1 RAPPEL : ARCHITECTURE NT

1.1 Architecture NT

NT offre une architecture modulaire en couche. Chaque couche est responsable de tâches spécifiques.

- Matériel (carte mère, cartes filles)
- Couche d'abstraction matérielle : HAL
- KERNEL
- Gestionnaire des entrées sorties, mémoire virtuelle …

} Mode privilégié

- Sous système WIN 32
- Application WIN 32

} Mode utilisateur

1.3.2 Le mode privilégié :

Fournit la totalité de la mémoire et du matériel => zone mémoire physique protégée des autres applications

1.3.3 Le mode utilisateur :

Ne permet pas l'accès direct au matériel (=> driver)
Espace adressable limité avec possibilité de swapping disque
Niveau de priorité de traitement des messages

1.3.4 Les sous systèmes :

Permet d'exécuter des applications écrites pour d'autres systèmes d'exploitation en émulation (OS/2 et POSIX)
(les applications Win16 ou DOS sont exécutées sous Win32 dans une machine virtuelle 16 bits Windows 3.x)

1.3.5 Services de l'exécutif

- Les gestionnaires

Gestion de sE/S, des IPC, RPC, mémoire virtuelle, processus …

- Les pilotes de périphériques

Permet l'accès au matériel

- Le kernel

Ordonnanceur des processus (multitâches, multithreads)

- La HAL

Couche d'abstraction matériel

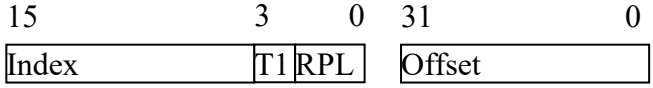
1.2 Pagination mémoire

Grâce à la technique de mémoire virtuelle paginée et l'utilisation d'une portion plus ou moins grande d'espace du disque en plus de la RAM disponible il sera possible, par un adressage 32 bits, d'accéder à 4 Go d'informations.
Cette zone est virtuellement découpée en deux parties de 2Go, une pour le système d'exploitation, l'autre pour l'application.
L'adresse virtuelle est associée à une adresse physique. La mémoire physique est divisée en pages de 4ko permettant ainsi un déplacement aisé d'une page sur le disque (swapping, mode protégé des processeurs Intel)
Le swapping s'exécute suivant l'algorithme Least Recent Used (LRU) : la page la moins récemment utilisée est déplacée sur le disque.

La demande par une application d'une page non disponible en mémoire provoque le swapping inverse.

1.3 Segmentation mémoire

La zone de mémoire linéaire affectée à une application est divisée en segments.
Chaque segment à une adresse de base et une limite.
Une adresse physique sera transformée en une adresse logique : Segment + Offset de la forme :



Index : entrée dans la table des descripteurs permettant d'obtenir l'adresse de base du segment
T1 : sélecteur de type de table (LDT = 1 ou GDT = 0)
RPL : accès sécurisé , mode user , mode kernel
La table de descripteurs contient les caractéristiques du segments (CODE, DATA HEAP…) notamment : Adresse de base, limite (taille du segment) granularité (1 octet, 4 ko …)

La GDT contient les caractéristiques des segments globaux (utilisés par le noyau) et les adresses des LDT.

2 NOTIONS DE THREADS ET DE PROCESS

2.1 Process ou tâche

C'est une entité logique définissant son environnement de travail suivant 4 axes :

- Le contexte matériel

Registres CPU et périphériques virtualisés dédiés au process

- Le contexte logiciel

Privilèges, droits, propriétaire …

- L'adressage virtuel
- L'image

Image mémoire de l'exécutable stocké sur disque

2.2 Ordonnancement et priorité

On distingue trois classes d'exécution :

1. Exécution Batch (en différé) : un processus ne pourra s'exécuter que si on lui a préparé toutes les données dont il a besoin
2. Exécution Itérative : un processus demande à son environnement les données dont il a besoin pour continuer son déroulement.
3. Exécution temps réel : c'est une cas particulier d'exécution itérative avec des contraintes fortes
4.

NT est un système multitâches préemptif.
Un processus démarre avec une priorité comprise entre 0 et 31 :

NT définit 4 classes de priorités pour les process :

- IDLE_PRIORITY_CLASS (basse) : batch et process exécutés si le système n'a rien d'autre à faire ($2 \leq$ priorité ≤ 5)

- NORMAL_PRIORITY_CLASS (normale) : applications interactives (6 ≤ priorité ≤ 10)
- HIGH_PRIORITY_CLASS (haute) : applications critiques (11 ≤ priorité ≤ 15)
- REALTIME_PRIORITY_CLASS (Temps réel) : réservées aux applications temps réel. De priorité supérieure au scheduler, elle ne peuvent pas être préemptées. (Une boucle sans fin dans un process de classe temps réel bloque le système) (21 ≤ priorité ≤ 25)

L'ordonnanceur est donc basé sur la gestion de priorité dynamique . Les process de même priorité sont gérés en temps partagé (Round Robin, time slice).

2.3 Threads :généralités

Pour économiser le coût inhérent à la commutation de contexte lors de la préemption d'un process par un autre les threads, en introduisant la notion de tâche légère, offrent une alternative.
Sous NT un process est composé d'au moins un thread. Il est possible de faire cohabiter dans le même environnement de process plusieurs threads avec les conséquences suivantes :

- Les threads partagent le même espace adressable (un trap mémoire provoqué par un thread pénalisera tout les threads du process).
- La communication entre thread est simplifiée par l'utilisation de variables globales

NT alloue un quota de temps à tous les threads des process d'un niveau de priorité donné.
Au sein d'un process on distingue la priorité des threads en 7 classes de priorités croissantes :

THREAD_PRIORITY_IDLE
THREAD_PRIORITY_LOWEST
THREAD_PRIORITY_BELOW_NORMAL
THREAD_PRIORITY_ NORMAL
THREAD_PRIORITY_ABOVE_NORMAL
THREAD_PRIORITY_HIGHEST
THREAD_PRIORITY_TIME_CRITICAL

De plus afin d'éviter (autant que possible) les risques d'étreinte fatale ou de famine, la priorité d'un thread est modifiée comme suit :

- Les threads en attente d'une E/S reçoivent une priorité supérieur (réactivité)
- Les threads en attente volontaire (mutex, sémaphores) reçoivent une priorité supérieure
- Augmentation de priorité périodique afin d'éviter les étreintes fatales
- Threads de calcul (pas dE/S) => priorité diminuée

❖ L'utilitaire « Process viewer » fourni avec VC++ permet d'afficher des informations détaillées concernant les process et threads s'exécutant sur la machine.

3 QUELQUES FONCTIONS DE BASE

Le programme ci dessous illustre l'utilisation de quelques fonctions utiles (voir aide pour les détails et autres fonctions)
Ex 3.1

```
#include <windows.h>
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
	char Chaine[20];
	HANDLE HProcess;
	SYSTEM_INFO Syst_info;
	cout << "\t\t Info systeme  " << endl;
	GetSystemInfo(&Syst_info);
	cout << " dwPageSize                  : " << Syst_info.dwPageSize << endl
	    << " lpMinimumApplicationAddress      : " << Syst_info.lpMinimumApplicationAddress << endl
		<< " lpMaximumApplicationAddress : " << Syst_info.lpMaximumApplicationAddress << endl
		<< " dwActiveProcessorMask                 : " << Syst_info.dwActiveProcessorMask << endl
		<< " dwNumberOfProcessors                  : " << Syst_info.dwNumberOfProcessors << endl
		<< " dwProcessorType            : " << Syst_info.dwProcessorType << endl
		<< " dwAllocationGranularity       : " << Syst_info.dwAllocationGranularity << endl
		<< " wProcessorLevel :              " << Syst_info.wProcessorLevel << endl
		<< " wProcessorRevision  :          " << Syst_info.wProcessorRevision << endl;
	cout << "\t\t Parametres de la ligne de commande  " << endl;
	cout << " ligne de commande :" << GetCommandLine() << endl;
	cout << "\t\t Affichage de variable d'environnement  " << endl;
	GetEnvironmentVariable("UserName", Chaine, 19);
	cout << " User : " << Chaine << endl;
	cout << "\t\t  Affichage des ID process et Thread  " << endl;
	DWORD PID = GetCurrentProcessId();
	DWORD TID = GetCurrentThreadId();
	cout << " id process : " << PID << endl;
	cout << " id thread : " << TID << endl;
	cout << "\t\t Version de l'OS " << endl;
	DWORD VersionOS;
	VersionOS = GetProcessVersion(PID);
	cout << " Version de l'OS : " << HIWORD(VersionOS) << ":" << LOWORD(VersionOS) << endl;
	cout << "\t\t Recuperer le pseudohandle  sur un process  " << endl;
	HProcess = GetCurrentProcess();
	cout << " Handle du process : " << HProcess << endl;
	cout << "\t\t Recuperer la taille memoire physique allouee au process  " << endl;
	DWORD TailleMini, TailleMaxi;
	GetProcessWorkingSetSize(HProcess, &TailleMini, &TailleMaxi);
	cout << " TailleMini : " << TailleMini << " TailleMaxi : " << TailleMaxi << endl;
	return 0;
}
```

4 LES THREADS DANS WIN 32

4.1 Créer un thread

La fonction **CreateThread()** permet de créer un Thread

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes,      // pointer to security attributes
  DWORD dwStackSize,                             // initial thread stack size
  LPTHREAD_START_ROUTINE lpStartAddress,         // pointer to thread function
  LPVOID lpParameter,                            // argument for new thread
  DWORD dwCreationFlags,                         // creation flags
  LPDWORD lpThreadId                             // pointer to receive thread ID
);
```

lpThreadAttributes	permet de préciser si le Handle du thread est héritable par un process fils NULL = non héritable
DwStackSize	taille de la pile 0 => taille par défaut du thread principal
LpStartAddress	fonction de signature : DWORD Fonction(LPVOID param)
LpParameter	adresse du paramètre passé au thread

DwCreationFlags CREATE_SUSPENDED le thread est crée endormi (réveillé par ResumeThread())
lpThreadId adresse d'un DWORD recevant l'ID du thread

Ex 4.1

```
#include <windows.h>
#include <conio.h>
#include <iostream>
using namespace std;
typedef enum { MARCHE, ARRET }T_ETAT;
WORD FoncThread(T_ETAT* p_Etat)
{
	while (*p_Etat != ARRET)
	{
		cout << " je suis le thread ... " << endl;
	}
	cout << "Fin de thread " << endl;
	return 0;
}

int main(int argc, char* argv[])
{
	DWORD Tid;   HANDLE HThread;
	T_ETAT Etat = MARCHE;
	cout << "Appuyez sur une touche pour demarrer  Nouvel appui => arret " << endl;
	_getch();
	HThread = CreateThread(NULL, 0,(LPTHREAD_START_ROUTINE)FoncThread, &Etat, 0, &Tid);
	if (HThread == NULL)
	{
		DWORD Erreur = GetLastError();
		char Msg[80];
		FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, NULL, Erreur, NULL, Msg, 80, NULL);
		cout << " erreur : " << Erreur << ": " << Msg << endl;
		exit(1);
	}
	_getch();
	Etat = ARRET;
	Sleep(1000);
	cout << "Appuyez sur une touche " << endl;
	_getch();
	return 0;
}
```

4.2 Suspendre un thread et modifier sa priorité

Les fonctions SetThreadPriority() ResumeThread() SuspendThread() permettent respectivement de fixer la priorité d'un thread, de le réveiller et de le suspendre.
Ex 4.2

```
#include <windows.h>
#include <iostream>
#include <conio.h>

using namespace std;
typedef enum { MARCHE, ARRET }T_ETAT;
DWORD  Compteur_Thread_1 = 0;
DWORD  Compteur_Thread_2 = 0;
WORD Thread_1(T_ETAT* p_Etat)
{
	while (*p_Etat != ARRET)
	{
		for (int L_Index = 0; L_Index < 40; L_Index++)
		{
			for (DWORD L_Temp = 0; L_Temp < 40000; L_Temp++);
			cout << "-";
		}
		cout << endl;
		Compteur_Thread_1++;
	}
	return 0;
}
WORD Thread_2(T_ETAT* p_Etat)
{
	while (*p_Etat != ARRET)
	{
		for (int L_Index = 0; L_Index < 40; L_Index++)
		{
			for (DWORD L_Temp = 0; L_Temp < 40000; L_Temp++);
			cout << "*";
		}
		cout << endl;
		Compteur_Thread_2++;
	}
	return 0;
}

int main(int argc, char* argv[])
{
	DWORD Tid1, Tid2;
	T_ETAT Etat = MARCHE;
	HANDLE HThread1, HThread2;

	HThread1 = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Thread_1, &Etat, CREATE_SUSPENDED, &Tid1);
	HThread2 = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Thread_2, &Etat, CREATE_SUSPENDED, &Tid2);
	cout << "Appuyez sur une touche pour démarrer " << endl;
	cout << "Nouvel appui => arreter " << endl;
	_getch();
	SetThreadPriority(HThread1, THREAD_PRIORITY_NORMAL);
	 //THREAD_PRIORITY_NORMAL
	  //THREAD_PRIORITY_TIME_CRITICAL //  THREAD_PRIORITY_ABOVE_NORMAL
	  // THREAD_PRIORITY_LOWEST //THREAD_PRIORITY_BELOW_NORMAL
	  // THREAD_PRIORITY_IDLE  // THREAD_PRIORITY_HIGHEST
	ResumeThread(HThread1);
	ResumeThread(HThread2);
	_getch();
	SuspendThread(HThread2);
	_getch();
	ResumeThread(HThread2);
	_getch();
	Etat = ARRET;
	Sleep(100);
	cout << endl << "Compteur thread 1 :" << Compteur_Thread_1 << endl;
	cout << "Compteur thread 2 :" << Compteur_Thread_2 << endl;
	cout << "Appuyez sur une touche pour quitter " << endl;
	_getch();
	return 0;
}
```

4.3 Synchronisation

Les fonctions WaitForSingleObject() WaitForMultipleObjects() permettent d'attendre

pendant un temps donné ou indéfiniment d'être signalé par un ou plusieurs objets
(mutex, évènement …)

Ex 4.3

```
// voir ex 4.2

 int main()
 {
            DWORD Tid1,Tid2,ret,ret2;
            T_ETAT Etat=MARCHE;
            HANDLE HThread[2];

            HThread[0]= CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Thread_1,&Etat,0,&Tid1);
            HThread[1]= CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Thread_2,&Etat,0,&Tid2);
            cout << "appui => arreter "<<endl;
            _getch();
            Etat = ARRET;

            WaitForMultipleObjects(2,HThread,TRUE,INFINITE);

            cout  <<"Compteur thread 1 :" <<Compteur_Thread_1 <<endl;
            cout << "Compteur thread 2 :" <<Compteur_Thread_2 <<endl;
            cout << "Appuyez sur une touche "<< endl;
            _getch();
            return 0;
 }
```

4.4 Signaux et évènements

Les fonctions CreateEvent(), SetEvent() et ResetEvent() permettent respectivement de créer, signaler et réinitialiser un événement.
Un thread pourra se mettre en attente d'un signal grâce à l'une des deux fonctions vues ci-dessus.

Ex 4.4

```
 // voir ex 4.2
…..
 HANDLE Event_Start;

 WORD Thread_1(T_ETAT* p_Etat)
 {
            WaitForSingleObject(Event_Start,INFINITE);
            while (*p_Etat != ARRET)
            {
                        //  comme ex 4.2
            }
 return 0;
 }

WORD Thread_2(T_ETAT* p_Etat)
 {
      //      comme ex 4.2
 }

suite -->
```

Ex 4.4 suite

```
int main()
{
        DWORD Tid1,Tid2;
        T_ETAT Etat=MARCHE;
        HANDLE HThread[2];
        Event_Start =CreateEvent(NULL,FALSE,FALSE,"Start");
        HThread[0]= CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Thread_1,&Etat,0,&Tid1);
        HThread[1]= CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Thread_2,&Etat,0,&Tid2);
        SetThreadPriority(HThread[0],THREAD_PRIORITY_NORMAL); //TIME_CRITICAL);
        Sleep(2000);
        SetEvent(Event_Start);
        cout << "appui => arreter "<<endl;
        _getch();
        Etat = ARRET;
        WaitForMultipleObjects(2,HThread,TRUE,INFINITE);
        cout  <<"Compteur thread 1 :" <<Compteur_Thread_1 <<endl;
        cout << "Compteur thread 2 :" <<Compteur_Thread_2 <<endl;
        cout << "Appuyez sur une touche "<< endl;
        _getch();
        return 0;
}
```

4.5 Les timers

La fonction ***timeSetEvent()*** permet de créer un timer. Celui-ci peut provoque l'exécution une fois ou périodiquement d'une fonction CALLBACK.
La fonction ***timeKillEvent()*** permet de détruire un timer.
Les fonctions ***timeBeginPeriod()*** et ***timeEndPeriod()*** permettent de fixer la résolution d'un timer dans un environnement temps reel.

```
MMRESULT timeSetEvent(
UINT uDelay,                    // delay en ms
UINT uResolution,               // résolution en ms   0 => résolution maximale
LPTIMECALLBACK lpTimeProc,      // pointeur sur fonction CALLBACK
DWORD dwUser,                   // donnée utilisateur passée à la fonction CALLBACK
UINT fuEvent                    // TIME_ONESHOT  évènement une fois au bout de uDelay ms
                                // TIME_PERIODIC évènement les uDelay ms
);
```

Ex 4.5

```
#include <windows.h>
#include <iostream>
#include <conio.h>
#include <mmsystem.h>
#pragma comment (lib, "winmm.lib")  // charge la librairie winmm

using namespace std;
typedef enum { MARCHE, ARRET }T_ETAT;
HANDLE Event_Start;
void CALLBACK On_Timer(UINT uID, UINT, DWORD param, DWORD, DWORD)
{
   cout << " uID : " << uID << " param : " << param << endl;
   SetEvent(Event_Start);
}
WORD Thread_1(T_ETAT* p_Etat)
{
   while (*p_Etat != ARRET)
   {
      WaitForSingleObject(Event_Start, INFINITE);
      cout << timeGetTime() << endl;
   }
   return 0;
}

// suite →
```

Ex 4.5 suite

```
int main()
{
	DWORD Tid;
	DWORD LeParam;
	T_ETAT Etat = MARCHE;
	HANDLE HThread;
	Event_Start = CreateEvent(NULL, FALSE, FALSE, "Start");
	HThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Thread_1, &Etat, 0, &Tid);
	LeParam = 52;
	MMRESULT Le_Timer = timeSetEvent(1000, 0, On_Timer, LeParam, TIME_PERIODIC);
	_getch();
	timeKillEvent(Le_Timer);
	Etat = ARRET;
	SetEvent(Event_Start);		// pour eviter un dead lock
	WaitForSingleObject(HThread, INFINITE);
	cout << "appui => quitter " << endl;
	_getch();
	return 0;
}
```

5 GESTION DE LA CONCURRENCE

5.1 Exclusion mutuelle: le mutex

Soit l'exemple suivant: ex 5.1.1

```
#include <iostream>
#include "conio.h"
#include "string.h"
#include "windows.h"
using namespace std;
typedef enum { MARCHE, ARRET }T_ETAT;

void Affiche(const char* p_Chaine)
{
	for (UINT L_Index = 0; L_Index < strlen(p_Chaine); L_Index++)
	{
		cout << p_Chaine[L_Index];
		for (DWORD L_Temps = 0; L_Temps < 70000; L_Temps++);
	}
	cout << endl;
}

WORD Thread_1(T_ETAT* p_Etat)
{
	while (*p_Etat != ARRET)
	{
		Affiche("Les sanglots longs de l'automne bercent mon coeur d'une langueur monotone");
	}
	return 0;
}

WORD Thread_2(T_ETAT* p_Etat)
{
	while (*p_Etat != ARRET)
	{
		Affiche("Il etait une bergere heriheron petit patapon qui gardait ses moutons ronron");
	}
	return 0;
}

int main()
{
	DWORD Tid1, Tid2;
	T_ETAT Etat = MARCHE;
	HANDLE HThread[2];
	HThread[0] = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Thread_1, &Etat, 0, &Tid1);
	HThread[1] = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Thread_2, &Etat, 0, &Tid2);
	_getch();
	Etat = ARRET;
	WaitForMultipleObjects(2, HThread, TRUE, INFINITE);
	cout << "Appuyez sur une touche " << endl;
	_getch();
	return 0;
}
```

Deux threads de même priorité accèdent concurremment à la même ressource (l'écran).
On obtient l'affichage suivant :

```
LIels  estaanigtl outnse  lboenrggse rdee  hle'raiuhteormonne  pbeetricte npta tmaopno nc oqeuuir  gda'rudnaei tl asnegsu emuoru
tmoonnso troonner
Loens
I ls aentgaliott su nleon gbse rdgee rle'a uhteorminhee rboenr cpeentti tm opna tcaopeounr  qdu'iu ngea rldaanigtu esuers
mmoonuototnosn er
Loensr osna
Inlg leottasi tl ounnges  dbee rlg'earuet ohmenrei hbeerrocne npte tmiotn  pcaoteaupro nd 'quunie  glaarndgauietu rs emso nmootuotneo
Lness  rsoannrgolno
Itls  eltoanigts  udnee  lb'earugteormen eh ebreirhceernotn  mpoent icto epura tda'puonne  qluain gguaerudra imto nsoetson em
oLuetso nssa nrgolnortosn
…..
```

On remarque que les thread se préemptant l'un l'autre , l'affichage devient anarchique.

On protège donc la ressource par un Mutex garantissant que tant qu'un process la détient, l'autre ne peut pas y accéder. Cette portion de code devient donc une ***section critique***.
La fonction **CreateMutex()** permet de créer un mutex.
La fonction ***WaitForSingleObject(LeMutex,INFINITE)*** joue le rôle de Mutex.P
Si le mutex est pris, le process exécutant cette fonction est mis en attente.
La fonction ***ReleaseMutex(LeMutex)*** joue le rôle de Mutex.V (on dit signaler le mutex).

D'ou le code de l'ex 5.1.2

```
// Voir  ex 5.1.1

HANDLE LeMutex;

void Affiche(char p_Chaine[])
{
        WaitForSingleObject(LeMutex,INFINITE);
        for (UINT L_Index=0;L_Index<strlen(p_Chaine);L_Index++)
        {
                cout <<p_Chaine[L_Index];
                for( DWORD L_Temps =0;L_Temps<70000 ; L_Temps++);
        }
        cout << endl;
        ReleaseMutex(LeMutex);
}

WORD Thread_1(T_ETAT* p_Etat)
{   // Voir  ex 5.1.1
}

WORD Thread_2(T_ETAT* p_Etat)
{       // Voir  ex 5.1.1
}

int main()
{
        DWORD Tid1,Tid2;
        T_ETAT Etat=MARCHE;
        HANDLE HThread[2];
        LeMutex =CreateMutex(NULL,FALSE,"Mutex1");
        HThread[0]= CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Thread_1,&Etat,0,&Tid1);
        HThread[1]= CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Thread_2,&Etat,0,&Tid2);
        _getch();
        Etat=ARRET;
        WaitForMultipleObjects(2,HThread,TRUE,INFINITE);
        cout << "Appuyez sur une touche "<< endl;
        _getch();
        return 0;
}
```

5.2 Définition d'une section critique

Le fonctions **InitializeCriticalSection() DeleteCriticalSection() EnterCriticalSection() (ou TryEnterCriticalSection()) et LeaveCriticalSection()** permettent de remplacer un mutex qui n'est utilisé qu'entre thread d'un même process.
Le principe est le suivant :

```
CRITICAL_SECTION  LaSectionCritique ;
 ..............
 ..............
InitializeCriticalSection(&LaSectionCritique ) ;
 ..............
 ..............
EnterCriticalSection(&LaSectionCritique ) ;   // <=> mutex.P
 ..............
 ..............
LeaveCriticalSection(&LaSectionCritique ) ;   // <=> mutex.V
..............
 ..............
DeleteCriticalSection(&LaSectionCritique ) ;
 ..............
```

5.3 Interverrouillage d'une variable globale

Lorsque deux thread partagent une données globale qu'ils sont susceptibles de modifier (incrémenter, décrémenter ou changer la valeur), il est important que celui qui a commencé une telle opération ne soit pas préempté.
Pour cela on dispose des fonctions
InterlockedIncrement() et **InterlockedDecrement() InterlockedExchange()**

5.4 Le Semaphore

Le mutex ne permet l'accès à une ressource qu'à un seul thread à la fois. La ressource est dite non partageable.
Dans le cas de ressource n-partageable on peut utiliser un sémaphore.

Pour créer un sémaphore, on dispose de la fonctions **CreateSemaphore() ,**

```
HANDLE CreateSemaphore(
 LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,        // pointeur sur les attributs de sécurité
 LONG lInitialCount,                                 // valeur initiale
 LONG lMaximumCount,                                 // valeur maximale
 LPCTSTR lpName                                      // pointeur sur le nom du sémaphore
);
```

HANDLE LeSemaphore = CreateSemaphore(NULL,3,3,NULL); crée un sémaphore non nommé initialisé à 3
OpenSemaphore() permet d'utiliser un sémaphore nommé crée par un autre process.
WaitForSingleObject(LeSemaphore,INFINITE); <=> Semaphore.P
ReleaseSemaphore(LeSemaphore,1,NULL); **<=> Semaphore.V** avec incrément de 1

Ex 5.4

```
#include <iostream>
#include <conio.h>
#include <string.h>
#include <windows.h>
#include <iomanip>
using namespace std;
typedef enum { MARCHE, ARRET }T_ETAT;
#define NB_THREADS 5
#define NB_MAX_TH_IN_RES 3

typedef struct {
		T_ETAT Etat;
		DWORD Tid;
}T_INFO_THREAD;

HANDLE LeSemaphore;
LONG Nombre_Threads = 0;   // variable globale à proteger
void Table(DWORD p_Tid)
{
		WaitForSingleObject(LeSemaphore, INFINITE);
		InterlockedIncrement(&Nombre_Threads);
		//cout <<setw(5)<< "Tid "<< p_Tid << "\t Nombre de threads utilisant la ressource  " << Nombre_Threads <<endl;
		cout << "\t Nombre de threads utilisant la ressource  " << Nombre_Threads << endl;

		InterlockedDecrement(&Nombre_Threads);
		Sleep(10);
		ReleaseSemaphore(LeSemaphore, 1, NULL);
}

WORD Thread(T_INFO_THREAD* p_Info)
{
		while (p_Info->Etat != ARRET)
		{
			Table(p_Info->Tid);
		}
		return 0;
}

int main()
{
		DWORD Tid[NB_THREADS];
		T_INFO_THREAD Info[NB_THREADS];
		HANDLE HThread[NB_THREADS];
		UINT L_Nbr_Threads = 0;
		LeSemaphore = CreateSemaphore(NULL, NB_MAX_TH_IN_RES, NB_MAX_TH_IN_RES, NULL);
		for (L_Nbr_Threads = 0; L_Nbr_Threads < NB_THREADS; L_Nbr_Threads++)
		{
			HThread[L_Nbr_Threads] = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Thread,
				&Info[L_Nbr_Threads], CREATE_SUSPENDED, &Tid[L_Nbr_Threads]);

			Info[L_Nbr_Threads].Etat = MARCHE;
			Info[L_Nbr_Threads].Tid = Tid[L_Nbr_Threads];
			cout << "tid thread " << L_Nbr_Threads << " " << Tid[L_Nbr_Threads] << endl;
		}
		_getch();
		for (L_Nbr_Threads = 0; L_Nbr_Threads < 5; L_Nbr_Threads++)
		{
			ResumeThread(HThread[L_Nbr_Threads]);
			Sleep(100);
		}

		Sleep(100 / NB_THREADS);
		for (L_Nbr_Threads = 0; L_Nbr_Threads < NB_THREADS; L_Nbr_Threads++)
		{
			Info[L_Nbr_Threads].Etat = ARRET;
		}
		WaitForMultipleObjects(NB_THREADS, HThread, TRUE, INFINITE);
		cout << "Appuyez sur une touche " << endl;
		_getch();
		return 0;
}
```

6 LES PROCESS

Comme indiqué dans l'introduction NT exécute l'image chargée en mémoire d'un process. Celui-ci dispose de 2Go.

6.1 Création d'un process

La fonction CreateProcess() présentée ci-dessous permet de créer un process :

```
BOOL CreateProcess(
 LPCTSTR lpApplicationName,                    // pointer to name of executable module
 LPTSTR lpCommandLine,                         // pointer to command line string
 LPSECURITY_ATTRIBUTES lpProcessAttributes,    // process security attributes
 LPSECURITY_ATTRIBUTES lpThreadAttributes,     // thread security attributes
 BOOL bInheritHandles,                         // handle inheritance flag
 DWORD dwCreationFlags,                        // creation flags
 LPVOID lpEnvironment,                         // pointer to new environment block
 LPCTSTR lpCurrentDirectory,                   // pointer to current directory name
 LPSTARTUPINFO lpStartupInfo,                  // pointer to STARTUPINFO
 LPPROCESS_INFORMATION lpProcessInformation    // pointer to PROCESS_INFORMATION
);
```

Voir l'aide pour les détails.

On crée donc deux modules exécutables : le père et le fils
Ex 6.1.fils

```
#include "windows.h"
#include "winbase.h"
#include <iostream>
#include "conio.h"
using namespace std;
void Erreur()
{
        char Msg[80];
        DWORD Cause = GetLastError();
        FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, NULL, Cause, NULL, Msg, 80, NULL);
        cout << " Erreur : " << Cause << " : " << Msg << endl;
        _getch();
}

int main()
{
        DWORD LeFichier;
        DWORD NbrEcrit;
        BOOL success;
        cout << "  Je suis le fils" << endl;
        cout << " Ligne de commande " << GetCommandLine() << endl;
        cout << "Donnez le Handle du fichier svp  ";
        cin >> LeFichier;
        success = WriteFile((HANDLE)LeFichier, "Je suis le fils", 15, &NbrEcrit, NULL);
        if (success != TRUE)
        {
                Erreur();
                _getch();
                exit(1);
        }
        cout << " Ecriture fichier OK  " << endl << "appuyez sur une touche pour quitter " << endl;
        _getch();
        return 0;
}
```

Ex 6.1 père

```
#include "windows.h"
#include "winbase.h"
#include <iostream>
#include "conio.h"
using namespace std;
void Erreur()
{
	char Msg[80];
	DWORD Cause = GetLastError();
	FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, NULL, Cause, NULL, Msg, 80, NULL);
	cout << " Erreur : " << Cause << " : " << Msg << endl;
	_getch();
}

int main()
{
	PROCESS_INFORMATION Process_Info;
	STARTUPINFO Process_Startup;
	SECURITY_ATTRIBUTES Securite;
	DWORD NbrEcrit;
	cout << "  Je suis le pere " << endl;
	memset(&Process_Startup, 0, sizeof(STARTUPINFO));
	Process_Startup.cb = sizeof(STARTUPINFO);

	Securite.nLength = sizeof(SECURITY_ATTRIBUTES);
	Securite.lpSecurityDescriptor = NULL;
	Securite.bInheritHandle = TRUE;

	HANDLE LeFichier = CreateFile("C:\\temp\\test.txt", GENERIC_READ | GENERIC_WRITE, 0, &Securite,
		CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
	BOOL success = CreateProcess("c:\\temp\\fils.exe",		// pointer sur le fichier exécutable
				NULL,
		NULL,
		NULL,
		NULL,				//flag pour héritage du handle   CREATE_NEW_CONSOLE, //
flags de creation
		TRUE,
		CREATE_NEW_CONSOLE,
		NULL,
		NULL,
		&Process_Startup,		// pointeur sur STARTUPINFO
		&Process_Info);			// pointeur sur PROCESS_INFORMATION
	if (0 == success)
	{
		Erreur();
		exit(1);
	}

	WriteFile(LeFichier, "BONJOUR", 7, &NbrEcrit, NULL);
	cout << "J ai ecrit dans le fichier ayant le  HANDLE:  : " << (DWORD)LeFichier << endl;
	CloseHandle(Process_Info.hProcess);
	cout << "appuyez sur une touche pour quitter " << endl;
	_getch();
	CloseHandle(LeFichier);
	return 0;
}
```

Créer le fichier fils.exe et copiez le dans un répertoire c:\temp
Créez père.exe et lancez le.

6.2 Communication inter process : Le pipe nommé

Un pipe nommé est une section de mémoire partagée que des process utilisent pour communiquer.
Le process qui crée le pipe est appelé serveur, celui qui s'y connecte est appelé client.
Un process écrit des données dans le pipe, un autre process lit ces informations dans le pipe.
Le serveur crée un pipe nommé grâce à la fonction CreateNamedPipe()
Le client se connecte au pipe nommé grâce à la fonction ConnectNamedPipe()

```
HANDLE CreateNamedPipe(
  LPCTSTR lpName,                          // pointer to pipe name
  DWORD dwOpenMode,                        // pipe open mode
  DWORD dwPipeMode,                        // pipe-specific modes
  DWORD nMaxInstances,                     // maximum number of instances
  DWORD nOutBufferSize,                    // output buffer size, in bytes
  DWORD nInBufferSize,                     // input buffer size, in bytes
  DWORD nDefaultTimeOut,                   // time-out time, in milliseconds
  LPSECURITY_ATTRIBUTES lpSecurityAttributes  // pointer to security attributes
);
```

```
BOOL ConnectNamedPipe(
  HANDLE hNamedPipe,                       // handle to named pipe to connect
  LPOVERLAPPED lpOverlapped                // pointer to overlapped structure
);
```

Ex 6.2 fils_pipe

```
#include "windows.h"
#include "winbase.h"
#include <iostream>
#include "conio.h"
using namespace std;
void Erreur()
{
        char Msg[80];
        DWORD Cause = GetLastError();
        FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, NULL, Cause, NULL, Msg, 80, NULL);
        cout << " Erreur : " << Cause << " : " << Msg << endl;
        _getch();
}
int main()
{
        bool L_success = false;
        int L_ret = 0;
        HANDLE hLePipe;
        char CaracEcrit ='R', CaracLu ='X';
        DWORD NbrEcrit, NbrLu;
        cout << "  Je suis le fils_pipe" << endl;
        hLePipe = CreateFile("\\\\.\\pipe\\MonPipe", GENERIC_READ | GENERIC_WRITE,
                0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
        if (hLePipe == INVALID_HANDLE_VALUE)
        {
                cout << "  Erreur ouverture pipe pipe" << endl;
                Erreur();
                L_ret = _getch();
                CloseHandle(hLePipe);
                exit(1);
        }
        if (hLePipe == INVALID_HANDLE_VALUE)
        {
                cout << "  Erreur lien au pipe" << endl;
                Erreur();
                L_ret = _getch();
                CloseHandle(hLePipe);
                exit(1);
        }
```

fils_pipe (suite)

```
		cout << " Lien au pipe OK" << endl;
		do {
			cout << "Lecture dans le pipe" << endl;
			L_success = ReadFile(hLePipe, &CaracLu, 1, &NbrLu, NULL);
			cout << "j'ai lu  "<< NbrLu << " caracteres : " << CaracLu << endl;
			Sleep(1000);
			cout << "Ecriture dans le pipe "  << endl;
			L_success = WriteFile(hLePipe, &CaracEcrit, 1, &NbrEcrit, NULL);
			cout << "j'ai ecrit  " << NbrEcrit << " caracteres : " << endl;
			} while (CaracLu != 'D');
			cout << " fermeture du pipe " << endl;
		CloseHandle(hLePipe);
		cout << "appuyez sur une touche pour quitter " << endl;
		L_ret = _getch();
		return 0;
}
```

Ex 6.2 pere_pipe

```
#include "windows.h"
#include "winbase.h"
#include <iostream>
#include "conio.h"
using namespace std;

void Erreur()
{
		int L_ret = 0;
		char Msg[80];
		DWORD Cause = GetLastError();
		FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, NULL, Cause, NULL, Msg, 80, NULL);
		cout << " Erreur : " << Cause << " : " << Msg << endl;
		L_ret = _getch();
}


int main()
{
		bool L_success = false;
		int L_ret = 0;
		PROCESS_INFORMATION Process_Info;
		STARTUPINFO Process_Startup;
		HANDLE hLePipe;
		char CaracEcrit='A',CaracLu='X';
		DWORD NbrLu, NbrEcrit;


		hLePipe = CreateNamedPipe("\\\\.\\pipe\\MonPipe", PIPE_ACCESS_DUPLEX,
			PIPE_TYPE_BYTE | PIPE_READMODE_BYTE | PIPE_WAIT,
			PIPE_UNLIMITED_INSTANCES, 10, 10, 100, NULL);
		if (hLePipe == INVALID_HANDLE_VALUE)
		{
			cout << "Erreur creation Pipe " << endl;
			Erreur();
			exit(1);
		}
		cout << "  Je suis le pere  pipe cree handle : " << hLePipe << endl;
		memset(&Process_Startup, 0, sizeof(STARTUPINFO));
		Process_Startup.cb = sizeof(STARTUPINFO);
		ZeroMemory(&Process_Info, sizeof(Process_Info));
		L_success = CreateProcess("c:\\temp\\fils_pipe.exe", NULL, NULL, NULL, TRUE,
			CREATE_NEW_CONSOLE, NULL, NULL, &Process_Startup, &Process_Info);
		if (FALSE == L_success)
		{
			cout << "Erreur creation Process fils " << endl;
			Erreur();
			exit(1);
		}

		ConnectNamedPipe(hLePipe, NULL);
		cout << "Lien pipe OK " << endl;
		Sleep(2000);
```

pere_pipe (suite)

```
do {
                        cout << "Ecriture dans le pipe : " << CaracEcrit << endl;
                        L_success = WriteFile(hLePipe, &CaracEcrit, 1, &NbrEcrit, NULL);
                        Sleep(1000);
                        cout << "Lecture dans le pipe" << endl;
                        L_success = ReadFile(hLePipe, &CaracLu, 1, &NbrLu, NULL);
                        cout << "j'ai lu " << CaracLu << endl;
                        CaracEcrit = CaracEcrit + 1;
            } while (CaracEcrit != 'E');
            WaitForSingleObject(Process_Info.hProcess, INFINITE);
            CloseHandle(Process_Info.hProcess);
            cout << "appuyez sur une touche pour quitter " << endl;
            L_ret = _getch();
            return 0;
}
```

Créer le fichier fils_pipe.exe et copiez le dans un répertoire c:\temp
Créez pere_pipe.exe et lancez le.

6.3 Pipe anonyme

La fonction **CreatePipe ()** permet de créer un pipe anonyme .
CreatePipe crée aussi deux handle (un pour la lecture et l'autre pour l'écriture) que les process utilisent pour écrire et lire dans le pipe anonyme grâce aux fonctions **ReadFile et WriteFile.**

```
BOOL CreatePipe(
  PHANDLE hReadPipe,                    // pointer to read handle
  PHANDLE hWritePipe,                   // pointer to write handle
  LPSECURITY_ATTRIBUTES lpPipeAttributes,  // pointer to security attributes
  DWORD nSize                           // pipe size
);
```

Un pipe anonyme ne peut pas être utilisé à travers un réseau ni entre des process non apparentés.

7 LES THREADS DANS MFC

Dans visual studio 2008, on peut créer deux sortes de threads : les threads de travail (worker threads) et les threads d'interface utilisateur.

Un thread de travail exécute une fonction (comme lors de l'utilisation de createThread() vu en début de cours). Il ne peut pas interagir avec les objet PFC et donc recevoir des messages utilisateur via des boites de dialogue par exemple.
Les threads d'interface utilisateurs″ permettent ces interactions. Ce sont des objets instances d'une classe dérivée de la classe MFC CWinThread.

Nous allons créer une application multithread basée sur la classe CWinThread fournie par MFC.
Nous lancerons plusieurs threads incrémentant un entier et l'affichant dans leur propre fenêtre. Des boutons permettrons de lancer le thread, de le suspende et de le terminer et des zones d'édition afficherons la valeur d'un compteur et le nombre de fois où le thread à été suspendu .

7.1 Manipulation des threads avec MFC.

7.1.1 Création d'un thread

Pour créer un thread qui s'exécutera dans l'espace mémoire du thread appelant il existe deux solutions :

Utilisez **AfxBeginThread** pour créer un objet thread et l'exécuter en une opération

Utilisez **CreateThread** si vous voulez réutiliser l'objet thread afin d'effectuer de multiples créations, terminaisons.

L'attribut public CWinThrread::m_bAutoDelete de type booléen spécifie si l'objet thread est détruit automatiquement (true) ou non (false) à sa terminaison.

7.1.2 Démarrer et suspendre un thread.

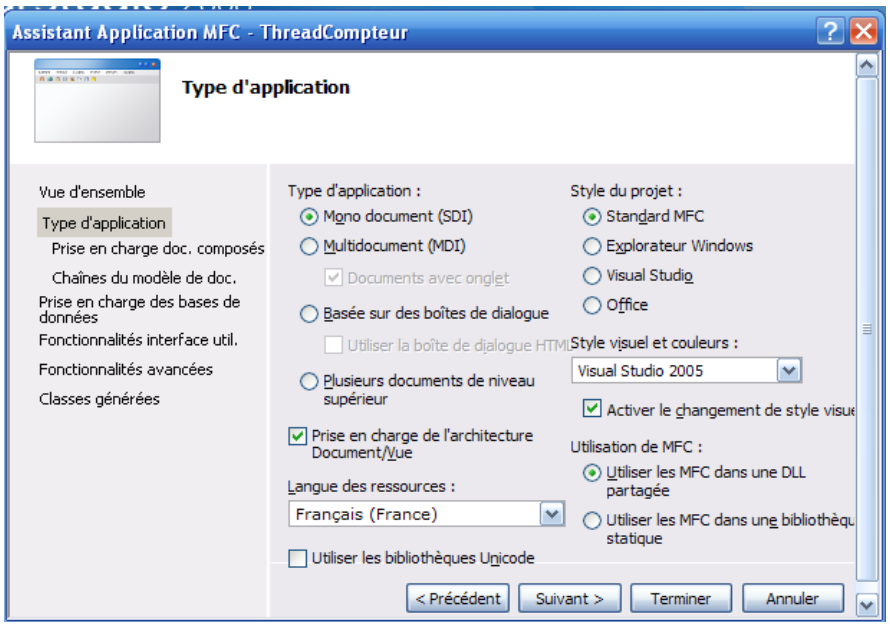
Utilisez les méthodes `CWinThread::ResumeThread` et `CWinThread::SuspendThread`

7.1.3 La fonction de traitement attachée au thread interface utilisateur

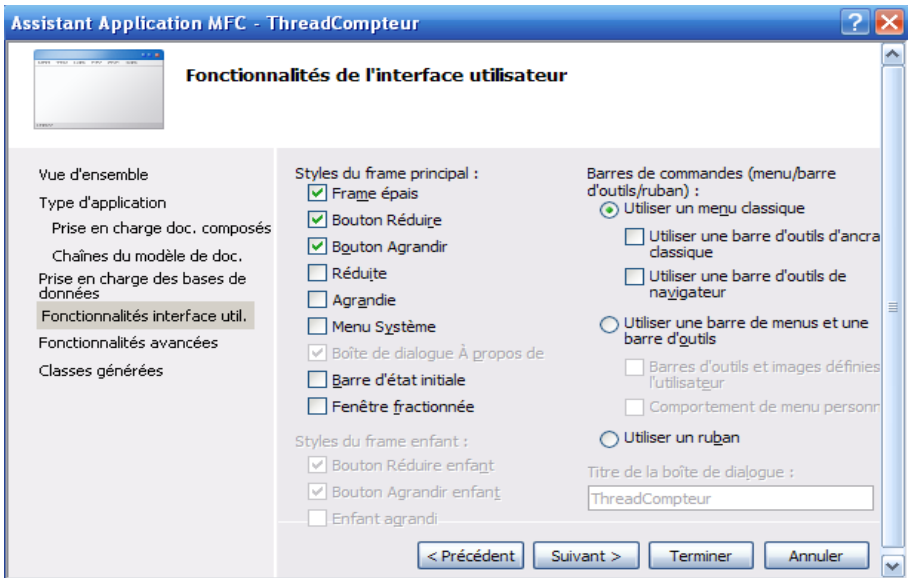
On surcharge la méthode `CWinThread::Run`

7.2 Création de notre application

Créez un projet nommée prjThreadCompteur de type " application MFC " configurer la fenêtre type d'application comme suit



cliquez sur suivant jusqu'à la fenêtre " fonctionnalité de l'interface utilisateur



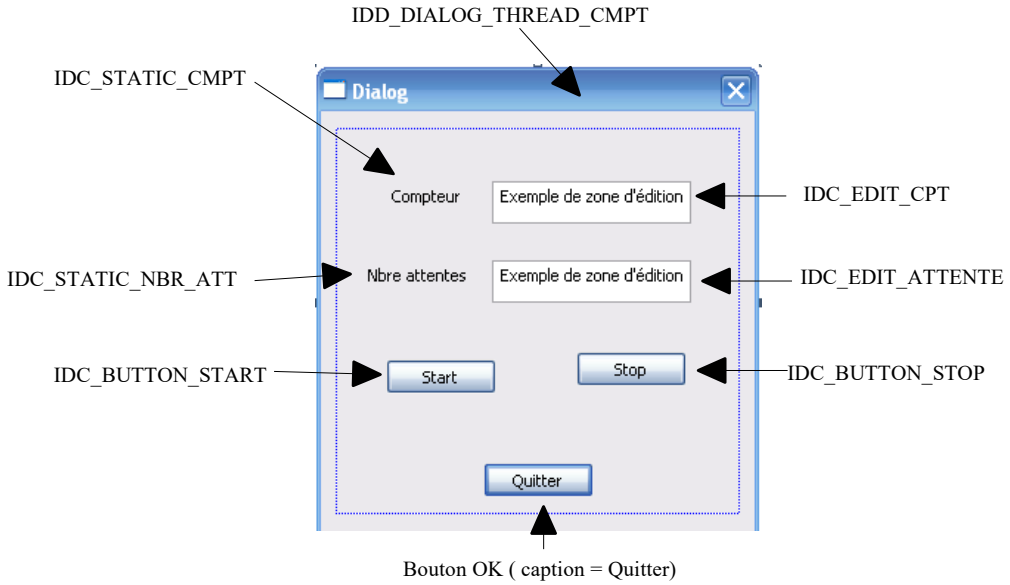
cliquez sur suivant .
dans la fenêtre " fonctionnalités avancées ", décochez toutes les options

cliquez sur terminer et lancez votre application.

7.3 Création de la boite de dialogue

Créez une boite de dialogue comme définie ci dessous et la classe CThreadDlg héritant de CDialog correspondante.

Cette boite permattra d'afficher la valeur du compteur du threadet le nombre de mises en attente . Les boutons *start* et *stop* permettront de lancer ou de suspendre le thread.



7.4 Le thread compteur

A l'aide de l'outils d'ajout de classes , ajoutez une classe **CCompteurThread** qui hérite de **CWinThread**.

On remarque qu'un thread possède (comme une application) les méthodes **virtual BOOL InitInstance()**; et **virtual int ExitInstance()**;

7.4.1 Attacher le thread compteur et sa boite de dialogue

Il faut que le thread *crée* la boite de dialogue qui affichera la valeur de son compteur ; On ajoute donc dans la classe **CCompteurThread** un attribut protected
CThreadDlg * pThreadDlg;

De plus cette boite de dialogue devra s'afficher dans la fenêtre principale . Il faut donc que le thread *connaisse* la fenêtre principale (la MainFrame) afin de créer sa boite de dialogue en lui passant le pointeur sur la fenêtre principale.
On ajoute donc dans la classe **CCompteurThread** un attribut protected
CMainFrame* laFenetrePrincipale;
en fin la boite de dialogue doit *connaître* le thread compteur.

On ajoute donc à la classe CThreadDlg un attribut protected
CCompteurThread * pMonThread;
et l'accesseur
void PrendPourThread(CCompteurThread * P_ptrLeThread);

```
void CThreadDlg::PrendPourThread(CCompteurThread * P_ptrLeThread)
{
        pMonThread = P_ptrLeThread;
}
```

On ajoute enfin à **CCompteurThread** la fonction membre publique
void CCompteurThread::CreerBoiteDeDialogue(CMainFrame* P_MainFrm) qui réalise cette connaissance mutuelle et affiche la boite de dialogue de façon non modale afin que l'on puisse encore agir sur le menu de l'applicaiton.

```
void CCompteurThread::CreerBoiteDeDialogue(CMainFrame*  P_MainFrm)
{
        laFenetrePrincipale = P_MainFrm;
        pThreadDlg=new CThreadDlg((CWnd*)laFenetrePrincipale);
        pThreadDlg->PrendPourThread((CCompteurThread *)this);
        pThreadDlg->Create(IDD_DIALOG_THREAD_CMPT,(CWnd*)laFenetrePrincipale);
        pThreadDlg->ShowWindow(SW_SHOWNORMAL);
}
```

7.4.2 Création du thread

La création du thread sera réalisée lorsque l'on choisit le menu Fichier -> Nouveau.
Comme le thread doit connaître la fenêtre principale pour l'indiquer à sa boite de dialogue , on implémentera la méthode **OnFileNew** dans la classe **CMainFrame** ce qui permettra d'utiliser le pointeur this.

Ouvrez le gestionnaire de ressource pour ajouter un gestionnaire d'événement **OnFileNew** dans la classe **CMainFrame** en réponse au click sur le menu Fichier -> Nouveau

```
void CMainFrame::OnFileNew()
{
    // TODO : ajoutez ici le code de votre gestionnaire de commande
    CCompteurThread * m_pCmptThread;
    m_pCmptThread = (CCompteurThread *) AfxBeginThread(RUNTIME_CLASS (CCompteurThread),
                        THREAD_PRIORITY_NORMAL,0,CREATE_SUSPENDED);
    m_pCmptThread ->CreerBoiteDeDialogue(this);
}
```

Il reste à y créer le thread compteur qui créera sa boite de dialogue.

La fonction AfxGeginThread reçoit en paramètre la classe de l'objet thread à créer t les flag de création (priorité normal, ayant une pile de la même taille que le thread appelant et initialement suspendu)

On indique alors au thread crée que son propriétaire et l'objet fenêtre principale (this) Il pourra ainsi recevoir et poster des messages.

7.4.3 La fonction compteur du thread

Nous voulons que le thread incrémente un compteur;

Ajoutons à la classe **CCompteurThread** un attribut protected **Cmpt** de type int.

Et surchargeons la méthode publique **virtual int Run();** qui incrémentera le **Cmpt** tant que le thread n'est pas terminé, ce qui est signalé par un attribut publique booléen **fin** que nous ajoutons également à la classe **CcompteurThread.**

La donnée membre **m_bAutoDelete** initialisée à false permet de faire en sorte que le thread ne soit pas détruit automatiquement lors de sa terminaison.

Il reste à initialiser les attributs du thread compteur dans le constructeur et à incrémenter compteur dans la méthode run tant que fin est différent de false.

```
CCompteurThread::CCompteurThread()
{
	Cmpt=100;
	fin = FALSE;
	m_bAutoDelete=FALSE;
}

int CCompteurThread::Run()
{
	// TODO: Add your specialized code here and/or call the base class
	while (!fin)
	{
			Cmpt ++;
			Sleep(250);
	};
	return 0;
}
```

7.5 Contrôler le thread depuis la boite de dialogue

7.5.1 Lancement du thread

Lorsque l'on clique sur le bouton *start* on va

- (re)démarrer le thread (qui est crée initialement suspendu) en appelant **ResumeThread()**;
- Positionner un flag pour se rappeler que le thread est démarré . Ce flag est implémenté par un attribut protected **Actif** de type **bool** initialisé à **false**;

- Mettre à jour le nombre d'attente avec la valeur de retour de **ResumeThread()**; (mémorisée dans un attribut protected **NbrSuspend** de type **int** initialisé à **0**
- Mettre à jour l'affichage en indiquant l'état du thread dans la barre de titre de la boite de dialogue.

On ajoute donc le gestionnaire d'événement correspondant au click sur le bouton start :

```
void CThreadDlg::OnBnClickedButtonStart()
{
	// TODO : ajoutez ici le code de votre gestionnaire de notification de contrôle
	NbrSuspend = pMonThread->ResumeThread();
	SetDlgItemInt(IDC_EDIT_ATTENTE,NbrSuspend);
	Actif = true;
	switch(NbrSuspend)
	{
	case 1:
			SetWindowText(" Le thread a ete remis en route");
			break;
	case 0:
			SetWindowText("Le thread etait déja en route ");
			break;
	default:
		SetWindowText("Le thread est suspendu ");
	}
}
```

7.5.2 Suspendre le thread

On ajoute de même un gestionnaire d'événement pour le bouton stop :

```
void CThreadDlg::OnBnClickedButtonStop()
{
	// TODO : ajoutez ici le code de votre gestionnaire de notification de contrôle
		NbrSuspend = pMonThread->SuspendThread();
	SetDlgItemInt(IDC_EDIT_ATTENTE,NbrSuspend);
	Actif = false;
	if(NbrSuspend >= 0)
			SetWindowText("Le thread est suspendu ");
}
```

7.5.3 Afficher la valeur du compteur

Il reste à afficher la valeur du compteur à chaque incrémentation

```
int CCompteurThread::Run()
{
	// TODO: Add your specialized code here and/or call the base class
	while (!fin)
	{
		Cmpt ++;
		pThreadDlg->SetDlgItemInt(IDC_EDIT_CPT,Cmpt);
		Sleep(250);
	};
	return 0;
}
```

7.6 Terminaison des threads

Lorsqu'on clique sur le bouton quitter (OK) ou lorsqu'on ferme la boite de dialogue il faut terminer le thread.

On surcharge donc la méthode OnClose et on ajoute un gestionnaire d'événement lorsque l'on clique sur Quitter (OnBnClickedOk) afin de terminer le thread proprement et de le signaler à l'utilisateur.

```
void CThreadDlg::OnClose()
{
	// TODO : ajoutez ici le code de votre gestionnaire de messages et/ou les paramètres par défaut des appels
	int L_cmpt=0;
	DWORD dwStatus;

	if (Actif==FALSE)
	{
			pMonThread->ResumeThread();
			Actif = true;
	}

	if (pMonThread!= NULL)
	{
			VERIFY(::GetExitCodeThread(pMonThread->m_hThread,&dwStatus));

			if (dwStatus == STILL_ACTIVE)
			{
				L_cmpt ++;
				pMonThread->fin = true;

			}
			else
			{
				delete pMonThread;
				pMonThread=NULL;
				MessageBox( "thread TERMINE");
			}
	}
	if (L_cmpt == 0)
			CDialog::OnClose();
	else
			PostMessage(WM_CLOSE,0,0L);
	CDialog::OnClose();
}

void CThreadDlg::OnBnClickedOk()
{
	// TODO : ajoutez ici le code de votre gestionnaire de notification de contrôle
	PostMessage(WM_CLOSE,0,0L);
	OnOK();
}
```