Notes de cours de SEM
December 25, 2019

1 Fonctions

```
BOOL WINAPI CallNamedPipe(NOM_CANAL,str,1+strlen(str),\
  str,sizeof str,&dwEcrits,NMPWAIT_WAIT_FOREVER)
  _In_  LPCTSTR lpNamedPipeName,
  _In_  LPVOID  lpInBuffer,
  _In_  DWORD   nInBufferSize,
  _Out_ LPVOID  lpOutBuffer,
  _In_  DWORD   nOutBufferSize,
  _Out_ LPDWORD lpBytesRead,
  _In_  DWORD   nTimeOut
  //NMPWAIT_WAIT_FOREVER
);
```

Retour indiquant une erreur : 0

```
BOOL WINAPI CloseHandle(
  _In_ HANDLE hObject
);
```

Retour indiquant une erreur : 0

```
BOOL WINAPI ConnectNamedPipe(
  _In_        HANDLE       hNamedPipe,
  _Inout_opt_ LPOVERLAPPED lpOverlapped
);
```

Retour indiquant une erreur : 0

```
HANDLE WINAPI CreateEvent(
  _In_opt_ LPSECURITY_ATTRIBUTES lpEventAttributes,
  _In_     BOOL                  bManualReset,
  _In_     BOOL                  bInitialState,
  _In_opt_ LPCTSTR               lpName
);
```

Retour indiquant une erreur : `NULL`

```
HANDLE WINAPI CreateFile(
  _In_     LPCTSTR               lpFileName,
  //0 GENERIC_READ GENERIC_WRITE  ou GENERIC_READ|GENERIC_WRITE,
  0,
  NULL,
  CREATE_ALWAYS ou OPEN_ALWAYS ou OPEN_EXISTING,
,FILE_ATTRIBUTE_NORMAL,NULL
);
```

- Retour indiquant une erreur : `INVALID_HANDLE_VALUE`
- le Handle retourné est à fermer avec `CloseHandle`.
- CREATE_ALWAYS écrase le fichier existant

- OPEN_EXISTING est à privilégier pour un pipe nommé, car il génère une erreur quand le fichier n'existe pas.
- ne pas utiliser le SECURITY_ATTRIBUTES lorsque le fichier est nommé.

```
HANDLE WINAPI CreateFileMapping(
  _In_     HANDLE                hFile,
//HANDLE de CreateFILE ou INVALID_HANDLE_VALUE
  _In_opt_ LPSECURITY_ATTRIBUTES lpAttributes,
//NULL ou adresse d'une structure de type SECURITY_ATTRIBUTES pour rendre le Handle hérit
  PAGE_READONLY ou PAGE_READWRITE,
  0,
0 ou taille de la zone mémoire,
NULL ou nom de la zone mémoire
);
```

- Retour indiquant une erreur : `NULL`
- Retour si le fichier existait déjà : `ERROR_ALREADY_EXISTS`
- Le Handle retourné doit être fermé avec `CloseHandle`.

```
HANDLE CreateMailslot(nom_de_la_boite_mail,0,MAILSLOT_WAIT_FOREVER,NULL);
```

- Retour indiquant une erreur : `INVALID_HANDLE_VALUE`
- Nom de la boite mail :

```
\\\\.\\mailslot\\boitemsg
```

```
HANDLE WINAPI CreateMutex(
  _In_opt_ LPSECURITY_ATTRIBUTES lpMutexAttributes,
  _In_     BOOL                  bInitialOwner,
  _In_opt_ LPCTSTR               lpName
);
```

- Retour indiquant une erreur : `NULL`
- GetLastError() contient ERROR_ALREADY_EXISTS si un Mutex avec ce nom existait déjà.

```
HANDLE WINAPI CreateNamedPipe(
  _In_     LPCTSTR               lpName,
  _In_     DWORD                 dwOpenMode,
  //PIPE_ACCESS_DUPLEX, PIPE_ACCESS_INBOUD, PIPE_ACCESS_OUTBOUND
  _In_     DWORD                 dwPipeMode,
  //PIPE_WAIT,PIPE_TYPE_MESSAGE
  _In_     DWORD                 nMaxInstances,
0,0,0,NULL
);
```

- Retour indiquant une erreur : `INVALID_HANDLE_VALUE`
- PIPE_ACCESS_DUPLEX est obligatoire pour permettre à la fois de lire et d'écrire sur le pipe

```
BOOL WINAPI CreatePipe(
  _Out_    PHANDLE               hReadPipe,
  _Out_    PHANDLE               hWritePipe,
  _In_opt_ LPSECURITY_ATTRIBUTES lpPipeAttributes,
  0
);
```

Retour indiquant une erreur : `FALSE`

```
BOOL CreateProcess(
              NULL,
  _Inout_opt_ LPTSTR                lpCommandLine,
              SECURITY_ATTRIBUTES   lpsaPr,
              SECURTIY_ATTRIBUTES   lpsaTh,
  _In_        BOOL                  bInheritHandles,
  //concerne que les Handle à transmettre et non les
  _In_        DWORD                 dwCreationFlags,
  //0 ou CREATE_NEW_CONSOLE
              NULL,
              NULL,
  _In_        LPSTARTUPINFO         lpStartupInfo,
  _Out_       LPPROCESS_INFORMATION lpProcessInformation
);
```

- Retour indiquant une erreur : `FALSE`
- lpsaPr est à remplir pour indiquer que ProcessInformation.Process est un Handle héritable.
- lpsaTh est à remplir pour indiquer que ProcessInformation.Thread est un Handle héritable.
- bInheritHandles quand il est à FALSE fait que le process créé n'hérite d'aucun HANDLE. Quand il est à vrai, il hérite de tous les Handles héritables qui ont déjà été créés au moment de la création de ce processus.

```
HANDLE WINAPI CreateSemaphore(
  _In_opt_ LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
  _In_     LONG                  lInitialCount,
  _In_     LONG                  lMaximumCount,
  _In_opt_ LPCTSTR               lpName
);
```

- Retour indiquant une erreur : `NULL`
- test pour savoir si le fichier existait déjà : `ERROR_ALREADY_EXISTS==GetLastError()`

```
HANDLE WINAPI CreateThread(
  _In_opt_  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  _In_      SIZE_T                 dwStackSize,
  _In_      LPTHREAD_START_ROUTINE lpStartAddress,
  _In_opt_  LPVOID                 lpParameter,
  _In_      DWORD                  dwCreationFlags,
  _Out_opt_ LPDWORD                lpThreadId
);
```

Retour indiquant une erreur : `NULL`

```
BOOL WINAPI DuplicateHandle(
  _In_  HANDLE   hSourceProcessHandle,
  _In_  HANDLE   hSourceHandle,
  _In_  HANDLE   hTargetProcessHandle,
  _Out_ LPHANDLE lpTargetHandle,
        0,
  _In_  BOOL     bInheritHandle,
        DUPLICATE_SAME_ACCESS,
);
```

Retour indiquant une erreur : FALSE

Le nouveau Handle est héritable si bInheritHandle est à TRUE, sinon mettre FALSE.

```
HANDLE GetCurrentProcess (VOID);
```

Cette fonction n'est pas à tester. La valeur retournée est un pseudo-handle.

```
HANDLE GetCurrentThread (VOID);
```

Cette fonction n'est pas à tester. La valeur retournée est un pseudo-handle.

```
BOOL WINAPI GetExitCodeProcess(
  _In_  HANDLE  hProcess,
  _Out_ LPDWORD lpExitCode
);
```

Retour indiquant une erreur : FALSE

Cette fonction GetExitCodeProcess requière le fait le processus soit terminée, du coup il est nécessaire d'appliquer la fonction WaitForSingleObject pour s'en assurer.

La valeur du GetExitCodeProcess est donné par la valeur du dernier thread tué si ce n'est pas le process lui-même qui est mis fin.

```
BOOL WINAPI GetExitCodeThread(
  _In_  HANDLE  hThread,
  _Out_ LPDWORD lpExitCode
);
```

Retour indiquant une erreur : FALSE

```
DWORD WINAPI GetFileSize(
  _In_      HANDLE  hFile,
  _Out_opt_ LPDWORD lpFileSizeHigh
);
```

Retour indiquant une erreur : INVALID_FILE_SIZE

```
int WINAPI GetThreadPriority(
  _In_ HANDLE hThread
);
```

Retour indiquant une erreur : THREAD_PRIORITY_ERROR_RETURN

```
LPVOID WINAPI MapViewOfFile(
  _In_ HANDLE hFileMappingObject,
  FILE_MAP_READ ou FILE_MAP_ALL_ACCESS,
  0,0,0
);
```

- Retour indiquant une erreur : NULL
- L'adresse retournée doit être fermée avec UnmapViewOfFile

```
BOOL WINAPI GetVersionEx(
  _Inout_ LPOSVERSIONINFO lpVersionInfo
);
```

Retour indiquant une erreur : FALSE

```
HANDLE WINAPI OpenEvent(
  EVENT_ALL_ACCESS,
  _In_ BOOL    bInheritHandle, //FALSE ou TRUE si le handle doit être héritable
  _In_ LPCTSTR lpName  //nom du Event
);
```

Retour indiquant une erreur : `NULL`

```
HANDLE WINAPI OpenMutex(
  MUTEX_ALL_ACCESS,
  _In_ BOOL    bInheritHandle,//FALSE ou TRUE si le handle doit être héritable
  _In_ LPCTSTR lpName //nom du Mutex à ouvrir
);
```

Retour indiquant une erreur : `NULL`

```
HANDLE WINAPI OpenSemaphore(
  SEMAPHORE_ALL_ACCESS,
  _In_ BOOL    bInheritHandle,//FALSE ou TRUE si le handle doit être héritable
  _In_ LPCTSTR lpName//nom du semaphore à ouvrir
);
```

Retour indiquant une erreur : `NULL`

```
BOOL WINAPI ReadFile(
  _In_        HANDLE       hFile,
  _Out_       LPVOID       lpBuffer, //chaîne de caractère où se trouvera le message lu
  _In_        DWORD        nNumberOfBytesToRead,//nombre de caractères disponibles pour é
  _Out_opt_   LPDWORD      lpNumberOfBytesRead,//vrai pointeur sur un DWORD
  NULL
);
```

Retour indiquant une erreur : `FALSE` Il semble que le quatrième argument soit vraiment nécessaire

```
BOOL WINAPI ReleaseMutex(
  _In_ HANDLE hMutex
);
```

Retour indiquant une erreur : `FALSE`

```
BOOL WINAPI ReleaseSemaphore(
  _In_      HANDLE hSemaphore,
  _In_      LONG   lReleaseCount,
  _Out_opt_ LPLONG lpPreviousCount
);
```

Retour indiquant une erreur : `FALSE`

```
BOOL WINAPI ResetEvent(
  _In_ HANDLE hEvent
);
```

Retour indiquant une erreur : `FALSE`

```
DWORD WINAPI ResumeThread(
  _In_ HANDLE hThread
);
```

Retour indiquant une erreur : -1

```
BOOL SetEnvironmentVariable (
   LPCTSTR lpName,
   LPCTSTR lpValue
);
```

Retour indiquant une erreur : `FALSE`

```
BOOL WINAPI SetEvent(
  _In_ HANDLE hEvent
);
```

Retour indiquant une erreur : `FALSE`

```
BOOL WINAPI SetNamedPipeHandleState(
  _In_     HANDLE  hNamedPipe,
  _In_opt_ LPDWORD lpMode,
    // &    ->PIPE_READMODE_MESSAGE
  _In_opt_ LPDWORD lpMaxCollectionCount,
  _In_opt_ LPDWORD lpCollectDataTimeout
);
```

Retour indiquant une erreur : `FALSE`

```
BOOL WINAPI SetThreadPriority(
  _In_ HANDLE hThread,
  _In_ int    nPriority
);
```

Retour indiquant une erreur : `FALSE`

```
DWORD WINAPI SuspendThread(
  _In_ HANDLE hThread
);
```

Retour indiquant une erreur : -1

```
BOOL WINAPI TerminateProcess(
  _In_ HANDLE hProcess,
  _In_ UINT   uExitCode
);
```

Retour indiquant une erreur : `FALSE`

```
BOOL WINAPI TerminateThread(
  _Inout_ HANDLE hThread,
  _In_    DWORD  dwExitCode
);
```

Retour indiquant une erreur : `FALSE`

On peut terminer le thread d'un processus et si c'est le seul en cours il termine le process.

```
BOOL TransactNamedPipe(hPipe,str,1+strlen(str),str,sizeof str,&dwEcrits,NULL)
         HANDLE       hNamedPipe,
         LPVOID       lpInBuffer,
         DWORD        nInBufferSize,
         LPVOID       lpOutBuffer,
         DWORD        nOutBufferSize,
         LPDWORD      lpBytesRead,
         LPOVERLAPPED lpOverlapped
);
```

Retour indiquant une erreur : 0

```
BOOL WINAPI UnmapViewOfFile(
  _In_ LPCVOID lpBaseAddress
);
```

Retour indiquant une erreur : `FALSE`

```
DWORD WINAPI WaitForSingleObject(
  _In_ HANDLE hHandle,
  _In_ DWORD  dwMilliseconds
//INFINITE ou 0
);
```

- Retour indiquant une erreur : `WAIT_FAILED`
- Retour indiquant le succès : `WAIT_OBJECT_0`
- Retour indiquant que le délai est dépassé : `WAIT_TIMEOUT`

```
DWORD WINAPI WaitForMultipleObjects(
  _In_       DWORD  nCount,
//nombre de Handle
  _In_ const HANDLE *lpHandles,
//tableau de HANDLE
  _In_       BOOL   bWaitAll,
//FALSE pour attente d'un objet, TRUE pour attendre tous les objets
  _In_       DWORD  dwMilliseconds
//INFINITE
);
```

Retour indiquant une erreur : `WAIT_FAILED`

```
BOOL WINAPI WriteFile(
  _In_         HANDLE       hFile,
  _In_         LPCVOID      lpBuffer,//chaîne à écrire
  _In_         DWORD        nNumberOfBytesToWrite,//1+longueur de la chaîne
  _Out_opt_    LPDWORD      lpNumberOfBytesWritten,//un vrai pointeur sur un DWORD.
  NULL
);
```

Retour indiquant une erreur : `FALSE` Il semble que le quatrième argument soit absolument nécessaire.

2 Communications entre deux thread

Entre deux thread appartenant au même process, pour faire communiquer un objet, on peut

1. Soit utiliser un objet non-nommé,
   - on passe l'adresse du Handle dans la mémoire et l'adresse mémoire est transmise au moment de la création du thread, ou le Handle est transformé en une variable globale.
   - Il ne doit y avoir qu'un unique CloseHandle pour l'ensemble des deux thread, parce qu'il s'agit du même Handle.
2. Soit utiliser un objet nommé,
   - L'objet est ouvert dans chaque thread.
   - L'objet est fermé dans chaque thread.

3 Communications entre process

Entre deux process, pour faire communiquer un objet, on peut :

1. Soit utiliser un objet non-nommé. On passee le Handle (et non l'adresse du Handle) dans une chaîne de caractère. La communication peut se faire au moment de la création d'un process avec :
   - ajout d'arguments
   - écriture et lecture dans une variable d'environnement

   Il y a deux conditions pour que le Handle fonctionne dans le nouveau processus :
   - Le Handle doit être héritable, soit par l'utilisation de `DuplicateHandle` et `TRUE` sur `bInheritHandle` ou en remplissant correctement une structure `SECURITY_ATTRIBUTES` avec `TRUE` sur `bInheritHandle` dans la fonction qui a créé ce Handle. Ceci doit être fait avant que la création du processus.
   - Au moment de la création du process, il faut mettre à `TRUE` l'argument `bInheritHandles`.
2. Soit utiliser un objet *nommé*. Un handle est ouvert et fermé dans chaque processus.

4 Constantes

4.1 Constantes pour `SetThreadPriority` et pour `GetThreadPriority`

```
THREAD_PRIORITY_ABOVE_NORMAL,
THREAD_PRIORITY_BELOW_NORMAL,
THREAD_PRIORITY_HIGHEST,
THREAD_PRIORITY_IDLE,
THREAD_PRIORITY_LOWEST,
THREAD_PRIORITY_NORMAL,
THREAD_PRIORITY_TIME_CRITICAL
```

4.2 pour les caractères

```
UNICODE
_UNICODE
```

5 Types

```
typedef struct ... {
  DWORD  nLength;  //mettre sizeof(SECURITY_ATTRIBUTES)
  LPVOID lpSecurityDescriptor; //mettre à NULL
  BOOL   bInheritHandle;       //mettre à TRUE pour héritabilité
} SECURITY_ATTRIBUTES;
```

Ce type peut-être à utiliser par exemple pour saPr et saTh dans `CreateProcess`. Ce type peut être utilisé pour `CreateMutex` ou `CreatePipe`.

```
typedef struct ... {
  DWORD dwOSVersionInfoSize;//mettre sizeof(SECURITY_ATTRIBUTES)
  DWORD dwMajorVersion;
  DWORD dwMinorVersion;
  ...
}OSVERSIONINFO;
```

Il faut initialiser à zéro la structure. `dwMajorVersion` vaut $5, 6, 10$ suivant que la version de Windows est XP, (7,8) ou 10. Il faut utiliser

Ce type est utilisé pour la fonction `GetVersionEx`.

6 Existence d'un objet et fermeture des HANDLE

Un objet existe tant qu'il existe un Handle non-fermé. En pratique Windows détruit l'objet lorsque le dernier process ayant possédé un des Handle portant sur l'objet est terminé.

7 Fonction d'erreur

```
VOID test_erreur (LPCTSTR msg_etape)
{
DWORD erreur, code_erreur;
LPTSTR msg_erreur;
/*-------------------------------------------------------------------*/
/* Détection d'une erreur dans un appel système */
/*-------------------------------------------------------------------*/
if ( (erreur=GetLastError()) == NO_ERROR ) return;
/*-------------------------------------------------------------------*/
/* Affichage d'un message d'erreur */
/*-------------------------------------------------------------------*/
code_erreur=0x0000FFFF & erreur;
FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER |
FORMAT_MESSAGE_FROM_SYSTEM,
NULL,erreur,
MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
(LPTSTR)&msg_erreur,0,NULL);
printf("\n*** ERREUR *** Code erreur : %d (%8.8lX)\n\n"
" Etape : %s\n Erreur : %s\n\n",
code_erreur,erreur,msg_etape,msg_erreur);
LocalFree(msg_erreur);
getchar(); //pause pour attendre
ExitProcess(code_erreur);
}
```

8 Initialiser à zéro une structure

dans `windows.h`

```
ZeroMemory(add_structure,sizeof(STRUCTURE));
```

dans `string.h`

```
void * memset ( void * ptr, int value, size_t num );
```

Cette fonction met à `value` la zone mémoire s'étalant de `ptr` jusqu'à `ptr+num-1`.

9 instructions classiques C

9.1 Déclarations

La norme C oblige à ce que toutes les déclarations des variables soient faites avant la première instruction. Certains environnements ne permettent pas la compilation lorsque le fichier a l'extension .c.

9.2 Fichiers

```
int     fclose( FILE * stream );
int     feof(FILE *stream) //0 si la fin n'est atteinte
char *  fgets ( char * str, int num, FILE * stream );
FILE *  fopen ( const char * filename, const char * mode );
errno_t fopen_s(FILE** pFile,const char *filename,const char *mode);
```

9.3 Chaîne de caractères

Utilisation de la librairie `#include <string.h>`

```
size_t strlen ( const char * str );   //renvoie la longueur de la chaîne de caractère
char *  strcpy ( char * destination, const char * source );
int     strncmp(const char *string1,const char *string2,size_t count);
        //valeur retour <0,0,>0
errno_t strcpy_s(char *strDestination,size_t numberOfElements,const char *strSource);
char * strchr (char * str, int character );
char * strstr (char * str1, const char * str2 );
        //recherche de la sous-chaîne str2 dans la chaîne str1,
//renvoie une valeur différente de NULL dans le cas favorable.
char * strtok ( char * str, const char * delimiters );
        //parse la chaîne de caractère str
//en fonction des caractères de délimitations contenues dans delimiters
//à utiliser avec une boucle tant que jusqu'à ce que la valeur renvoyée soit null.
//à la deuxième utilisation mettre NULL à la place de str.
void * memset ( void * ptr, int value, size_t num );
        //Fill block of memory
        //Sets the first num bytes of the block of memory pointed by ptr
        //to the specified value (interpreted as an unsigned char).
```

Pour écrire une adresse mémoire ou un Handle dans une chaîne de caractère, on utilise `sprintf(str,"%ld",add)`. Pour lire l'adresse mémoire ou le Handle, on utilise `(unsigned long int)atoi(str)`.

10 Dans une fenêtre dos

La commande `start cmd` permet d'ouvrir une autre fenêtre dos à partir d'une fenêtre dos.

La commande `prg > print.txt` permet de mettre dans un fichier print.txt l'ensemble de l'affichage fait par l'exécutable `prg.exe`.

11 Dans CodeBlock

Il est possible de modifier le répertoire où sont les exécutables

- Création projet
- Console application
- C
- ...
- Next
- Dossier output ...