


**INF2610**

# **Noyau d'un système d'exploitation**

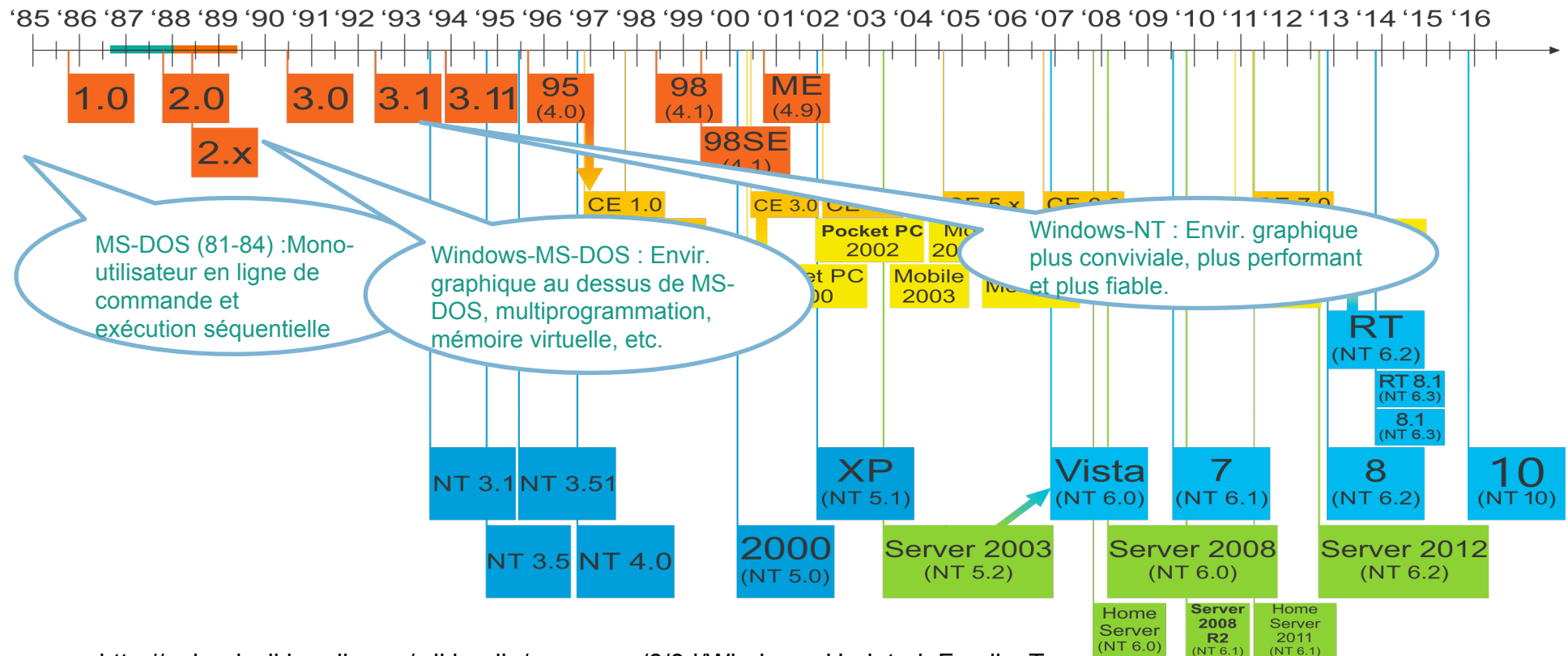


## **Chapitre 9 - Windows**

## **Chapitre 10 - Cas de Windows (API WIN32)**

-  . **Introduction**
- . **Processus et threads**
- . **Sémaphores et mutex**
- . **Communication interprocessus**
  - **Héritage d'objets**
  - **Tubes anonymes et redirection des E/S standards**
  - **Segments de données partagés**

# Introduction

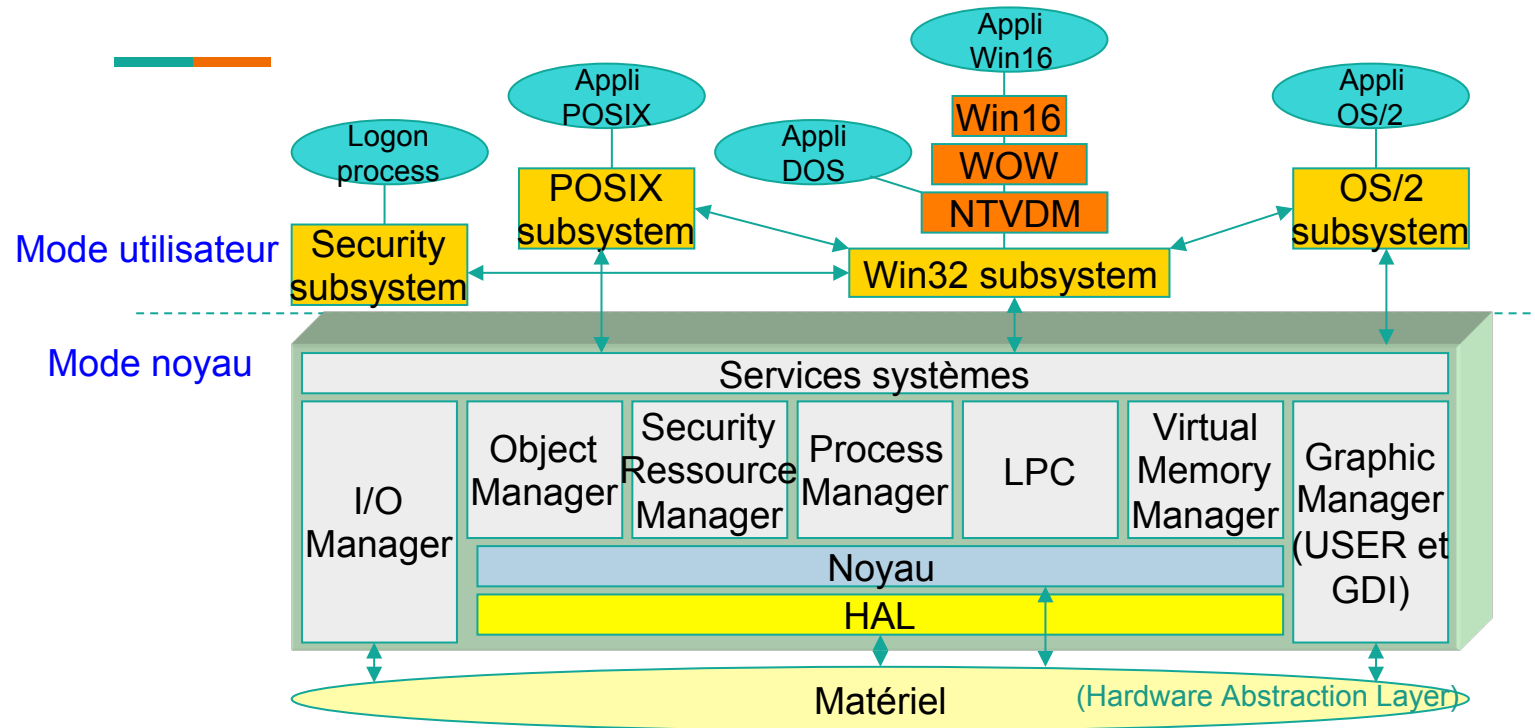


[http://upload.wikimedia.org/wikipedia/commons/6/6d/Windows\\_Updated\\_Family\\_Tree.png](http://upload.wikimedia.org/wikipedia/commons/6/6d/Windows_Updated_Family_Tree.png)

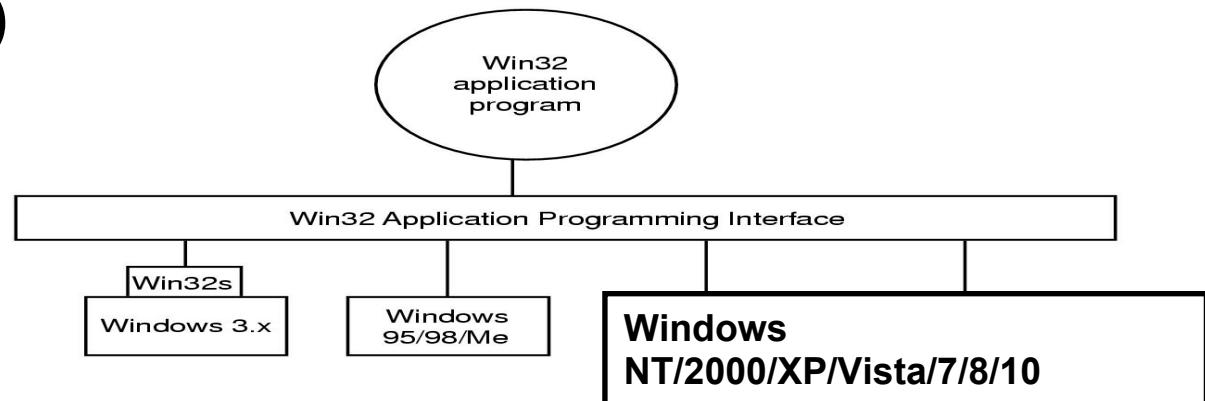
[https://fr.wikipedia.org/wiki/Histoire\\_de\\_Microsoft\\_Windows](https://fr.wikipedia.org/wiki/Histoire_de_Microsoft_Windows)

# Introduction (2)

<http://www-igm.univ-mlv.fr/~dr/XPOSE/archiNT/winnt4.html#NativeAPI>



## Introduction (3)



- 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.
- <http://msdn.microsoft.com/en-us/library/ms682425.aspx>


# 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* qui 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

- Win32 dispose d'un ensemble de fonctions (CreateXXXX, OpenXXXX) qui permettent de créer ou d'ouvrir des objets système (kernel-mode objects).
- Ces fonctions retournent des « handles » des objets créés ou ouverts via lesquelles ils sont accessibles.

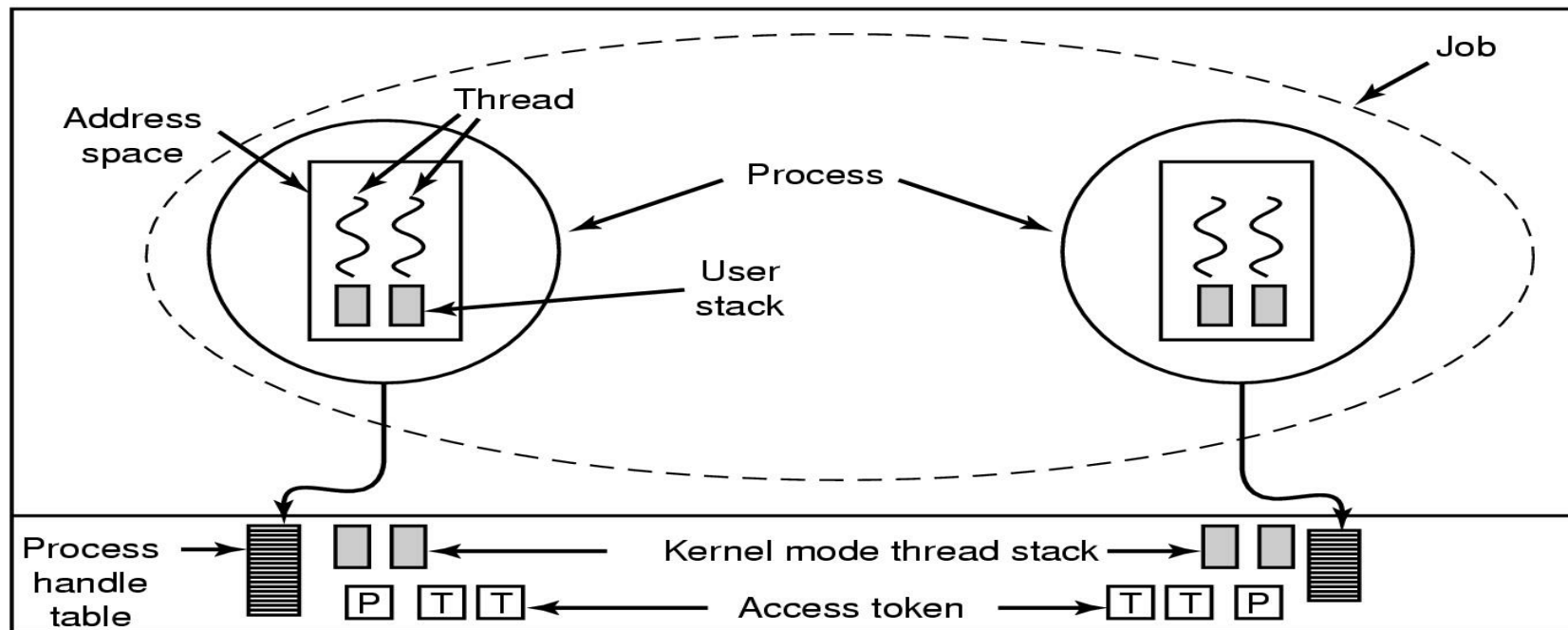
# Processus et threads



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 créée et ordonnancée au niveau utilisateur par l'application (thread utilisateur). Pour créer des fibres par un thread :
  - 1) Il commence par se convertir en fibre : **ConvertThreadToFiber()**
  - 2) Il crée ensuite des fibres **CreateFiber(...)**
  - 3) La fonction **SwitchToFiber(...)** permet de céder le CPU à une fibre.
- 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.

## Processus et threads (2)





# Processus et threads (3)

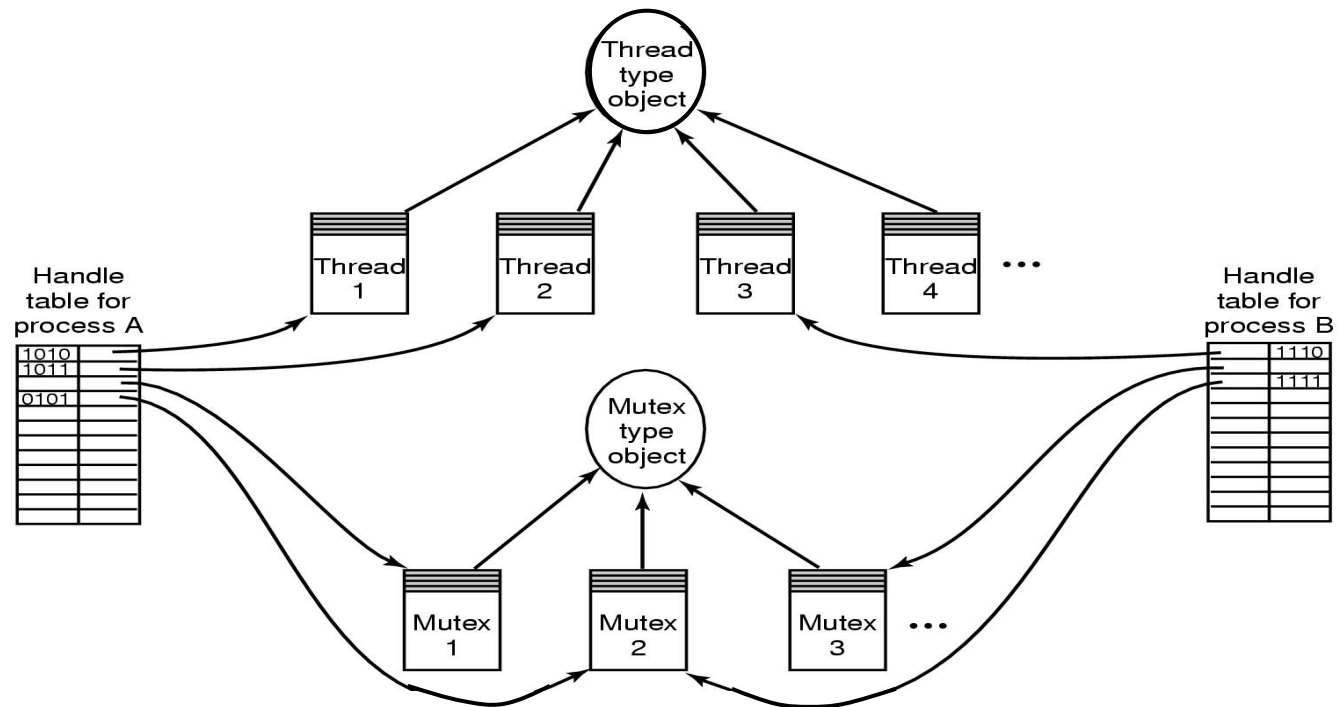
Win32 API Function	Description
CreateProcess	Create a new process
CreateThread	Create a new thread in an existing process
CreateFiber	Create a new fiber
ExitProcess	Terminate current process and all its threads
ExitThread	Terminate this thread
ExitFiber	Terminate this fiber
SetPriorityClass	Set the priority class for a process
SetThreadPriority	Set the priority for one thread
CreateSemaphore	Create a new semaphore
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

Un processus Windows est composé de :

- Un identificateur unique appelé ID du processus et un handle;
- Un espace d'adressage virtuel privé dans lequel est mappé le code, les données, les piles d'exécution, etc. et l'espace noyau.
- Une table des handles qui pointent vers diverses ressources du système telles que les sémaphores, les ports de communication, les fichiers, etc, 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, etc.

## Processus (2)



# Processus (3) : Création

BOOL WINAPI **CreateProcess**(

LPCTSTR path\_executable

LPTSTR CommandLine, // --> main (argc,argv[])

LPSECURITY\_ATTRIBUTES lpProcessAttributes,

LPSECURITY\_ATTRIBUTES lpThreadAttributes,

**BOOL bInheritHandles**, // true pour permettre au processus créé de partager les  
objets marqués héritables de son père.

DWORD dwCreationFlags, // Ex: CREATE\_NEW\_CONSOLE, classe de priorité, etc.

LPVOID lpEnvironment, // variables d'environnement

LPCTSTR lpCurrentDirectory, // répertoire courant

**LPSTARTUPINFO lpStartupInfo**, // sert notamment à indiquer les E/S standards et erreur.

**LPPROCESS\_INFORMATION lpProcessInformation** // sert à récupérer les identifiants et les  
handles du processus créé et de son thread

);

## Processus (4) : Terminaison

- La fonction **ExitProcess** permet de terminer le processus appelant (et tous ses threads) :

```
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 (5) : 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* (ou *INFINITE*):

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

- 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 *ms* *millisecondes* :

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



## Création d'un processus fils

`codes/Windows/CreateProcess`

Cet exemple repose sur deux codes utilisant l'API Win32. Le programme `parent.exe` crée un processus fils exécutant `child.exe`

# Threads : Création

**HANDLE WINAPI CreateThread(**

**LPSECURITY\_ATTRIBUTES** lpThreadAttributes, // attribut de sécurité

**SIZE\_T** dwStackSize, // taille de la pile

**LPTHREAD\_START\_ROUTINE** lpStartAddress, // la fonction du thread

**LPVOID** lpParameter, // argument de la fonction

**DWORD** dwCreationFlags, // ex. état de départ du thread créé suspendu ou prêt

**LPDWORD** lpThreadId // identifiant du thread

**);**



## Threads (2) : Terminaison

- La fonction *ExitThread* permet de terminer le thread appelant :

```
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 );
```

# Sémaphores : Création et initialisation

```
HANDLE WINAPI CreateSemaphore (  
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes, // attributs de sécurité  
    LONG lInitialCount, // valeur initiale  
    LONG lMaximumCount, // valeur maximale  
    LPCTSTR lpName // nom  
);  
  
DWORD WINAPI WaitForSingleObject ( // P du sémaphore  
    HANDLE hHandle,  
    DWORD dwMilliseconds  
);  
  
BOOL WINAPI ReleaseSemaphore ( // V du sémaphore  
    HANDLE hSemaphore,  
    LONG lReleaseCount,  
    LPLONG lpPreviousCount  
);
```

# Mutex : Création et initialisation



```
HANDLE WINAPI CreateMutex (  
    LPSECURITY_ATTRIBUTES lpMutexAttributes,  
    BOOL bInitialOwner, // false pour libre et true sinon  
    LPCTSTR lpName // nom  
);  
  
DWORD WINAPI WaitForSingleObject (  
    HANDLE hHandle,  
    DWORD milliseconds  
);  
  
BOOL WINAPI ReleaseMutex (  
    HANDLE hMutex  
);
```

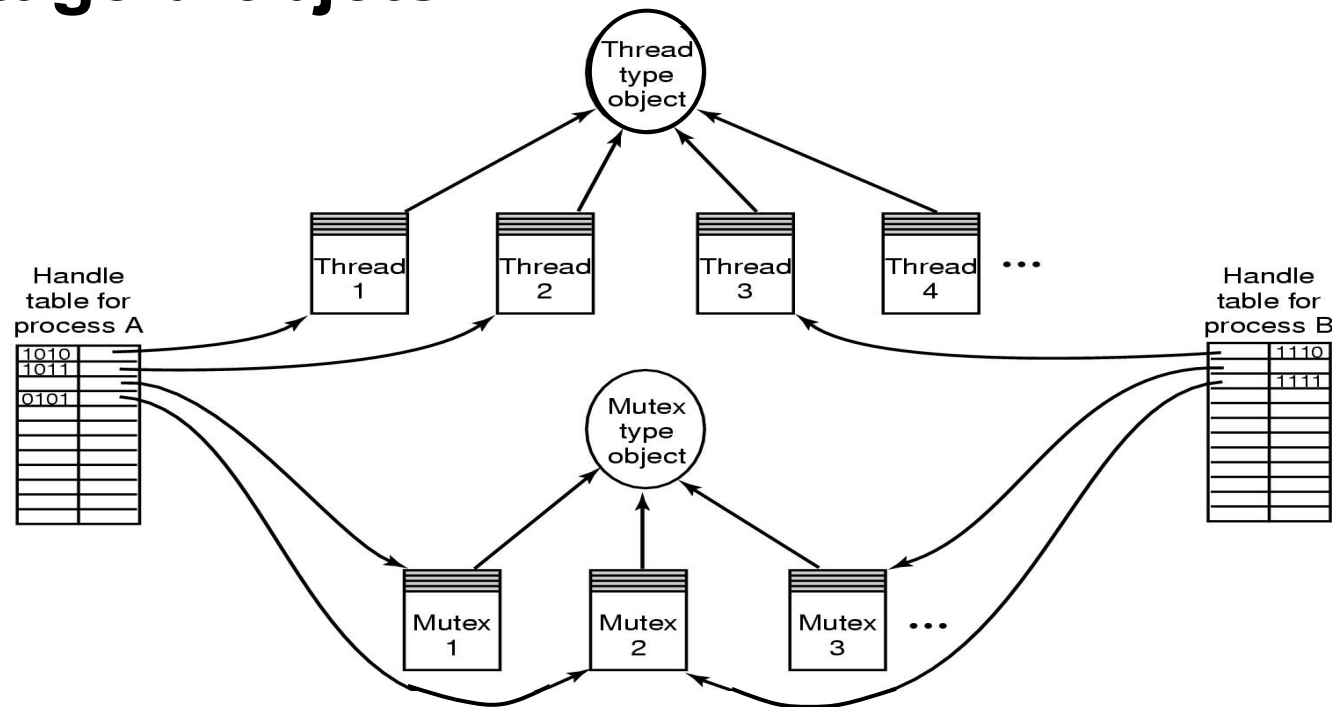


## Exemple sémaphore

codes/Windows/Semaphore

Cet exercice présente un programme qui partage une variable globale entre deux threads. l'accès à cette variable est protégé par un sémaphore binaire.

# Communication interprocessus : Héritage d'objets



# Communication interprocessus (2) :

## Héritage d'objets

- Un processus fils peut hériter des handles d'objets de son père (créés ou ouverts par 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, etc.
- Il ne peut pas hériter les handles retournés par **LocalAlloc**, **GlobalAlloc**, **HeapCreate**, et **HeapAlloc**, etc.

## Communication interprocessus (3) : Héritage d'objets

- 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.

# Tubes anonymes

```
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  
);
```

- La fonction **CreatePipe** crée un tube anonyme et retourne deux handles : un handle de lecture et un handle d'écriture.
- Les tubes anonymes Windows fonctionnent comme ceux de POSIX (pipe). Pour faire communiquer un processus père avec un processus fils via un tube anonyme, on peut procéder comme suit :
  - Le père crée un tube anonyme en spécifiant que les handles du tube sont héritables par ses descendants.
  - Le père crée un processus fils en spécifiant qu'il 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.
- Les fonctions **ReadFile** et **WriteFile** permettent de lire et écrire dans le tubes. La fonction **CloseHandle** permet de fermer un handle.





## Exemple tube anonyme

codes/Windows/Tubes

Cet exemple applique la procédure précédente pour faire communiquer un processus père avec son processus fils.

# Redirection des E/S standards

- L'avant dernier paramètre de type STARTUPINFO de la fonction CreateProcess sert notamment à préciser les E/S standards et erreur du processus à créer :
  - hStdInput
  - hStdOutput
  - hStdError
- Ces attributs permettent de rediriger les E/S standards et erreur d'un processus fils vers, par exemple, des fichiers ou des tubes anonymes.

```
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;  
    // Pour les redirections  
    HANDLE hStdInput;  
    HANDLE hStdOutput;  
    HANDLE hStdError;  
}  
STARTUPINFO, *LPSTARTUPINFO;
```



## **Exemple redirection des E/S**

codes/Windows/Redirection

Le programme tube3.c crée un tube anonyme et un processus fils en associant l'entrée standard du fils à la lecture du tube anonyme. Le père envoie ensuite un message au fils via le tube.



## Exemple mémoire partagée

codes/Windows/SharedMem

Cet exemple est composé de deux programmes shm1.c et shm2.c:

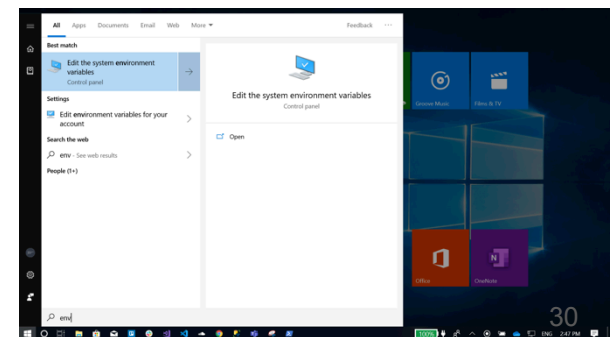
- shm1 crée un segment partagé (**CreateFileMapping**) puis l'attache à son espace d'adressage (**MapViewOfFile**),
- shm2 ouvre le segment partagé créé par shm1, l'attache à son espace d'adressage.
- shm1 et shm2 peuvent accéder au segment. Chacun détache le segment de son espace d'adressage (**UnmapViewOfFile**)

## Win32 - Types prédéfinis

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 <u>int</u>
WORD	16 bit unsigned <u>int</u>
HANDLE	opaque pointer to system data

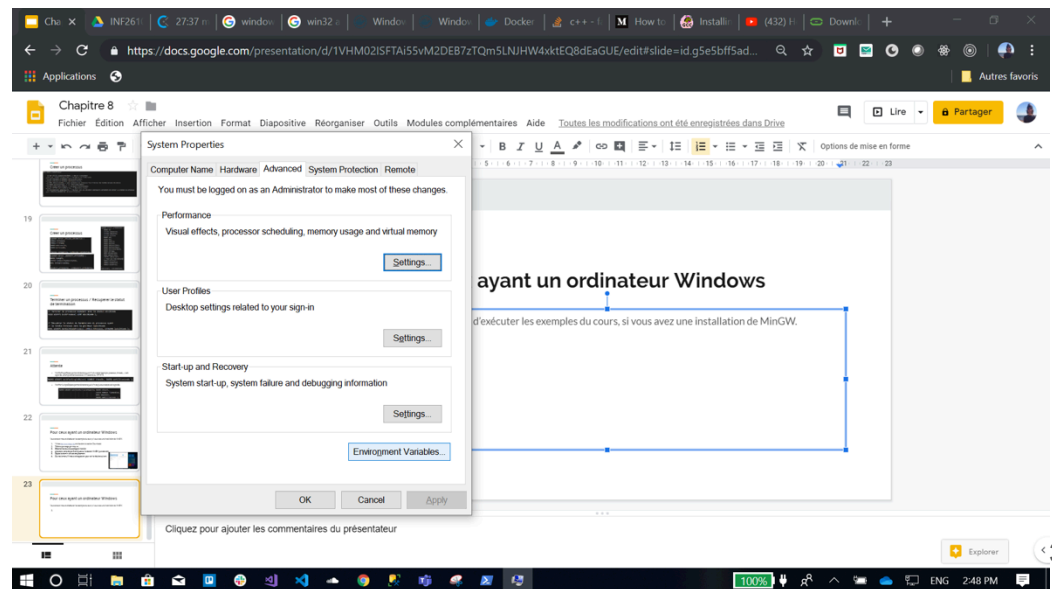
# Pour tester les exemples sous Windows

- Vous serez en mesure d'exécuter les exemples du cours, si vous avez une installation de MinGW.
  1. Visitez <http://www.mingw.org/> et allez dans la section Downloads
  2. Téléchargez **mingw-get-setup.exe**
  3. Sélectionnez tous les packages à installer
  4. Allez dans votre disque C et cliquez sur le dossier MinGW puis dans bin
  5. Copiez le chemin vers cet emplacement
  6. Ouvrez le menu Windows et tapez env pour voir le résultat suivant



# Pour tester les exemples sous Windows

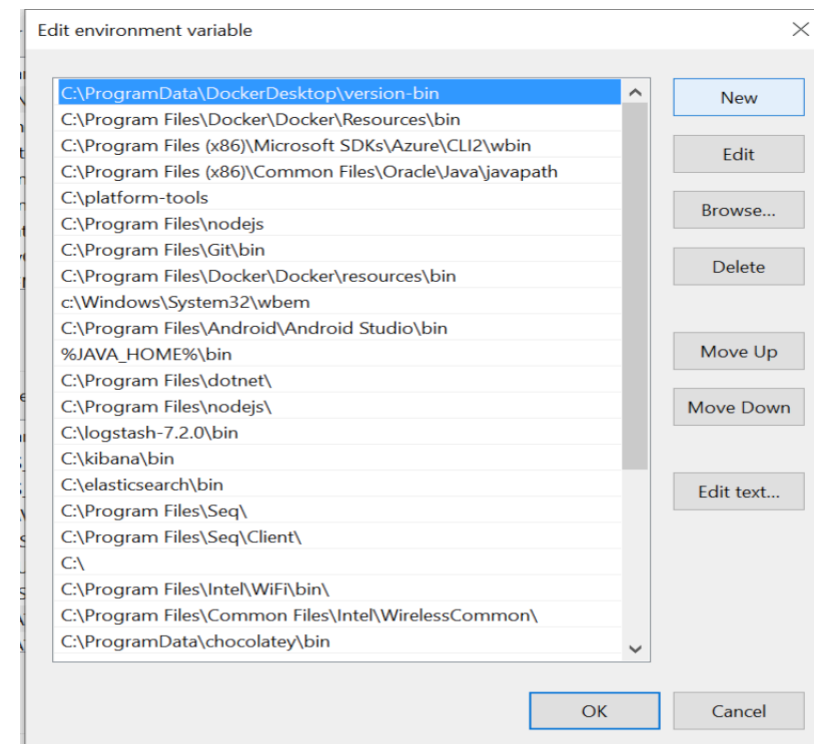
- La suite :
  1. Cliquez sur Environment Variables



# Pour ceux ayant un ordinateur Windows

La suite :

1. Dans la section system variables, choisissez la variable PATH et appuyez sur edit
2. Cliquez sur new et collez le chemin
3. Redémarrez vos terminaux
4. Vous devriez pouvoir exécuter gcc





# Références intéressantes



## 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>

## Comparaison de Windows et Linux

- <https://www.youtube.com/watch?v=19w5v9rTAu4>
- <https://technet.microsoft.com/en-us/library/bb496995.aspx>

## Exercice :

Vous devez implémenter une queue bloquante, de dimension maximale fixe, qui peut accepter des requêtes de plusieurs fils (threads) d'exécution, en utilisant l'API de Windows (par exemple pour les primitives de synchronisation comme les sémaphores et les mutex).

Fournissez la déclaration des champs de données de la classe Queue et fournissez une implémentation pour son constructeur, son destructeur et ses méthodes enqueue et dequeue dont la signature est fournie. Pour simplifier le problème, vous n'avez pas besoin de vérifier les valeurs de retour pour les conditions d'erreur.


```
Queue::Queue(int capacity);
```

```
Queue::~~Queue();
```

```
void Queue::enqueue(void *item);
```

```
void *Queue::dequeue();
```


## Exercise : Solution



```
void **queue;
HANDLE sem_free, sem_busy, mutex;
int size, ip, ic;

Queue::Queue(int capacity) {
    size = capacity;
    queue = new void*[size];
    ip = ic = 0;
    sem_free = CreateSemaphore(NULL, size, size, NULL);
    sem_busy = CreateSemaphore(NULL, 0, size, NULL);
    mutex = CreateMutex(NULL, FALSE, NULL);
}
```

## Exercise : Solution



```
void Queue::enqueue(void *item) {  
    WaitForSingleObject(sem_free, INFINITE);  
    WaitForSingleObject(mutex, INFINITE);  
    queue[ip] = item;  
    ip = (ip + 1) % size;  
    ReleaseMutex(mutex);  
    ReleaseSemaphore(sem_busy, 1, NULL);  
}
```

```
void **queue;  
HANDLE sem_free,  
        sem_busy,  
        mutex;  
int size, ip, ic;
```

```
void *Queue::dequeue() {  
    void *item;  
    WaitForSingleObject(sem_busy, INFINITE);  
    WaitForSingleObject(mutex, INFINITE);  
    item = queue[ic]; ic = (ic + 1) % size;  
    ReleaseMutex(mutex);  
    ReleaseSemaphore(sem_free, 1, NULL);  
    return item;  
}
```