PROGRAMMATION MULTITÂCHES SOUS WINDOWS

Ce document donne des bases de programmation multitâches sous Windows. Il aborde la gestion des processus et des threads (création, synchronisation, changement de priorité) et également du verrouillage de sections critiques. Loin d'être complet, il permet un lancement simple et efficace pour développer une première et simple application multitâches. Il est conseillé de l'utiliser avec une bonne base d'aide telle que le site MSDN ou le "*Win32 Programmer's Reference*" (guide de l'API Windows).

Les prototypes des fonctions utilisées dans ce document se trouvent dans le fichier `'windows.h'`. Les programmes donnés en exemple compilent sans erreur avec les compilateurs Borland.

LES PROCESSUS

Un processus est une entité de programme en exécution ou en attente d'exécution (état prêt). Une unique application peut être composée de plusieurs processus distincts qui se trouveront alors en concurrence pour l'accès au microprocesseur, leur exécution semblera alors simultanée. Un processus père et un processus fils sont deux entités réellement séparées étant donné qu'elles ont chacune un espace mémoire virtuel différent.

Cette partie décrit la méthode de création de processus fils au sein d'un processus père. Elle explique également la méthode permettant de changer les attributs d'un processus tels que sa priorité.

Il est bon de noter que l'exécution de n'importe quelle application Windows se traduit par la création d'un processus pouvant ensuite créer d'autres processus ou des threads. Pour modifier la priorité de toute la descendance d'un processus, il est plus simple de la spécifier au démarrage de l'application. Pour cela, la création d'un raccourci est nécessaire :

```
C:\Winnt\system32\Cmd.exe /c start <priorité> <programme>
```

`<priorité>` prend les valeurs `'/low'`, `'/normal'`, `'/high'` ou `'/realtime'`.

Création d'un processus

La fonction 'CreateProcess' permet de créer un processus fils au sein d'un autre processus. On doit lui fournir des indications sur le type de processus à créer, le programme qu'il va exécuter, comment il sera capable de créer de nouveaux processus ou threads. Elle retourne une valeur logique TRUE ou FALSE selon que l'opération a réussi ou non.

Le prototype de cette fonction est le suivant :

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

- 'lpApplicationName' est un pointeur vers une chaîne de caractères contenant le chemin d'accès à l'application à exécuter par le processus fils.
- 'lpCommandLine' est une chaîne de caractères contenant la ligne de commande à exécuter. Cette chaîne peut inclure le chemin d'accès à l'application, auquel cas le paramètre 'lpApplicationName' doit être NULL.
- 'lpProcessAttributes' est un pointeur vers un attribut de sécurité indiquant si les processus fils peuvent hériter du handle du processus père.
- 'lpThreadAttributes' est un pointeur vers un attribut de sécurité indiquant si les threads peuvent hériter du handle du processus père.
- 'bInheritHandles' indique si les handles héritables sont automatiquement hérités (TRUE) ou non (FALSE).
- 'dwCreationFlags' indique la priorité du processus fils et les différents flags de création. Si ce paramètre est zéro, aucun flag n'est spécifié.
- 'lpEnvironment' est un pointeur vers une zone mémoire destinée à accueillir l'environnement du nouveau processus. Si ce pointeur est NULL, l'environnement du processus père est utilisé pour le fils.

- 'lpCurrentDirectory' est un pointeur vers une chaîne de caractères donnant le chemin d'accès au répertoire de travail de l'application exécutée par le processus fils. Si ce pointeur est NULL, le répertoire de travail du père est utilisé.
- 'lpStartupInfo' est un pointeur vers une structure de type 'STARTUPINFO' indiquant les paramètres du bureau, des fenêtres, etc.
- 'lpProcessInformation' est un pointeur vers une structure de type 'PROCESS_INFORMATION' qui permet à la fonction de renvoyer les informations d'identification du processus fils.

Exemple – Processus.c

Le programme suivant crée un processus fils exécutant le programme 'calc.exe' (la calculatrice Windows) qu'il trouve dans le répertoire 'c:\temp'. Il attend ensuite la fin de son exécution pour se terminer.

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

void main()
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

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

    // Lancement du processus fils
    if( !CreateProcess( NULL, "c:\\temp\\calc.exe", NULL, NULL, FALSE,
        0, NULL, NULL, &si, &pi ) )
       {
        printf( "Création du processus echouee..." );
       }

    // Attente de la fin du processus fils
    WaitForSingleObject( pi.hProcess, INFINITE );

    // Ferme les handles de processus et de thread
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```

Changement de la priorité d'un processus

Il faut dans un premier temps demander au système de nous fournir le handle du processus courant pour pouvoir accéder à ses propriétés. On utilise pour cela la fonction 'GetCurrentProcess'.

```
h=GetCurrentProcess();
```

Le handle retourné sera ensuite passé en argument aux fonctions gérant l'accès à la table des processus.

Pour spécifier la priorité du processus courant, on utilise la fonction '[SetPriorityClass](#)' dont le prototype est le suivant :

```
BOOL SetPriorityClass(
  HANDLE hProcess,
  DWORD dwPriorityClass
);
```

'hProcess' correspond au handle retourné par la fonction 'GetCurrentProcess' et 'dwPriorityClass' est une constante déterminant le niveau de priorité. Elle peut prendre les valeurs suivantes :

IDLE_PRIORITY_CLASS ..... Priorité basse
NORMAL_PRIORITY_CLASS..... Priorité normale
HIGH_PRIORITY_CLASS ..... Priorité haute
REALTIME_PRIORITY_CLASS .... Priorité temps-réel

Exemple

```
#include <windows.h>

HANDLE h;

void main()
{
 h=GetCurrentProcess();   // Retourne le handle du process courant
 SetPriorityClass(h, REALTIME_PRIORITY_CLASS );   // Priorité temps-réel
 SetPriorityClass(h, HIGH_PRIORITY_CLASS );       // Priorité  haute
 SetPriorityClass(h, NORMAL_PRIORITY_CLASS );     // Priorité normale
 SetPriorityClass(h, IDLE_PRIORITY_CLASS );       // Priorité basse
}
```

LES THREADS

Un thread est une entité distincte du processus lui donnant naissance mais évoluant dans le même espace mémoire virtuel. Il partagera les ressources accordées au processus père tout au long de son exécution. La commutation de contexte (mise en attente d'un thread et mise en exécution d'un autre) est plus rapide étant donné que la zone mémoire est la même. Il n'est pas nécessaire de sauvegarder ni de restaurer les zones mémoire.

Création de threads

Un thread est représenté par une fonction du programme du processus principal qui sera associée au thread au moment de sa création. Cette fonction prend un seul paramètre de type quelconque (LPVOID) qu'il faudra caster[1] au moment de son utilisation et retourne un entier non signé de type DWORD. Son prototype est donc le suivant :

```
DWORD WINAPI Thread1( LPVOID lpParam );
```

La fonction '`CreateThread`', appelée dans le processus principal permet de créer le thread et de lancer son exécution. Cette fonction prend plusieurs arguments et retourne un handle[2] vers le thread créé en cas de succès, ou la valeur NULL en cas d'échec. Le prototype de cette fonction est le suivant :

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

- '`lpThreadAttributes`' est un pointeur vers une structure qui définit si un processus fils peut hériter du handle retourné. Si ce pointeur est NULL, aucun héritage n'est possible.
- '`dwStackSize`' indique la taille de la pile accordée au thread. Si cette valeur est zéro, une taille par défaut lui est accordée.
- '`lpStartAddress`' donne l'adresse de départ du code du thread. C'est en général un pointeur vers une fonction du code que l'on veut voir exécutée par le thread.
- '`lpParameter`' est un pointeur vers la variable à passer au thread. Il s'agit d'un pointeur car le type de la variable est inconnu au moment de la création du thread.

---

[1] Changer de type. Si la variable '`Var`' est déclarée comme '`int`', la ligne '`(char)Var`' aura pour effet de la transformer en '`char`'. Les bits de poids fort seront perdus.

[2] Pointeur représentant l'objet créé dans sa zone de mémoire virtuelle.

- 'dwCreationFlags' permet d'indiquer le mode d'exécution du thread. Si cette valeur est zéro, le thread est exécuté immédiatement après sa création.
- 'lpThreadId' est un pointeur vers une variable de type DWORD accueillant le numéro d'identification du thread. Cette valeur est retournée par la fonction à la création du thread. Si le pointeur est NULL, le numéro d'identification du thread n'est pas retourné.

Exemple – Thread.c

Le programme suivant crée un thread exécutant la fonction 'Thread1'.

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

int nb;
HANDLE hThread1;
DWORD Thread1ID, Thread1Param = 100;

DWORD WINAPI Thread1( LPVOID lpParam )
{
 DWORD var;
 var = *(DWORD*)lpParam;

 printf("\tExécution du thread :\n\r");

 for(nb=1 ; nbvar ; nb++)
       {
        printf("%d ", nb);
       }
 printf("\n\r");

 printf("\t# Parametre passe au thread : %d\n\r", var);
 printf("\tFin de l'execution du thread\n\r");

 return 0;
}


void main()
{
 printf("Creation d'un thread...\n\r");
 hThread1 = CreateThread(NULL, 0, Thread1, &Thread1Param, 0,
     &Thread1ID);
 if (hThread1 == NULL)
       {
        printf("Erreur de creation du thread !\n\r");
       }
 else
       {
        printf("Creation de thread reussie !\n\r");
       }

 getchar();
}
```

Changement de la priorité d'un thread

Il est possible d'accorder à un thread une priorité différente de celle de son processus père. Pour cela, on utilise la fonction 'SetThreadPriority' dont le prototype est le suivant :

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

Cette fonction retourne un flag indiquant si l'opération s'est bien déroulée. Elle prend comme arguments le handle du thread et une constante déterminant la priorité à affecter.

'nPriority' peut prendre les valeurs suivantes :

- 'THREAD_PRIORITY_ABOVE_NORMAL' : un point au dessus de la priorité de base.
- 'THREAD_PRIORITY_ABOVE_NORMAL' : un point en dessous de la priorité de base
- 'THREAD_PRIORITY_HIGHEST' : deux points au dessus de la priorité de base.
- 'THREAD_PRIORITY_LOWEST' : deux points en dessous de la priorité de base.
- 'THREAD_PRIORITY_TIME_CRITICAL' : priorité de 15 si le processus père est exécuté en priorité normale, basse ou haute; priorité de 31 si le processus père est exécuté en priorité temps-réel.

SYNCHRONISATION D'EXÉCUTION

Objet unique

Il est possible et indispensable dans certains cas de demander à un processus d'attendre la fin de l'exécution d'un thread, d'un autre processus ou d'attendre qu'une section critique soit libérée. La fonction 'WaitForSingleObject' permet d'attendre la fin d'un unique processus ou thread. Le prototype de cette fonction est le suivant :

```
DWORD WaitForSingleObject(
  HANDLE hHandle,
  DWORD dwMilliseconds
);
```

- 'hHandle' est le handle du thread à attendre.
- 'dwMilliseconds' indique la durée maximale d'attente.

La valeur retournée indique le type de signal qui a fait terminer l'attente.

- 'WAIT_ABANDONED' indique qu'un mutex est placé par un autre thread sur l'objet attendu, l'attente ne peut pas continuer.
- 'WAIT_OBJECT_0' indique que le signal attendu est arrivé.
- 'WAIT_TIMEOUT' indique que le délai maximal d'attente a été dépassé.
- 'WAIT_FAILED' indique que la fonction ne s'est pas terminée correctement.

Exemple

L'exemple est le même que le précédent. La fonction 'main()' est modifiée pour que le processus principal attende la fin de l'exécution du thread fils.

```
void main()
{
 printf("Creation d'un thread...\n\r");
 hThread1 = CreateThread(NULL, 0, Thread1, &Thread1Param, 0,
     &Thread1ID);
 if (hThread1 == NULL)
       {
        printf("Erreur de creation du thread !\n\r");
       }
 else
       {
        printf("Creation de thread reussie !\n\r");
        printf("Attente de la fin du thread...\n\r");
        WaitForSingleObject(hThread1, INFINITE);
       }

 getchar();
}
```

Objets multiples

Il est également possible de synchroniser l'exécution d'un processus sur la terminaison de plusieurs threads ou la libération de plusieurs sections critiques. Pour cela on utilise la fonction 'WaitForMultipleObjects' dont le prototype est le suivant :

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

```
);
```

- 'nCount' est le nombre d'objets à attendre.
- 'lpHandles' est un pointeur vers un tableau de handles.
- 'bWaitAll' indique s'il faut attendre tous les objets ou simplement l'occurrence du premier.
- 'dwMilliseconds' indique la durée d'attente maximale.

Exemple

L'exemple suivant crée deux threads puis attend la fin de leur exécution avant de se terminer.

```
void main()
{
 printf("Deux threads ecrivent a l'ecran sans mutex...\n\n\r");

 hThread1 = CreateThread(NULL, 0, Thread1, &Thread1Param, 0,
      &Thread1ID);
 hThread2 = CreateThread(NULL, 0, Thread2, &Thread2Param, 0,
      &Thread2ID);

 Tableau[0] = hThread1;
 Tableau[1] = hThread2;
 WaitForMultipleObjects(2, Tableau, TRUE, INFINITE);

 getchar();
}
```

Événements

Un événement est un objet de synchronisation dont l'état est contrôlé par un processus ou un thread. La fonction 'CreateEvent' permet de créer un événement à reset automatique ou manuel. Son prototype est le suivant :

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

Cette fonction retourne un handle vers l'événement créé et prend les arguments suivants :

- 'lpEventAttributes' indique si le handle peut être hérité par un processus ou thread fils.
- 'bManualReset' indique si l'événement est à reset manuel (par appel de la fonction 'ResetEvent') ou automatique (dès qu'un processus ou thread en attente de l'événement se trouve libéré).

- 'bInitialState' indique l'état de démarrage de l'événement.
- 'lpName' est un pointeur vers une chaîne de caractères contenant le nom de l'événement.

L'événement est activé par la fonction 'SetEvent' et reseté par la fonction 'ResetEvent' dont les prototypes sont les suivants :

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

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

Exemple – Event.c

L'exemple suivant crée deux threads (Thread2 et Thread3), l'un comptant et l'autre décomptant le temps pendant une durée définie. Un troisième thread (Thread1) permet de synchroniser les deux autres pour qu'ils s'exécutent chaque seconde.

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

#define      MINUTES_MAX       1
#define      SECONDES_MAX      30

HANDLE h, hThread1, hThread2, hThread3;
DWORD Thread1ID, Thread1Param, Thread2ID, Thread2Param, Thread3ID,
      Thread3Param;
BOOL Res, Boucler1, Boucler2;
HANDLE hEvent[2];
HANDLE Tableau[3];


DWORD WINAPI Thread1( LPVOID lpParam )
{
 while(Boucler1 | Boucler2 == TRUE)
 {
  Sleep(1000);
  SetEvent(hEvent[0]);
  SetEvent(hEvent[1]);
 }
 return 0;
}


DWORD WINAPI Thread2( LPVOID lpParam )
{
 char Minutes, Secondes;
 char i;

 Secondes = SECONDES_MAX;

 for(Minutes=MINUTES_MAX ; Minutes>=0 ; Minutes--)
 {
  for( ; Secondes>=0 ; Secondes--)
       {
        WaitForSingleObject(hEvent[0], INFINITE);
        printf("Duree restante : %0.2d:%0.2d\n\r", Minutes, Secondes);
       }
```

```
   Secondes = 59;
  }

  Boucler1 = FALSE;
  return 0;
}


DWORD WINAPI Thread3( LPVOID lpParam )
{
  char Minutes, Secondes;
  char i;

  for(Minutes=0 ; Minutes<MINUTES_MAX ; Minutes++)
  {
   for(Secondes=0 ; Secondes<60 ; Secondes++)
        {
         WaitForSingleObject(hEvent[1], INFINITE);
         printf("\t\t\tDuree ecoulee : %0.2d:%0.2d\n\r", Minutes,
Secondes);
        }
  }
  for(Secondes=0 ; Secondes<=SECONDES_MAX ; Secondes++)
  {
   WaitForSingleObject(hEvent[1], INFINITE);
   printf("\t\t\tDuree ecoulee : %0.2d:%0.2d\n\r", Minutes, Secondes);
  }

  Boucler2 = FALSE;
  return 0;
}


void main()
{
  Boucler1 = TRUE;
  Boucler2 = TRUE;
  printf("Sycnhronisation de deux threads avec un troisieme...\n\n\r");

  hEvent[0] = CreateEvent(NULL, FALSE, TRUE, "Thread2 Event");
  hEvent[1] = CreateEvent(NULL, FALSE, TRUE, "Thread3 Event");

  hThread1 = CreateThread(NULL, 0, Thread1, &Thread1Param, 0,
       &Thread1ID);
  hThread2 = CreateThread(NULL, 0, Thread2, &Thread2Param, 0,
       &Thread2ID);
  hThread3 = CreateThread(NULL, 0, Thread3, &Thread3Param, 0,
       &Thread3ID);

  Tableau[0] = hThread1;
  Tableau[1] = hThread2;
  Tableau[2] = hThread3;
  WaitForMultipleObjects(2, Tableau, TRUE, INFINITE);

  getchar();
}
```

SECTIONS CRITIQUES ET EXCLUSION MUTUELLE

Lorsque deux processus ou threads différents sont amenés à accéder à une même ressource (fichier, port d'entrées/sorties, matériel quelconque), il est indispensable de s'assurer qu'ils ne puissent pas y accéder en même temps. On a donc recours au mécanisme d'exclusion

mutuelle afin de protéger la section critique. Une section critique est une zone de code dans laquelle l'accès à la ressource partagée se fait.

Exemple – SectionCritique.c

Ce premier exemple montre l'effet que peut avoir l'accès à une ressource partagée par deux threads différents sans mécanisme d'exclusion mutuelle.

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

HANDLE h, hThread1, hThread2;
DWORD Thread1ID, Thread1Param, Thread2ID, Thread2Param;
BOOL Res;
HANDLE Tableau[2];

DWORD WINAPI Thread1( LPVOID lpParam )
{
 char *Chaine = "Je suis le thread 1\n";
 char i;
 int T;

 i=0;
 while(Chaine[i] != '\0')
       {
        printf("%c", Chaine[i]);
        i++;
        T = rand()/100;
        Sleep(T);
       }

    return 0;
}

DWORD WINAPI Thread2( LPVOID lpParam )
{
 char *Chaine = "Je suis le thread 2\n";
 char i;
 int T;


 i=0;
 while(Chaine[i] != '\0')
       {
        printf("%c", Chaine[i]);
        i++;
        T = rand()/100;
        Sleep(T);
       }

    return 0;
}


void main()
{
 printf("Deux threads ecrivent a l'ecran sans mutex...\n\n\r");

 hThread1 = CreateThread(NULL, 0, Thread1, &Thread1Param, 0,
      &Thread1ID);
 hThread2 = CreateThread(NULL, 0, Thread2, &Thread2Param, 0,
      &Thread2ID);

 Tableau[0] = hThread1;
```

```
 Tableau[1] = hThread2;
 WaitForMultipleObjects(2, Tableau, TRUE, INFINITE);

 getchar();
}
```

Les mutex

Un mutex est un flag indiquant si la section critique est prise ou non. Il ne permet qu'à un seul processus d'y entrer. Chaque processus entrant en section critique est contraint de s'assurer qu'aucun autre n'y est en testant l'état du mutex. Trois étapes sont nécessaires pour cette opération. Il faut dans un premier temps créer le mutex, puis attendre qu'il soit libre pour le prendre et enfin le relâcher.

La création d'un mutex se fait par la fonction '`CreateMutex`' qui est capable de créer un mutex libre ou déjà pris par le créateur. Le prototype de cette fonction est le suivant :

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

- '`lpMutexAttributes`' est un pointeur vers une structure de type '`SECURITY_ATTRIBUTES`' qui définit les attributs d'héritage du mutex par des processus fils.
- '`bInitialOwner`' indique si le mutex créé est pris par son créateur (TRUE) ou non (FALSE).
- '`lpName`' est une chaîne de caractères indiquant le nom du mutex.

L'attente de la libération d'un mutex se fait par la fonction '`WaitForSingleObject`'. Comme pour la synchronisation de threads, on lui passe un handle qui est, dans ce cas, le handle du mutex.

On libère un mutex par la fonction '`ReleaseMutex`' dont le prototype est le suivant :

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

☐*Exemple – Mutex.c*

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

HANDLE h, hThread1, hThread2;
HANDLE hMutex;
DWORD Thread1ID, Thread1Param, Thread2ID, Thread2Param;
BOOL Res;
HANDLE Tableau[2];

DWORD WINAPI Thread1( LPVOID lpParam )
{
 char *Chaine = "Je suis le thread 1\n";
 char i;
 int T;

 WaitForSingleObject(hMutex, INFINITE);
 i=0;
 while(Chaine[i] != '\0')
       {
        printf("%c", Chaine[i]);
        i++;
        T = rand()/100;
        Sleep(T);
       }

 ReleaseMutex(hMutex);
 return 0;
}

DWORD WINAPI Thread2( LPVOID lpParam )
{
 char *Chaine = "Je suis le thread 2\n";
 char i;
 int T;


 WaitForSingleObject(hMutex, INFINITE);
 i=0;
 while(Chaine[i] != '\0')
       {
        printf("%c", Chaine[i]);
        i++;
        T = rand()/100;
        Sleep(T);
       }

 ReleaseMutex(hMutex);
 return 0;
}


void main()
{
 printf("Deux threads ecrivent a l'ecran avec mutex...\n\n\r");

 hMutex = CreateMutex(NULL, FALSE, "Seb's Mutex");

 hThread1 = CreateThread(NULL, 0, Thread1, &Thread1Param, 0,
      &Thread1ID);
 hThread2 = CreateThread(NULL, 0, Thread2, &Thread2Param, 0,
      &Thread2ID);

 Tableau[0] = hThread1;
 Tableau[1] = hThread2;
 WaitForMultipleObjects(2, Tableau, TRUE, INFINITE);

 getchar();
```

```
}
```

Les sémaphores

Le sémaphore, contrairement au mutex, permet de définir le nombre de processus pouvant accéder en même temps à une section critique. On crée un sémaphore avec la fonction '`CreateSemaphore`' dont le prototype est le suivant :

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

- '`lpSemaphoreAttributes`' est un pointeur vers une structure de données de type '`SECURITY_ATTRIBUTES`' qui indique si les threads ou processus fils peuvent hériter du sémaphore.
- '`lInitialCount`' est la valeur initiale du compteur du sémaphore. Elle doit être supérieure ou égale à zéro et inférieure ou égale à '`lMaximumCount`'. Le compteur étant inversé, quand il vaut zéro, aucun autre processus ne peut entrer en section critique.
- '`lMaximumCount`' indique le nombre maximal de processus ou threads pouvant entrer dans la section critique.
- '`lpName`' est un pointeur vers une chaîne de caractères représentant le nom du sémaphore.

Pour libérer un sémaphore, on utilise la fonction '`ReleaseSemaphore`' dont le prototype est le suivant :

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

- '`hSemaphore`' est le handle du sémaphore.
- '`lReleaseCount`' quantité dont le sémaphore doit être incrémentée.
- '`lpPreviousCount`' pointeur vers une variable de type '`LONG`' destinée à recevoir l'ancienne valeur du sémaphore.

Exemple – Semaphore.c

L'exemple suivant crée trois différent threads se partageant une ressource à laquelle seulement deux peuvent accéder.

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

HANDLE h, hThread1, hThread2, hThread3;
HANDLE hSem;
DWORD Thread1ID, Thread1Param, Thread2ID, Thread2Param, Thread3ID,
   Thread3Param;
BOOL Res;
HANDLE Tableau[3];

DWORD WINAPI Thread1( LPVOID lpParam )
{
 char *Chaine = "Je suis le thread 1\n";
 char i;
 int T;

 WaitForSingleObject(hSem, INFINITE);
 i=0;
 while(Chaine[i] != '\0')
 {
  printf("%c", Chaine[i]);
  i++;
  T = rand()/100;
  Sleep(T);
 }

 ReleaseSemaphore(hSem, 1, NULL);
 return 0;
}

DWORD WINAPI Thread2( LPVOID lpParam )
{
 char *Chaine = "Je suis le thread 2\n";
 char i;
 int T;

 WaitForSingleObject(hSem, INFINITE);
 i=0;
 while(Chaine[i] != '\0')
 {
  printf("%c", Chaine[i]);
  i++;
  T = rand()/100;
  Sleep(T);
 }

 ReleaseSemaphore(hSem, 1, NULL);
 return 0;
}


DWORD WINAPI Thread3( LPVOID lpParam )
{
 char *Chaine = "Je suis le thread 3\n";
 char i;
 int T;

 WaitForSingleObject(hSem, INFINITE);
 i=0;
 while(Chaine[i] != '\0')
```

```
 {
  printf("%c", Chaine[i]);
  i++;
  T = rand()/100;
  Sleep(T);
 }

 ReleaseSemaphore(hSem, 1, NULL);
 return 0;
}


void main()
{
 printf("Deux threads ecrivent a l'ecran avec mutex...\n\n\r");

 hSem = CreateSemaphore(NULL, 2, 2, "Seb's Semaphore");

 hThread1 = CreateThread(NULL, 0, Thread1, &Thread1Param, 0,
     &Thread1ID);
 hThread2 = CreateThread(NULL, 0, Thread2, &Thread2Param, 0,
     &Thread2ID);
 hThread3 = CreateThread(NULL, 0, Thread3, &Thread3Param, 0,
     &Thread3ID);

 Tableau[0] = hThread1;
 Tableau[1] = hThread2;
 Tableau[2] = hThread3;
 WaitForMultipleObjects(3, Tableau, TRUE, INFINITE);

 getchar();
}
```

LIENS UTILES

Les sources des programmes d'exemple – MultiTâches.zip

La bibliothèque MSDN – Centre d'aide au développement de Microsoft.

LCC-Win32 – un compilateur gratuit pour application Windows portables. Ce site distribue un doc. sur l'API Windows au format HLP.