Tutorial d'initiation
A la programmation avec l'API Windows

24 novembre 2003

Par Bob et CGi

Chapitre 1

Les bases d'un programme Windows

1. Création d'une boîte de dialogue

Cours théorique :

Bien comprendre le principe de fonctionnement d'une application Windows est essentiel avant de commencer à programmer. Je vais ici exposer le principe global de fonctionnement d'une application créant une fenêtre.

Tout d'abord, il convient de bien garder à l'esprit qu'un programme fonctionnant sous Windows et gérant une fenêtre doit rester en dialogue permanent avec Windows. Le programme ne connait (a priori) rien sur son environnement. C'est Windows qui signale à l'application si elle doit redessiner le contenu d'une de ses fenêtres, ou encore si l'utilisateur essaie de fermer la fenêtre.

Cette communication se fait au travers de messages que Windows envoie à chaque fenêtre concernée. C'est au programme d'effectuer la réception de ces messages et de les transmettre aux fonctions gérant les différentes fenêtres. La réception de ces messages ne doit donc pas prendre de retard, et le traitement de chacun des messages doit être bref. En cas de retard, le redessinement des fenêtres n'est donc plus assuré, ce qui a pour conséquence des fenêtres blanches, indéplaçables, similaires à celles des programmes "plantés".

Chaque fenêtre est associée à une fonction ou procédure de fenêtre (Window Proc). Parfois plusieurs fenêtres peuvent être associées à une même procédure. Chaque message reçu est transmis à la procédure correspondante qui se chargera de traiter ce message (redessiner la fenêtre, la redimensionner, afficher un caractère entré par l'utilisateur...).

Une partie du travail de rafraichissement de la fenêtre est pris en charge par Windows. Le programme n'a à redessiner que la zone client de sa fenêtre (et non pas la barre de titre, les menus éventuels...).

Projet N°1 :

- Objectif : créer une boîte de dialogue et une procédure très simple chargée de la gérer.

- Réalisation : la boîte de dialogue sera créée grâce à l'éditeur de ressources de VC++. Le programme sera chargé de la réception des messages et du traitement des plus simples.

Télécharger le projet ici.

- Le projet pas à pas :

Tout d'abord, il s'agit de créer le projet dans VC++ : **File**/**New**/**Projects**/**Win32 Application**. Donnez ensuite un nom à votre projet et cliquez 'Ok'. Ensuite, sélectionnez 'An empty project' puis 'Finish' et 'Ok'. Nous venons de créer un projet vide, il ne contient aucun fichier de code.

Ajoutons maintenant un fichier de code et fichier de ressources. Allez dans **File**/**New**/**Files**/**C++ Source File**. Appelez ce fichier main.cpp et validez. Refaites la même opération mais avec un fichier de type **Ressource Script** nommé res. Fermez le fichier qui s'est ouvert dans l'éditeur. Le script de ressources est associé à un fichier d'en-tête créé et mis à jour automatiquement, mais qu'il convient d'ajouter. Dans le feuillet **FileView** du panneau de gauche, faites un clic droit sur **Header Files** et sélectionnez **Add Files to Folder**. Ajoutez le fichier 'ressource.h' qui doit être dans le même répertoire que votre projet.

Créons à présent le modèle de notre boîte de dialogue. Allez dans le feuillet **RessourceView** du panneau de travail (à gauche). Faites un clic droit sur **res ressources** et sélectionnez **Insert**/**Dialog**/**New**. Nous ne modifirons pas cette boîte de dialogue.

Occupons-nous maintenant du code qui va constituer le programme. Tout d'abord, les fichiers d'en-tête, deux sont nécessaires, **Windows.h** et **ressource.h** que nous avons inclus précédemment.

Le point d'entrée du programme est la fonction WinMain(). Elle est exécutée automatiquement par Windows lorsque le programme est démarré. Mais avant cette fonction, nous aurons la définition de la procédure de notre fenêtre principale, MainPoc().

```
#include <Windows.h>
#include "ressource.h"


LRESULT CALLBACK MainProc(HWND Dlg,UINT message,WPARAM wParam,LPARAM lParam);


int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     int       nCmdShow)
{
```

Nous allons maintenant créer la boîte de dialogue grâce à la fonction CreateDialog(). Le paramètre HINSTANCE est un identifiant de notre application, on passe l'identifiant fourni par Windows en paramètre de la fonction WinMain(). On donne ensuite l'identifiant ressource de notre boîte de dialogue. Elle n'a pas de fenêtre parent et la procédure est la fonction MainProc(). La fonction retourne une variable de type HWND qui identifie notre fenêtre. Puis, on affiche cette fenêtre. Toute fenêtre créée est invisible par défaut.

```
    HWND hDlg;
    hDlg=CreateDialog(hInstance,(LPCTSTR)IDD_DIALOG1,NULL,(DLGPROC)MainProc);
    ShowWindow(hDlg,SW_SHOW);
```

Occupons-nous maintenant de la réception des messages. La réception des messages est prise en charge par la fonction GetMessage(). Cette fonction retourne FALSE dès que le message WM_QUIT est reçu, ce qui indique que l'application doit être fermée.

```
    MSG msg;
```

```
   while(GetMessage(&msg,NULL,0,0)==TRUE)
   {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
   }
   return 0;
}
```

Il ne reste plus qu'à définir le comportement de notre fenêtre dans quelques cas simples: elle doit être fermée et l'application quittée si l'utilisateur presse 'ok' ou essaie de quitter. La fermeture de la fenêtre ne signifie pas forcément l'arrêt de l'application. Ces deux comportements correspondent à deux messages distincs: WM_QUIT indiquie que l'application doit se terminer, WM_CLOSE indique la fermeture de la fenêtre. Mais dans ce cas, on a une fenêtre un peu particulière, puisqu'il s'agit d'une boîte de dialogue. Sa procédure est définie de la même manière que celle d'une fenêtre.

```
LRESULT CALLBACK MainProc(HWND Dlg,UINT message,WPARAM wParam,LPARAM lParam)
{
   int Select;
   switch(message)
   {
   case WM_COMMAND:
      Select=LOWORD(wParam);
      switch(Select)
      {
      case IDOK:
         EndDialog(Dlg,0);
         PostQuitMessage(0);
         return TRUE;
      case IDCANCEL:
         EndDialog(Dlg,Select);
         PostQuitMessage(0);
         return TRUE;
      }
   default:
      return FALSE;
   }
}
```

Cette procédure est très simple, elle ne réagit qu'à la pression des boutons 'Ok' et 'Cancel' (le bouton ou la "croix"). En réponse à l'un ou l'autre de ces événements, la procédure demande la fermeture de la boîte de dialogue et termine l'application en envoyant un message WM_QUIT grâce à la fonction PostQuitMessage().

2. Création d'une fenêtre

Cours théorique :

Dans la première partie de ce chapitre, nous avons vu comment créer une boite de dialogue. La création d'une boite de dialogue grâce à la fonction CreateDialog() est relativement simple puisqu'une grande partie du travail est effectué par CreateDialog().

Nous allons donc voir maintenant comment créer une fenêtre. Cette fenêtre pourra être personnalisée et ne réferrera à aucune resource. Notre programme n'utilisera donc pas de resources.

Comme nous avons pu le voir dans la partie précédente, le programme n'intervient en aucun cas dans le dessin de la fenêtre (barre de titre, menu...). Le programme n'est responsable que du redessinement de la zone client de la fenêtre. Comme il existe plusieurs types de fenêtre (avec un menu, sans menu, pouvant être minimisée ou pas...) il est nécessaire que de créer un 'protoype de fenêtre'. Ce prototype est aussi appelé classe de la fenêtre. C'est sur ce prototype que Windows se basera pour créer la fenêtre.

Une fois le prototype de fenêtre créé, nous demanderons à Windows de créer une fenêtre suivant ce prototype. La fenêtre créée devra elle aussi disposer d'une procédure permettant de traiter les messages qu'elle reçoit. Cette procédure est très similaire à celle gérant la boite de dialogue de la partie précédante. Elle ne traite cependant pas exactement les messages de la même manière.

Projet N°2 :

- Objectif : créer une fenêtre sans utiliser de resource et une procédure permettant sa gestion.

- Réalisation : la fenêtre sera créée sans utiliser de resource et sa procédure traitera les messages les plus simples.

Télécharger le projet ici.

- Le projet pas à pas :

Il s'agit tout d'abord de créer un projet dans Microsoft Visual C++. Le projet sera créé de la même manière que dans la partie précédente, mais sans y ajouter de **Ressource Script**.

Le point d'entrée WinMain() est un standard à toute application Windows, ce programme débutera donc de la même manière que le précédant.

```
#include <Windows.h>


LRESULT CALLBACK MainProc(HWND Dlg,UINT message,WPARAM wParam,LPARAM lParam);


int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     int       nCmdShow)
{
```

Il faut mintenant créer une nouvelle classe de fenêtre. Cette classe est créée au moyen de la fonction RegisterClassEx(). La création de cette classe paut paraître fastidieuse et comprend un grand nombre de paramètres. nous n'utiliserons pas tous les paramètres, donc ne vous effrayez pas si vous ne comprenez pas totalement l'utilité de chaqun d'eux.

```
WNDCLASSEX principale;
principale.cbSize=sizeof(WNDCLASSEX);
principale.style=CS_HREDRAW|CS_VREDRAW;
principale.lpfnWndProc=MainProc;
principale.cbClsExtra=0;
principale.cbWndExtra=0;
principale.hInstance=hInstance;
principale.hIcon=LoadIcon(NULL,IDI_APPLICATION);
principale.hCursor=LoadCursor(NULL,IDC_ARROW);
principale.hbrBackground=reinterpret_cast<HBRUSH>(COLOR_WINDOW+1);
principale.lpszMenuName=NULL;
principale.lpszClassName="std";
principale.hIconSm=LoadIcon(NULL,IDI_APPLICATION);
RegisterClassEx(&principale);
```

principale.style=CS_HREDRAW|CS_VREDRAW indique que notre fenêtre devra être redessinée en cas de redimentionnement horizontal ou vertical. Nous indiquons ensuite quelle procédure sera chargée de gérer la fenêtre. L'instance de notre application est elle aussi passée en paramètre. Les curseurs ou icones de notre fenêtre sont ceux par défaut de Windows. La fenêtre n'as pas de menu, la couleur de fond est celle de Windows par défaut. Le nom de la classe de la fenêtre 'std' est un choix personnel. Il n'a d'importance que pour nous. Une fois la structure remplie, un appel à RegisterClassEx() demande à Windows de mémoriser la classe.

Maintenant que nous disposons de notre classe de fenêtre, il ne reste plus qu'a la créer grâce à la fonction CreateWindowEx().

```
HWND hWnd;
hWnd=CreateWindowEx(
   WS_EX_CLIENTEDGE,
   "std",
   "Notre fenêtre",
   WS_OVERLAPPEDWINDOW,
   CW_USEDEFAULT,
   CW_USEDEFAULT,
   CW_USEDEFAULT,
   CW_USEDEFAULT,
   NULL,
   NULL,
   hInstance,
   NULL
);

ShowWindow(hWnd,SW_SHOW);
```

La réception des messages se fait exactement de la même manière que dans le projet précédent.

```
MSG msg;
while(GetMessage(&msg,NULL,0,0)==TRUE)
{
   TranslateMessage(&msg);
```

```
      DispatchMessage(&msg);
   }
   return 0;
}
```

La définition de la procédure de cette fenêtre est la même que dans le programme précédent. Mais son fonctionnement interne est différent.

Tout d'abord, cette fenêtre ne contient pas de bouton. Sa fermeture se fera lors d'un clic sur la 'croix'. Les messages non traités sont passés à DefWindowProc(). Les messages traités seront WM_PAINT et WM_DESTROY. WM_PAINT indique que nous devons redessiner la zone client. Ici le traitement de ce message ne fera rien et pourrait être ignoré. Il est inclu dans le seul but de comprendre le mécanisme de redessinement d'une fenêtre. Le message WM_DESTROY indique que la fenêtre est en train d'être détruite (probablement suite au clic sur la 'croix'). Le fermeture de la fenêtre ne signifie pas forcément l'arret de l'application. Mais comme dans ce cas notre application n'est formée que d'une fenêtre, nous demanderons à terminer l'application dans ce cas en postant un message WM_QUIT.

```
LRESULT CALLBACK MainProc(HWND hWnd, UINT mes, WPARAM wParam, LPARAM lParam)
{
   HDC hDC;
   PAINTSTRUCT paintst;
   switch (mes)
   {
   case WM_PAINT:
      hDC=BeginPaint(hWnd,&paintst);
      EndPaint(hWnd,&paintst);
      return 0;
   case WM_DESTROY:
      PostQuitMessage(0);
      return 0;
   default:
      return DefWindowProc(hWnd, mes, wParam, lParam);
   }
}
```

Le traitement du message WM_PAINT peut vous paraitre étrange. En effet, pourquoi effectuer un BeginPaint() et un EndPaint() alors que nous ne dessinons rien. La raison est très simple. A partir du moment ou nous décidons de traiter ce message, nous devons assurer un traitement minimum par la suite de ces deux fonction, même si rien n'est dessiné.

3. Affichage dans une fenêtre

Cours théorique :

Maintenant que nous savons créer une fenêtre, nous allons voir comment dessiner dans sa zone client. L'affichage dans la zone client d'une fenêtre se fait au niveau du message WM_PAINT. Pour provoquer l'envoi d'un tel message, il faut demander le redessinement d'une zone de la fenêtre. Lors du dessin dans la partie WM_PAINT, il sera impossible d'afficher hors de la zone de redessinement.

Pour dessiner dans une fenêtre, nous allons utiliser un contexte d'affichage. Il est propre à la fenêtre dans laquelle nous dessinons et nous est fourni par Windows. Il identifie en fait la zone de l'écran dans laquelle nous allons dessiner.

Projet N°3 :

- Objectif : créer une fenêtre et afficher l'heure.

- Réalisation : la création de la fenêtre se fera de manière identique au projet précédent. Seule sera ajoutée la partie permettant l'affichage de l'heure courante.

Télécharger le projet ici.

- Le projet pas à pas :

La création de la fenêtre se fait exactement de la même manière que dans le projet précédent. Nous allons simplement utiliser les fonctions contenues dans stdio.h.

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


LRESULT CALLBACK MainProc(HWND Dlg,UINT message,WPARAM wParam,LPARAM lParam);


int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     int       nCmdShow)
{
   WNDCLASSEX principale;
   principale.cbSize=sizeof(WNDCLASSEX);
   principale.style=CS_HREDRAW|CS_VREDRAW;
   principale.lpfnWndProc=MainProc;
   principale.cbClsExtra=0;
   principale.cbWndExtra=0;
   principale.hInstance=hInstance;
   principale.hIcon=LoadIcon(NULL,IDI_APPLICATION);
   principale.hCursor=LoadCursor(NULL,IDC_ARROW);
   principale.hbrBackground=reinterpret_cast<HBRUSH>(COLOR_WINDOW+1);
   principale.lpszMenuName=NULL;
   principale.lpszClassName="std";
   principale.hIconSm=LoadIcon(NULL,IDI_APPLICATION);
   RegisterClassEx(&principale);

   HWND hWnd;
   hWnd=CreateWindowEx(
```

```
      WS_EX_CLIENTEDGE,
      "std",
      "Notre fenêtre",
      WS_OVERLAPPEDWINDOW,
      CW_USEDEFAULT,
      CW_USEDEFAULT,
      CW_USEDEFAULT,
      CW_USEDEFAULT,
      NULL,
      NULL,
      hInstance,
      NULL
   );
   ShowWindow(hWnd,SW_SHOW);
```

Comme nous allons afficher l'heure, il est important que nous soyons prévenus de manière régulière. Nous allons demander à Windows de nous avertir toutes les secondes, de manière à afficher l'heure régulièrement. Notre fenêtre recevra un message WM_TIMER toutes les secondes.

```
   SetTimer(hWnd,NULL,1000,NULL);
```

La réception des messages se fait exactement de la même manière que dans le projet précédent.

```
   MSG msg;
   while(GetMessage(&msg,NULL,0,0)==TRUE)
   {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
   }
   return 0;
}
```

La procédure fonctionne de manière identique que précédemment. Nous allons ajouter le traitement du message WM_TIMER qui sera recu toutes les secondes. Ce message demandera le redessinement d'une zone de l'écran (la zone ou nous afficherons l'heure).

```
LRESULT CALLBACK MainProc(HWND hWnd, UINT mes, WPARAM wParam, LPARAM lParam)
{
   HDC hDC;
   PAINTSTRUCT paintst;
   RECT rcClient;
   switch (mes)
   {
      case WM_TIMER:
         rcClient.top=0;
         rcClient.left=0;
         rcClient.right=100;
         rcClient.bottom=50;
         RedrawWindow(hWnd,&rcClient,NULL,RDW_ERASE|RDW_INVALIDATE|RDW_ERASE ✂
NOW|RDW_NOCHILDREN);
```

```
        return 0;
```

Un message WM_PAINT sera donc reçu toutes les secondes et à chaque fois que la fenêtre doit être redessinée. La fenêtre doit être redessinée si elle est couverte puis découverte par une autre fenêtre (par exemple).

Nous allons afficher l'heure en utilisant une police spécifique. Une fois cette police créée, il est nécessaire d'indiquer que nous allons l'utiliser dans notre contexte d'affichage (HDC). Le contexte d'affichage (ou Device Context) réfère à la zone client de notre fenêtre et contient tous ses paramètres graphiques (disposition sur l'écran, nombre de couleurs affichables...). Il est identifié par la variable hDC qui nous est passée par Windows au travers de la fonction BeginPaint().

Une fois la police créée et le texte dessiné, il convient de détruire la police.

L'heure courante sera récupérée grace à la fonction GetLocalTime().

```
   case WM_PAINT:
      char buf[256];
      SYSTEMTIME CurrentTime;
      HFONT hFont;
      hFont=CreateFont(20,0,0,0,700,FALSE,FALSE,FALSE,0,OUT_DEFAULT_PRECIS,CLIP_ ✂
DEFAULT_PRECIS,DEFAULT_QUALITY,DEFAULT_PITCH|FF_DONTCARE,"Comic Sans MS");
      hDC=BeginPaint(hWnd,&paintst);
      SelectObject(hDC,hFont);
      GetLocalTime(&CurrentTime);
      sprintf(buf,"%d : %d : %d",CurrentTime.wHour,CurrentTime.wMinute,CurrentTi ✂
me.wSecond);
      TextOut(hDC,0,0,buf,strlen(buf));
      EndPaint(hWnd,&paintst);
      DeleteObject(hFont);
      return 0;
   case WM_DESTROY:
      PostQuitMessage(0);
      return 0;
   default:
      return DefWindowProc(hWnd, mes, wParam, lParam);
   }
}
```

Il est essentiel de bien comprendre le principe d'affichage dans une fenêtre car il est très utilisé par Windows. Quelque soit le cas, l'affichage se fait toujours entre deux fonctions qui indiquent le début et la fin de l'affichage. Il ne faut jamais oublier de libérer le contexte d'affichage après l'avoir utilisé. Ici, la libération du contexte d'affichage se fait par la fonction EndPaint().

4. Lecture dans un fichier et boîte de dialogue

Cours théorique :

Nous allons maintenant revenir sur un programme utilisant des ressources. En effet, l'utilisation de boites de dialogues créées en ressources permettent bien souvent de gagner un temps précieux pour des programmes de petites tailles et de faible complexité. C'est le cas du programme que nous allons réaliser ici.

Nous allons donc aborder deux thèmes principaux: les boites de dialogues et les controles prédéfinis, ainsi que la lecture d'un fichier par des fonctions de l'API Windows, donc sans utiliser les fonctions du C traditionnel. Les controles prédéfinis sont les 'edit boxes', 'combo boxes', 'buttons', etc. Leur utilisation est très simple car ils sont gérés de manière entièrement autonome. Ils envoient à la fenêtre qui les a créés des messages de notification pour indiquer les actions de l'utilisateur: clic sur le boutton, saisie d'un caractère...

La lecture du fichier se fait de manière relativement simple, et fonctionne par un systeme de HANDLE (comme beaucoup de fonctions de l'API Windows) qu'il faut créer puis détruire.

Projet N°4 :

- Objectif : créer une boite de dialogue permettant la saisie d'un fichier et affichant les 500 premiers octets de ce fichier.

- Réalisation : nous allons ici utiliser une ressource pour créer la boite de dialogue. La lecture du fichier se fera au niveau de la procédure de la boite de dialogue car elle est très brève. Nous utiliserons ici une fonction qui permet de créer la boite de dialogue et de relever les messages, la partie 'main' du programme sera donc très courte (2 lignes).

Télécharger le projet ici.

- Le projet pas à pas :

Nous allons créer un programme avec un fichier de ressources, inclure les fichiers correspondants (voir parties précédentes). La fonction 'WinMain()' est excessivement simple puisque la fonction DialogBox() s'occupe de créer la boîte de dialogue et de récupérer ses messages. Cette fonction retourne seulement une fois que la boite de dialogue est fermée.

```
#include <Windows.h>


LRESULT CALLBACK MainProc(HWND Dlg,UINT message,WPARAM wParam,LPARAM lParam);


int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     int       nCmdShow)
{
   DialogBox(hInstance,(LPCTSTR)IDD_MAIN,NULL,(DLGPROC)MainProc);
   return 0;
}
```

La procédure de la boîte de dialogue est très simple, elle ne gère que trois messages provenant de trois bouttons. IDOK est envoyé lors d'un clic sur le

boutton OK, IDCANCEL est envoyé lors d'un clic sur la 'croix' et IDC_LIRE est un message envoyé lors d'un clic sur le boutton IDC_LIRE que nous avons créé. Les variables créées en début de procédure serviront à la lecture du fichier que nous verrons après.

```
LRESULT CALLBACK MainProc(HWND Dlg,UINT message,WPARAM wParam,LPARAM lParam)
{
   int Select;
   char buf[501];
   HANDLE hFile;
   DWORD Read;
   switch(message)
   {
   case WM_COMMAND:
      Select=LOWORD(wParam);
      switch(Select)
      {
      case IDC_LIRE:
```

La réelle nouveauté de ce programme va se faire ici. Il s'agit tout d'abord de récupérer le nom de fichier entré par l'utilisateur. Comme il est entré dans un contrôle prédéfini, nous allons utiliser la fonction GetDlgItemText() et nous placerons ce nom de fichier dans le buffer créé au debut de cette procédure sous le nom de 'buf'. Une fois ce nom de fichier récupéré, nous allons essayer d'ouvrir le fichier donné en lecture. Si nous n'y parvenons pas, un message d'erreur sera affiché grâce à la fonction 'MessageBox()'. Le traitement du message sera alors arrêté.

Remarque: la fonction MessageBox() ne retourne qu'une fois que la boîte de dialogue créée est fermée. Mais la récupération des messages de notre boîte de dialogue est assurée par MessageBox(). Le traitement prolongé de ce message n'est donc pas gênant dans ce cas puisque les messages continueront d'arriver à notre procédure.

```
         GetDlgItemText(Dlg,IDC_FILENAME,buf,256);
         hFile=CreateFile(buf,GENERIC_READ,FILE_SHARE_READ,NULL,OPEN_EXISTING,FI ✂
LE_ATTRIBUTE_NORMAL,NULL);
         if(hFile==INVALID_HANDLE_VALUE)
         {
            MessageBox(Dlg,"Erreur, impossible d'ouvrir le fichier spécifié.","E ✂
rreur",MB_OK);
            return 0;
         }
```

Arrivés à ce stade nous savons que le fichier saisi est valide et qu'il a pu être ouvert. Il ne reste plus qu'à lire les 500 premiers octets. Ils seront placés dans le buffer. Comme cette chaîne vas être afichée, elle doit être terminée par un caractère NULL. C'est pour cette raison que 'buf' a été défini comme un tableau de 501 caractères (500 pour la lecture et 1 pour le caractère NULL). Comme la lecture peut faire moins de 500 octets, nous nous baserons sur la valeur retournée par ReadFile() pour ajouter le caractère NULL. Puis nous afficherons ce buffer dans l'edit box créé a cet effet grace à SetDlgItemText().

```
         ReadFile(hFile,buf,500,&Read,NULL);
         CloseHandle(hFile);
         buf[Read]='';
         SetDlgItemText(Dlg,IDC_TEXT,buf);
         return 0;
```

Le reste du traitement de la procédure ne présente pas de difficulté particulière. Il est similaire aux traitements précédents. La seule différence notalbe est lors de la fermeture de la boîte de dialogue. Comme elle a été créée avec la fonction DialogBox() il ne faut pas utiliser PostQuitMesage() pour la quitter mais EndDialog().

```
      case IDOK:
         EndDialog(Dlg,0);
         return TRUE;
      case IDCANCEL:
         EndDialog(Dlg,0);
         return TRUE;
      }
   default:
      return FALSE;
   }
}
```

Ce programme est relativement simple. La gestion de l'interface graphique ne prend que quelques lignes. Nous voyons donc bien l'intérêt de l'utilisation des ressources et donc contrôles prédéfinis. Sans l'aide des ressources, ce programme aurait été beaucoup plus compliqué à réaliser.

5. Pas à pas pour C++ Builder

Nous allons reprendre la 4ème partie : **"Lecture dans un fichier et boîte de dialogue."** mais adaptée à **C++ Builder**. Deux procédures sont traitées dans ce document une pour BCB6 et une pour BCB4.

Pour les projets sans boîte de dialogue la procédure est la même sauf qu'il n'y a pas besoin de rajouter de fichiers ressources.

BCB 6

Faire "Nouveau". (appelle la boîte de dialogue Nouveaux éléments)

Sur la boîte de dialogue "Expert console" :

Sur "Type de source" sélectionner "C" ou "C++". (Pour l'exemple cela n'a pas d'importance)

Aucune des CheckBox à droite ne doit être cochée.

Clicker sur le bouton "Ok".

Maintenant nous avons le code minimum pour ce type d'application:

```
#include <windows.h>
#pragma hdrstop


//---------------------------------------------------------------------------
```

```
#pragma argsused
WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,int ✂
 nCmdShow)
{
        return 0;}
```

Enregister le projet sous le nom choisi. Ici "main.cpp" et "main.bpr". (Faire "Fichier" puis "Tout enregistrer")

J'ai utilisé "Resource WorkShop" (livré avec BCB) pour faire le fichier Dialog1.rc mais il peut se faire avec n'importe quel éditeur de ressource.

Puis il faut ajouter le fichier ressource au projet:

Sur le menu faire "Projet" puis "Ajouter au projet"

Selectionner le type de fichier voulu ici "Fichier ressource (*.rc)"

Puis sélectionner le fichier ici "Dialog1.rc"

Puis faire Ouvrir

Le code source est identique a l'exemple Visual C++ (J'ai mis les mêmes noms d'identificateur)

Les source des fichiers "Dialog1.rc" et "ressource.h" sont a la fin de ce document.

Fichier "main.cpp":

```
#include <windows.h>
#pragma hdrstop //Pas obligatoire
#include "ressource.h"


//---------------------------------------------------------------------------



LRESULT CALLBACK MainProc(HWND Dlg,UINT message,WPARAM wParam,LPARAM lParam);


#pragma argsused //ajouté automatiquement mais pas nécéssaire.


WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, in ✂
t nCmdShow)


{
     DialogBox(hInstance,(LPCTSTR)DIALOG_1,NULL,(DLGPROC)MainProc);
     return 0;
}
//---------------------------------------------------------------------------
```

```
LRESULT CALLBACK MainProc(HWND Dlg,UINT message,WPARAM wParam,LPARAM lParam)
{
   int Select;
   char buf[501];
   HANDLE hFile;
   DWORD Read;
   switch(message)
   {
   case WM_COMMAND:
      Select=LOWORD(wParam);
      switch(Select)
      {
      case IDC_LIRE:
         GetDlgItemText(Dlg,IDC_FILENAME,buf,256);
         hFile=CreateFile(buf,GENERIC_READ,FILE_SHARE_READ,NULL,OPEN_EXISTING,FI
LE_ATTRIBUTE_NORMAL,NULL);
         if(hFile==INVALID_HANDLE_VALUE)
         {
            MessageBox(Dlg,"Erreur, impossible d'ouvrir le fichier spécifié.","E
rreur",MB_OK);
            return 0;
         }
         ReadFile(hFile,buf,500,&Read,NULL);
         CloseHandle(hFile);
         buf[Read]='';
         SetDlgItemText(Dlg,IDC_TEXT,buf);
         return 0;



      case IDOK:
         EndDialog(Dlg,0);
         return TRUE;
      case IDCANCEL:
         EndDialog(Dlg,0);
         return TRUE;
      }
   default:
      return FALSE;
   }
}
```

Faire "Nouveau". (appelle la boîte de dialogue Nouveaux éléments)

Sur la boîte de dialogue "Nouveaux éléments" Onglet "Nouveau" DbClk sur l'icône "Expert console". (appelle la boîte de dialogue "Expert d'application console")

Sur la boîte de dialogue "Expert d'application console" :

Sur "Type de fenêtre" sélectionner "Windows (GUI)".

Sur "Type d'exécution" sélectionner "EXE".

Clicker sur le bouton "Terminer".

Maintenant nous avons le code minimum pour ce type d'application:

```
#include <windows.h>
#pragma hdrstop
#include <condefs.h>


//---------------------------------------------------------------------------
#pragma argsused
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
        return 0;
}
```

Enregister le projet sous le nom choisi. Ici "main.cpp". (Faire "Fichier" puis "Tout enregistrer")

J'ai utilisé "Resource WorkShop" (livré avec BCB) pour faire le fichier Dialog1.rc mais il peut se faire avec n'importe quel éditeur de ressource.

Puis il faut ajouter le fichier ressource au projet:

Sur le menu faire "Projet" puis "Ajouter au projet"

Selectionner le type de fichier voulu ici "Fichier ressource (*.rc)"

Puis sélectionner le fichier ici "Dialog1.rc"

Puis faire Ouvrir

Cela va ajouter cette ligne dans le code :

USERC("Dialog1.rc");

Le reste du code est identique a l'exemple Visual C++ (J'ai mis les mêmes noms d'identificateur)

Fichier "main.cpp":

```
#include <windows.h>
#pragma hdrstop
#include <condefs.h>
#include "ressource.h"
```

```
//---------------------------------------------------------------------------
USERC("Dialog1.rc");
//---------------------------------------------------------------------------


LRESULT CALLBACK MainProc(HWND Dlg,UINT message,WPARAM wParam,LPARAM );


#pragma argsused //ajouté automatiquement mais pas nécéssaire.
WINAPI WinMain(HINSTANCE hInstance, HINSTANCE, LPSTR, int)
{
   DialogBox(hInstance,(LPCTSTR)DIALOG_1,NULL,(DLGPROC)MainProc);
   return 0;
}


LRESULT CALLBACK MainProc(HWND Dlg,UINT message,WPARAM wParam,LPARAM )
{
   int Select;
   char buf[501];
   HANDLE hFile;
   DWORD Read;
   switch(message)
   {
   case WM_COMMAND:
      Select=LOWORD(wParam);
      switch(Select)
      {
      case IDC_LIRE:
         GetDlgItemText(Dlg,IDC_FILENAME,buf,256);
         hFile=CreateFile(buf,GENERIC_READ,FILE_SHARE_READ,NULL,OPEN_EXISTING,FI ✂
LE_ATTRIBUTE_NORMAL,NULL);
         if(hFile==INVALID_HANDLE_VALUE)
         {
            MessageBox(Dlg,"Erreur, impossible d'ouvrir le fichier spécifié.","E ✂
rreur",MB_OK);
            return 0;
         }
         ReadFile(hFile,buf,500,&Read,NULL);
         CloseHandle(hFile);
         buf[Read]='';
         SetDlgItemText(Dlg,IDC_TEXT,buf);
         return 0;


      case IDOK:
```

```
         EndDialog(Dlg,0);
         return TRUE;
     case IDCANCEL:
         EndDialog(Dlg,0);
         return TRUE;
      }
   default:
      return FALSE;
   }
}
```

Autres fichiers

Identique pour BCB4 et BCB6

Fichier "Dialog1.rc"

```
#include "ressource.h"

DIALOG_1 DIALOG 14, 30, 212, 227
STYLE WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "DIALOG_1"
FONT 8, "MS Sans Serif"
{
 CONTROL "Ok", IDOK, "BUTTON", BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE | WS_TABSTO
P, 56, 193, 36, 14
 CONTROL "Cancel", IDCANCEL, "BUTTON", BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE | W
S_TABSTOP, 115, 193, 38, 14
 CONTROL "", IDC_FILENAME, "EDIT", ES_LEFT | WS_CHILD | WS_VISIBLE | WS_BORDER |
 WS_TABSTOP, 26, 18, 91, 12
 CONTROL "", IDC_TEXT, "EDIT", ES_LEFT | ES_MULTILINE | WS_CHILD | WS_VISIBLE |
WS_BORDER | WS_TABSTOP, 18, 49, 175, 119
 CONTROL "Lire", IDC_LIRE, "BUTTON", BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE | WS_
TABSTOP, 136, 18, 50, 14
}
```

Fichier "Ressource.h"

```
#define DIALOG_1    1


#define IDC_FILENAME 101
#define IDC_TEXT 102
#define IDC_LIRE 103
```

6. Lecture d'un fichier WAV

Cours théorique :

A présent, nous allons réaliser un nouveau projet qui sera déjà un programme 'utile' en soi. Evidemment, il restera très sommaire et se contentera de lire un fichier WAV sélectionné par l'utilisateur.

Nous inviterons tout d'abord l'utilisateur à choisir le fichier qu'il veut lire à partir d'une boîte de dialogue 'parcourir'. Une fois que l'utilisateur aura choisi le fichier qu'il désire lire, il pourra choisir de l'écouter. La lecture du fichier WAV se fera grâce à une fonction multimédia fournie par Windows: sndPlaySound(). Cette fonction est entièrement autonome et lira le fichier en arrière plan sans la moindre intervention de la part du programme. Cet automatisme est très pratique puisqu'il permet de lire le fichier sans se soucier de rien. Cependant, les possibilités de cette fonction restent restreintes. On pourra se satisfaire de cette fonction dans quasiment toutes les applications multimédias.

Projet N°5 :

- Objectif : créer une boite de dialogue permettant la saisie d'un fichier WAV puis sa lecture.

- Réalisation : nous allons utiliser un fichier ressources qui sera ici très adapté. Le programme ne contiendra que la fonction WinMain() et une procédure pour la boite de dialogue.

Télécharger le projet ici.

- Le projet pas à pas :

Comme dans le programme précédent, nous allons créer un projet d'application Win32, ainsi qu'un fichier de ressources. La fonction WinMain() contiendra simplement l'appel de notre boîte de dialogue. Remarquons tout de même que la fonction sndPlaySound() que nous allons utiliser se trouve dans la librairie 'winmm.lib' qui n'est pas une librairie standard. Il faut donc ajouter cette librairie. Ceci peut être fait grâce au menu Project/Setting/Link/Input. Dans le champ contenant déjà les librairies par défaut, ajoutez 'winmm.lib'. Sans cet ajout, nous obtiendrions une erreur à la compilation indiquant que la fonction ne peut pas être trouvée.

La boîte de dialogue contiendra un champ 'Edit Box' (IDC_NOMDEFICHIER)dans lequel sera affiché le nom du fichier à lire. Nous cocherons la case 'Read Only' dans les options de ce champ de manière à empêcher l'utilisateur de saisir lui même un nom de fichier. De cette manière, la possibilité de choix erronés se trouve déjà limitée.

Nous ajouterons un bouton 'Parcourir' avec pour identifiant 'IDC_PARCOURIR' et un bouton 'Lire' (IDC_LIRE). Pour quitter, l'utilisateur devra utiliser la 'croix' (ID_CANCEL).

La fonction WinMain() ne diffère pas par rapport au programme précédent (si ce n'est dans les entêtes à inclure). Nous ajouterons une variable globale de manière à pouvoir accéder à l'instance du programme dans toutes les fonctions du programme (et ici dans la procédure de la boîte de dialogue).

```
#include <Windows.h>
#include <mmsystem.h>
#include "ressource.h"
```

```
HINSTANCE hInst;


LRESULT CALLBACK MainProc(HWND Dlg,UINT message,WPARAM wParam,LPARAM lParam);


int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     int       nCmdShow)
{
   hInst=hInstance;
   DialogBox(hInstance,(LPCTSTR)IDD_MAIN,NULL,(DLGPROC)MainProc);
   return 0;
}
```

La procédure de la boîte de dialogue n'est pas très compliquée. Il faudra toutefois y ajouter les boutons 'Lire' et 'Parcourir'. Le bouton parcourir sera le plus difficile à ajouter.

Nous allons réserver un buffer de taille MAX_PATH (qui est une constante marquant la taille maximale d'un nom de fichier), ainsi qu'une structure OPENFILENAME qui nous servira pour la boîte de dialogue 'Parcourir'.

```
LRESULT CALLBACK MainProc(HWND Dlg,UINT message,WPARAM wParam,LPARAM lParam)
{
   int Select;
   char buf[MAX_PATH];
   OPENFILENAME DlgInfs;

   switch(message)
   {
   case WM_COMMAND:
      Select=LOWORD(wParam);
      switch(Select)
      {
      case IDC_PARCOURIR:
```

La fonction qui affiche la boîte de dialogue 'Parcourir' est GetOpenfileName(). De manière à personnaliser cette boîte de dialogue à nos besoins, nous passons une structure contenant les caractéristiques de la boîte de dialogue. Comme nous n'utiliserons pas une grande partie des membres de cette structure, nous allons tous les mettre à 0 grâce à la fonction memset() du C standard. Ceci constitue une bonne habitude à prendre. Pour plus d'informations sur les paramètres et les possibilités de cette fonction référez vous à l'aide de fournie par Microsoft.

```
         memset(&DlgInfs,0,sizeof(OPENFILENAME));

         DlgInfs.lStructSize=sizeof(OPENFILENAME);
         DlgInfs.hwndOwner=Dlg;
```

```
            DlgInfs.hInstance=hInst;
            DlgInfs.lpstrFilter="Fichiers WAV*.WAV";
            DlgInfs.lpstrFile=buf;
            DlgInfs.nMaxFile=MAX_PATH;
            DlgInfs.lpstrTitle="Choisissez le fichier à lire.";
            DlgInfs.Flags=OFN_PATHMUSTEXIST|OFN_FILEMUSTEXIST;
            if(GetOpenFileName(&DlgInfs))
               SetDlgItemText(Dlg,IDC_NOMDEFICHIER,buf);
            return TRUE;
```

Lorsque l'utilisateur pressera le bouton 'Lire', il faudra récupérer le nom de fichier saisi dans le champ 'IDC_NOMDEFICHIER' et le lire grâce à la fonction sndPlaySound().

```
         case IDC_LIRE:
            GetDlgItemText(Dlg,IDC_NOMDEFICHIER,buf,MAX_PATH);

            sndPlaySound(buf,SND_ASYNC);
            return TRUE;
```

Il ne reste plus qu'à gérer la sortie de la boîte de dialogue. Dans le cas ou l'utilisateur quitte le programme, il faut stopper la lecture du son en cours. Pour cela on utilisera encore la fonction sndPlaySound() avant de quitter.

```
         case IDCANCEL:
            sndPlaySound(NULL,NULL);
            EndDialog(Dlg,0);
            return TRUE;
         }
      default:
         return FALSE;
      }
}
```

Ce programme ne présente aucune difficulté particulière. Il se suffit à lui même bien qu'étant particulièrement pauvre. Ceci permet de donner une idée de la gestion des boîtes de dialogues qui seront dans le futur intégrées à un programme bien plus important. Bien que la gestion de telles boîtes de dialogue constitue un détail dans un programme plus important, il est nécessaire de bien comprendre leur fonctionnement de manière à ne pas perdre de temps et pouvoir créer rapidement une interface graphique agréable.

7. Création d'une application console

Cours théorique :

Avant de s'intéresser aux applications consoles, il convient de bien comprendre de quoi il s'agit. Une application console n'est pas un programme DOS. C'est un programme Windows utilisant les fonctions de l'API Windows mais qui s'affiche dans une console DOS au lieu d'une fenêtre classique. Un tel

programme ne peut pas être démarré en mode DOS réel.

L'avantage de ces programmes est qu'il est inutile de se préoccuper de l'affichage de la fenêtre puisque toutes les interventions sur la fenêtre sont prises en compte par le système. Ce type d'application est très utile pour de petits utilitaires fonctionnant en ligne de commande (bien qu'ils paraissent austères!).

Cependant, ces programmes fonctionnent de la même manière que les applications DOS (pour ce qui est de l'entrée sortie sur l'écran). On pourra donc utiliser les fonctions printf() et scanf() du C standard pour afficher des données à l'écran. De même les fonctions de flux standard C++ cin et cout sont utilisables.

Cette partie ne fera pas l'objet d'un projet, mais les applications console seront utilisées plus tard pour illustrer des programmes simples et ne nécessitant pas ou peu de communications avec l'utilisateur, de manière à réduire la taille des programmes.

8. Fenêtre classique ou ressource ?

Cours théorique :

Maintenant que nous avons créé et utilisé ces deux types de fenêtres, il convient de s'interroger sur leur utilisation. Faut il préférer les ressources ou plutôt les fenêtres classiques ? Quels sont les avantages des une et des autres ?

Les fenêtres créées à partir de ressources (souvent les boîtes de dialogue) sont évidemment bien plus rapides à créer. De plus leur utilisation nécessite beaucoup moins de code. Dans ce cas, pourquoi continuer à utiliser des fenêtres classiques créées avec l'API Windows ?

Si les ressources peuvent séduire, il faut cependant s'en méfier. Elles conviennent très bien pour des boîtes de dialogue invitant l'utilisateur à saisir des valeurs, ou à définir certains paramètres. C'est d'ailleurs leur rôle principal. En les utilisant dans ce contexte on gagne un temps très important par rapport à des fenêtres créées 'manuellement'. Cependant, l'affichage dans une boîte de dialogue ne se prête que peu à la personnalisation.

Pour afficher des données comme des images, des polices personnalisées ou autre, il reste préférable d'utiliser une fenêtre classique. De même si la fenêtre doit gérer des interventions utilisateur comme une touche entrée, un clic, il faut alors passer à une fenêtre classique. Dans ces cas, la gestion automatisée devient en fait un obstacle à l'interception de ces événements.

Avant de se lancer dans la programmation d'une fenêtre classique ou au contraire de fenêtres ressources, il faut donc réfléchir à l'utilisation qui sera faite de ces fenêtres. Ni l'une ni l'autre de ces méthodes ne sont à bannir. Au contraire, il faut connaître les deux et savoir utiliser l'une ou l'autre dès qu'il le faut. La facilité des ressources ne doit pas en faire un réflexe systématique.

Bien entendu on ne peut pas établir des caractéristiques types pour lesquelles il faut absolument utiliser tel ou tel type de fenêtre. Cette décision doit prendre en compte les exigences du programme... Je ne me risquerais pas à dresser une liste exhaustive de tous les cas ou l'utilisation de telle ou telle méthode est préférable mais j'invite seulement le programmeur à réfléchir avant d'utiliser les ressources.

Chapitre 2

Les boîtes de dialogue

1. Fonctionnement général

Cours théorique :

Avant de se lancer dans la programmation d'applications plus complexes, il est nécessaire de bien comprendre le schéma de fonctionnement de la boîte de dialogue. Il se rapproche du fonctionnement d'une fenêtre classique sans pour autant être le même. Il conviendra donc d'éviter les analogies entre ces deux types de fenêtres.

La procédure qui gère une boîte de dialogue peut être excessivement simple. Pour toute notification, Windows appelle la procédure et lui demande de traiter le message. Si elle est en mesure de le traiter, elle doit retourner TRUE. Dans le cas contraire elle retourne FALSE et Windows effectuera le traitement par défaut pour le message. Le minimum est donc d'assurer la fermeture de la boite de dialogue grâce à la fonction EndDialog(). En effet, la fermeture de la boite de dialogue ne fait pas partie du traitement par défaut des messages, ce qui est logique puisqu'il est impossible à Windows de 'deviner' le bouton qui doit quitter la boîte de dialogue.

Dès qu'une boite de dialogue traite un message, elle doit donc retourner TRUE (sauf indication contraire). Remarquez la structure du switch avec le default qui retourne FALSE. Dès qu'un message n'est pas traité, la procédure retourne donc FALSE.

On peut considérer le fonctionnement d'une boîte de dialogue en trois temps. Tout d'abord l'initialisation. C'est à ce moment que les différents contrôles sont initialisés. Par exemple, le statut des 'check boxes' sera défini. Cette phase est généralement indispensable, mais dans certains cas, une boîte de dialogue n'aura besoin d'aucune valeur par défaut. L'initialisation se fait avant que la boite de dialogue ne soit affichée, grâce à un message (WM_INITDIALOG). Si ce message n'est pas traité, c'est à dire si la procédure ne contient pas de 'case WM_INITDIALOG', aucun champ n'aura de valeur particulière.

La deuxième phase de fonctionnement d'une boite de dialogue est celle durant laquelle l'utilisateur effectue les modifications qu'il désire. Il modifie des champs, 'coche' des options... Ce traitement peut se faire de manière entièrement autonome si aucune action particulière n'est nécessaire. Cette phase ne se reflétera donc pas forcément dans la procédure. Dans certains cas cette phase sera inutile, par exemple pour un message d'erreur.

Une fois que l'utilisateur a fait les modifications qu'il désire, il va falloir les 'récupérer'. Le statut de tous les contrôles doit être analysé de manière à modifier les variables au sein même du programme. En effet, les modifications effectuées par l'utilisateur ne se reflètent en rien au niveau des variables du programme. Une fois que le programme à mis à jour toutes les variables nécessaires, la boite de dialogue peut se terminer.

2. Initialisation

Cours théorique :

L'initialisation d'une boîte de dialogue se fait grâce au message WM_INITDIALOG. Si une boîte de dialogue ne désire pas effectuer de traitement

particulier à sa création, elle ignore simplement ce message. Lorsque ce message est envoyé à la boite de dialogue, celle-ci n'est pas encore affiché. C'est à ce moment que les valeurs par défaut des contrôles sont définies. Ces valeurs seront définies grâce à des fonctions de l'API Windows.

De plus, comme la fenêtre n'est pas encore visible, il est possible de modifier sa taille, sa position, sans que cela apparaisse à l'utilisateur (fonction SetWindowPos() par exemple).

La valeur retournée après le traitement du message WM_INITDIALOG détermine si le champ par défaut aura ou non le focus. Le focus est en fait l'entrée clavier. Si un champ possède le focus et qu'une touche est tapée au clavier, c'est à ce champ et à lui seul que cela sera signalé. Dans un champ demandant un mot de passe (par exemple), il est très utile de passer la focus au champ par défaut, de manière à éviter à l'utilisateur de devoir cliquer dans le champ avant de saisir sont mot de passe. Pour que Windows donne le focus au champ par défaut, la procédure doit retourner TRUE. Dans le cas contraire, elle doit retourner FALSE.

3. Le contrôle 'Edit'

Cours théorique :

Ce contrôle à déjà été vu dans les projets précédents. C'est un des plus simples à utiliser. Comme pour tous les contrôles, on peut distinguer deux aspects à l'utilisation de celui-ci. Le premier aspect est son apparence. Elle est modifiée grâce à l'éditeur de ressources du compilateur. Si vous utilisez Visual C++, il vous suffit de double cliquer sur le contrôle pour accéder à ses propriétés. La deuxième partie du fonctionnement de ce contrôle se fait au travers de la procédure qui gère la boîte de dialogue. Elle consiste en la définition ou la récupération de l'état du contrôle (ici le texte qu'il contient).

Avant d'utiliser le contrôle, il faut donc définir son apparence, ainsi qu'un identifiant, permettant au programme de l'identifier. L'identifiant d'un contrôle est généralement de la forme 'IDC_' suivi du nom du contrôle tout en majuscules. Le compilateur se charge de maintenir à jour un fichier définissant toutes les constantes utilisées. Pour Visual C++, ce fichier est 'ressource.h'.

Il est possible de personnaliser de nombreuses caractéristiques pour un contrôle de type 'Edit'. Voici une liste regroupant les styles les plus utilisés. Cette liste n'est pas exhaustive, référez vous à l'annexe A pour plus de précisions. Si vous utilisez un éditeur de ressources, vous devriez pouvoir modifier les styles grâce à une interface graphique. Si éditez vos ressources en mode texte, les styles sont mis entre crochets.

Read Only [ES_READONLY] : Afficher le contrôle sur fond gris. Il est impossible de modifier le texte, mais il peut toutefois être sélectionné et copié.

Disabled : Le contrôle est désactivé. Il apparaît en gris et il est impossible de sélectionner le texte. Cette option ne sera accessible qu'à partir d'un éditeur de ressources car elle ne constitue pas un style. Pour désactiver un contrôle, utilisez la fonction EnableWndow() lors de l'initialisation de la boîte de dialogue.

Number [ES_NUMBER] : Indique que l'on ne peut saisir que des chiffres dans ce contrôle. Le point n'est pas accepté. Pour entrer un nombre à virgule, il faudra donc utiliser un contrôle classique.

Password [ES_PASSWORD] : Affiche des étoiles à la place des caractères saisis. Le texte ne peut pas être copié.

Multiline [ES_MULTILINE] : Indique que le contrôle peut contenir plusieurs lignes. Pour effectuer un retour à la ligne, il faut afficher le caractère '\r\n'. Au clavier, il faut taper CTRL+ENTER.

AutoHScroll [ES_AUTOHSCROLL] - AutoVScroll [ES_AUTOVSCROLL] : Indique si le contrôle doit défiler horizontalement ou verticalement. Pour permettre à un contrôle de défiler verticalement il faut désactiver le défilement horizontal.

Une fois que l'apparence du contrôle est convenable, il faut pouvoir lui assigner une valeur ou récupérer la valeur entrée par l'utilisateur. Ces manipulations se font au travers de 4 fonctions :

SetDlgItemInt() : permet de définir la valeur d'un contrôle grâce à un entier. Il est possible d'indiquer si cet entier est signé ou non. L'identifiant du contrôle est celui définit dans les ressources (IDC_... en général).

GetDlgItemInt() : permet de récupérer la valeur numérique d'un contrôle. Cette fonction échouera si le contrôle contenait du texte.

SetDlgItemText() : permet d'afficher une chaîne de caractères dans le contrôle. Cette chaîne de caractères peut à priori avoir n'importe quelle longueur.

GetDlgItemText() : permet de récupérer la chaîne de caractères contenue dans le contrôle. Il est indispensable de préciser la longueur maximale de la chaîne qui sera récupérée (elle correspond à la taille du buffer dans lequel elle sera stockée).

4. Le contrôle 'Check Box'

Cours théorique :

Le contrôle 'Check Box' fait lui aussi partie de l'un des plus simples. Il peut contenir trois états, coché, non coché ou indéterminé. On peut lui attribuer différents styles qui modifient son apparence et parfois son fonctionnement. Par défaut, ce contrôle ne peut être que dans deux états, coché ou non coché. En modifiant le style on peut permettre à l'utilisateur de spécifier un troisième état, indéterminé, qui correspond à une case cochée mais grisée. Voici une liste des styles principaux pouvant être appliqués à ce contrôle (les styles modifiant l'apparence graphique du contrôle ne sont pas traités ici). Pour une liste exhaustive des différents types consultez l'annexe A.

Tri-state [BS_3STATE] [BS_AUTO3STATE] : le contrôle peut prendre 3 états au lieu de deux.

Auto : [BS_AUTO3STATE] [BS_AUTOCHECKBOX] : permet de créer un contrôle dont l'état est modifié automatiquement lorsque l'utilisateur l'utilise. Si ce style n'est pas activé, la boîte de dialogue reçoit un message WM_COMMAND et doit elle-même modifier l'état du contrôle.

Comme on pouvait s'y attendre, il suffit de deux fonctions pour le manipuler, une pour modifier son état, et l'autre pour récupérer son état.

CheckDlgButton() : permet de modifier l'état du contrôle. Il suffit de spécifier la boite de dialogue qui le contient ainsi que son identifiant et un paramètre commandant l'état du contrôle. Les trois états possibles sont BST_CHECKED, BST_INDETERMINATE et BST_UNCHECKED. Le contrôle sera redessiné automatiquement, c'est à dire que lorsque la fonction retournera, l'état graphique du contrôle aura été modifié.

IsDlgButtonChecked() : permet de récupérer l'état du contrôle. Cette fonction retourne un entier non signé qui peut prendre une des trois valeurs présentées ci-dessus.

L'utilisation de contrôles 'Check box' se résume donc à peu de choses. Elle pourra être mise en parallèle avec les contrôles 'Radio buttons', qui fonctionnement de manière similaire, mais avec un système de groupes.

5. Le contrôle 'Radio Button'

Cours théorique :

Le contrôle 'Radio Button' est assez similaire au contrôle 'Check Box', si ce n'est qu'un seul de ces contrôles peut être coché à la fois. Il se présente généralement sous forme d'un rond à cocher. Bien entendu, vous pouvez utilisez plusieurs de ces contrôles dans un même fenêtre et créer des groupes, de sorte qu'un seul des contrôle par groupe ne puisse être sélectionné, et non pas un seul pour l'ensemble de la fenêtre.

Si vous désirez créer seulement un groupe dans une fenêtre, il est inutile de créer un groupe. En effet, par défaut, un seul contrôle peut être sélectionné dans l'ensemble de la fenêtre.

Si vous utilisez l'éditeur de ressources de Visual, la création de groupes est simplifiée. Pour créer un groupe de radio buttons, il faut en insérer un, et lui mettre le style 'Group'. Puis on insère les autres radios du groupe mais sans mettre le style 'Group'. Pour créer un nouveau groupe, il suffit de refaire la même manipulation autant de fois que nécessaire.

Si vous éditez vos ressources en mode texte, le principe est le même. On met le style **WS_GROUP** au premier des radios puis on insère les autres à la suite.

L'ensemble des styles est décrit dans l'annexe A. La plupart modifient seulement l'apparence graphique du contrôle et ne sont pas très difficiles à utiliser. Retenez simplement un style assez important :

Auto [BS_AUTORADIOBUTTON] : ce contrôle indique que les radios sont gérés automatiquement. Vous n'avez pas à demander de décocher les radios lorsqu'un autre est coché. Généralement, ce style est toujours utilisé, sauf cas très particuliers.

Pour cocher ou décocher les radios, il faut utiliser les fonctions **ChekDlgButton()** et **IsDlgButtonChecked()**. Ces fonctions sont identiques à celles utilisées avec les contrôles 'Check Box'. Bien entendu, le style indéterminé n'est pas disponible ici. Remarquons que lorsque vous utilisez CheckDlgButton(), les autres radios ne sont pas décochés, même en utilisant le style Auto. Enfin, vous pouvez utilisez la fonction **CheckRadioButton()** pour cocher un radio et décochez les autres. Pour cela, les radios doivent avoir étés créés dans l'ordre (les identifiants doivent être en ordre croissant). Il suffit ensuite de passer l'identifiant du premier radio, celui du dernier, ainsi que le radio à sélectionner.

6. Le contrôle 'List Box'

Cours théorique :

Le contrôle 'List Box' est plus complexe que ceux étudiés précédemment. Il ne permet pas seulement de contenir un état entré par l'utilisateur, il permet d'afficher des informations sous forme de liste, de trier la liste par ordre alphabétique... Ce contrôle aura donc de multiples utilisations, d'autant plus qu'il est relativement simple à utiliser. Il ne se manipule pas grâce à des fonctions mais au travers de messages, qui seront envoyés par l'intermédiaire de la fonction SendMessage().

La liste peut être présentée sous deux formes : soit en une seule colonne et avec un défilement vertical, soit sur plusieurs colonnes et avec un

défilement horizontal. Les colonnes auront obligatoirement la même taille. Elles ne peuvent pas comporter d'entêtes.

Voici les styles applicables à ce contrôle. Pour une liste exhaustive, consultez l'annexe A. Remarquez que par défaut, c'est la sélection simple qui est utilisée.

Sort [LBS_SORT] : détermine si le contrôle trie ou non la liste par ordre alphabétique.

Multi-Column [LBS_MULTICOLUMN] : permet d'afficher la liste sur une ou plusieurs colonnes. Si ce style est activé, il faut utiliser un défilement horizontal.

Horizontal Scroll [WS_HSCROLL] : défilement horizontal de la liste.

Vertical Scroll [WS_VSCROLL] : défilement vertical de la liste.

Multiple Selection [LBS_MULTIPLESEL] : autorise la sélection multiple. L'utilisateur peut sélectionner les éléments de la liste en cliquant dessus.

Extended Selection [LBS_EXTENDEDSEL] : autorise la sélection multiple. L'utilisateur sélectionne les éléments de la liste grâce aux touches SHIFT et CTRL.

No Selection [LBS_NOSEL] : interdit toute sorte de sélection.

L'utilisation du contrôle est relativement simple. Nous ne verrons ici que les tâches élémentaires, ajout et suppression d'un élément, récupération de la sélection courante. La méthode de récupération de la sélection courante dépend bien entendu du mode de sélection choisi. Pour une sélection unique, cette action est simple. Elle s'avère un peu plus délicate pour un contrôle à sélections multiples. L'ensemble des messages utilisables avec ce contrôle est présenté dans l'annexe A. Voici les messages les plus simples :

LB_ADDSTRING : ajoute un élément à la liste et demande le tri de la liste si celle-ci à le style 'Sort'.

LB_INSERTSTRING : ajoute un élément à la liste et le place à une position déterminée. Après l'insertion, la liste n'est pas triée.

LB_SETITEMDATA : permet d'associer une valeur 32 bits à un élément de la liste. Cette valeur peut par exemple être un pointeur sur les données que représente cet élément.

LB_DELETESTRING : supprime l'élément désigné.

LB_GETCURSEL : retourne la sélection courante pour un contrôle à sélection unique.

LB_GETSELITEMS : place dans un tableau l'ensemble des éléments sélectionnés dans un contrôle à sélections multiples.

LB_SETCURSEL : modifie la sélection courante dans un contrôle à sélection unique.

LB_GETCOUNT : retourne le nombre total d'éléments contenus par le contrôle.

7. Le contrôle 'Progress bar'

Cours théorique :

Le contrôle 'Progress Bar' est l'un des plus simples à utiliser. Ses styles étant définis au niveau ressources, l'utilisation se borne à la définition de la

position. Pour cela, il faut tout d'abord donner 2 valeurs numériques qui constituent le minimum et le maximum de la barre de progression. Une fois cela fait, il ne reste plus qu'à fournir une position courante.

Avant de pouvoir utiliser ce contrôle, il faut penser à l'initialiser. On doit placer en début de programme un appel à la fonction InitCommonControls() ou à la fonction InitCommonControlsEx(), de manière à forcer le chargement des Dlls permettant d'utiliser ce contrôle. L'appel à l'une de ces deux fonctions peut être fait à n'importe quel moment (mais avant d'utiliser le contrôle) et ne doit être fait qu'une seule fois. Si cet appel n'est pas fait, la boîte de dialogue de s'affichera pas.

Les styles pouvant être utilisés avec ce contrôle sont simples, ils sont donnés dans l'Annexe A.

Voici les messages utilisés pour commander ce contrôle :

PBM_SETRANGE permet de définir les valeurs numériques du minimum et du maximum pouvant être atteints. Les valeurs de cet intervalle, tout comme la position courante sont des entiers, il faut donc prévoir un intervalle assez large pour ne pas que l'on distingue de saccades lors de la progression. Cet intervalle doit donc être au moins de la taille (en pixels) de la barre. De manière générale, on peut opter pour un intervalle 0 - 1000.

PBM_SETPOS définit la position actuelle de progression. Cette valeur doit bien entendu être comprise dans l'intervalle fourni par le message PBM_SETRANGE.

8. Le contrôle 'Combo box'

Cours théorique :

Le contrôle 'Combo Box' est l'un des plus complexes. Il peut servir dans différents 'modes de fonctionnement'. On peut l'utiliser pour permettre à l'utilisateur de faire un choix dans un menu déroulant ou permettre à l'utilisateur d'entrer un choix personnalisé. Nous ne verrons ici que l'utilisation de ce contrôle en tant que menu déroulant permettant à l'utilisateur de faire un choix dans une liste prédéterminée. C'est l'utilisation la plus pratique de ce contrôle car elle permet de s'assurer que le choix fait est valide.

Voici un aperçu des styles applicables à ce contrôle :

CBS_DROPDOWN est le style correspondant au type de 'Combo box' que nous avons décrit. La liste est affichée si l'utilisateur clique sur le contrôle et elle ne peut pas être modifiée.

CBS_SORT permet d'activer le tri automatique du contrôle par ordre alphabétique.

Pour utiliser ce contrôle, vous pouvez utiliser les messages suivants :

CB_ADDSTRING permet d'ajouter une entrée à la liste. La liste sera triée si le style CBS_SORT est utilisé.

CB_SETCURSEL permet de définir la sélection courante.

CB_GETCURSEL permet de retourner la sélection courante.

CB_SETITEMDATA permet d'associer une valeur 32 bits à une entrée. Cette valeur peut être utilisée pour identifier les entrées si la liste est triée par exemple.

CB_GETITEMDATA retourne la valeur 32 bits associée avec l'entrée spécifiée.

Pour déterminer la taille de ce contrôle (notamment la hauteur de la liste), on pourra utiliser la fonction **SetWindowPos()** qui sera vue plus tard.

9. Synthèse sur les contrôles

Cours théorique :

Les contrôles qui ont été présentés précédemment l'ont été de manière brève. Ce tutorial ne prétend absolument pas fournir une référence exhaustive de toutes les possibilités d'utilisation des contrôles. Cette présentation des contrôles les plus courants permettra simplement de se familiariser avec leur utilisation. En effet, quel que soit le type de contrôle utilisé, la méthode pour le manipuler reste toujours la même.

Une fois le principe de fonctionnement des contrôles compris, il sera facile de l'étendre à une utilisation plus complexe. Les contrôles prédéfinis offrent une excellente manière de guider l'utilisateur et de le limiter. De plus, leur utilisation demande relativement peu de code et permet de créer rapidement une interface agréable. Les contrôles offrent de nombreuses fonctionnalités qu'il faut savoir exploiter.

On remarquera cependant que ces contrôles sont assez difficilement personnalisables. Pour créer une interface graphique plus poussée ou plus personnalisée, il faudra donc recourir à d'autres méthodes que des contrôles. Il est tout de même possible de personnaliser les contrôles (couleurs, etc...) mais ces méthodes ne permettant une personnalisation complète de l'interface.

Il faut tout de même rappeler que les ressources ne sont pas le seul moyen de mettre en place les contrôles, ils constituent simplement un 'raccourci' pour leur utilisation. Tous les contrôles peuvent être intégrés de manière dynamique dans une fenêtre ou un boîte de dialogue en utilisant la fonction **CreateWindowEx()**. Dans le cas d'une interface dynamique, c'est à dire ne se présentant pas systématiquement sous la même forme, l'utilisation des ressources est fortement déconseillée car elle 'fixe' les contrôles au moment de la compilation. La création des contrôles au moment de l'affichage permet donc une plus grande souplesse, bien qu'exigeant une mise en place plus contraignante.

10. Deux projets récapitulatifs

Cours théorique :

Voici 2 projets qui présentent des utilisations extrêmement simples de quelques contrôles élémentaires. Seul le projet 06a sera décrit ici, le second étant très similaire.

Projet N°6a :

Ces projets présentent une utilisation extrêmement simple de quelques contrôles élémentaires. Aucune vérification n'est faite quant à la validité des entrées faites par l'utilisateur. Remarquons que du fait des contrôles utilisés, l'utilisateur ne peut pas fournir de sélection non valide (à part entrer un nom

vide). C'est d'ailleurs pour ça qu'il est indispensable d'initialiser les contrôles 'Radio' ou 'Combo box' car sans cela, on pourrait avoir une sélection non valide (aucune sélection en fait). Les contrôles sont bien entendus initialisés lors du message **WM_INITDIALOG** (à ce moment, la boîte de dialogue n'est pas encore affichée).

La liste n'est modifiée qu'au moment où l'utilisateur presse le bouton 'Ajouter'. A ce moment, on récupère le statut de tous les contrôles que l'on a placés et on vérifie éventuellement la validité des données entrées. Si tout est correct, on ajoute les données voulues à la liste. Ici, on aurait quelques peines à récupérer les informations contenues dans la liste au moment de la validation de la boîte de dialogue. Pour faciliter cela, il faudrait stocker chaque ligne dans une structure appropriée. On associerait ensuite à chaque entrée de la liste son index dans notre tableau grâce au message **LB_SETITEMDATA**.

Télécharger les projets commentés [VC++] : Projet 06a - Projet 06b.

Télécharger les projets commentés [BCB] : Projet 06a [BCB] - Projet 06b [BCB].

11. MessageBox()

Cours théorique :

La fonction **MessageBox()** permet d'afficher une boite de dialogue prédéfinie contenant le texte spécifié. Elle est très utile pour afficher des messages d'erreurs, informer l'utilisateur. Elle retourne l'identifiant du bouton qui a été pressé pour la quitter. Cette fonction ne permet quasiment aucune personnalisation de la boîte de dialogue qu'elle affiche, elle n'est donc réellement utile que dans le cas de messages envoyés à l'utilisateur, souvent de manière informative avec pour seul choix le bouton 'Ok'.

Les types de boites de dialogue pouvant être affichés sont :

MB_ABORTRETRYIGNORE

MB_OK

MB_OKCANCEL

MB_RETRYCANCEL

MB_YESNO

MB_YESNOCANCEL

Les icônes suivantes peuvent être affichées sur la boîte de dialogue :

MB_ICONEXCLAMATION

MB_ICONWARNING

MB_ICONINFORMATION

MB_ICONASTERISK

MB_ICONQUESTION

MB_ICONSTOP

MB_ICONERROR

MB_ICONHAND

La fonction retournera l'une des valeurs suivantes :

IDABORT

IDCANCEL

IDIGNORE

IDNO

IDOK

IDRETRY

IDYES

12. Parcourir l'arborescence

Cours théorique :

Pour permettre à l'utilisateur de sélectionner un dossier ou un fichier, on utilisera respectivement les fonctions **SHBrowseForFolder()** et **GetOpenFileName()**. Ces deux fonctions affichent les boîtes de dialogue standard de Windows pour parcourir l'arborescence à la recherche d'un dossier ou d'un fichier. Notons que la constante **MAX_PATH** désigne la longueur standard pour un chemin. Les buffers chargés de recevoir les noms de fichiers ou les chemins sont supposés faire au moins cette taille.

Voici un exemple d'utilisation de la fonction **SHBrowseForFolder()** :

```
BROWSEINFO bi;
LPITEMIDLIST Item;
// Ici, la taille du buffer ne peut pas être passée
// buffer est supposé être de taille MAX_PATH (ou plus)
char buffer[MAX_PATH];

// On met tous les champs inutilisés à 0
memset(&bi,0,sizeof(BROWSEINFO));
// hDlg est le HWND de la boite de dialogue qui demande l'ouverture
// Ou NULL si la boite de dialogue n'a pas de fenêtre parent
bi.hwndOwner=Dlg;
// Contient le répertoire initial ou NULL
bi.pidlRoot=NULL;
bi.pszDisplayName=buffer;
```

```
bi.lpszTitle="Répertoire courant";
bi.ulFlags=NULL;
bi.lParam=NULL;
Item=SHBrowseForFolder(&bi);
if(Item!=NULL)
{
    // buffer contient le nom du répertoire sélectionné
    SHGetPathFromIDList(Item,buffer);
    // buffer contient le chemin complet de la sélection
}
```

Voici un exemple d'utilisation de la fonction **GetOpenFileName()** :

```
OPENFILENAME st;
char buffer[MAX_PATH];

// Pas de fichier par défaut
buf[0]='';
// On met tous les champs inutilisés à 0
memset(&st,0,sizeof(OPENFILENAME));
st.lStructSize=sizeof(OPENFILENAME);
// hDlg est le HWND de la boite de dialogue qui demande l'ouverture
// Ou NULL si la boite de dialogue n'a pas de fenêtre parent
st.hwndOwner=hDlg;
// La syntaxe est : Description1Filtre1Description2Filtre2
st.lpstrFilter="Executables - Fichiers de commandes*.exe;*.bat";
st.lpstrFile=buffer;
st.nMaxFile=MAX_PATH;
st.lpstrTitle="Recherche de l'exécutable";
st.Flags=NULL;
// Contient le répertoire initial ou NULL
st.lpstrInitialDir=NULL;
if(GetOpenFileName(&st))
    // buffer contient notre chemin
```

Chapitre 3

Les fenêtres

1. Fenêtres et boîtes de dialogue

Cours théorique :

Les boîtes de dialogue sont semblables aux fenêtres dans le sens où ce sont des fenêtres. Cependant, il faut être bien attentif à marquer la différence qu'il existe entre ces deux types de fenêtres. Les boîtes de dialogue sont gérées en grande partie par le système. Créer une boite de dialogue est relativement simple. Les couleurs de fond, le type de fenêtre ont des valeurs par défaut si l'on utilise un éditeur de ressources.

Dans le cas des fenêtres, il va falloir tout définir lors de l'exécution. Lors de la création d'une fenêtre, il sera nécessaire de préciser un grand nombre de paramètres qui étaient passés implicitement au système grâce aux ressources avec les boîtes de dialogue. Il faut donc bien considérer les boîtes de dialogue comme un cas 'simplifié' de fenêtres.

Cependant, on ne peut pas appliquer toutes les méthodes fonctionnant sur les fenêtres aux boîtes de dialogues ou inversement. Le mode de gestion des boîtes de dialogue est différent de celui des fenêtres, particulièrement au niveau de l'affichage. De manière générale, il faut utiliser les boîtes de dialogue avec des contrôles prédéfinis. Pour créer des fenêtres contenant un affichage personnalisé, des images, des graphiques, du texte utilisant différentes polices, il est préférable d'utiliser une fenêtre. C'est donc pour ce type de travaux que les boîtes de dialogues trouvent leurs limites.

Il faut donc retenir que les fenêtres et les boîtes de dialogue ne fonctionnement pas de la même manière (malgré de nombreuses similitudes). Copier - Coller la procédure d'une fenêtre pour une boîte de dialogue ne serait pas une bonne idée car certains messages sont spécifiques aux fenêtres et d'autres aux boîtes de dialogue.

2. Fonctionnement général d'une fenêtre

Cours théorique :

Le fonctionnement général des fenêtres a déjà été vu au cours du premier chapitre. Un petit rappel ainsi que quelques précisions est cependant nécessaire.

Lorsqu'une fenêtre est créée, le système lui attribue un identifiant. Cet identifiant est appelé 'handle'. Il identifié la fenêtre dans son ensemble. Toute modification de la fenêtre (déplacement, modification de la taille) se fait grâce à cet identifiant. Une application utilisant plusieurs fenêtres pourra donc être amenée à définir des variables globales contenant les identifiants des différentes fenêtres pour plus de confort. Cette définition globale n'est absolument pas nécessaire et peut être évitée dans le cas d'applications simples (le passage en paramètre des identifiants devient rapidement impossible).

La fenêtre d'une application ne correspond absolument pas à l'application elle même. Une application (ou processus) peut être exécutée et ne pas avoir de fenêtre. La création de la fenêtre est effectuée par le programme lui même. De plus, elle est totalement facultative. Aucune fenêtre ne sera créée par le système au moment de l'exécution de l'application. De plus, la fermeture de la fenêtre n'entraîne pas la fermeture de l'application. C'est au programmeur de faire concorder ces deux événements si il le désire. Bien entendu la majorité des applications créent une fenêtre au démarrage et se terminent lors de la fermeture de cette même fenêtre, mais ce n'est en rien une obligation. De plus, une fenêtre peut avoir plusieurs états, visible, cachée, minimisée... Lorsqu'une fenêtre est créée, elle n'est pas visible. Elle existe, ont peut la déplacer, afficher dans sa zone client... Cependant, rien de tout ceci n'est visible à l'utilisateur. Dès qu'on donne à une fenêtre le statut 'visible', elle apparaît à la même position que lorsqu'elle était invisible. Lorsqu'on détruit une fenêtre, elle n'est plus affichée à l'écran, mais dans ce cas, on ne peut plus la modifier. Elle n'existe plus pour le système. Il ne faut donc pas confondre une fenêtre cachée, qui existe toujours bien que n'étant pas visible, et une fenêtre détruite, qui elle n'existe plus.

Une fenêtre contient différentes zones. La zone client est la zone (généralement blanche) dans laquelle les données de la fenêtre sont affichées. Les menus, la barre de titre, ne font pas partie de la zone client. La zone client est la seule partie de la fenêtre qui soit gérée par l'application. La taille de la fenêtre sur l'écran est donc différente de celle de la zone client. La zone client est nécessairement de taille égale ou inférieure à l'espace occupé par la fenêtre. Pour afficher des données dans une fenêtre, on utilise des coordonnées relatives à la zone client. Pour déplacer une fenêtre, on utilise des coordonnées relatives à l'écran (coin supérieur gauche).

Pour identifier la zone client, on utilise un contexte d'affichage. Il définit l'ensemble des propriétés relatives à l'affichage dans la zone client (nombre de couleurs disponibles, type de police...). Un contexte d'affichage identifie une zone dans laquelle ont peut afficher n'importe quel type de données. Cette zone n'est pas nécessairement affichée à l'écran, on peut par exemple afficher des données dans une fenêtre cachée en utilisant le même contexte d'affichage. Dans ce cas, le contexte d'affichage est lié à la fenêtre, on peut donc l'obtenir à partir de l'identifiant de la fenêtre concernée. Pour afficher dans la zone client, il n'est donc jamais nécessaire de se préoccuper de la position de la fenêtre sur l'écran, il suffit de posséder un 'handle' sur le contexte d'affichage.

3. Récupération des messages

Cours théorique :

La réception des messages et leur transmission à la procédure de fenêtre appropriée sont essentiels. L'application ne doit jamais cesser 'd'écouter' au cas où de nouveaux messages arriveraient. Si l'application arrête de transmettre les messages, la fenêtre semblera 'bloquée' (affichage persistant, impossibilité de la déplacer...).

La récupération des messages peut se faire grâce à deux fonctions : **GetMessage()** ou **PeekMessage()**. La différence entre ces deux fonctions est fondamentale, il ne faut donc pas les assimiler.

GetMessage() est une fonction bloquante. Elle attend qu'au moins un message soit présent dans la file d'attente de la fenêtre pour retourner. Le temps d'attente est infini, tant qu'aucun message n'est placé dans la file d'attente.

PeekMessage() est une fonction non bloquante. Elle vérifie si des messages sont présents et retourne immédiatement. Si au moins un message était présent, elle le retourne, sinon elle ne retourne aucun message. De plus, cette fonction permet de ne pas supprimer les messages de la file d'attente. On peut donc consulter des messages et laisser la boucle de messages principale effectuer le traitement.

L'utilisation de ces deux fonctions s'inscrit dans des cadres totalement différents. En général, **GetMessage()** est utilisé dans une boucle 'while'. Si aucun message n'est présent, **GetMessage()** signale au système qu'il peut donner le processeur à une autre application. Placer cette fonction dans une boucle ne génère donc pas d'occupation processeur tant qu'aucun message n'est présent dans la file d'attente. **PeekMessage()** ne fonctionne pas de cette manière. Placer cette fonction dans une boucle 'while' génère une occupation processeur de 100%. La fonction sera effectivement appelée un nombre indéfini de fois, occupant ainsi le processeur. La récupération principale des messages doit donc se faire avec la fonction **GetMessage()**.

Cependant, si l'application est occupée, elle peut appeler de manière régulière **PeekMessage()** suffisamment de fois pour traiter tous les messages. Par exemple, toute les secondes, **PeekMessage()** est appelée jusqu'à épuisement de la file d'attente. Le traitement des messages sera plus lent mais il

sera effectué quand même. Cette solution évite de devoir faire appel au multi-threading (exécution simultanée de plusieurs parties du programme).

La différence entre ces deux fonctions doit donc être clairement comprise. L'utilisation de l'une ou l'autre de ces fonctions à tort peut s'avérer catastrophique pour les performances du programme et du système dans son ensemble. Dans le cas d'une application n'effectuant pas de tâches lourdes, l'utilisation de **PeekMessage()** ne sera pas nécessaire. L'interruption du traitement des messages pour une durée courte (jusqu'à un dixième de secondes) est sans conséquences. Il faut tout de même remarquer que **GetMessage()** ne sera pas rappelée tant que le traitement du message courant n'est pas terminé. Le traitement des messages par la procédure doit donc si possible être bref.

Voici un exemple classique de boucle réalisant le traitement de messages :

```
MSG msg;
while(GetMessage(&msg,NULL,0,0))
{
  TranslateMessage(&msg);
  DispatchMessage(&msg);
}
```

La fonction **GetMessage()** retourne **FALSE** si le message **WM_QUIT** est reçu. Ce message est envoyé à la fenêtre principale pour demander l'arrêt de l'application. Par défaut, ce message n'est pas envoyé lors de la destruction de la fenêtre.

4. Création d'une fenêtre

Cours théorique :

Avant de pouvoir créer une fenêtre, il faut définir un type. C'est ce type de fenêtre qui sera utilisé pour afficher la fenêtre nouvellement créée. Il est possible d'utiliser des types prédéfinis ou de créer son propre type de fenêtres. Les contrôles utilisés par les boites de dialogues sont des types de fenêtres prédéfinis. Pour définir un nouveau type de fenêtres, on utilise la fonction **RegisterClassEx()** (cf. chapitre 1).

Une fois le type de fenêtre définit, il faut créer la fenêtre. Pour cela, on utilise la fonction **CreateWindowEx()** (cf. chapitre 1). On lui passe le type (la classe) de fenêtre désiré ou encore une classe prédéfinie. Lorsque la fenêtre est créée, un message **WM_CREATE** sera envoyé. C'est à ce moment que l'application doit faire les initialisations qu'elle désire.

La fenêtre créée est initialement cachée. Il faut donc faire un appel à **ShowWindow()** pour l'afficher. Il est possible de passer à **ShowWindow()** le paramètre **nCmdShow** que Windows passe à **WinMain()**, de cette manière l'état de base de la fenêtre peut être paramétré dans un raccourci.

5. Destruction d'une fenêtre

Cours théorique :

La destruction d'une fenêtre se traduit par l'envoi à la fenêtre concernée d'un message **WM_DESTROY**. Il ne faut pas confondre ce message avec le

message **WM_QUIT**. **WM_QUIT** demande la fermeture totale de l'application, le message **WM_DESTROY** demande simplement la destruction de la fenêtre en question. Le fait de cliquer sur la 'croix' provoque l'envoi d'un message **WM_DESTROY** à la fenêtre. Pour faire quitter l'application en même temps que la fermeture de la fenêtre, on peut utiliser la fonction **PostQuitMessage()** qui elle, envoie un message **WM_QUIT**. Cette fonction doit être appelée dès la réception du message **WM_DESTROY**. Le 'handle' de la fenêtre devient invalide dès la destruction de la fenêtre. Pour tester la validité d'un 'handle', on peut utiliser la fonction **IsWindow()**.

Lorsque ce message est reçu, la fenêtre est déjà en cours de destruction. Il est possible de traiter le message **WM_CLOSE** pour demander une autre action que la destruction de la fenêtre. On peut par exemple, masquer la fenêtre. Une pression sur la 'croix' provoquera donc simplement la disparition à l'écran de la fenêtre, mais pas au niveau système. La fenêtre pourra par la suite être réaffichée dans le même état, par un appel à la fonction **ShowWindow()**.

6. Contexte d'affichage

Cours théorique :

Un contexte d'affichage (DC) identifie une zone dans laquelle l'application peut afficher tout type de données. La zone identifiée peut être située à n'importe quel endroit de l'écran, elle peut être déplacée, ou même masquée par d'autres fenêtres, sans que l'application en ai conscience. Le fait de déplacer une fenêtre ne change donc rien à la manière dont l'application redessine son contenu. Il en est de même pour une fenêtre cachée ou n'étant pas au premier plan. Toute fenêtre possède un contexte d'affichage associé à sa zone client. Pour obtenir ce contexte, on peut utiliser la fonction **GetDC()** en spécifiant la fenêtre concernée. Si on spécifie un paramètre **NULL**, la fonction retourne un contexte d'affichage sur la totalité de l'écran. La zone identifiée couvre l'ensemble de l'écran et est située au dessus de toutes les fenêtres affichées. Sauf cas très particulier, on n'utilise jamais le contexte d'affichage de la zone écran pour dessiner. Il peut être utilisé lors de la création de bitmaps, etc...

Le contexte d'affichage contient toutes les informations relatives à l'affichage (police, nombre de couleurs)... Il n'identifie pas nécessairement une zone écran. Il peut par exemple identifier une feuille dans le cas d'une impression. On pourra donc utiliser les mêmes fonctions pour afficher à l'écran ou pour envoyer des données imprimables mais en modifiant le contexte d'affichage utilisé.

Lorsqu'on modifie une propriété du contexte d'affichage (ex. : la police courante), on dit qu'on sélectionne un objet dans le contexte. Cet sélection est effectuée grâce à la fonction **SelectObject()**. Différents types d'objets peuvent être sélectionnés dans un contexte d'affichage (polices, images, pinceaux de dessin...).

7. Gestion de base d'une fenêtre

Cours théorique :

La gestion d'une fenêtre se fait à deux niveaux : au niveau de la procédure qui gère tous les messages reçus par la fenêtre et par des fonctions permettant diverses manipulation sur la fenêtre.

Voici un rapide aperçu des fonctions les plus utiles pour manipuler les fenêtres :

SetWindowText() modifie le texte de la barre de titre.

GetWindowText() retourne le texte de la barre de titre.

SetWindowPos() modifie la taille de la fenêtre, sa position sur l'écran, ainsi que sa position par rapport aux autres fenêtres (Z-order).

GetWindowPlacement() retourne des informations sur le statut courant de la fenêtre spécifiée.

GetClientRect() retourne la taille de la zone client de la fenêtre.

GetWindowRect() retourne la taille totale occupée par la fenêtre sur l'écran.

AdjustWindowRect modifie la taille de la fenêtre spécifiée.

GetParent() retourne la fenêtre parent de la fenêtre spécifiée.

SetParent() modifie la fenêtre parent de la fenêtre spécifiée.

ShowWindow() modifie l'état courant d'une fenêtre.

IsIconic(), **IsZoomed()**, **IsWindowVisible()** retournent des informations sur l'état de la fenêtre courante.

GetWindow() retourne la fenêtre ayant une relation spécifiée dans le Z-order par rapport à la fenêtre passée en paramètre.

La procédure de fenêtre, quant à elle, gère l'ensemble des messages envoyés à une fenêtre. Comme les messages envoyés à une fenêtre sont extrêmement nombreux, il serait très pénible de devoir tous les traiter, en particulier pour des fenêtres simples. L'API Windows fournit la fonction **DefWindowProc()** qui propose un traitement par défaut de tous les messages envoyés à une fenêtre. Pour un traitement personnalisé du message, il ne faut pas appeler la fonction **DefWindowProc()**. Le traitement par défaut de la plupart des messages est en général satisfaisant. Cependant, pour imposer des comportements personnalisés à une fenêtre, il s'avère obligatoire d'effectuer un traitement personnalisé de certains messages (ex. : pour masquer la fenêtre lors d'une pression sur la 'croix').

Voici un aperçu rapide des messages les plus courants :

WM_CREATE est envoyé à une fenêtre au moment de sa création.

WM_PAINT est envoyé lorsqu'une partie de la zone client doit être redessinée.

WM_ERASEBKGND est utilisé pour demander l'effacement d'une partie de la zone client.

WM_SYSCOMMAND est envoyé pour le traitement de différents évènements pouvant survenir (fenêtre minimisée, restaurée, ...).

WM_ACTIVATE est envoyé lorsqu'une fenêtre est activée ou désactivée.

WM_MOVE est envoyé lorsque la fenêtre a été déplacée.

WM_SIZE est envoyé lorsque la taille de la fenêtre à été modifiée.

WM_CLOSE indique que l'utilisateur demande la fermeture de l'application (en cliquant sur la 'croix' ou en pressant ALT+F4).

WM_DESTROY indique que la fenêtre est détruite.

Ces messages sont les premiers à utiliser pour le maniement d'une fenêtre. De nombreux autres seront utilisés pour gérer les interactions avec

l'utilisateur (souris, clavier). Ils seront vus plus tard. Dans la plupart des cas, seul un nombre relativement peu élevé de messages sera traité par l'application elle même. Les autres seront passés à la fonction **DefWindowProc()**.

8. Interactions avec l'utilisateur

Cours théorique :

Une fenêtre, peut recevoir des données de la part de l'utilisateur de deux manières : par le clavier, ou par la souris. Cependant, la fenêtre ne recevra d'information du système que si elle est active. Si la fenêtre n'est pas active elle ne sera informée ni des événements clavier, ni des événements souris. La fenêtre sera informée de différents événements : pression d'une touche, relâchement d'une touche, mouvement de la souris, pression d'un bouton de la souris… Une fenêtre est informée dès que la souris bouge au dessus de sa zone client (si elle est active). Le nombre de messages envoyés lorsque la souris se déplace est assez important. Aussi, le traitement de ces messages (**WM_MOUSEMOV**) doit être bref.

Les messages suivants sont utilisés pour traiter les événements provenant du clavier :

WM_KEYDOWN est envoyé lorsqu'une touche est enfoncée. Ce message indique le 'Virtual Key Code' correspondant à la touche.

WM_KEYUP est envoyé lorsqu'une touche est relevée. Ce message indique le 'Virtual Key Code' correspondant à la touche.

WM_CHAR indique le caractère ASCII correspondant à la touche pressée.

WM_SYSKEYUP, **WM_SYSKEYDOWN**, **WM_SYSCHAR** sont utilisés pour identifier les événements dus à des touches systèmes.

Remarquons tout de même la différence entre un code ASCII et un 'Virtual Key Code' : le code ASCII identifie un caractère (a,A,E,* ont des codes ASCII différents). Un 'Virtual Key Code' identifie une touche physique, a et A ont donc le même code, mais dans le 2e cas, SHIFT est aussi pressé (en ASCII SHIFT n'a pas de code, ce qui est normal car cette touche ne correspond pas à un caractère).

Les messages suivants sont utilisés pour traiter les événements provenant de la souris :

WM_MOUSEMOVE est envoyé à chaque déplacement de la souris (plusieurs dizaines de fois par seconde). Les coordonnées de la souris sont passées en paramètres lors de l'envoi de ce message. Si le message est reçu avec un retard, la position reçue sera celle enregistrée lorsque le message a été envoyé.

WM_LBUTTONDOWN est envoyé lorsque le bouton gauche de la souris est pressé.

WM_LBUTTONUP est envoyé lorsque le bouton gauche de la souris est relevé.

WM_MBUTTONDBLCLK est envoyé lors d'un double clic du bouton gauche de la souris. Ce message n'est envoyé que si le style **CS_DBLCLKS** est utilisé lors de la création du style de la fenêtre.

Les mêmes messages sont envoyés pour les boutons centraux et droits de la souris. (**WM_MBUTTON_** ou **WM_RBUTTON_**).

Pour obtenir la position courante de la souris, une application peut utiliser la fonction **GetCursorPos()**. Cette position peut ensuite être convertie grâce aux fonctions **ScreenToClient()** et **ClientToScreen()**. Les fonctions **GetKeyState()** et **GetKeyboardState()** peuvent être utilisées pour connaître l'état d'une touche ou de l'ensemble du clavier.

9. Les timers

Cours théorique :

Parfois, il peut être utile d'exécuter des tâches à intervalles fixes. Il existe de nombreux moyens d'effectuer cela, mais une des méthodes les plus simples est d'utiliser les 'timers' fournis par l'API Windows. Les 'timers' ne conviennent qu'à des résolutions de temps faibles, en général pas plus de 10 Hz. Un message est envoyé à intervalles réguliers à une fenêtre spécifiée. L'utilisation des timers pour des fréquences trop importantes peut nuire aux performances générales du système du fait des nombreux messages à envoyer. De plus, le temps de traitement des messages fait qu'il est impossible d'obtenir des intervalles réguliers inférieurs à 50ms. Les timers ne constituent donc qu'une méthode simple pour synchroniser un programme dans des conditions assez restreintes. Cependant, ils conviennent pour la plupart des applications bureautiques.

Les timers sont généralement utilisés avec des fenêtres, cependant, ils peuvent être utilisés sans fenêtres. Dans ce cas, le message sera envoyé au thread qui a demandé la création du timer. La fonction **GetMessage()** doit donc être appelée avec un argument **hWnd** égal à **NULL** de manière à na pas associer la réception des messages à une fenêtre spécifique.

Pour créer un timer, on utilise la fonction **SetTimer()** en précisant l'intervalle de temps désiré. Un message **WM_TIMER** sera envoyé à la fenêtre à chaque intervalle de temps. Pour stopper un timer, on utilise la fonction **KillTimer()**. Comme les timers ont des identifiants, il est possible de créer différents timers avec des intervalles différents. L'identifiant du timer est passé en paramètre lors de la réception du message **WM_TIMER**.

Voici un exemple utilisant un timer sans fenêtre :

```
MSG msg;
SetTimer(NULL,NULL,60000,NULL);

while(GetMessage(&msg,NULL,0,0))
{
  if(msg.message==WM_TIMER)
    MessageBox(NULL,"Déjà une minute de passé!","Info",MB_OK);
}
```

10. Un projet récapitulatif

Cours théorique :

Le projet présenté ici est simple. Il reprend la même création de fenêtre qu'au chapitre 1.2. La différence réside dans le traitement des messages. Tout d'abord, un timer est utilisé pour permettre l'affichage de la fenêtre après deux secondes seulement. La fenêtre est initialement cachée. Le timer est créé lors de la réception du message **WM_CREATE**. Ce n'est pas une nécessité. Il aurait aussi bien pu être créé après l'appel à **CreateWindowEx()** mais le fait de créer le timer dans la procédure permet de regrouper l'ensemble des traitements relatifs à la fenêtre dans sa procédure.

Des boîtes de dialogue sont affichées lorsque divers événements sont détectés, pression d'un bouton (gauche ou droit), pression d'une touche. Les

clics hors de la zone client ne sont pas pris en charge. Comme la position récupérée par **GetCursorPos()** est relative à l'écran, il faut la convertir en coordonnées relatives à la zone client. La fonction **ScreenToClient()** fournie pas l'API Windows est donc la plus appropriée.

La destruction de la fenêtre est suivie de la fermeture de l'application grâce un appel de **PostQuitMessage()**.

Télécharger le projet commenté : Projet 07.

11. Affichage de texte dans une fenêtre

Cours théorique :

Pour afficher du texte dans la zone client d'une fenêtre, il faut posséder un 'handle' sur le contexte d'affichage de cette fenêtre. Le texte qui sera affiché le sera avec la police courante ainsi que les couleurs de premier plan et de fond courantes. Pour afficher du texte, il existe diverses méthodes. La première est d'utiliser la fonction **TextOut()**. Cette fonction affiche le texte sur une seule ligne, sans jamais calculer de retour à la ligne. Si le texte dépasse la taille de la fenêtre, il sera tronqué. La fonction **DrawText()** permet quant à elle de définir un rectangle dans lequel le texte doit être écrit. Si le texte dépasse la largeur, il sera mis à la ligne, et ceci autant de fois qu'il faudra.

La position passée pour dessiner le texte correspond par défaut au coin supérieur gauche de la chaîne de texte qui sera dessinée. Le texte sera donc dessiné vers la droite et vers le bas par rapport au point donné. On peut modifier le point de référence pour le positionnement du texte avec la fonction **SetTextAlign()**. Pour aligner du texte à droite, on pourra par exemple placer ce point dans le coin supérieur droit. Le texte sera donc dessiné à gauche et en bas de la position passée pour dessiner le texte.

Les fonctions **SetTextColor()** et **SetBkColor()** permettent de définir les couleurs courantes de premier plan et d'arrière plan pour l'affichage du texte. La fonction **SetBkMode()** modifie les propriétés du fond (transparent ou opaque). Une fois ces fonctions appelées, le texte dessiné le sera systématiquement avec ces propriétés. Pour dessiner du texte en blanc sur fond noir, il suffira donc d'un seul appel à chacune de ces fonctions. Il faudra faire un nouvel appel pour repasser en texte noir sur fond blanc.

Pour connaître la taille d'un texte à l'écran (en pixels), il faut utiliser la fonction **GetTextExtentPoint32()**. Cette fonction retourne la taille qu'occupera un texte donné dans un contexte d'affichage donnée. En effet, la taille du texte dépend du contexte d'affichage car la modification de la police entraîne une modification de la taille occupée par le texte. L'utilisation de polices personnalisées sera vue dans le chapitre suivant.

La fonction suivante dessine un texte en rouge centré dans la zone client de la fenêtre spécifiée.

```
void DrawCenteredText(HWND hWnd,char *text)
{
  HDC hDC;
  RECT rcClient;
  GetClientRect(hWnd,&rcClient);
  hDC=GetDC(hWnd);
  SetTextColor(hDC,0x000000FF);
  SetBkMode(hDC,TRANSPARENT);
  SetTextAlign(hDC,TA_CENTER|TA_TOP);
  TextOut(hDC,(int)((float)rcClient.right/2),5,text,strlen(text));
```

```
  ReleaseDC(hWnd,hDC);
}
```

12. Utilisation de polices personnalisées

Cours théorique :

Pour utiliser une police personnalisée pour l'affichage du texte, il faut tout d'abord créer la police spécifiée. Pour cela, on réalise un appel à **CreateFont()** pour déterminer la police voulue, la taille, le style... Il faut ensuite sélectionner la police dans le contexte d'affichage avec la fonction **SelectObject()**. Une fois qu'une police est sélectionnée dans un contexte d'affichage, chaque texte affiché le sera avec cette police.

SelectObject() retourne le 'handle' de la police précédemment sélectionnée. Une fois l'utilisation de la police terminée, l'application doit resélectionner l'ancienne police par un nouvel appel à **SelectObject()**. Ensuite, elle doit détruire la police inutilisée par un appel à **DeleteObject()**.

La suppression des objets créés par une application est faite automatiquement lorsque celle-ci se termine. Cependant, ils occupent de la place inutilement durant l'exécution de l'application. Il est donc important de supprimer les objets inutilisés. De plus, le nombre d'objets disponible pour une application donnée est limité, si l'application recrée une nouvelle police à chaque modification de l'affichage sans supprimer l'ancienne, elle va rapidement atteindre le nombre maximal d'objets.

Voici un exemple d'appel à **CreateFont()** :

```
HFONT Police;
Police=CreateFont(
  20,
  0,
  0,
  0,
  700,
  FALSE,
  FALSE,
  FALSE,
  0,
  OUT_DEFAULT_PRECIS,
  CLIP_DEFAULT_PRECIS,
  DEFAULT_QUALITY,
  DEFAULT_PITCH | FF_DONTCARE,
  "Comic Sans MS"
  );
```

13. Affichage d'une image

Cours théorique :

Avant de pouvoir afficher une image, il faut tout d'abord la charger. Nous ne traiterons ici que les images bitmap. En effet, ce sont les seules images prises en charge par l'API Windows. Pour effectuer des traitements sur les images avec des fonctions du GDI (Graphic Device Interface), il faut donc tout d'abord les convertir au format bitmap. Les images sont considérées comme des objets du GDI, au même titre que les polices. On peut donc utiliser les fonctions de manipulation d'objet (**ex. : SelectObject()** ou **DeleteObject()**) aussi bien sur des images que sur des polices.

On peut charger une image à partir du disque ou encore à partir des ressources du programme. Pour cela, on peut utiliser deux fonctions : **LoadBitmap()** ou **LoadImage()**. La fonction **LoadBitmap()** est beaucoup moins complète que **LoadImage()**, il est donc préférable d'utiliser cette dernière. Ce n'est cependant pas une obligation. Une fois l'image chargée en mémoire, le GDI retourne un 'handle' (de type **HBITMAP**) sur l'image. Elle peut alors être affichée.

Le résultat de l'affichage dépend bien entendu du contexte d'affichage utilisé (en particulier du nombre de couleurs qu'il comprend). Pour afficher l'image, on utilisera la fonction **DrawState()**. Cette fonction n'affiche pas nécessairement des bitmaps, elle peut aussi afficher des icônes par exemple. De plus, cette fonction est capable de modifier la taille d'affichage du bitmap. Il n'est cependant pas recommandé d'agrandir la taille des images à l'affichage car l'agrandissement est alors de très mauvaise qualité.

Voici un exemple de fonction dessinant une image chargée depuis un fichier dans la fenêtre spécifiée :

```
void PrintBmp(HWND hWnd, char *filename)
{
  HBITMAP hBmp;
  HDC hDC;
  hBmp=(HBITMAP)LoadImage(NULL,filename,IMAGE_BITMAP,0,0,LR_LOADFROMFILE);
  hDC=GetDC(hWnd);
  DrawState(hDC,NULL,NULL,(LPARAM)hBmp,NULL,0,0,0,0,DST_BITMAP);
  DeleteObject(hBmp);
  ReleaseDC(hWnd,hDC);
}
```

14. Dessin

Cours théorique :

L'affichages de 'dessins' (lignes, cercles, points) dans une fenêtre passe par le GDI. Les dessins se font avec un pointeur. Par défaut, le pointeur est un pixel unique. Tracer une ligne donne donc le résultat attendu : une ligne de 1 pixel de large. Pour dessiner une ligne plus large, il va falloir remplacer le pointeur courant par un pointeur plus large. Pour cela, on utilise le fonction **CreatePen()**. Elle retourne une variable de type **HPEN**. Ensuite, il faut sélectionner le pointeur ainsi créé dans le contexte d'affichage courant. Bien entendu, il faudra, tout comme avec les polices, supprimer les pointeurs dès que ceux-ci ne sont plus utilisés.

Voici un aperçu des fonctions utilisées pour dessiner :

SetPixel() dessine un pixel unique, indépendamment du pointeur courant.

GetPixel() retourne la couleur du pixel à la position spécifiée.

MoveToEx() déplace la position courante du curseur de dessin.

LineTo() trace une ligne depuis la position courante du curseur de dessin jusqu'à la position spécifiée. La ligne est tracée avec le pointeur courant.

Arc(), **ArcTo()** et **AngleArc()** dessinent des ellipses ou des portions d'ellipses.

Polyline() dessine une série de lignes connectées.

15. Fond personnalisé

Cours théorique :

Jusqu'à présent, nous avions laissé Windows effacer le fond. L'effacement de la zone client avant le redessinement n'est pas nécessaire, la méthode la plus simple et la plus rapide est de redessiner par dessus l'ancienne zone. Toutefois, si l'on veut redessiner une partie de la zone client seulement ou si le nouveau dessin ne couvre pas toute la zone (du texte avec fond transparent par exemple) il est nécessaire d'effacer d'abord.

Pour cela, il faut utiliser le message **WM_ERASEBKGND**. Lorsque ce message est envoyé, un 'handle' sur le contexte d'affichage à effacer est passé en paramètre. Il faut impérativement utiliser le 'handle' passé en paramètre pour effectuer toutes les opérations de dessin. L'application dispose de plusieurs solutions pour effacer la zone client. Elle peut effectuer des tâches complexes, afficher des images, puis un texte... Si les images ne couvrent pas toutes la zone, il faut effacer les parties qui resteront visibles. Pour cela, on peut utiliser la fonction **FillRect()**. Cette fonction ne prend pas en paramètre une couleur mais un **HBRUSH**. On peut créer des 'brushes' à partir d'images bitmap avec **CreatePatternBrush()** ou encore à partir d'une couleur unie avec **CreateSolidBrush()**. Si l'application désire simplement modifier la couleur de fond de sa zone client, elle n'est pas obligée de traiter le message **WM_ERASEBKGND**. Elle peut simplement utiliser **CreateSolidBrush()** et passer le résultat au membre **hbrBackground** de la structure **WNDCLASSEX** lors de la création de la classe de la fenêtre. De cette manière la fonction **DefWindowProc()** écrasera le fond elle même avec la couleur demandée.

16. Commandes systèmes

Cours théorique :

Pour personnaliser une application, on peut être amené à modifier le traitement standard des commandes systèmes. Pour cela, on utilise le message **WM_SYSCOMMAND**. Ce message est envoyé en diverses occasions : déplacement de la fenêtre, utilisation du menu de contrôle sur la barre de titre (fermer, minimiser, restaurer), activation de l'écran de veille, déroulement du menu démarrer... En interceptant ce message, une application peut définir des réactions personnalisées aux traitements standard Windows. Par exemple, masquer la fenêtre au lieu de la détruire lors d'un clic sur la 'croix'. Comme

le même message est envoyé (**WM_SYSCOMMAND**) quel que soit la commande système, il faut impérativement appeler **DefWindowProc()** pour traiter les commandes systèmes non prises en charge par l'application.

17. Menus

Cours théorique :

Il existe deux manières d'insérer des menus dans une application. La première méthode est de créer un menu grâce à un éditeur de ressources puis de l'insérer dans le programme. La seconde méthode est dynamique, grâce aux fonctionnalités de l'API Windows. De manière générale, les menus obtenus par des ressources seront utilisés pour les menus principaux des fenêtres, pour créer des barres de menus. Pour les menus contextuels, boîtes à outils, ..., il est préférable d'utiliser des menus dynamiques. A chaque clic dans un des menus, la fenêtre possédant le menu reçoit un message **WM_COMMAND** indiquant l'identifiant de l'option choisie.

Pour obtenir une barre de menus à partir d'un fichier de ressources, il suffit de créer le menu dans l'éditeur de ressources puis de passer son identifiant au membre **lpszMenuName** de la structure **WNDCLASSEX**. Le menu sera inséré automatiquement dans la fenêtre.

Pour un menu dynamique, on utilisera l'API Windows. Les menus sont identifiés par une variable **HMENU**. Il faut tout d'abord créer le menu. Il sera initialement vide. Pour cela, il faut utiliser la fonction **CreatePopupMenu()**. Une fois le menu créé, on peut lui ajouter des éléments (options, sous menus, séparateurs) avec les fonctions **AppendMenu()** ou **InsertMenuItem()**. Une fois le menu créé, il faut le dessiner. Pour cela, on utilise les fonctions **TrackPopupMenu()** ou **TrackPopupMenuEx()**. Pour une boîte à outils, il suffira de dessiner le menu à la position courante obtenue par un appel à **GetCursorPos()**. Si le menu est associé à une fenêtre, il sera supprimé automatiquement lors de la destruction de la fenêtre. Dans le cas contraire, l'application doit appeler la fonction **DestroyMenu()** pour libérer les ressources.

La fonction suivante dessine un menu contextuel :

```
void PrintMenu(HWND hWnd)
{
  HMENU hMenu;
  POINT pt;
  GetCursorPos(&pt);
  hMenu=CreatePopupMenu();
  AppendMenu(hMenu,MF_STRING,1,"Item 1");
  AppendMenu(hMenu,MF_STRING,2,"Item 2");
  AppendMenu(hMenu,MF_SEPARATOR,NULL,NULL);
  AppendMenu(hMenu,MF_STRING,3,"Item 3");
  TrackPopupMenu(hMenu,NULL,pt.x,pt.y,0,hWnd,NULL);
}
```

Dans cet exemple, les valeurs 1, 2 et 3 passées à **AppendMenu()** représentent les identifiants des options. Ces valeurs seront passées en paramètre lors de l'envoi du message **WM_COMMAND**. Elles n'ont aucune signification particulière, si ce n'est pour le programmeur lui même.

18. Création dynamique d'un contrôle

Cours théorique :

La création dynamique de contrôles ne présente aucune difficulté particulière. Elle se fait grâce à la fonction **CreateWindowEx()**, en utilisant des classes de fenêtres prédéfinies. Une fois le contrôle créé, il faut tout de même remarquer que son utilisation est quelque peu différente. En effet, des fonctions telles que **SetDlgItemText()** ne fonctionneront plus. Il faudra utiliser la fonction **SetWindowText()** à la place, puisque le contrôle est une fenêtre. La fonction **SetDlgItemText()** n'est en fait qu'un raccourci qui appelle **SetWindowText()** après avoir utilisé **GetDlgItem()** pour obtenir le 'handle' du contrôle correspondant.

Les styles utilisés dans l'éditeur de ressources sont les mêmes. La fonction **CreateWindowEx()** acceptera donc comme styles l'ensemble des styles utilisés dans l'éditeur de ressources.

Les classes suivantes sont prédéfinies : **BUTTON**, **COMBOBOX**, **EDIT**, **LISTBOX**, **MDICLIENT**, **RichEdit**, **RICHEDIT_CLASS**, **SCROLLBAR**, **STATIC**.

Chapitre 4

Le système de fichier

1. Introduction

Cours théorique :

L'API Windows fournit une suite de fonctions permettant d'accéder au système de fichier. Bien entendu, il est toujours possible d'utiliser les fonctions classiques du C (fopen(), fprintf()) pour accéder au système de fichier. Cependant, ces fonctions ne supportent pas toutes les possibilités du système de fichier Windows (ex. : la gestion du partage des ressources dans l'environnement Windows). Il est donc en général préférable d'utiliser les fonctions fournies avec l'API Windows pour manipuler le système de fichier.

Les fonctions de manipulation de fichiers (ouverture, écriture, lecture) seront donc abordées dans ce paragraphe. De plus, l'utilisation de fichiers dans un contexte multi-tâches impose des précautions d'emploi supplémentaires. Le non respect de certaines précautions simples peut entrainer des erreurs ou même la perte ou l'écrasement de données.

Le problème de la recherche de fichiers dans le système de fichier sera également abordé.

Ce paragraphe a donc pour but de présenter de manière générale les opérations de base lors de la manipulation du système de fichier Windows grâce aux fonctions de l'API Windows.

2. Création et ouverture d'un fichier

Cours théorique :

L'API Windows fournit la fonction **CreateFile()** pour ouvrir ou créer des fichiers. Cette fonction permet de gérer le partage du fichier au niveau système, les attributs de ce fichiers. Il est important de bien comprendre les paramètres principaux de cette fonction.

Avant la description plus en détail de cette fonction, il est important de bien comprendre le fonctionnement du système de fichiers sous Windows. Comme cet environnement est multi-tâches, il est possible qu'un fichier soit accédé en même temps par plusieurs programmes ou même par le même programme. Le premier programme à demander l'accès au fichier doit préciser si il souhaite ou non le partager. Si le fichier n'est pas partagé, le système refusera tout autre appel à **CreateFile()** pour accéder à ce fichier. Le fichier ne sera donc plus accessible au reste du système. Le fichier restera 'réservé' jusqu'à ce que le programme se termine ou jusqu'à ce que la fonction **CloseHandle()** soit appelée (cette dernière fonction sera expliquée plus tard). En ne partageant pas le fichier le programme s'assure qu'il est le seul à accéder au fichier.

Un fichier peut également être partagé en lecture. Dans ce cas, seul le programme ayant accédé au fichier en premier aura un accès en lecture/écriture. Les appels suivants à **CreateFile()** permetteront seulement d'accéder au fichier en lecture.

Si le fichier est partagé en lecture/écriture, alors l'accès simultané en lecture ou écriture à ce fichier est autorisé. Plusieurs programmes peuvent donc lire ou écrire des données simultanément dans ce fichier. Ce dernier mode de partage est très dangereux et ne doit être utilisé qu'en connaissance de cause.

Lors du premier appel à **CreateFile()**, le mode de partage du fichier est défini. Il ne pourra pas être modifié par un autre appel. Même si le fichier est partagé, son mode de partage ne pourra pas être modifié. Si l'appel à la fonction **CreateFile()** réussit, le système retourne un **HANDLE** au programme. Ce **HANDLE** identifie le fichier. La fonction **CloseHandle()** permet de refermer le fichier. A partir de cet appel, le **HANDLE** identifiant le fichier n'est plus valide.

L'identifiant du fichier stocke également la position courante du pointeur. Le système stockera donc la position courante d'écriture pour chaque nouvel appel à **CreateFile()**. Si le fichier est ouvert 10 fois simultanément, le système stockera 10 positions. Le partage du fichier n'entraîne donc aucune modification dans les méthodes de lecture/écriture sur le fichier (le pointeur courant ne sera pas modifié même si un autre programme accède le fichier). Cependant, si deux threads utilisent le même **HANDLE** pour accéder à un fichier, alors le pointeur pourra être modifié par l'un ou par l'autre. Si le même thread fait deux appels successifs à l'API pour lire le fichier, rien ne garantit qu'il lira deux blocs successifs. En effet, si le deuxième thread a modifié le pointeur entre les deux appels, alors la lecture se poursuivra à la nouvelle position.

Pour éviter des complications, le plus simple est généralement de ne pas partager les fichiers ou d'autoriser seulement le partage en lecture.

Voici un détail de principaux arguments de la fonction **CreateFile()** :

```
HANDLE CreateFile(
  LPCTSTR lpFileName,          // pointer to name of the file
  DWORD dwDesiredAccess,       // access (read-write) mode
  DWORD dwShareMode,           // share mode
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
```

```
                              // pointer to security attributes
  DWORD dwCreationDisposition,  // how to create
  DWORD dwFlagsAndAttributes,  // file attributes
  HANDLE hTemplateFile         // handle to file with attributes to
                              // copy
);
```

lpFileName est le nom du fichier à ouvrir ou à créer.

DwDesiredAccess est le type d'accès demandé : lecture (**GENERIC_READ**), écriture (**GENERIC_WRITE**) ou les deux. Si le fichier est partagé en lecture seulement et qu'un accès en écriture est demandé, la fonction retournera une erreur.

DwShareMode indique le mode de partage demandé. Ce mode peut être non partagé (**NULL**), partagé en lecture (**FILE_SHARE_READ**), partagé en écriture (**FILE_SHARE_WRITE**) ou les deux.

DwCreationDisposition indique la méthode utilisée pour lire ou créer le fichier. CREATE_NEW demande la création d'un fichier. Si le fichier existe déjà, la fonction retournera une erreur. **CREATE_ALWAYS** demande la création d'un fichier. Si le fichier existe déjà, il est écrasé. **OPEN_EXISTING** demande l'ouverture d'un fichier existant. Si le fichier n'existe pas, le fonction retourne une erreur. **PEN_ALWAYS** demande l'ouverture du fichier. Si le fichier n'existe pas il est créé.

dwFlagsAndAttributes indique les attributs utilisés pour créer le fichier.

La fonction retournera un **HANDLE** en cas de succès ou **INVALID_HANDLE_VALUE**, c'est à dire le pointeur 0xFFFFFFFF est cas d'erreur.

Dès que le fichier n'est plus utilisé, l'application peut utiliser **CloseHandle()** pour signifier qu'elle n'utilise plus le fichier spécifié. Toute autre application pourra alors avoir accès à ce fichier si elle le demande.

3. Lecture/Ecriture dans un fichier

Cours théorique :

Avant de pouvoir lire ou écrire dans un fichier, il faut réaliser un appel à **CreateFile()**. Le **HANDLE** retourné permettra d'identifier le fichier dans lequel on souhaite effectuer les opérations. La fonction **ReadFile()** permet la lecture dans un fichier. Elle déplace le pointeur courant à la nouvelle position. **ReadFile()** lit le nombre d'octets spécifiés et retourne un indicateur booléen, ainsi que le nombre d'octets lus. **ReadFile()** retournera toujours **TRUE** si le pointeur sur le fichier est valide. Pour détecter la fin du fichier, il faut comparer le nombre d'octets lus au nombre d'octets demandés. Si 256 octets sont demandés à la lecture et que **ReadFile()** indique que seulement 18 octets ont été lus, alors la fin du ficher est atteinte.

La fonction **WriteFile()** a un mode de fonctionnement similaire à **ReadFile()**. Elle écrit le nombre d'octets demandé, retourne un indicateur de succès ainsi que le nombre d'octets effectivement écrits.

Pour déplacer le pointeur courant on utilise la fonction **SetFilePointeur()**. La nouvelle position peut être spécifiée à partir du début du fichier, de la position courante du fichier ou de la fin du fichier. Cette fonction retourne la nouvelle position. Pour obtenir la position courante du pointeur, on appelle **SetFilePointeur()** en demandant un déplacement nul à partir de la position courant. On récupère ainsi la position courante du pointeur.

La fonction **SetEndOfFile()** détermine la position de la fin du fichier. La fin du fichier est alors placée à la position courante du pointeur.

4. Manipulations sur les fichiers

Cours théorique :

D'autres fonctions peuvent être utilisées lors de la manipulations de fichiers. Pour obtenir la taille d'un fichier, il faut utiliser **GetFileSize()**. Cette fonction retourne sur 32 ou 64 bits la taille du fichier spécifié.

Les fonctions **MoveFileEx()**, **CopyFileEx()** et **DeleteFile()** peuvent être utilisées pour les manipulations courantes sur les fichiers. Ces fonctions ne nécessitent pas que le fichier soit précédemment ouvert par **CreateFile()**. La fonciton **DeleteFile()** supprime un fichier sans le faire passer par la corbeille. Il est donc impossible d'annuler la suppression du fichier. Pour envoyer simplement un fichier dans la corbeille, il faut utiliser la fonction **SHFileOperation()**. Cette fonction permet à la fois la suppression et l'envoi à la corbeille d'un fichier.

Le système de fichier VFAT (utilisé sous Windows 95/98) manipule des noms courts et des noms longs. Les noms de fichiers courts (format 8.3) sont un héritage des précédents systèmes de fichiers utilisés. Pour convertir des nom de fichiers en version courte ou longue on utilise les fonctions **GetShortPathName()** et **GetFullPathName()**.

5. Enumération de fichiers

Cours théorique :

Pour énumérer des fichiers, l'API Windows offre une sélection de 3 foncitons : **FindFirstFile()**, **FindNextFile()** et **FindClose()**. La fonction **FindFirstFile()** initialise la recherche et retourne le premier fichier trouvé (si il y en a un). Elle retourne un **HANDLE** sur la recherche en cours, pour permettre de poursuivre la recherche. Le fonction **FindNextFile()** doit ensuite être appelée pour récupérer l'ensemble des fichiers trouvés. Généralement cette fonction sera appelée dans une boucle. La fonction **FindClose()** referme le **HANDLE** et libère la mémoire occupée.

Cette suite de fonctions permet d'effectuer des recherches dans un dossier. Attention, les sous dossiers ne seront pas parcourus. La recherche peut porter sur un listage non exhaustif, par exemple, listage des fichiers '*.txt'. Pour un listage exhaustif, il faudra demander le listage des fichiers '*.*'. Le type de recherche, ainsi que le dossier de recherche sont passés à la fonction **FindFirstFile()** sous forme d'une chaîne de caractère de type : Chemin\\Masque (ex. c:*.*). Aucune fonction n'est fournie pour le parcours récursif des dossiers de manière à explorer les dossiers et sous dossiers. Pour effectuer une recherche complète, il faut effectuer des appels successifs à ces 3 fonctions de manière à parcourir l'ensemble des sous dossiers. Lors d'une recherche, les pseudo-dossiers '.' et '..' sont également listés. Il peut donc être utile d'effectuer un test pour supprimer ces dossiers du listage final. Les informations concernant les fichiers et dossiers listés sont placées dans une structure **WIN32_FIND_DATA** (attributs, nom, taille...).

6. Manipulations sur les dossiers

Cours théorique :

L'API Windows offre peu de manipulations possibles sur les dossiers. La fonction **CreateDirectory()** permet la création d'un dossier. Attention toutefois au fait que cette fonction ne crée pas les sous dossiers nécessaires. Si la création du dossier demandé demande la création implicite de plusieurs dossiers, la fonction retournera une erreur. Pour ex., le dossier c:\a\b ne sera créé que si le dossier c:\a existait déjà. Pour créer ce dossier, il faudra faire deux appels à **CreateDirectory()** pour créer respectivement les dossiers c:\a et c:\a\b.

Pour supprimer un dossier, on utilisera la fonction **RemoveDirectory()**. Cette fonction ne supprimera le dossier que si il est vide. Pour supprimer un dossier non vide il faudra donc lister l'ensemble des fichiers et dossiers et les supprimer avant de supprimer le dossier parent.

La fonction **GetCurrentDirectory()** retourne le répertoire courant. Elle peut être utilisée par exemple pour obtenir le chemin du processus courant. En effet, généralement, les programmes sont exécutés avec pour dossier courant, le dossier contenant l'exécutable. Ceci est toujours vrai sauf mention contraire de l'utilisateur. La récupération du chemin de l'exécutable courant par cette méthode n'est donc pas garantie mais elle marchera dans la plupart des cas.

La fonction **GetTempPath()** permet de récupérer le chemin du dossier temporaire (généralement Windows\temp). Les fonctions **GetWindowsDirectory()** et **GetSystemDirectory()** permettent de récupérer le chemin du dossier contenant Windows et du dossier système (system ou system32).

7. Manipulations des chemins

Cours théorique :

Windows fournit une collection de fonctions permettant de travailler sur les fichiers, les dossiers ou les chemins. Voici une liste non exhaustive des fonctions les plus utilisées :

PathIsDirectory() permet de déterminer si le chemin spécifié est un dossier.

PathFileExist() détermine si le fichier spécifié existe.

PathCanonicalize() supprimer toutes les références symboliques du chemin et le convertit en chemin absolu. Ex. : (c:\a\.. sera converti en c:\).

PathCompactPath() permet de réduire la taille occupée par le chemin. Cette fonction remplace une partie de chemin par des points.

PathFindExtension() retourne l'extension du nom de fichier spécifié.

PathFindFileName() retourne le nom de fichier dans le chemin spécifié.

PathGetArgs() retourne les arguments du chemin spécifié.

PathRelativePathTo() permet de relativiser un chemin par rapport à un autre chemin.

PathRemoveArgs() supprime les arguments du chemin passé en paramètre.

PathRemoveExtension() supprime l'extension du chemin, si celle-ci existe.

Pour une liste exhaustive, reportez vous à l'annexe B.

Chapitre 5

Le multithreading

1. Introduction

Cours théorique :

Commençons tout d'abord par une explication sur le terme de multithreading. Le multithreading est le fait d'exécuter des tâches simultanément (multitâche). Les termes multitâche et multithread ne sont pas totalement similaires. Le terme multithread indique une application qui effectue différentes taches simultanément. Le terme de multitâche est un terme plus général et plus commun pour parler d'un système capable de gérer plusieurs applications simultanément.

Dans un programme classique, les instructions sont traitées de manière linéaire, "ligne à ligne". En ignorant le reste du système on pourrait dire que le processeur exécute les instructions de la ligne courante, puis passe à la suivante. En réalité ce n'est pas tout à fait vrai car les autres applications doivent elles aussi avoir accès au processeur, ainsi que le système d'exploitation lui même.

Un thread est en fait une unité d'exécution. Chaque thread peut avoir accès au processeur et exécuter des instructions. Un thread peut se comporter exactement comme si il ne partageait pas le processeur (mais dans ce cas, il faudrait disposer d'autant de processeurs qu'il y a de threads). Pour éviter cet inconvénient, le système donne l'accès au processeur à chaque thread durant un temps très court (quelques millisecondes), et ceci de manière circulaire. Si le nombre de thread augmente, chaque thread devra attendre un délai plus long avant d'obtenir de nouveau l'accès au processeur.

De cette manière, chaque application peut fonctionner comme si elle était seule à utiliser le processeur. La seule différence est que l'exécution est plus lente. Si le processeur est suffisamment rapide, l'exécution d'un nombre modéré de threads peut s'effectuer très rapidement. C'est le cas actuellement. Dans un système utilisant Windows, le nombre de threads avoisine 50 au repos. Il dépasse rapidement 100.

L'utilisation du multithread consiste à créer un programme qui comporte plusieurs threads. Ce programme peut donc exécuter plusieurs instructions "simultanément". On peut alors se demander l'utilité d'une telle pratique. En effet, puisque le processeur est partagé, l'exécution du programme ne sera pas plus rapide. Elle sera seulement fragmentée. Un exemple simple permet de comprendre l'utilité du multithreading.

Considérons une application réalisant un transfert de fichier. Cette application dispose d'une interface graphique présentant une barre de progression ainsi qu'un bouton 'annuler'. En effet l'utilisateur désire être tenu au courant de l'avancement de la copie. De plus il veut pouvoir stopper celle ci à tout moment si il le désire. Le programme doit donc réaliser une boucle de manière à recevoir les messages. Mais il doit également s'occuper de la copie des fichiers, ce qui reste son rôle principal. Si le programme écrit sur le disque, il ne peut plus recevoir les messages. La fenêtre ne sera donc plus rafraîchie. De plus, si l'utilisateur clique 'annuler', l'application ne traitera pas cette demande puisqu'elle ne recevra même pas le message... Une telle application utilisera

donc 2 threads. Le premier thread s'occupera de la réception et du traitement des messages. Le deuxième thread s'occupera de la copie des fichiers. De cette manière, la réception des messages sera effectuée même au cours de la copie. Si l'utilisateur demande l'arrêt de la copie, le premier thread devra simplement stopper l'exécution du second. La copie sera alors stoppée.

2. Notion de processus et de thread

Cours théorique :

Il ne faut pas confondre les processus et les threads. Un processus est une application. Tout processus dispose d'au moins un thread. Il peut cependant en posséder plusieurs. Pour l'utilisateur moyen, les threads sont invisibles. Ils sont masqués par les processus. La différence entre un processus et un thread est qu'un processus ne constitue en rien une unité d'exécution. L'exécution simultanée de plusieurs processus est réalisée grâce au multithreading. Le processus représente simplement un application vis à vis du système. Lorsque l'utilisateur demande la terminaison d'une application, tous les threads appartenant à cette application devront être terminés.

Tout processus doit comporter au moins un thread, de manière à exécuter le point d'entrée du programme. Ce thread ne pourra jamais être terminé sans que l'application ne soit elle même terminée. Le processus ne doit en aucun cas être assimilé à ce thread. Le thread constitue simplement l'unité d'exécution de base du processus. Le système ne communiquera jamais avec le processus mais toujours avec l'un des threads de ce processus. En effet, le processus n'étant pas une unité d'exécution il ne réalise aucune tâche. Le premier thread est créé par le système. C'est pour cette raison que dans un programme comportant un seul thread, aucune référence n'est faite aux fonctions de l'API relatives aux threads. La création de nouveaux threads devra être explicite. C'est ce qui sera étudié tout au long de ce chapitre.

3. Partage des tâches

Cours théorique :

Avant d'utiliser des threads, il est nécessaire de bien comprendre comment le système gère ces threads. En effet, un programme multithread mal conçu peut conduire à une très forte baisse des performances système.

Il faut bien comprendre que si le système d'exploitation accorde à chaque thread un temps d'accès au processeur, le thread n'est pas obligé d'utiliser totalement ce temps. Le système accorde simplement un temps maximum d'utilisation. Au delà de cette période, le contrôle du processeur sera passé à un autre thread. Si le thread n'a plus aucune tâche à effectuer, il doit repasser lui même le contrôle au système. De cette manière, une application inactive ne monopolisera pas inutilement le processeur. Si une application inactive n'effectue pas ce passage, elle utilisera totalement son quota d'accès. Le système affichera alors une charge processeur de 100% et les performances seront fortement ralenties.

Le système d'exploitation dispose également d'un système de priorités permettant de gérer les quotas d'accès alloués aux processus. Voici un résumé rapide du fonctionnement de ce système de priorité :

Time Critical : ce thread requiert l'utilisation totale du processeur. Les quotas alloués à ce thread sont infinis. Si un tel thread ne repasse pas spontanément le contrôle au reste du système, le système restera en attente. Le contrôle du processeur ne sera accordé à aucun autre thread. Tout le système sera alors bloqué jusqu'à ce que le thread repasse le contrôle. Ce type d'application n'est en réalité quasi jamais utilisé. Il doit être réservé à des utilisations très particulières (ex : un gestionnaire de souris).

High : ce thread recevra le contrôle du processeur en priorité. Un tel thread ne peut pas être bloquant pour le système mais il pourra ralentir énormément les performances. Un thread utilisant cette priorité et effectuant un accès permanent au processeur causera un effondrement des performances générales du système. Cette priorité ne doit généralement être accordé qu'à des applications effectuant des tâches très brèves ou étant en premier plan. Les performances de ce thread ne seront quasiment pas affectées par l'exécution d'autres applications.

Normal : ce thread a une priorité normale. Il recevra le processeur exactement autant de fois que les autres threads de la même priorité. Si deux threads utilisent cette priorité et utilisent le processeur en permanence, les performances de chacun des thread seront divisées par 2. La quasi totalité des threads utilise cette priorité (c'est d'ailleurs la plus recommandée hors cas très particuliers).

Idle : ce thread recevra le contrôle du processeur seulement si aucune autre application ne demande le contrôle. Si ce thread effectue des tâches en continu et que toutes les autres applications sont inactives, il disposera de l'accès processeur dans sa quasi totalité. Si un autre thread effectue des tâches continues, ce thread n'aura plus aucun accès au processeur. Ce type de priorité peut être accordé à des applications effectuant des tâches de fond. De cette manière les performances du système resteront inchangées. La charge du processeur sera alors continuellement de 100%. Le partage sera alors : 0% pour le thread Idle si le système utilise le processeur. 100% pour le thread Idle si le système est inactif.

De manière générale il est inutile de modifier la priorité des threads. La modification des priorités ne doit être faite que dans les cas particuliers et en connaissance de cause. En particulier, les priorités 'Time Critical' et 'High' ne doivent jamais être utilisées abusivement.

De manière à conserver des performances système optimales, un thread doit repasser le contrôle au système dès qu'il devient inactif. Pour cela il dispose de plusieurs méthodes. Prenons l'exemple d'un thread qui désire attendre 150ms. Si ce thread effectue une boucle en testant l'heure système, il utilisera totalement son quota et le système affichera une charge de 100%. Si eu lieu de cela le thread effectue un appel à la fonction Sleep(), il repassera le contrôle au reste du système pour une durée de 150ms. Le contrôle du processeur ne lui sera plus accordé durant cette durée. La charge processeur sera alors de 0%. Le thread sera alors en attente. De cette manière les autres threads auront accès au processeur comme ci ce thread n'était pas présent et les performances système seront conservées.

Un autre exemple est la récupération des messages. La fonction GetMessage() repasse le contrôle au système dès que tous les messages ont été traités. Si l'application est inactive, le contrôle est donc repassé immédiatement au système et la charge processeur affichée est de 0%. Si un thread utilise PeekMessage() dans une boucle, le thread conservera l'accès au processeur même si aucun message n'est présent dans la file d'attente. La charge processeur sera donc de 100% même si l'application est inactive. Ce type d'erreurs ne doit donc surtout pas être commis sous peine de diminuer gravement les performances globales du système.

Il est également indispensable de comprendre qu'on ne peut jamais prévoir l'ordre dans lequel les threads recevront l'accès au processeur. Une application multithread devra toujours utiliser les fonctions de synchronisation (qui seront détaillées plus tard) et ne devra jamais tenter de prévoir l'ordre d'exécution des instructions. De plus, deux lignes consécutives d'une fonction ne seront pas forcément effectuées à la suite car le contrôle du processeur pourra être passé à d'autres threads entre temps. Il faut donc bien avoir en tête les contraintes imposées par un programme multithread avant de se lancer dans sa réalisation.

4. Synchronisation

Cours théorique :

Les fonctions de synchronisation sont indispensables dans un programme multithread. La mauvaise synchronisation d'une application peut conduire à des plantages ou à des performances totalement désastreuses.

Etudions l'exemple d'un processus contenant deux threads, chacun utilisant le même tableau global. Un tel programme peut donner des résultats surprenant dans le cas d'une mauvaise synchronisation. Par exemple, deux évaluations consécutives de la même variable pourront retourner des valeurs différentes. Ce genre de comportement n'est généralement pas souhaité. Les deux threads devront alors se synchroniser avant d'accéder aux ressources partagées. Dans ce cas, la synchronisation consistera en une exclusion mutuelle (d'autres types de synchronisation seront étudiées plus tard). L'exclusion mutuelle consiste à faire patienter tous les threads demandant l'accès aux données partagées tant que le thread les utilisant n'a pas déclaré qu'il avait terminé. Chaque thread utilise une fonction pour signaler qu'il entre dans une section protégée. Il utilisera ensuite une seconde fonction pour signaler qu'il sort de la section protégée. Si un autre thread demande à entrer dans un bloc protégé, il sera mis en attente. Le thread qui est entré dans la section protégée doit impérativement signaler sa sortie de la section protégée sinon les autres threads resteront en attente. L'accès aux données partagées doit donc être aussi bref que possible.

Les fonctions de synchronisation sont un très bon moyen de mettre des threads en attente tout en conservant une charge processeur nulle. Si un thread entre dans une phase d'attente, son utilisation processeur restera très faible durant toute la période d'attente. Le système lui accordera de nouveau le contrôle dès que cela sera possible.

La synchronisation des threads devra donc toujours être effectuée par des fonctions de l'API et jamais par des boucles. L'utilisation de boucles ne repassant pas le contrôle au système rendrait les performances désastreuses et ralentirait l'exécution du processus lui même car l'accès du processeur serait accordé à des threads en attente, ce qui ne présente aucun intérêt.

5. Premier exemple

Cours théorique :

Ce premier exemple montre comment créer un thread. Le point d'entrée du thread peut être n'importe quel fonction respectant un certain typage. Dès la fonction constituant le point d'entrée retourne, le thread est terminé par le système.

La fonction **CreateThread()** qui démarre le thread retourne un HANDLE ainsi qu'un identifiant. L'identifiant est une valeur de type DWORD. Dans cet exemple, le HANDLE et l'identifiant ne nous seront d'aucune utilité car le thread se terminera spontanément. Le HANDLE ne sera donc même pas récupéré. Il est indispensable de passer un pointeur non nul pour l'identifiant. Dans le cas contraire, la fonction **CreateThread()** échouera. L'identifiant ou le HANDLE sont utilisés pour communiquer avec le thread.

Notons que l'ensemble des variables globales est accessible à la procédure du thread.

Télécharger le projet commenté : Projet 08

6. Arrêt d'un thread

Cours théorique :

Le problème de l'arrêt d'un thread est délicat. Un thread peut s'arrêter spontanément en terminant sa fonction principale. Dans ce cas, on considèrera que l'arrêt était prévu. Dans le cas où l'application doit se terminer, l'arrêt ne peut pas être prévu. Dans ce cas, il existe différentes méthodes de terminer les threads en cours.

Voici tout d'abord la méthode la plus évidente, et aussi la plus mauvaise. La fonction **TerminateThread()** fournie par l'API permet de terminer un thread. Le thread est alors immédiatement stoppé. Bien que cette fonction soit fournie en standard avec l'API Windows, elle ne sera en pratique quasiment jamais utilisée. Son utilisation doit être réservée à des cas très particuliers et en connaissance de cause. Comme ces cas sont généralement très rares on peut considérer que dans un programme bien conçu, cette fonction ne devra jamais être utilisée. En effet, cette fonction termine le thread courant sans appeler les fonctions de désallocation de mémoire. Ceci est dû au fait qu'il n'existe pas de point de sortie prédéfini à un thread. La fonction **TerminateThread()** ne peut donc pas effectuer la désallocation de mémoire. Toute la mémoire allouée de manière dynamique sera donc perdue. De même, les dlls chargées par le thread ne seront pas libérées... Cette fonction ne doit donc être utilisée que dans le cas d'un thread n'utilisant que des allocations de mémoire statique, et aucune ressource extérieure...

Pour éviter l'utilisation de cette fonction, chaque thread doit être muni d'un ou de plusieurs points de sortie. Les systèmes de communication inter-threads seront utilisés pour mener le thread vers ce point de sortie. De cette manière la désallocation de la mémoire ainsi que la libération des ressources extérieures pourra être effectuée. La communication inter-threads sera traitée plus tard. On peut tout de même citer les variables globales et les messages qui sont les principaux moyens de communication entre threads. Bien que plus complexes à mettre en place, les méthodes de communication inter-threads seront donc toujours à préférer.

L'arrêt de l'application entraîne automatiquement l'arrêt de l'ensemble des threads. Le système appellera alors la fonction **TerminateThread()** pour arrêter les threads encore actifs. Une application doit donc toujours s'assurer de l'arrêt de l'ensemble des threads avant de se terminer. Les fonctions de synchronisation sont très adaptées à ce type de tâche, comme la fonction **WaitForMultipleObjects()** qui sera vue en détails plus tard.

7. Récupération des messages

Cours théorique :

La récupération des messages peut avoir deux fonctions : récupérer les messages de communication liés à une fenêtre ou récupérer les messages de communication inter-thread. La réception des messages doit être effectuée de manière différente selon le type de messages. De manière générale, les messages associés à des fenêtres doivent être traités avec la fonction **GetMessage()** dans une boucle. Cette fonction permet de passer le contrôle aux

autres threads du système dès que la file d'attente est vide. De cette manière les performances sont optimales. De plus, les messages associés aux fenêtres doivent être traités rapidement. L'utilisation de cette fonction dans une boucle permet donc un traitement rapide tout en conservant de bonnes performances.

Le traitement des messages dans le cadre de la communication inter-thread est différent (il sera vu plus en détail plus tard). Il concerne par exemple l'envoi d'un message d'abandon de la tâche courante. La fonction **GetMessage()** est très peu adaptée au traitement de ce type de messages car elle est bloquante. Le thread sera donc suspendu tant que la file d'attente sera vide. Un thread ne s'occupant pas de la réception des messages associés à une fenêtre a généralement une tâche bien précise à exécuter. Il serait donc absurde d'utiliser une fonction bloquante pour le traitement des messages. De plus, ces messages ne nécessitent pas un traitement rapide. On préférera donc utiliser la fonction **PeekMessage()** qui elle n'est pas bloquante. Il faudra donc utiliser cette fonction dans une boucle. Comme l'utilisation de cette fonction dans une boucle provoque de graves problèmes de performances, il ne faudra pas effectuer cette boucle trop fréquemment. Prenons l'exemple d'un thread gérant un transfert de fichier. Le transfert de fichier est réalisé par une boucle qui lit les données puis les écrit à une place différente. Selon la durée moyenne d'une boucle, on déterminera un intervalle d'appel de **PeekMessage()**. Par exemple, toutes les 100 itérations de la boucle principale, la fonction **PeekMessage()** sera appelée dans une boucle jusqu'à épuisement de la fille d'attente. Ce type de traitement est adapté à un faible nombre de messages, de manière à ne pas ralentir le travail de la boucle principale. En supposant que la durée moyenne d'une itération de la boucle soit de 5ms, les messages seront donc traités 2 fois par seconde. Si un message demandant l'arrêt de la tâche en cours est envoyé, il sera traité au plus 500ms plus tard, ce qui est très acceptable. De plus, cette technique ne nuit que très peu aux performances de la boucle principale puisque seulement une itération sur 100 sera sensiblement ralentie.

Si la rapidité du traitement des messages n'a que peu d'importance on pourra espacer les appels à **PeekMessage()** de manière à conserver autant que possibles les performances de la boucle principale. Cette fonction sera donc particulièrement adaptée au traitement de messages de notifications envoyés à des thread effectuant une tâche précise.

Dans le cas d'un thread effectuant des tâches 'sur commande' on préférera bien entendu la fonction bloquante **GetMessage()** puisque le thread sera en attente de tâche (et ne consommera donc pas de ressources).

8. Communication inter-threads

Cours théorique :

La communication inter-threads permet aux différent threads d'une application de dialoguer entre eux. Comme la réception des messages peut poser problème, on tentera généralement de réduire au minimum la communication inter-threads et donc de réaliser des threads aussi indépendants que possible.

Outre les fonctions de synchronisation qui ne sont pas à proprement parler des fonctions de communication, il existe deux moyens principaux de communication : les variables globales et les messages. Les variables globales sont un moyen de communication facile à mettre en place. Toutefois, leur utilisation devra être faite avec rigueur pour éviter des problèmes dus à l'accès de variables partagées. De plus dans certains cas, l'utilisation de variables globales peut nuire aux performances.

Une variable globale ne doit jamais être utilisée comme moyen de synchronisation. En effet, il est possible de placer la valeur d'une variable globale à 0 et d'effectuer une boucle avec un test sur cette variable de manière à attendre tant que la valeur reste à 0. Un second thread peut alors débloquer

l'attente en modifiant la valeur de cette variable globale. Bien que fonctionnant, cette méthode ne devra jamais être utilisée car elle provoque une forte charge processeur inutile. Dans ce cas, les fonctions de synchronisation devront être utilisées car elles permettent de laisser l'accès processeur au reste du système tant que le thread est en attente.

De plus, deux threads ne doivent jamais accéder en écriture à une même variable globale. Ce type d'accès peut provoquer des plantages dus à la manière dont les variables globales sont utilisées (ces variables sont parfois placées dans des registres pour optimisation). L'utilisation de variables globales pour la communication inter-threads implique une communication à sens unique. Un thread accède à cette variable en écriture et les autres threads réagissent en fonction de cette valeur. Une telle utilisation ne posera pas de problèmes. Le partage de données mémoire en écriture implique des précautions particulières comme la synchronisation.

Les variables globales pourront donc être utilisées comme drapeau. Par exemple pour signifier à un thread qu'il doit se terminer. Si le thread effectue une boucle, on peut placer dans cette boucle un test sur la variable globale. Le thread principal place cette variable à une valeur principale au moment de quitter, ce qui provoque l'arrêt des autres threads. Notons tout de même que le thread principale devra attendre que l'ensemble des threads ait terminé avant de stopper l'application. En effet, l'arrêt de l'application provoque une fermeture de tous les threads en cours (ce qui revient à un appel de **TerminateThread()**). Il faudra donc placer un système d'attente dans le thread principal (fonctionnant via d'autres variables globales ou par un système de messages de notification).

L'utilisation de variables globales est donc une méthode rapide de communication. Toutefois cette méthode est soumise à un certain nombre de contraintes qui ne doivent surtout pas être ignorées.

Les messages constituent la deuxième méthode de communication. La fonction **PostThreadMessage()** permet d'envoyer des messages à un thread, même si celui-ci ne gère pas de fenêtres. Le thread pourra alors utiliser les fonctions **GetMessage()** ou **PeekMessage()** pour récupérer les messages. Bien entendu, il sera inutile d'utiliser les fonctions **TranslateMessage()** et **DispatchMessage()** puisque ces messages ne sont pas destinés à des fenêtres.

Il existe plusieurs méthodes pour créer des messages personnalisés. Tout d'abord, il est possible de définir une constante au niveau du préprocesseur (#define). Les valeurs valides pour les messages définis par l'utilisateur doivent être compris entre les valeurs WM_USER et 0x7FFF. La constante WM_USER est définie dans les headers Windows.

La deuxième méthode consiste à utiliser la fonction **RegisterWindowMessage()**. Cette fonction retourne une valeur utilisable pour un message en fonction d'une chaîne de caractère. Le système garantit l'unicité du message en fonction de la chaîne de caractère. Si deux applications distinctes utilisent cette fonction avec la même chaîne, elles obtiendront la même valeur. Cette fonction doit donc seulement être utilisée dans le cadre de la communication inter-applications, c'est à dire au niveau système. Si les messages sont locaux à l'application, il est recommandé de définir des constantes.

Si un thread désire communiquer une plus grande quantité d'information, il peut joindre au message 2 valeurs (lParam et wParam). Deux choix sont alors possibles. Utiliser ces valeurs comme paramètres ou joindre un pointeur. La méthode à utiliser pour joindre un pointeur est la suivante : le thread qui poste le message alloue dynamiquement de la mémoire et poste le pointeur. Le thread émetteur ne doit pas désallouer cette mémoire. C'est le thread récepteur qui s'occupera de la désallocation de la mémoire. Cette méthode assure que le pointeur sera toujours valide à la réception du message.

9. Sections Critiques

Cours théorique :

Les sections critiques sont utilisées dans le cadre de la synchronisation, de manière à réaliser des exclusions mutuelles. Elles permettent de créer une section de code qui sera protégé, c'est à dire qu'un seul thread à la fois aura accès à ce code. Les sections critiques sont très utiles pour protéger des données partagées.

L'utilisation des sections critiques est relativement simple. Pour créer une section critique, l'application doit déclarer une variable de type CRITICAL_SECTION, généralement globale. Puis au moins un thread doit initialiser la section critique en appelant la fonction **InitializeCriticalSection()**. Cette fonction peut être appelée un nombre indéfini de fois. Par exemple, chaque thread qui va utiliser la section critique peut s'assurer qu'elle est valide en appelant cette fonction.

Deux fonctions seront ensuite utilisées pour signifier l'entrée et la sortie d'un bloc protégé. La fonction **EnterCriticalSection()** permet de signaler l'entrée dans un bloc protégé. Si un thread a déjà appelé cette fonction et n'est pas encore sorti du bloc protégé, cette fonction mettra en attente n'importe quel autre thread. Le même thread peut appeler cette fonction plusieurs fois sans être bloqué, mais il devra alors appeler la fonction de sortie de bloc un nombre de fois égal pour indiquer sa sortie du bloc protégé.

Un thread indique qu'il sort d'un bloc protégé en appelant la fonction **LeaveCriticalSection()**. Un des threads en attente sera alors débloqué et pourra à son tour exécuter le bloc protégé.

10. Fonctions d'attente

Cours théorique :

Les fonctions d'attente permettent d'attendre de manière explicite qu'un événement se produise. Les fonctions d'attente sont **WaitForSingleObject()** et **WaitForMultipleObjects()**. Ces fonctions permettent d'attendre que l'état d'un objet change. Le thread appelant une de ces fonctions peut définir un temps maximal d'attente (qui sera éventuellement infini).

Différents objets peuvent être utilisés. Par exemple, la fonction **WaitForSingleObject()** peut être utilisée avec le HANDLE d'un thread. La fonction attendra alors que le thread se termine pour retourner. Pour attendre que plusieurs threads se terminent, une application pourra alors appeler la fonction **WaitForMultipleObjects()** en spécifiant qu'elle désire attendre que l'ensemble des threads ait terminé leur exécution. Ces fonctions mettent le thread qui les appelle en attente et sont donc excellentes au niveau performance. Leur utilisation est donc hautement recommandée.

Un événement créé explicitement peut également être utilisé avec ces fonctions. Ce cas sera étudié dans la partie consacrée aux événements. De même, le HANDLE d'un processus peut être utilisé de manière à attendre qu'il se termine.

Dans le cas de la communication inter-threads nous avons vu qu'une application pouvait utiliser une variable globale pour spécifier à ses threads qu'ils devaient se terminer. L'application doit ensuite attendre que les threads se terminent avant de quitter. Cette attente sera typiquement réalisée par un appel à la fonction **WaitForMultipleObjects()**. Lorsque cette fonction retournera, l'application sera alors assurée que l'ensemble des ses threads auront terminés

et pourra quitter sans risque.

11. Evénements

Cours théorique :

Les événements peuvent être utilisés dans les fonctions d'attente. L'API Windows fournit des fonctions permettant de créer des événements de manière explicite. L'état de ces événements pourra ensuite être modifié de manière explicite. L'état d'un événement peut être signalé ou non signalé. Lorsqu'un événement est en état non signalé, l'appel à une fonction d'attente provoquera la suspension du thread appelant jusqu'à ce que l'événement passe en état signalé.

Les fonctions **CreateEvent()** et **CloseHandle()** peuvent être utilisées pour la création et la destruction d'événements. Les fonctions **SetEvent()**, **ResetEvent()** et **PulseEvent()** pourront ensuite être utilisées pour modifier l'état de l'objet.

Les événements peuvent être configurés de différentes manières. La configuration d'un événement se fait lors de sa création. Si l'objet est en mode 'réinitialisation manuelle' il restera en état signalé tant que son état ne sera pas explicitement réinitialisé grâce à la fonction **ResetEvent()**. Si plusieurs threads étaient en attente de cet objet, ils seront alors tous libérés au moment ou l'état de l'objet passera à 'signalé'.

Si l'objet est en mode 'réinitialisation automatique', son état sera automatiquement replacé à 'non signalé' dès qu'un thread en attente sera libéré. Si plusieurs threads sont en attente d'un même événement, alors un seul thread sera libéré eu moment ou l'état sera passé à 'signalé'.

Les événements sont un moyen pratique de synchroniser plusieurs threads de manière explicite.

Les événements peuvent par exemple être utilisés pour mettre un thread en attente tandis qu'un autre thread initialise des données. Dès que les données seront initialisées, le ou les threads seront débloqués par le thread ayant initialisé les données.

Tout comme les sémaphores, les événements peuvent être nommés ou non. Si l'événement est nommé, son nom doit être unique dans l'ensemble du système. Si une autre application tente de créer un événement du même nom, la fonction **CreateEvent()** retournera un HANDLE sur l'événement précédemment créé. Ceci peut être utilisé pour synchroniser des applications entre elles.

12. Sémaphores

Cours théorique :

Les sémaphores sont utilisés pour limiter le nombre de threads en fonctionnement. Prenons le cas d'un serveur. On désire limiter le nombre maximal de connections simultanées. Dans ce cas, on utilisera la synchronisation fournie par les sémaphores.

Pour utiliser un sémaphore, une application doit créer une objet de type sémaphore en appelant la fonction **CreateSemaphore()**. Un sémaphore est constitué d'un compteur. Le statut d'un sémaphore est signalé lorsque le compteur est supérieur à 0 et non signalé lorsque le compteur est à 0. Lors de la création du sémaphore, on spécifie la valeur initiale du compteur, ainsi que sa valeur maximale. La valeur maximale du compteur correspond au nombre

maximum de threads qui pourront être exécutés simultannément.

Chaque thread désirant se synchroniser doit utiliser une fonction d'attente (**WaitForSingleObject()** par exemple). Lorsqu'un thread appelant une fonction d'attente est libéré, le compteur du sémaphore est décrémenté de 1. Dès que la valeur du compteur atteint 0, les threads appelant les fonctions d'attente seront bloqués. Lorsqu'un thread sort d'un bloc synchronisé, il doit appeler la fonction **ReleaseSemaphore()**. Cette fonction incrémente de 1 le compteur du sémaphore, permettant ainsi à un autre thread d'être débloqué.

Le compteur du sémaphore ne pourra jamais dépasser la valeur maximale spécifiée au début. De cette manière, on peut très facilement limiter le nombre de threads effectuant une tâche spécifiée, et conserver ainsi les performances système. Une augmentation trop importante du nombre de threads en exécution simultanée poserait en effet des problèmes de performances.

Il est possible de spécifier un nom lors de la création du sémaphore. De cette manière, des applications différentes peuvent se synchroniser. La première crée un sémaphore nommé. La seconde application tente ensuite de créer un sémaphore du même nom. Ceci aura pour effet de lui retourner un HANDLE sur le sémaphore déjà existant. Le nom d'un sémaphore doit donc être unique dans l'ensemble du système. Remarquons qu'un sémaphore ne doit pas nécessairement être nommé.

13. Trois projets simples

Cours théorique :

Voici 3 projets présentant l'utilisation des fonctions de synchronisation étudiées : sections critiques, événements et sémaphores. Ces 3 projets ont pour but d'expliquer l'utilité des différents types de synchronisation. De plus, ils pourront aider à comprendre la mise en place d'une synchronisation rudimentaire.

Il ne faut jamais oublier que les accès à la console Windows doivent être synchronisés de manière à éviter des résultats totalement imprévisibles. Une section critique est parfaitement adaptée pour résoudre ce genre de problèmes.

Télécharger les projets commentés : Projet 09 - Projet 10 - Projet 11.

14. Performances

Cours théorique :

La création d'applications multithread est généralement très performante. Toutefois, il faudra faire attention à préserver les performances systèmes. La synchronisation devra toujours être effectuées grâce aux objets fournis par l'API comme les sections critiques, les événements ou encore les sémaphores. Il faut toujours penser qu'un thread inactif doit laisser le contrôle au système et non pas effectuer une boucle. Bien qu'elles fonctionnent parfaitement, les boucles sont désastreuses au niveau performances système et on perd rapidement l'intérêt de l'application multithread.

Il ne faut pas non plus tenter de segmenter à tout prix les tâches. Un nombre trop important de threads peut nuire aux performances système. Le

travail qui peut être effectué par un thread ne doit donc pas être confié à 2 threads.

Les applications clients réseau sont souvent très performantes grâce au multithreading. En effet, le fait d'établir plusieurs connections avec la machine hôte permet d'améliorer le débit réseau. Toutefois, un nombre trop important de threads finira par nuire aux performances de l'application car le système perd du temps en passant le contrôle du processeur à un nombre trop important de threads. Les applications sont alors considérablement ralenties.

De plus, la réalisation d'applications multithread est souvent source d'erreurs. En effet, il ne faut jamais oublier que des instructions situées dans des threads seront exécutées dans un ordre quelconque. Il ne faut jamais tenter de prévoir l'ordre d'exécution. A partir du moment où l'ordre d'exécution n'est pas certain, il faudra impérativement utiliser les fonctions de synchronisation.

Il faut également comprendre que le fait d'utiliser des fonctions de synchronisation nuit aux performances de l'application. Les fonctions de synchronisation ne doivent donc pas être utilisées si elles ne sont pas nécessaires. De plus, si un nombre important de threads partage des données, le temps d'attente pour obtenir l'accès à ces données va rapidement devenir important. Chaque thread passera donc un temps important à attendre, ce qui est d'autant plus problématique si la tâche qu'il doit effectuer est courte.

Il faut également éviter impérativement des entrées - sorties successives dans des sections critiques. En effet, si le temps d'exécution entre les sections critiques est trop court, le thread aura très rapidement un temps d'attente supérieur à son temps de calcul. Ce problème se manifestera encore plus avec un nombre important de threads. Si un thread doit faire des accès successifs à des données partagées, il peut copier ces données dans un espace non partagé ou rester dans la section critique le temps d'exécuter sa tâche. Si le temps d'exécution de la tâche est court, un thread aura tout intérêt à ne pas sortir de la section critique.

Il faut donc impérativement prendre en compte les questions de performance lors de la réalisation d'applications multithread. De plus, les plantages dus à des problèmes de partage de données ne surviendront pas forcément lors des tests car les threads ne s'exécutent jamais de la même manière. Il faut donc prévoir les problèmes de partage et non pas compter sur les tests pour les déceler.