contexte du processus.
- Un thread est l'unité de base à laquelle le système d'exploitation assigne du temps processeur.
- Une fibre est une unité d'exécution ordonnancée au niveau utilisateur (thread utilisateur). Les fibres s'exécutent dans le contexte des threads qui les ordonnancent (chaque thread peut avoir plusieurs fibres et une fibre peut être associée à plusieurs threads).
- Un job permet à des groupes de processus d'être considérés comme une unité. Les opérations exécutées sur un job affectent tous les processus du job. On peut imposer à un job des limites par rapport aux ressources du système.

Jobs, processus, threads et fibres (2)



Jobs, processus, threads et fibres (3)



Jobs, processus, threads et fibres (4)

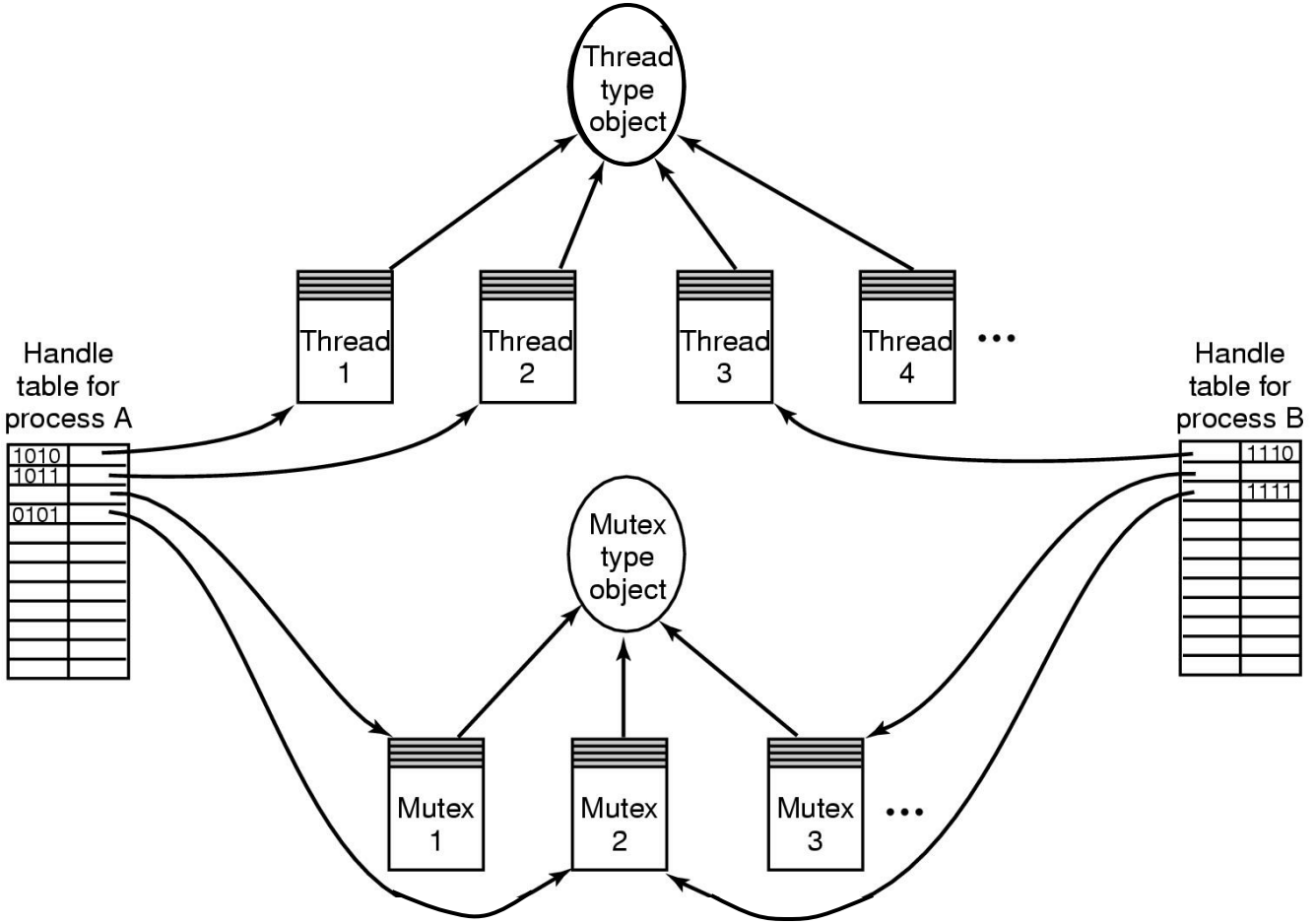
Win32 API Function	Description
CreateProcess	Create a new process
CreateThread	Create a new thread in an existing process
CreateFiber	Create a new fiber
ExitProcess	Terminate current process and all its threads
ExitThread	Terminate this thread
ExitFiber	Terminate this fiber
SetPriorityClass	Set the priority class for a process
SetThreadPriority	Set the priority for one thread
CreateSemaphore	Create a new semapahore
CreateMutex	Create a new mutex
OpenSemaphore	Open an existing semaphore
OpenMutex	Open an existing mutex
WaitForSingleObject	Block on a single semaphore, mutex, etc.
WaitForMultipleObjects	Block on a set of objects whose handles are given
PulseEvent	Set an event to signaled then to nonsignaled
ReleaseMutex	Release a mutex to allow another thread to acquire it
ReleaseSemaphore	Increase the semaphore count by 1
EnterCriticalSection	Acquire the lock on a critical section
LeaveCriticalSection	Release the lock on a critical section

Processus Windows

Un processus Windows est composé de :

- Un identificateur unique appelé ID du processus;
- Un espace d'adressage virtuel privé qui est un ensemble d'adresses mémoires virtuelles utilisables par le processus;
- Un programme exécutable qui définit le code et les données initiaux et qui est mappé dans l'espace d'adressage virtuel du processus;
- Une table des handles ouverts qui pointent vers diverses ressources du système telles que les sémaphores, les ports de communication, les fichiers …., qui sont accessibles aux threads du processus;
- Un jeton de sécurité qui identifie l'utilisateur, les groupes de sécurité et les privilèges associés au processus;
- Au moins un thread d'exécution.

Processus Windows (2)



Processus Windows (3) : Types prédéfinis

Win32 uses an extended set of data types, using C's typedef mechanism These include

BYTE	unsigned 8 bit integer
DWORD	32 bit unsigned integer
LONG	32 bit signed integer
LPDWORD	32 bit pointer to DWORD
LPCSTR	32 bit pointer to constant character string
LPSTR	32 bit pointer to character string
UINT	32 bit unsigned int
WORD	16 bit unsigned int
HANDLE	opaque pointer to system data

Processus Windows (4) : Création

```
BOOL WINAPI CreateProcess(

__in_opt     LPCTSTR lpApplicationName, // myProg.exe
__inout_opt  LPTSTR lpCommandLine,      //  -> main (argc,argv[])

  __in_opt     LPSECURITY_ATTRIBUTES lpProcessAttributes,
  __in_opt     LPSECURITY_ATTRIBUTES lpThreadAttributes,

  __in       BOOL bInheritHandles, // permettre au fils d'accéder aux objets du père

  __in       DWORD dwCreationFlags,// Ex: CREATE_NEW_CONSOLE
  __in_opt     LPVOID lpEnvironment, // variables d'environnement
  __in_opt     LPCTSTR lpCurrentDirectory, // répertoire courant

  __in       LPSTARTUPINFO lpStartupInfo,
  __out      LPPROCESS_INFORMATION lpProcessInformation
);
```

Processus Windows (5) : Création

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

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

Processus Windows (6) : Terminaison

- La fonction TerminateProcess permet d'arrêter un processus et tous ses threads

  BOOL WINAPI **TerminateProcess**(HANDLE *hProcess*, UINT *uExitCode*);

  VOID WINAPI **ExitProcess**(UINT *uExitCode*);

- La fonction GetExitCodeProcess permet de récupérer le code (valeur) de retour d'un processus

  BOOL WINAPI **GetExitCodeProcess**(HANDLE *hProcess*, LPDWORD *lpExitCode*);

Processus Windows (7) : Attendre la fin d'un objet

- La fonction WaitForSingleObject permet d'attendre qu'un objet (ex. un processus) soit signalé (ex. un processus se termine). Le temps d'attente est limité à *dwMilliseconds millisecondes :*

  DWORD WINAPI **WaitForSingleObject**(HANDLE *hHandle*, DWORD *dwMilliseconds*);

- La fonction WaitForMultipleObjects permet d'attendre qu'un objet (ou tous les objets) d'un ensemble donné d'objets soit signalé (soient signalés). Le temps d'attente est limité à *dwMilliseconds millisecondes* :

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

Processus Windows (8) : Exemple 1

http://msdn.microsoft.com/en-us/library/windows/desktop/dd374083(v=vs.85).aspx

```
/* parent1.c   Création de processus */
#include <windows.h>
#include <stdio.h>

void ErrorExit (LPTSTR lpszMessage);
int main()
{    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    ZeroMemory(&si, sizeof(si));
    ZeroMemory(&pi, sizeof(pi));
    si.cb = sizeof(si);
   if( !CreateProcess("fils.exe", //name or path of executable
        NULL,  // no 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.
    )
   {   ErrorExit("CreateProcess failed."); }
```

```
// Wait until child process exits.
    WaitForSingleObject(pi.hProcess, INFINITE);
    // Close process and thread handles.
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
    return 0;
}

void ErrorExit(LPTSTR lpszMessage)
{   fprintf(stderr, "%s\n", lpszMessage);
   ExitProcess(1);
}
```

```
/* fils1.c */
#include <windows.h>
#include <stdio.h>
int main()
{   printf("Child sleeps 5s\n »);
    Sleep(5000);
    printf("done\n");
    return 0;
}
```

Processus Windows (9) : Exemple 2

```
/* parent2.c  Valeur de retour d'un processus */
#include <windows.h>
#include <stdio.h>
void main()
{    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    DWORD status ; // address to receive termination status
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));
    if(!CreateProcess( "child.exe",NULL,NULL,NULL,FALSE,0,NULL,NULL, &si,&pi)) {
       printf("CreateProcess failed (%d).\n", GetLastError());
       ExitProcess(1); }
    WaitForSingleObject(pi.hProcess, INFINITE);
    if(GetExitCodeProcess(pi.hProcess, &status)) {
       printf("I am the father. My child [%d:%d] has terminated with (%d).\n",
              pi.dwProcessId, pi.dwThreadId, status);
    } else {  printf("GetExitCodeProcess failed (%d).\n", GetLastError());  }
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
    printf("I am the father. Going to terminate.\n");
    ExitProcess(0);
}
```

Processus Windows (10) : Exemple 2

```
/* fils2.c */
#include <windows.h>
#include <stdio.h>
#include <tchar.h>
void main()
{
    printf("I am the child. My Pid and Tid:  (%d, %d).\n",
          GetCurrentProcessId(), GetCurrentThreadId());
    printf( "I am the child. Going to sleep for 5 seconds. \n");
    Sleep(5000);
    ExitProcess(10);   // exit code
}
```

```
I am the child. My Pid and Tid (1084, 1404).
I am the child. Going to sleep for 5 seconds.
I am the father. My child [1084:1404] has terminated with (10).
I am the father.  Going to terminate.
```

Threads Windows : Création

```
HANDLE WINAPI CreateThread(
  __in_opt   LPSECURITY_ATTRIBUTES lpThreadAttributes,
  __in       SIZE_T dwStackSize,
  __in       LPTHREAD_START_ROUTINE lpStartAddress, // fonction
  __in_opt   LPVOID lpParameter,                    // arguments
  __in       DWORD dwCreationFlags,
  __out_opt  LPDWORD lpThreadId // retourne l'id du thread
);
```

Threads Windows (2) : Terminaison

- La fonction TerminateThread permet d'arrêter un thread :

  BOOL WINAPI **TerminateThread**(HANDLE *hThread*, DWORD *dwExitCode*);

  VOID WINAPI **ExitThread**(DWORD *dwExitCode*);

- La fonction GetExitCodeThread permet de récupérer le code de retour d'un thread :

  BOOL WINAPI **GetExitCodeThread**(HANDLE *hThread*, LPDWORD *lpExitCode*);

Threads Windows (3) : Exemple 3

```
/* thread1.c   création de threads */
#include <windows.h>
#include <stdio.h>
#include <tchar.h>
#define true 1
int glob=100;
DWORD WINAPI FuncThread1(LPVOID arg)  // paramètre = handle du thread
{   int i;
    for(i=0; i<=10;i++) {
       glob = glob+10;
       printf("ici thread %d : glob = %d \n",*((HANDLE*)arg), glob);
    }
    return 0;
}
DWORD WINAPI FuncThread2(LPVOID arg)
{   int i;
    for(i=0; i<=10;i++) {
        glob = glob-10;
        printf("ici thread %d : glob = %d \n",*((HANDLE*)arg), glob);
    }
    return 0;
}
```

Threads Windows (4) : Exemple 3

```
int main()
{
    HANDLE th[2];
    DWORD thId1, thId2;
    th[0]=CreateThread(NULL, 0, FuncThread1, (LPVOID)&th[0],0,&thId1);
    if (th[0]== NULL) {
        printf("CreateThread failed: %d\n",GetLastError());
        return -1;
    }
    th[1]=CreateThread(NULL, 0, FuncThread2, (LPVOID)&th[1], 0, &thId2);
    if (th[1]== NULL) {
        printf("CreateThread failed: %d\n", GetLastError());
        return -1;
    }

    WaitForMultipleObjects(2, th, true, INFINITE);
    printf("Fin des threads %d , %d \n", th[0], th[1]);
    CloseHandle(th[0]);
    CloseHandle(th[1]);
    return 0;
}
```

Threads Windows (5) : Exemple 3

```
ici thread 2556 : glob = 110
ici thread 2556 : glob = 120
ici thread 2556 : glob = 130
ici thread 2556 : glob = 140
ici thread 2556 : glob = 150
ici thread 2556 : glob = 160
ici thread 2556 : glob = 170
ici thread 2556 : glob = 180
ici thread 2556 : glob = 190
ici thread 2556 : glob = 200
ici thread 2556 : glob = 210
ici thread 2552 : glob = 200
ici thread 2552 : glob = 190
ici thread 2552 : glob = 180
ici thread 2552 : glob = 170
ici thread 2552 : glob = 160
ici thread 2552 : glob = 150
ici thread 2552 : glob = 140
ici thread 2552 : glob = 130
ici thread 2552 : glob = 120
ici thread 2552 : glob = 110
ici thread 2552 : glob = 100
Fin des threads 2556, 2552
```

Exercice :

1) Augmentez le nombre d'itérations dans les fonctions des threads.

2) Remplacez les instructions glob = glob-10; et glob = glob+10; par respectivement :

```
int r = glob;
        for(int j=0; j<10000; j++);
        r= r – 10;
        glob = r;
```

et

```
int r = glob;
        for(int j=0; j<10000; j++);
        r= r + 10;
        glob = r;
```

Sémaphores, mutex et événements Windows

- Chaque objet de synchronisation (sémaphore, mutex et événement) a un état (signalé, non-signalé, etc.)

Objets section critique et variables de condition :
http://msdn.microsoft.com/en-us/library/windows/desktop/ms686908(v=vs.85).aspx

http://msdn.microsoft.com/en-us/library/windows/desktop/ms686903(v=vs.85).aspx

- Chaque sémaphore a deux états :
  - Signalé, si sa valeur > 0;
  - Non-signalé, si sa valeur = 0.

- Chaque mutex a trois états :
  - Signalé, s'il est libre;
  - Non-signalé, s'il est détenu par un thread vivant;
  - Abandonné, s'il est détenu par un thread terminé.

- L'état initial (signalé ou non-signalé) est fixé lors de la création.

- Les fonctions **WaitForMultipleObjects**(…, INFINITE) et **WaitForSingleObject**(…, INFINITE) bloquent le thread appelant si le ou les objets spécifiés sont à l'état non-signalé ~ de la fonction P d'un sémaphore

Sémaphores, mutex et événements Windows

- Chaque événement peut être crée selon deux modes :
  - « reset » manuel ou
  - « reset » automatique.

- Un événement a deux états :
  - Signalé, l'événement s'est produit; SetEvent(…) permet de signaler un événement,
  - Non-signalé, l'événement ne s'est pas produit ou n'est plus en cours.

- L'état initial est spécifié lors de la création.

- Un événement à reset manuel devient non-signalé par ResetEvent(…). Le changement d'état n'est pas automatique.

- Les fonctions **WaitForMultipleObjects**(…, INFINITE) et **WaitForSingleObject**(…, INFINITE) bloquent le thread appelant si le ou les objets spécifiés sont à l'état non-signalé.

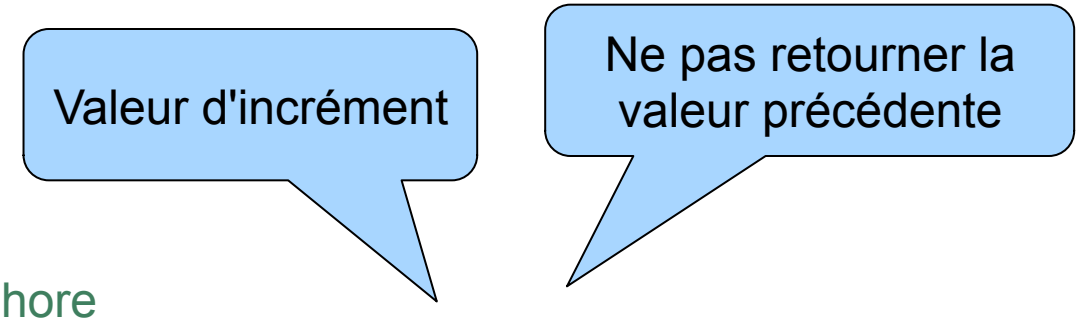
Sémaphores Windows : Création et initialisation

```
/* sem1.c */
#include <windows.h>
#include <stdio.h>
HANDLE hSemaphore;
LONG cMax = 10;

int main() {
   // Créer un objet de type sémaphore dont la valeur initiale est 10.
   hSemaphore = CreateSemaphore(
         NULL, // sécurité par défaut
         cMax, // valeur initiale
         cMax, // valeur maximale
         NULL); // sémaphore anonyme

   if (hSemaphore == NULL) {
      printf("CreateSemaphore error: %d\n", GetLastError());
   }
```

Sémaphores Windows (2) : Attente de sémaphores


```
DWORD dwWaitResult;
// Essayer d'obtenir le sémaphore ~ P du sémaphore
dwWaitResult = WaitForSingleObject(hSemaphore, 0);
switch (dwWaitResult) {
// Le sémphore était signalé
case WAIT_OBJECT_0:
    break;
// Le sémpahore était non-signalé – time-out
case WAIT_TIMEOUT:
    break;
}
```

Sémaphores Windows (3) : Libération




```
// ~ du V du sémaphore
if (!ReleaseSemaphore(hSemaphore, 1, NULL)) {
    printf("ReleaseSemaphore error: %d\n", GetLastError());
}
```

Sémaphores Windows (4) : Exemple 4

```
/* mux1.c  Sémaphore binaire */
int glob=100;
HANDLE hSem;
DWORD WINAPI FuncThread1(LPVOID arg)
{   int i;
    for(i=0; i<=10; i++) {
        WaitForSingleObject(hSem, INFINITE);
        glob = glob+10;
        printf("ici thread %p : valeur de glob = %d \n",*((HANDLE*)arg), glob);
        ReleaseSemaphore(hSem, 1, NULL);
    }
    return 0;
}
DWORD WINAPI FuncThread2(LPVOID arg)
{  for(int i=0; i<=10; i++) {
        WaitForSingleObject(hSem, INFINITE);
        glob = glob-10;
        printf("ici thread %p : valeur de glob = %d \n",*((HANDLE*)arg), glob);
        ReleaseSemaphore(hSem, 1, NULL);
    }
    return 0;
}
```

Sémaphores Windows (5) : Exemple 4

```
int main()
{   HANDLE th[2];
    DWORD dwLoginThreadId1, dwLoginThreadId2;
    hSem = CreateSemaphore(NULL,1,1,NULL);
    if (hSem == NULL) {
        printf("CreateSemaphore error: %d\n", GetLastError());
        return -1;
    }

    th[0] = CreateThread(NULL, 0, FuncThread1, (LPVOID) &th[0], 0, &dwLoginThreadId1);

    th[1] = CreateThread(NULL, 0, FuncThread2, (LPVOID) &th[1], 0, &dwLoginThreadId2);

    WaitForMultipleObjects(2, th,true,INFINITE);
    printf("Fin des threads %d , %d \n", th[0], th[1]);
    CloseHandle(hSem);
    CloseHandle(th[0]);
    CloseHandle(th[1]);
    return 0;
}
```

Sémaphores Windows (6) : Exemple 4

```
ici thread 1780 : valeur de glob = 110
ici thread 1780 : valeur de glob = 120
ici thread 1780 : valeur de glob = 130
ici thread 1780 : valeur de glob = 140
ici thread 1780 : valeur de glob = 150
ici thread 1780 : valeur de glob = 160
ici thread 1780 : valeur de glob = 170
ici thread 1780 : valeur de glob = 180
ici thread 1776 : valeur de glob = 170
ici thread 1776 : valeur de glob = 160
ici thread 1776 : valeur de glob = 150
ici thread 1776 : valeur de glob = 140
ici thread 1776 : valeur de glob = 130
ici thread 1776 : valeur de glob = 120
ici thread 1776 : valeur de glob = 110
ici thread 1776 : valeur de glob = 100
ici thread 1776 : valeur de glob = 90
ici thread 1776 : valeur de glob = 80
ici thread 1776 : valeur de glob = 70
ici thread 1780 : valeur de glob = 80
ici thread 1780 : valeur de glob = 90
ici thread 1780 : valeur de glob = 100
Fin des threads 1780, 1776
```

Exercice :

1) Augmentez le nombre d'itérations dans les fonctions des threads.

2) Remplacez les instructions glob = glob-10; et glob = glob+10; par respectivement :

```
int r = glob;
        for(int j=0; j<10000; j++);
        r= r – 10;
        glob = r;

et
int r = glob;
        for(int j=0; j<10000; j++);
        r= r + 10;
        glob = r;
```

Mutex Windows : Exemple 5

```
/* mux2.c  Mutex  */
#include <windows.h>
#include <stdio.h>
#define THREADCOUNT 2
HANDLE ghMutex;
DWORD WINAPI WriteToDatabase();
int main()
{
   HANDLE aThread[THREADCOUNT];
   DWORD ThreadID; int i;
   // Create a mutex with no initial owner
   ghMutex = CreateMutex(
      NULL,              // default security attributes
      FALSE,             // initially not owned
      NULL);             // unnamed mutex
   for(i=0; i < THREADCOUNT; i++) {
      aThread[i] = CreateThread( NULL, 0,
            (LPTHREAD_START_ROUTINE) WriteToDatabase,  NULL, 0, &ThreadID);
   }
   WaitForMultipleObjects(THREADCOUNT, aThread, TRUE, INFINITE);
   for(i=0; i < THREADCOUNT; i++)
      CloseHandle(aThread[i]);
   CloseHandle(ghMutex);
   return 0;
}
```

Écritures dans une même base de données synchronisées par un mutex

Créer les threads écrivains

Mutex Windows (2) : Exemple 5

```
DWORD WINAPI WriteToDatabase(LPVOID lpParam)
{
   DWORD dwCount=0, dwWaitResult;
   // Request ownership of mutex.
   while( dwCount < 20 ) {
      dwWaitResult = WaitForSingleObject(ghMutex, INFINITE);
      switch (dwWaitResult) {
      case WAIT_OBJECT_0:
         printf("Thread %d writing to database...\n",  GetCurrentThreadId());
         dwCount++;
         ReleaseMutex(ghMutex);
         break;
      //The thread got ownership of an abandoned mutex.
      case WAIT_ABANDONED:
         return FALSE;
      }
   }
   return TRUE;
}
```

Attendre le mutex

Signaler le mutex

Mutex Windows (3) : Exemple 5

Thread 2284 writing to database...
Thread 2284 writing to database...
Thread 2284 writing to database...
Thread 2284 writing to database...
Thread 2284 writing to database...
Thread 2284 writing to database...
Thread 2284 writing to database...
Thread 3240 writing to database...
Thread 3240 writing to database...
Thread 3240 writing to database...
Thread 3240 writing to database...
Thread 3240 writing to database...
Thread 3240 writing to database...
Thread 3240 writing to database...
Thread 3240 writing to database...
Thread 3240 writing to database...
Thread 3240 writing to database...
Thread 3240 writing to database...
Thread 3240 writing to database...
Thread 3240 writing to database...
Thread 3240 writing to database...

Thread 3240 writing to database...
Thread 3240 writing to database...
Thread 3240 writing to database...
Thread 2284 writing to database...
Thread 2284 writing to database...
Thread 2284 writing to database...
Thread 2284 writing to database...
Thread 2284 writing to database...
Thread 2284 writing to database...
Thread 2284 writing to database...
Thread 2284 writing to database...
Thread 2284 writing to database...
Thread 2284 writing to database...
Thread 2284 writing to database...
Thread 2284 writing to database...
Thread 2284 writing to database...
Thread 3240 writing to database...
Thread 3240 writing to database...
Thread 3240 writing to database...

Événements Windows : Exemple 6

http://msdn.microsoft.com/en-us/library/ms686915(VS.85).asp

```
/* Event.c  Accès partagé en lecture*/
#include <windows.h>
#include <stdio.h>
#define THREADCOUNT 4
HANDLE ghWriteEvent;
HANDLE ghThreads[THREADCOUNT];
DWORD WINAPI ThreadProc(LPVOID);
char Buffer[1024];

int main(int argc, char* argv[])
{   DWORD dwWaitResult;
    int i;
    DWORD dwThreadID;
    // Create a manual-reset event. The main thread
    // sets this object to signaled  when it
    // finishes writing to the buffer.
    ghWriteEvent = CreateEvent (
        NULL,              // default security attributes
        TRUE,              // manual-reset event
        FALSE,             // initial state is nonsignaled
        TEXT("WriteEvent") );  // object name
```

Créer un événement (reset manuel, état initial non-signalé)

Événements Windows (2) : Exemple 6

```
 if (ghWriteEvent == NULL) {
    printf("CreateEvent failed (%d)\n", GetLastError());
    return 1;
}

// Create multiple threads to read from the buffer.
for(i = 0; i < THREADCOUNT; i++) {
    ghThreads[i] = CreateThread(
        NULL,              // default security
        0,                 // default stack size
        ThreadProc,        // name of the thread function
        NULL,              // no thread parameters
        0,                 // default startup flags
        &dwThreadID);

    if (ghThreads[i] == NULL) {
        printf("CreateThread failed (%d)\n", GetLastError());
        return 2;
    }
}
```

Créer un thread

Événements Windows (3) : Exemple 6

```
    printf("Main thread writing to the shared buffer...\n");
    strcpy(Buffer,"API WIN32-CreateEvent-Example\n");

    // Set ghWriteEvent to signaled
    if (!SetEvent(ghWriteEvent)) {
        printf("SetEvent failed (%d)\n", GetLastError());
        return 3;
    }

    printf("Main thread waiting for threads to exit...\n");
    // The handle for each thread is signaled when the thread is  terminated.
    dwWaitResult = WaitForMultipleObjects(THREADCOUNT, ghThreads, TRUE, INFINITE);
    switch (dwWaitResult) {
    case WAIT_OBJECT_0: // All thread objects were signaled
        printf("All threads ended, cleaning up for application exit...\n");  break;
    default: // An error occurred
        printf("WaitForMultipleObjects failed (%d)\n", GetLastError());
        return 4;
    }
    CloseHandle(ghWriteEvent); // Close the event to clean up
} // end of main
```

Écrire dans le buffer

Signaler l'événement

Événements Windows (4) : Exemple 6

```
DWORD WINAPI ThreadProc(LPVOID lpParam) {
    DWORD dwWaitResult;
    printf("Thread %d waiting for write event...\n", GetCurrentThreadId());

    dwWaitResult = WaitForSingleObject(ghWriteEvent, INFINITE);

     printf("Thread %d reading from buffer: %s\n", GetCurrentThreadId(), Buffer);
     Sleep(1000);

    printf("Thread %d exiting\n", GetCurrentThreadId());
    return 3;
}
```

ghWriteEvent reste signalé

Lire du buffer

Événements Windows (5) : Exemple 6

```
Main thread writing to the shared buffer...
Main thread waiting for threads to exit...
Thread 968 waiting for write event...
Thread 968 reading from buffer: API WIN32-CreateEvent-Example
Thread 4812 waiting for write event...
Thread 3288 waiting for write event...
Thread 3288 reading from buffer: API WIN32-CreateEvent-Example
Thread 4812 reading from buffer: API WIN32-CreateEvent-Example
Thread 2056 waiting for write event...
Thread 2056 reading from buffer: API WIN32-CreateEvent-Example
Thread 968 exiting
Thread 3288 exiting
Thread 4812 exiting
Thread 2056 exiting
All threads ended, cleaning up for application exit...
```

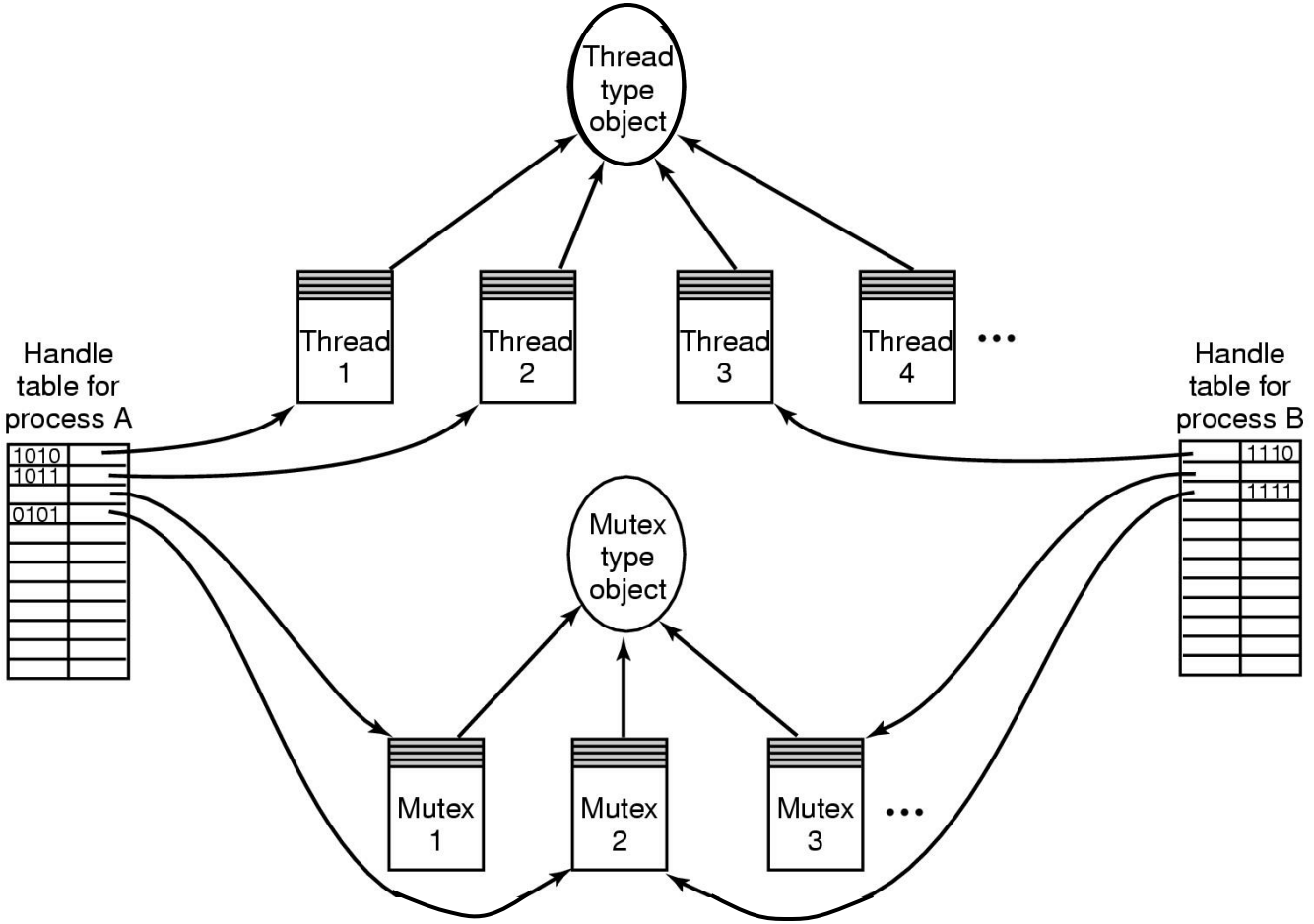
Événements Windows (6) : Exemple 6'

```
ghWriteEvent = CreateEvent(
        NULL,               // default security attributes
        FALSE,               //  non manual-reset event
        FALSE,              // initial state is nonsignaled
        TEXT("WriteEvent")  // object name
        );
```

```
// after reading, signal the event
if (! SetEvent(ghWriteEvent) )
    {   printf("SetEvent failed (%d)\n", GetLastError());
        return 3;   }
```

```
Thread 5004 waiting for write event...
Main thread writing to the shared buffer...
Thread 2484 waiting for write event...
Thread 4792 waiting for write event...
Thread 4016 waiting for write event...
Main thread waiting for threads to exit...
Thread 5004 reading from buffer: API WIN32-CreateEvent-Example
Thread 5004 signals the event  and ends
Thread 2484 reading from buffer: API WIN32-CreateEvent-Example
Thread 2484 signals the event and ends
Thread 4792 reading from buffer: API WIN32-CreateEvent-Example
Thread 4792 signals the event and ends
Thread 4016 reading from buffer: API WIN32-CreateEvent-Example
Thread 4016 signals the event and ends
All threads ended, cleaning up for application exit...
```

Communication interprocessus : Héritage d'objets



Communication interprocessus (2) : Héritage d'objets

- Un processus fils peut hériter des propriétés et des ressources de son père.

- Il est possible aussi d'empêcher le processus fils d'hériter des propriétés ou des ressources de son père.

- Le processus fils peut hériter :
  - Les handles retournés par **CreateFile**, **CreateProcess**, **CreateThread**, **CreateMutex**, **CreateEvent**, **CreateSemaphore**, **CreateNamedPipe**, **CreatePipe**, et **CreateFileMapping**.
  - Les variables d'environnement.
  - Le répertoire courant.
  - La console, …..

- Il ne peut pas hériter :
  - Les handles retournés par **LocalAlloc**, **GlobalAlloc**, **HeapCreate**, et **HeapAlloc**…

Communication interprocessus (3) : Héritage d'objets

- Un processus fils peut hériter des handles d'objets de son père (créés ou ouverts par son père) .

- Pour permettre à un processus fils d'hériter un handle (descripteur d'objet) de son processus père, il suffit de :
  - Créer le descripteur (handle) en initialisant le membre "**bInheritHandle"** de la structure **SECURITY_ATTRIBUTES** à TRUE.
  - Créer le processus fils en appelant la fonction **CreateProcess.** Le paramètre ***bInheritHandles*** doit être positionné à TRUE.

- Le handle peut être passé :
  - comme paramètre au programme exécuté par le fils (int main (int argc, char*Argv[]) ou
  - communiqué au processus fils en utilisant un des mécanismes de communication.

Communication interprocessus (4) : Héritage d'objets

- La fonction **DuplicateHandle** crée un autre descripteur, pour un même objet, à utiliser dans le contexte du processus courant ou d'un autre processus (deux descripteurs pour un même objet) .

- Si un processus duplique un de ses descripteurs pour un autre processus, le descripteur créé est valide uniquement dans le contexte de l'autre processus.

```
BOOL WINAPI DuplicateHandle(
    HANDLE hSourceProcessHandle, // le processus propriétaire du handle à dupliquer
    HANDLE hSourceHandle,        // le handle à dupliquer
    HANDLE hTargetProcessHandle, // le processus concerné par cette duplication
    LPHANDLE lpTargetHandle,     // pour récupérer le handle créé
    DWORD dwDesiredAccess,       // droits d'accès pour le nouveau handle
    BOOL bInheritHandle,         // true si le handle peut être hérité par
                                 // les fils qui seront crées par
                                 // le processus hTargetProcessHandle
  DWORD dwOptions     // 0, DUPLICATE_CLOSE_SOURCE, DUPLICATE_SAME_ACCESS
);
```

Tubes anonymes

- La fonction **CreatePipe** crée un tube anonyme et retourne deux handles : un handle de lecture et un handle d'écriture.

- Pour faire communiquer un processus père avec un processus fils au moyen d'un tube anonyme, on peut procéder comme suit :
  - Le père crée un tube anonyme en spécifiant que les handles du tube peuvent être hérités par ses descendants.
  - Le père crée un processus fils en spécifiant que son fils va hériter les handles marqués « héritables ».
  - Le père peut communiquer le handle approprié du pipe au fils en le passant comme paramètre au programme exécuté par le fils. Il peut aussi utiliser un autre mécanisme de communication.

- Un processus peut également créer un tube, dupliquer un handle du tube en utilisant la fonction DuplicateHandle et envoyer le hundle créé à un processus indépendant (par exemple via une mémoire partagée).

Tubes anonymes (2)

- Pour lire du pipe, il faut utiliser le handle de lecture et la fonction **ReadFile**.
- Pour écrire dans le tube, il faut utiliser le handle d'écriture et la fonction **WriteFile.**
- La fonction **CloseHandle** permet de fermer un handle.
- Un pipe anonyme existe tant qu'il y a au moins un handle ouvert pour ce pipe.

Tubes anonymes (3)

```
BOOL WINAPI CreatePipe(
        PHANDLE hReadPipe,   // handle pour lire
        PHANDLE hWritePipe,   // handle pour écrire
        LPSECURITY_ATTRIBUTES lpPipeAttributes,
    // indique  si le pipe peut être
    // hérité par les fils du créateur
        DWORD nSize     // taille du pipe
);
```

Tubes anonymes (4) : Exemple 7

```
/* tube1.c  Processus parent communique avec son fils au moyen d'un pipe anonyme */
#include <windows.h>
#include <stdio.h>
#include <tchar.h>
#define BUFSIZE 4096
HANDLE FD[2];

int main() {
   // Créer un tube anonyme
   // Initialisation de SECURITY_ATTRIBUTES du pipe à créer.
   SECURITY_ATTRIBUTES saAttr;
   saAttr.nLength = sizeof(SECURITY_ATTRIBUTES);
   saAttr.bInheritHandle = TRUE; // les handles du pipe peuvent être hérités par ses fils
   saAttr.lpSecurityDescriptor = NULL;

   if (!CreatePipe(&FD[0], &FD[1], &saAttr, 0)) {
      printf("CreatePipe a échoué (%d).\n", GetLastError());
      ExitProcess(1);
   } else
      printf("FD[0]= %d, FD[1]=%d\n", FD[0], FD[1]);
```

1- créer un pipe

Tubes anonymes (5) : Exemple 7

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

//convert the handle FD[0] to char[]
char buf[10];
sprintf(buf, "tube2.exe %d", FD[0]);

// Créer un processus fils.
if (!CreateProcess("tube2.exe", buf, NULL, NULL, TRUE, 0, NULL, NULL, &si, &pi)) {
    printf("CreateProcess a échoué (%d).\n", GetLastError());
    ExitProcess(1);
}
CloseHandle(FD[0]);
DWORD dwWritten;
CHAR chBuf[BUFSIZE] = "Bonjour du Pere au Fils\0";
[...]
```

2- créer un processus et permettre l'héritage des handles

3- fermer le handle non utilisé

Tubes anonymes (6) : Exemple 7

```
    [...]
    if (!WriteFile(FD[1], chBuf, strlen(chBuf) + 1, &dwWritten, NULL)) {
        printf("WriteFile pipe a échoué (%d).\n", GetLastError());
        return -1;
    } else
        printf("WriteFile pipe a réussi  (%d).\n", dwWritten);

    CloseHandle(FD[1]);
    // Wait until child process exits.
    WaitForSingleObject(pi.hProcess, INFINITE );

    // Close process and thread handles.
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
    ExitProcess(0);
    return 0;
}
```

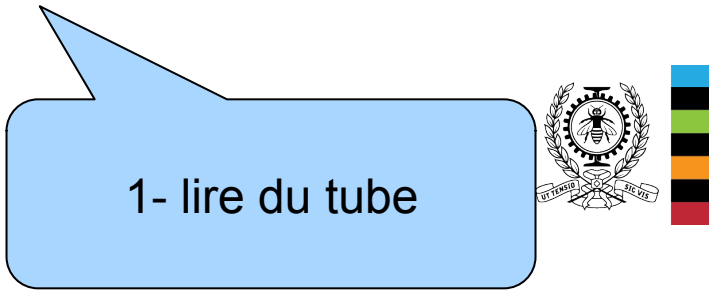
4- écrire dans le tube

5- fermer le handle d'écriture

Tubes anonymes (7) : Exemple 7

```
/* tube2.c */
//processus pipeChild
#include <windows.h>
#include <stdio.h>
#include <tchar.h>
int _tmain(int argc, char * argv[]) {
    DWORD dwRead;
    char chBuf[4096];
    printf("I am the child. My Pid and Tid:  (%d, %d).\n",
          GetCurrentProcessId(), GetCurrentThreadId());

    HANDLE fd0 = (HANDLE) atoi(argv[1]);
    printf("argc = %d, argv[0]=%s, argv[1]=%s, fd0=%d\n", argc, argv[0], argv[1], fd0);

    if (!ReadFile(fd0, chBuf, 4096, &dwRead, NULL) || dwRead == 0) {
        printf("ReadFile Pipe échoue (%d)\n", GetLastError());
        CloseHandle(fd0);
        ExitProcess(1);
    }
    printf("Message recu : %s , de taille %d \n", chBuf, dwRead);
    CloseHandle(fd0);
    ExitProcess(0);
}
```

```
FD[0]= 1828, FD[1]=1784
WriteFile pipe a réussi (24).
I am the child. My Pid and Tid:  (3140, 3756).
argc = 2, argv[0]=pipeChild, argv[1]=1828, fd0=1828
Message recu : Bonjour du Pere au Fils , de taille 24
Press any key to continue
```



Tubes anonymes (8)
Redirections des E/S standards

- Pour faire communiquer un processus père avec un processus fils, au moyen d'un tube anonyme, en redirigeant l'entrée ou la sortie standard d'un processus fils vers un pipe, on peut procéder comme suit :

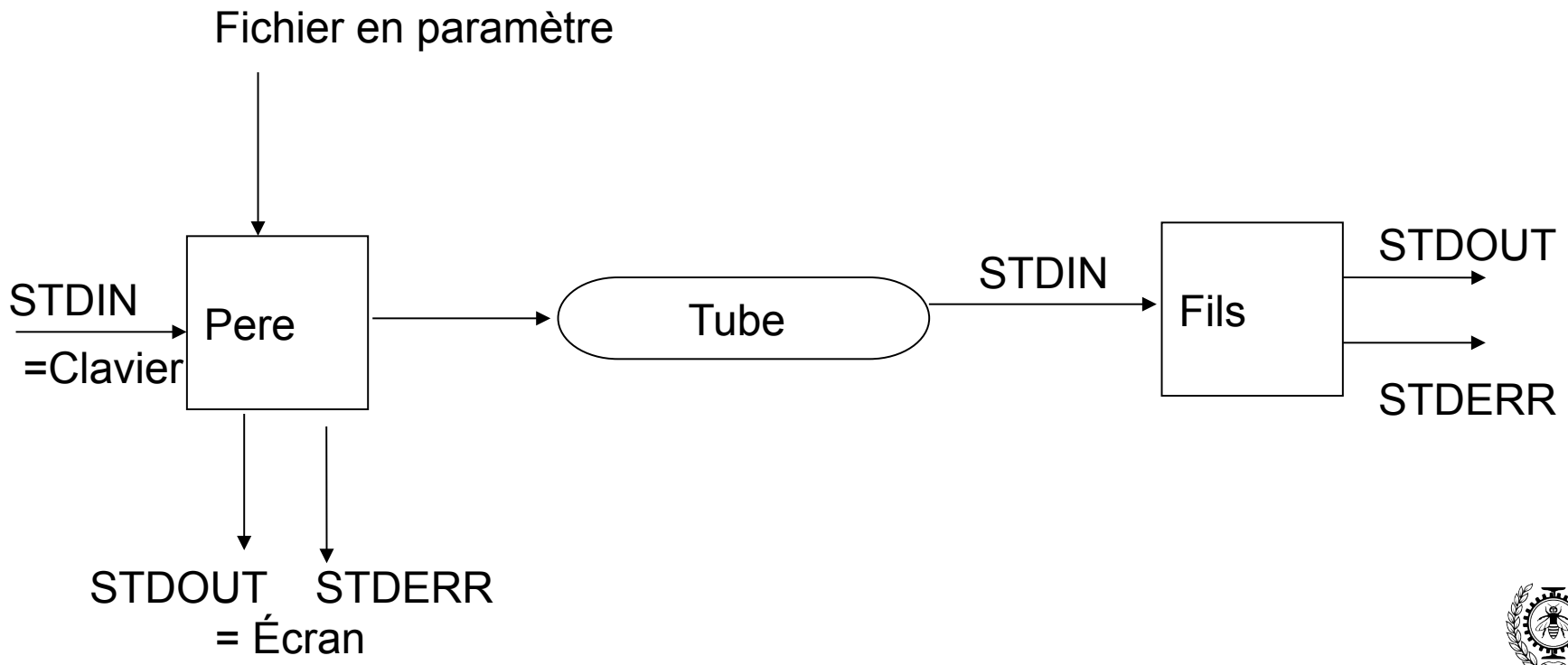
  – Le père crée un tube anonyme en spécifiant que les handles du tube peuvent être hérités par ses descendants.

  – Le père initialise adéquatement les membres **hStdInput, hStdOutput, hStdError** de la structure STARTUPINFO à faire passer à la fonction **createProcess** (ex. hStdInput = handle de lecture du pipe).

  – Le père crée un processus fils en spécifiant que son fils va hériter les handles marqués « héritables ».

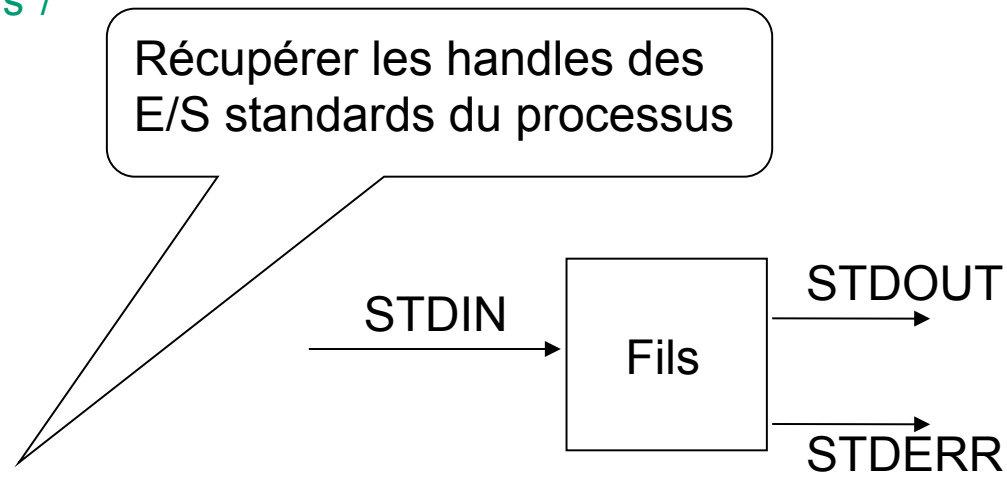
Tubes anonymes (9) : Exemple 8

Redirection des E/S standards



Tubes anonymes (10) : Exemple 8

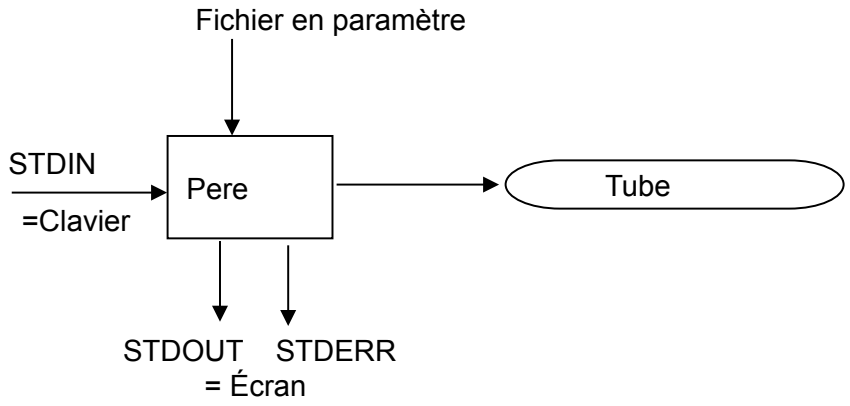
```
/* std1.c  Redirections des E/S standards*/
#include <windows.h>
#define BUFSIZE 4096

VOID main(VOID) {
    CHAR chBuf[BUFSIZE];
    DWORD dwRead, dwWritten;
    HANDLE hStdin, hStdout;
    BOOL fSuccess;

    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    hStdin = GetStdHandle(STD_INPUT_HANDLE);
    if ((hStdout == INVALID_HANDLE_VALUE) || (hStdin == INVALID_HANDLE_VALUE))
        ExitProcess(1);
    for (;;) {  // Read from standard input.
                fSuccess = ReadFile(hStdin, chBuf, BUFSIZE, &dwRead, NULL);
                if (!fSuccess || dwRead == 0)    break;
                // Write to standard output.
                fSuccess = WriteFile(hStdout, chBuf, dwRead, &dwWritten, NULL);
               if (!fSuccess)  break;
               }
}
```



Source : http://msdn.microsoft.com/en-us/library/windows/desktop/ms682499(v=vs.85).aspx

Tubes anonymes (11) : Exemple 8

```
/* std2.c */
#include <windows.h>
#include <stdio.h>
#define BUFSIZE 4096

BOOL CreateChildProcess(VOID);
VOID WriteToTube(VOID);
VOID ErrorExit(LPTSTR);

HANDLE hFd0, hFd1, hInputFile, hStdout, hStderr;

DWORD main(int argc, char *argv[]) {
   SECURITY_ATTRIBUTES saAttr;
   BOOL fSuccess;
   // Initialisation de SECURITY_ATTRIBUTES du pipe à créer
   saAttr.nLength = sizeof(SECURITY_ATTRIBUTES);
   saAttr.bInheritHandle = TRUE;
   saAttr.lpSecurityDescriptor = NULL;

   // récupérer le handle du STDOUT actuel.
   hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
```



Paramètre du programme = Nom de fichier

Les handles du pipe sont héritables par les fils.

Tubes anonymes (12) : Exemple 8

```
// Créer un tube pour le STDIN du processus enfant.
if (!CreatePipe(&hFd0, &hFd1, &saAttr, 0))
    ErrorExit("Stdout tube creation failed\n");

// Le processus enfant a juste besoin de lire du tube.
// Ne pas lui faire hériter son handle d'écriture
SetHandleInformation(hFd1, HANDLE_FLAG_INHERIT, 0);
hStderr = GetStdHandle(STD_ERROR_HANDLE);

fSuccess = CreateChildProcess();
if (!fSuccess)  ErrorExit("Create process failed with");

// ouvrir le fichier (passé en argument) en lecture et récupérer son handle.
if (argc == 1)
    ErrorExit("Please specify an input file");
printf("Debug: argv[1] = %s\n", argv[1]);
hInputFile = CreateFile(argv[1], GENERIC_READ, 0, NULL, OPEN_EXISTING,
        FILE_ATTRIBUTE_READONLY, NULL);
if (hInputFile == INVALID_HANDLE_VALUE)
    ErrorExit("CreateFile failed");
WriteToTube();
return 0;
}
```

créer un tube

créer un processus

écrire dans le tube

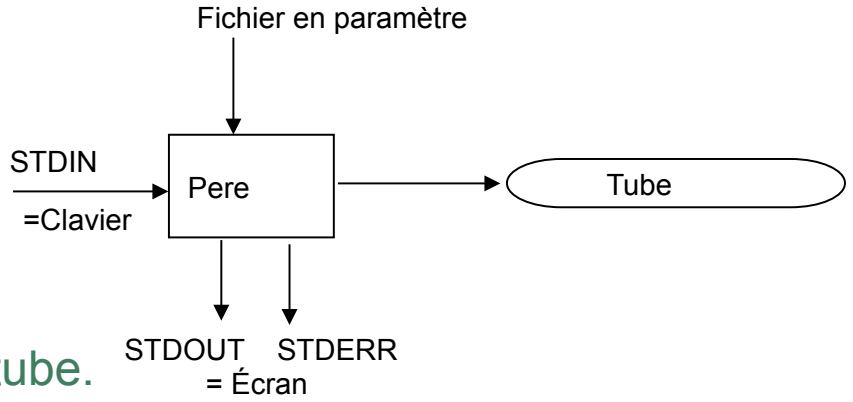
Tubes anonymes (13) : Exemple 8

```
BOOL CreateChildProcess() {
   PROCESS_INFORMATION piProcInfo;
   STARTUPINFO siStartInfo;
   BOOL bFuncRetn = FALSE;
   ZeroMemory(&piProcInfo, sizeof(PROCESS_INFORMATION));

  // Initialiser la structure STARTUPINFO.
   ZeroMemory(&siStartInfo, sizeof(STARTUPINFO));
   siStartInfo.cb = sizeof(STARTUPINFO);
   siStartInfo.hStdError = hStderr;
   siStartInfo.hStdOutput = hStdout;
   siStartInfo.hStdInput = hFd0;
   siStartInfo.dwFlags |= STARTF_USESTDHANDLES;

  // Créer le processus enfant.
   bFuncRetn = CreateProcess("std1.exe", NULL, NULL, NULL, TRUE, 0,
                             NULL, NULL,  &siStartInfo, &piProcInfo);

   return bFuncRetn;
}
```

Indiquer les E/S standards et la sortie erreur du processus à créer

Permettre l'héritage des handles

Tubes anonymes (14) : Exemple 8

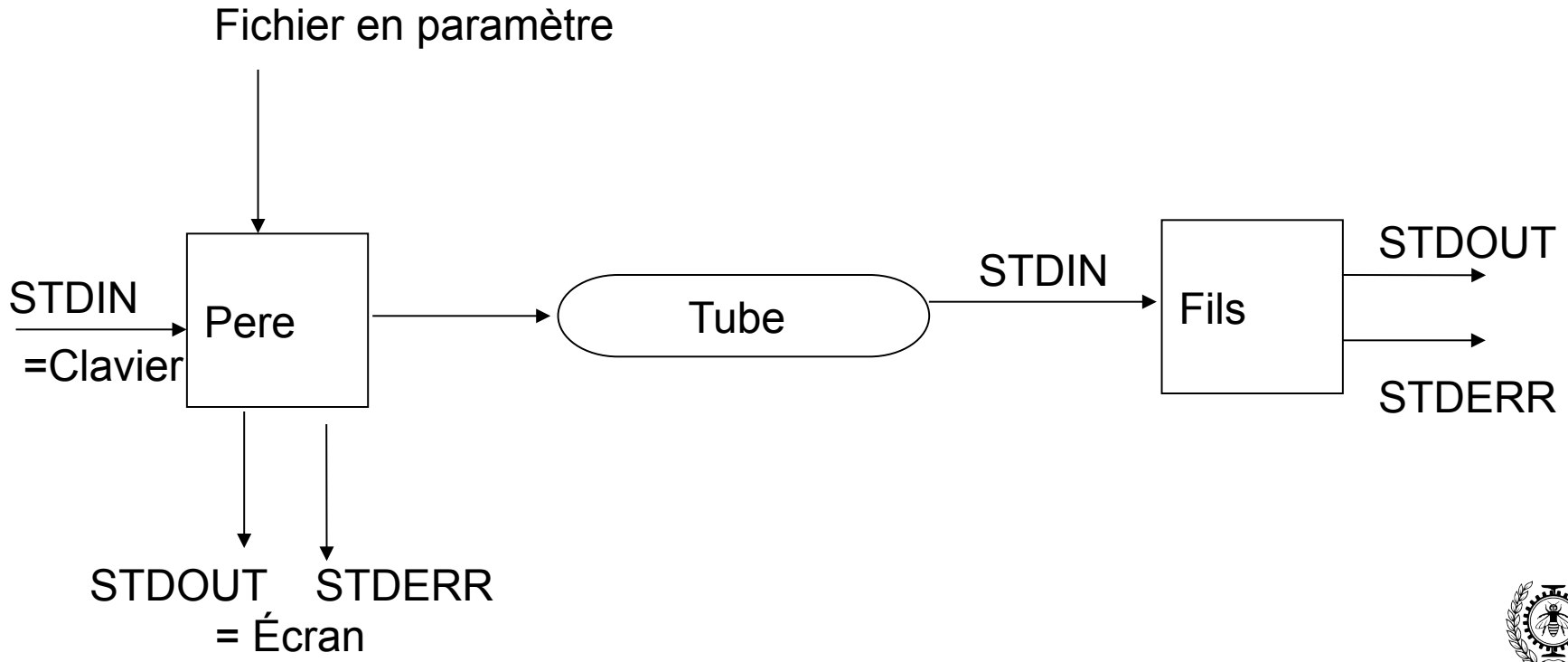



```
VOID WriteToTube(VOID) {
    DWORD dwRead, dwWritten;
    CHAR chBuf[BUFSIZE];
    // Lire le fichier et envoyer son contenu sur le tube.
    for (;;) {
        if (!ReadFile(hInputFile, chBuf, BUFSIZE, &dwRead, NULL) || dwRead == 0)
            break;
        if (!WriteFile(hFd1, chBuf, dwRead, &dwWritten, NULL))
            break;
    }
    // Fermer le handle d'écriture pour que le proc enfant arrête de lire.
      CloseHandle(hFd1);
}
```

```
VOID ErrorExit (LPTSTR lpszMessage)
{
   fprintf(stderr, "%s\n", lpszMessage);
   ExitProcess(0);
}
```

Tubes anonymes (15) : Exemple 8

Résumé



Annexe 1 : Threads – allocation dynamique

```
/* mem1.c  Allocation dynamique*/
#include <windows.h>
#include <stdio.h>
#define MAX_THREADS 3

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

DWORD WINAPI ThreadProc(LPVOID lpParam) {
    PMYDATA pData;
    // Un simple cast.
    pData = (PMYDATA) lpParam;
    // Imprimer sur la console les valeurs des variables.
    printf("Parameters = %d, %d of thread %d \n", pData->val1, pData->val2,
            GetCurrentThreadId());
    // Libérer la mémoire.
    HeapFree(GetProcessHeap(), 0, pData);
    return 0;
}
```

Threads – allocation dynamique (2)

```
int main() {
   PMYDATA pData; DWORD dwThreadId[MAX_THREADS];
   HANDLE hThread[MAX_THREADS];
   for (int i = 0; i < MAX_THREADS; i++) { // Créer MAX_THREADS threads
      // Allocation dynamique de mémoire
      pData = (PMYDATA) HeapAlloc(GetProcessHeap(),
            HEAP_ZERO_MEMORY, sizeof(MYDATA));
      if (pData == NULL)
         ExitProcess(2);
      pData->val1 = i;
      pData->val2 = i + 100;
      hThread[i] = CreateThread(NULL, // attributs de sécurité par défaut
            0,                  // taille de pile par défaut
            ThreadProc,         // pointeur vers la fonction ThreadProc
            pData,              // argument de la fonction ThreadProc
            0,                  // création de flags par défaut
            &dwThreadId[i]);    // retourne le handle du thread
      if (hThread[i] == NULL)
         ExitProcess(i); // Vérifier si le handle est valide.
   }
   WaitForMultipleObjects(MAX_THREADS, hThread, TRUE, INFINITE);
   for (i = 0; i < MAX_THREADS; i++)
      CloseHandle(hThread[i]); // Fermer tous les handles.
}
```

Threads – allocation dynamique (3)

```
Parameters = 0, 100  of thread 5184
Parameters = 1, 101  of thread 5644
Parameters = 2, 102  of thread 5524
Parameters = 3, 103  of thread 3124
Parameters = 4, 104  of thread 6016
```

Annexe 2 : Mémoire partagée

- Les processus peuvent communiquer via des zones de données partagées, de fichiers ou de fichiers mappés en mémoire.

- Pour créer une zone de données partagée par deux processus P1 et P2, il suffit de suivre les étapes suivantes :

Processus P1 :

- Crée un objet de type "fichier mappé" en appelant la fonction **CreateFileMapping** en spécifiant INVALID_HANDLE_VALUE comme premier paramètre et un nom pour l'objet à créer. On peut également spécifier les droits d'accès à l'objet. Si on utilise le flag PAGE_READWRITE, le processus aura le droit d'accéder en lecture et en écriture à la zone de données.

- Utilise le handle de l'objet créé et la fonction **MapViewOfFile** pour créer une vue du "fichier mappé" dans l'espace d'adressage du processus (attacher la zone de données à l'espace d'addressage du processus). La fonction **MapViewOfFile** retourne un pointeur vers la zone de données.

- Ferme l'object créé (**CloseHandle**), lorsque il aura fini d'utiliser l'objet. La zone de données sera libérée lorsque tous les handles de l'objet seront fermés.

Mémoire partagée (2)

Processus P2 :

- Appelle la fonction **OpenFileMapping pour ouvrir l'objet créé par P1.**

- Utilise le handle de l'objet ouvert et la fonction **MapViewOfFile** pour créer une vue du "fichier mappé" dans l'espace d'adressage du processus P2 (attacher la zone de données à l'espace d'addressage du processus).

- Ferme l'object créé (**CloseHandle**), lorsque le processus aura fini d'utiliser l'objet. La zone de données sera libérée lorsque tous les handles de l'objet seront fermés.

- L'adresse retournée par **MapViewOfFile** peut être utilisée pour lire et/ou écrire dans la zone de données. On peut également utiliser la fonction **CopyMemory** pour écrire des données dans la zone de données partagée.

  void CopyMemory(PVOID *Destination*, const VOID **Source*, SIZE_T *Length*);

Mémoire partagée (3)

```
HANDLE WINAPI CreateFileMapping(
  __in      HANDLE hFile,
  __in_opt  LPSECURITY_ATTRIBUTES lpAttributes,
  __in      DWORD flProtect,
  __in      DWORD dwMaximumSizeHigh,
  __in      DWORD dwMaximumSizeLow,
  __in_opt  LPCTSTR lpName
);
```

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

Mémoire partagée (4) : Exemple 12

```
/* shm1.c */  // Processus P1
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <tchar.h>
#define BUF_SIZE 256
TCHAR szName[] = TEXT("MyFileMappingObject");
TCHAR szMsg[] = TEXT("Message from first process.");
int main() {
   HANDLE hMapFile;
   LPCTSTR pBuf;
   hMapFile = CreateFileMapping(
         INVALID_HANDLE_VALUE, // use paging file
         NULL,                 // default security
         PAGE_READWRITE,       // read/write access
         0,                    // max. object size
         BUF_SIZE,             // buffer size
         szName);              // name of mapping object
   if (hMapFile == NULL) {
      printf(TEXT("Could not create file mapping object (%d).\n"), GetLastError());
      return 1;
   }
```

Créer un segment de données

Mémoire partagée (5) : Exemple 12

Attacher le segment de données à son espace d'adressage

```
    pBuf = (LPTSTR) MapViewOfFile(hMapFile, // handle to map object
           FILE_MAP_ALL_ACCESS,           // read/write permission
           0, 0, BUF_SIZE);
    if (pBuf == NULL) {
       printf(TEXT("Could not map view of file (%d).\n"), GetLastError());
       CloseHandle(hMapFile);
       return 1;
    }
    CopyMemory((PVOID)pBuf, szMsg, (_tcslen(szMsg) * sizeof(TCHAR)));
    _getch();
    UnmapViewOfFile(pBuf);
    CloseHandle(hMapFile);
    return 0;
}
```

Écrire dans le segment de données

Mémoire partagée (6) : Exemple 12

```
/* shm2.c */ // Processus P2
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <tchar.h>
#pragma comment(lib, "user32.lib")
#define BUF_SIZE 256
TCHAR szName[] = TEXT("Global\\MyFileMappingObject");
int _tmain() {
   HANDLE hMapFile;
   LPCTSTR pBuf;
   hMapFile = OpenFileMapping(
            FILE_MAP_ALL_ACCESS, // read/write access
            FALSE,               // do not inherit the name
            szName);             // name of mapping object
   if (hMapFile == NULL) {
      printf(TEXT("Could not open file mapping object (%d).\n"),
                        GetLastError());
      return 1;
   }
```

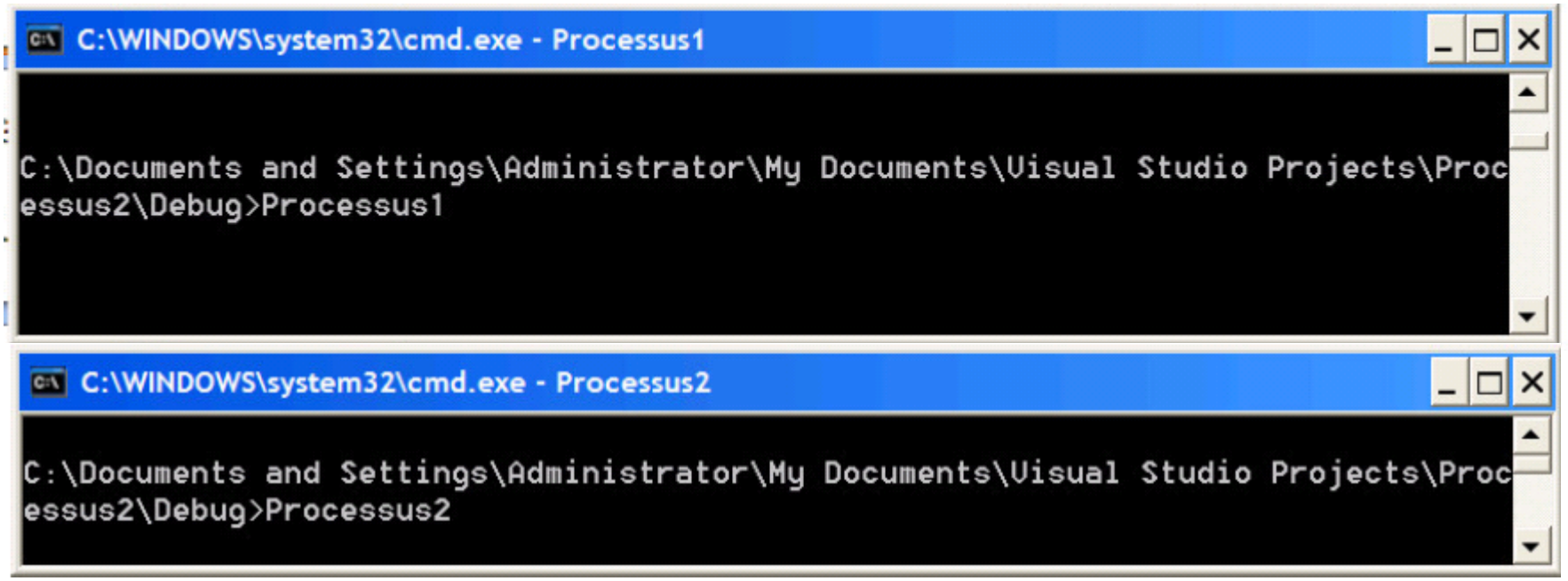
Ouvrir un segment de données

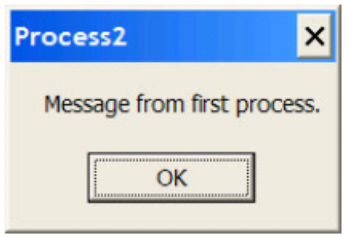
Mémoire partagée (7) : Exemple 12

Attacher le segment de données à son espace d'adressage

```
    pBuf = (LPTSTR) MapViewOfFile(
                hMapFile,           // handle to map object
                FILE_MAP_ALL_ACCESS, // read/write permission
                0, 0, BUF_SIZE);
    if (pBuf == NULL) {
        printf(TEXT("Could not map view of file (%d).\n"), GetLastError());
        CloseHandle(hMapFile);
        return 1;
    }
    MessageBox(NULL, pBuf, TEXT("Process2"), MB_OK);
    UnmapViewOfFile(pBuf);
    CloseHandle(hMapFile);
    return 0;
}
```

Mémoire partagée (8) : Exemple 12

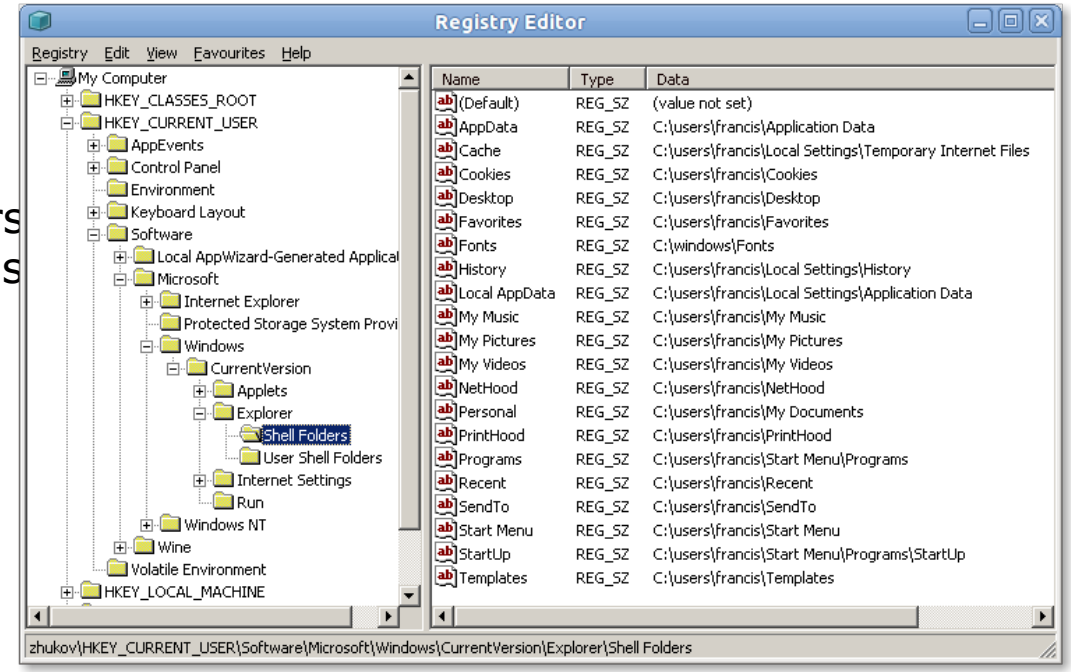




Registre Windows

http://msdn.microsoft.com/en-us/library/microsoft.win32.registry.aspx

- Registre est un croisement entre Base de données et systèmes de fichiers (nécessite des logiciels spéciaux pour gérer la complexité).
- Il est organisé en volumes séparés (ruches) conservés dans des fichiers du volume d'amorcage (C:\Windows\system32\config\)
- La ruche SYSTEM contient des informations de configuration utilisées par le programme d'amorçage.
- Regedit (interface utilisateur graphique) et PowerShell (langage de script) permettent d'ouvrir et d'explorer les répertoires (clés). Procmon surveille les accès au registre.



Suggestions de lecture et exemples de code

Processus et threads

https://msdn.microsoft.com/en-us/library/windows/desktop/ms684841%28v=vs.85%29.aspx

Gestion de la mémoire

https://msdn.microsoft.com/en-us/library/windows/desktop/aa366912%28v=vs.85%29.aspx

https://msdn.microsoft.com/en-us/library/windows/desktop/cc441804%28v=vs.85%29.aspx

https://msdn.microsoft.com/en-us/library/windows/desktop/ms682050%28v=vs.85%29.aspx

https://msdn.microsoft.com/en-us/library/windows/desktop/ms682623%28v=vs.85%29.aspx