Introduction aux thread sous Win32

Quentin Pouplard

16 avril 2001

Résumé

Cette article est une introduction (très basique) au thread sous Win32. Normalement vous devriez être capable de créer des threads et de les synchroniser un minimum. Cette article n'a pas la prétention d'être une référence... vous aurez besoin de MSDN pour appliquer les quelques concepts présentés dans cette article.

Table des matières

1 Introduction

Bon ben j'avais envie d'écrire quelque chose pour ce site, je me suis donc décidé à écrire mon premier article... et comme j'ai voulu faire quelque chose d'intéressant, j'ai choisi de parler (un peu, ce n'est qu'une intro) des threads de Windows [1]. Les threads ne sont pas (évidemment) un concept propre à Win32, mais elles présentent certaines particularité sous Win32, je parle uniquement des threads sous Win32 dans cette article, mais la plupart des choses que je vais dire sont valable dans un environement Unix par exemple.

1.1 Qu'est ce qu'un thread ?

Bon, vous savez que Windows peut exécuter plusieurs programmes (plusieurs tâche - multitask) en même temps (ce qui est d'ailleurs plutôt théorique sous Windows 95 et 98, mais c'est marqué dans la pub (y'a aussi marqué stable dans la pub) enfin soit...), et bien les threads sont des

[1] dénomé Win32 dans cet article, car les threads ne sont vraiment accessibles que sur un environnement 32 bit, càd : Windows 95, 98, NT, 2000 et le futur Windows XP.

sous-tâche exécuter dans les programmes, autrement dit, le même programme peut exécuter deux choses en même temps... (plusieurs thread - multithread, évidemment, ce n'est qu'une illusion, on a toujours qu'un cpu qui ne fait qu'une seule chose à la fois). Cela permet par exemple d'afficher les images contenues dans un dossier sous forme de petite vignettes, tout en laissant l'utilisateur changer de dossier... (tiens ça aurait été un bon exemple de code... mais vaut mieux pas que je fasse du MFC, et c'est preque inhumain de ne pas employer MFC pour une telle application, tanpis...). C'est d'ailleurs exactement ce que fais l'explorateur de Win32. Il faut également savoir que tous programmes contient au moins une thread (sinon ce ne serait pas un programme...), elle est crée par Windows, et terminer lors de la fermeture du programme. Cette thread est appelé 'thread principale'...(original non...on parle aussi de thread primaire) Note : Windows alloue le processeur aux thread selon leur priorités (que vous définissez et changer lors de la création de la thread ou avec *SetThreadPriority*). En fait il s'en fout pas mal de savoir si une application a 20 threads ou 1, il répartit les ressources processeurs globalement (càd, si il y'a 45 threads dans le système ayant la même priorité (utopique, mais bon) elles auront chacune 100/45 % du processeur (si elles ont toutes besoin du processeur et moins ce que le kernel demande... soit quelques %), même si 20 de ces 45 threads font parties du même process (et donc du même programme)... Ca c'est pour le principe, dans la réalité vrai, l'algo de scheduling est un peu plus pénible que cela... mais ce n'est pas le but de cette article (et à vrai dire, je n'ai pas les compétences de faire un bon article la dessus ;-)

1.2 Pourquoi les threads ?

Si vous avez déjà été confronté au problème d'exécuter plusieurs choses en même temps (un long calcul avec un bouton "annuler"...) vous savez sans doute que l'on peut éviter les threads en vérifiant l'existance de message dans la pile de message durant le calcul, mais cela revient à décomposer le calcul en petite étape... et cela pose plusieurs problème : tout d'abord vous devez décomposer le calcul pour appeler l'API *GetMessage* , ce qui à tendance à bousiller la pile, à modifier les régistres et donc à vous demander d'abandoner l'espoir d'une optimisation idéale... Ensuite cela ne donne pas une évolution régulière de votre calcul (selon le nombre de message à traiter...) et cela devient vite ingérable, faites le test... à moins d'utiliser une architecture C++ super bien pensée, cela devient vite inhumain...

1.3 Les différent type de thread.

On peut séparer les threads Windows en deux types distinct, les *thread de service* et les *thread d'interface*, les thread de service n'ont pas de pompe à message, n'affiche rien à l'écran (mais peuvent demander à la thread principale de le faire...), en bref, elles n'ont pas besoin de fenêtre pour exister... il s'agit en fait d'une simple fonction à laquelle, on passe un paramètre (de type *DWORD* (32bit), autrement dit on va lui passer un pointeur vers notre structure qui contiendra tous ce dont on aura besoin...), en gros c'est comme un fonction main(...)[2]. (ça veut dire que vous pourrez virtuellement faire tous ce que vous voudrez mais n'ayant pas de pompes à messages, ben vous ne pourrez pas utiliser toute l'api de Windows) Les threads d'interfaces, elles ont leur pompe à message, elles ne sont pratiquement pas utiliser et les utiliser revient souvent à créer une application dans une application. Un exemple de thread d'interface : L'explorateur de Windows, en fait si vous aller dans le Gestionnaires de Tâche (NT ou 2000) ou que vous appuyez sur Ctrl+Alt+Del (sous Win9x) vous ne verrez jamais qu'un seul explorer.exe lancer...(même si vous avez 10 fenêtres de l'exploration lancée) cela s'explique par le fait que les différentes fenêtre de l'explorateur sont en faite des threads du même process. Pourquoi cela ? et bien ça a l'avantage, que comme toutes les threads se partagent le même espace de mémoire... les nombreuses resources consommes par l'explorateur n'existe qu'une seule fois en mémoire... (les icônes, les informations sur les fichiers, les caches... etc...) mais ça à le désavantage que si une fenêtre de l'explorateur

[2] WinMain dans un environnement graphique Win32

plante, toute les fenêtres seront tués... (et si vous êtes sous 9x, vous avez 9 chances sur 10 que tous soit tué... alors vive le bouton reset)

2 La création des threads.

Bon c'est décidé on va créer des threads, (rien que pour pouvoir faire le malin sur #codefr) Pour créer une thread, cela n'est en fait pas vraiment compliqué, il suffit d'appeler l'API *CreateThread* , en allant faire un petit tour dans MSDN [3], vous apprendrez qu'il faut 6 paramètres, on va élaguer cela :

```
1  HANDLE CreateThread(
2    LPSECURITY_ATTRIBUTES lpThreadAttributes,
3    DWORD dwStackSize,
4    LPTHREAD_START_ROUTINE lpStartAddress,
5    LPVOID lpParameter,
6    DWORD dwCreationFlags,
7    LPDWORD lpThreadId
8  );
```

- le premier, un pointeur vers *SECURITY_ATTRIBUTES* , vous programmez sous Win9x, tant mieux, laisser tomber cela... (passer *NULL* à la fonction), vous programmez sur et pour Windows NT, à moins de devoir faire une application pouvant s'exécuter dans plusieurs process et devant partager les handles de thread... (et dans ce cas, vous en avez sans doute plus que moi...) laisser tomber, en gros passer *NULL* pour ce paramètres.
- Le second (déjà plus interressant) : un *DWORD* (*dwStackSize*), définissant la taille de la pile pour le thread... vous vous demander ce qu'il faut mettre comme taille... normal vous le saurez une fois la thread implémenter... le plus simple est de passer *NULL* , de cette façon Win32 prendra une valeur par défaut... (qui est bien dans la majorités des cas... et bon vu le prix de la mémoire, y'a plus grand intêret à minimiser la taille de la pile... quelques Ko sur des milliers, c'est rien... là je sens que je vais me faire taper sur les doigts ;-)
- Ensuite vous lui donnerez le paramètre que vous allez passer à votre thread... (un truc sur 32bit, tiens bizarre, c'est la taille d'un pointeur ;-)
- Et puis (enfin) la fonction qui correspondra à votre thread (à l'image du main() d'un programme dos, cette fonction est appelé, quand elle se termine, la thread est finie... voir plus bas, pour la déclaration de cette fonction)
- Le suivant signale si la thread est lancé au début du programme ou pas (*CREATE_SUSPENDED* si la thread est suspendue au démarrage... utilisé *ResumeThread(...)* pour la lancer et 0 pour la lancer dès sa création).
- Et enfin un pointeur vers un *DWORD* qui contiendra le 'thread identifier' (un numéro identifiant le thread de façon unique...). Attention : si vous n'avez rien à faire de ce numéro... ne passer pas *NULL* créer un *DWORD* , donner son addresse, car Windows 9x, semble avoir des problèmes si l'on passe *NULL* pour ce paramètres... (vous me croyez pas ? essayer, mais n'oubliez pas de sauver tous vos documents ouvert avant de... planter)

La procédure de la thread est tout simplement :

```
9  DWORD WINAPI ThreadProc( LPVOID lpParameter );
10 // (ou vous mettez ce que vous voulez comme nom...)
```

note: T

[3] MSDN : si vous l'avez pas, il vous le faut sinon laisser tomber la prog Windows... en attendant, vous pouvez aller voir sur le site msdn.microsoft.com, tout y est mais... online... si vous avez Visual C++ 5.0 c'est dans la doc qu'il faut regarder (qui est en fait une partie de MSDN)

iens cette fonction renvoie un *DWORD* ! et bien vous aller pouvoir connaitre ce *DWORD* en appelant : *GetExitCodeThread(...)* , vous devrez lui passer le *HANDLE* renvoyer par *CreateThread* ...

2.1 Un petit exemple

L'exemple suivant est un exemple basique de création d'une thread, pour le compiler créer un nouveau projet "console" et copier/coller le texte, après avoir insérer les headers nécessaires[4].

```
11 DWORD WINAPI Thread1(LPVOID lpParameter)
12 {
13 printf("Ceci est ma thread\n");
14 }
15
16 int main()
17 {
18 DWORD threadID;
19 CreateThread(NULL, 0, Thread1, NULL, 0, &threadID);
20
21 Sleep(1000);
22
23 printf("fin du programme de test\n");
24 return 0;
25 }
```

Que fait ce programme ? Il ne fait qu'afficher deux lignes de textes dans la console. Rien de très compliqué, la seule chose qui peut paraitre bizarre : le "*Sleep(1000) ;* ", que fait-il là ? Il faut alors se rappeler quand Win32 termine un programme : le programme est considéré comme fini... quand le premier thread (le thread principal) est terminé, autrement dit : sans le délai d'attente, le Thread1 pourrait ne jamais être exécuté, la fin du main marquant la fin du thread principal.

Passons maintenant à un exemple un peu plus compliqué : la création de plusieurs threads, et l'utilisation du paramètre de la fonction de thread :

```
26 DWORD WINAPI Thread1(int param)
27 {
28     printf("Ceci est ma thread n°%i\n", param);
29 }
30
31 int main()
32 {
33     DWORD threadID;
34     CreateThread(NULL, 0, Thread1, 1, 0, &threadID);
35     CreateThread(NULL, 0, Thread1, 2, 0, &threadID);
36     CreateThread(NULL, 0, Thread1, 3, 0, &threadID);
37     CreateThread(NULL, 0, Thread1, 4, 0, &threadID);
38
39     Sleep(1000);
40
41     printf("fin du programme de test\n");
42     return 0;
43 }
```

Voilà, ce programme crée 4 threads... toutes les mêmes, et les exécute...

[4] Nous ne sommes pas ici pour apprendre le C !

On peut alors se poser plusieurs questions : comment éviter le "*Sleep(1000) ;* ", comment dire à une thread d'attendre une autre ? (par exemple, imaginons un programme lancant deux thread : une chargeant un fichier, et l'autre affichant le contenu de ce fichier, et ce de façon à laisser l'interface graphique disponible à l'utilisateur... il faudrait que le thread d'affichage attende le thread de chargement...)

3 Les bases de la synchronisation

Cette section va vous donner quelques moyens pour synchroniser les threads, normalement, vous ne devriez pas en avoir besoin d'autre...

3.1 Les events

Nous en sommes resté au problème de faire attendre la thread... la solution qui saute aux yeux, définir une variable globale, contenant un état : 1 ou 0... et pour faire attendre le thread, on ferait quelque chose du genre :

```
44 while(ma_variable != 1);
45 // la on exécute le code du thread
```

Mais cela n'est pas à faire ! Pourquoi ? tous simplement parce qu'une boucle consomme du temps cpu, temps qui pourrait être beaucoup mieux utilisé ailleurs... Il faut donc chercher autre chose... mais pourtant le principe est bon, en fait ce qu'il faudrait c'est remplacer la boucle par autre chose, un appel Win32, qui aurait le même rôle, mais qui serait correctement géré par le kernel afin de ne pas consommer bêtement du cpu... cela s'appelle les "events" (évènement).

Il s'agit exactement d'une variable définie à 1 ou 0 (signalé ou non signalé), pour créer ce type de varibale, on utilise la fonction :

```
46 HANDLE CreateEvent(
47   LPSECURITY_ATTRIBUTES lpEventAttributes, // SD
48   BOOL bManualReset,                       // reset type
49   BOOL bInitialState,                      // initial state
50   LPCTSTR lpName                           // object name
51 );
```

Et en remplacement de notre boucle, on utilise la fonction[5]

```
52 DWORD WaitForSingleObject(
53   HANDLE hHandle,        // handle to object
54   DWORD dwMilliseconds   // time-out interval
55 );
```

Comment utiliser ces deux fonctions ? les paramètres de *CreateEvent()* contiennet : un paramètre de sécurité, que nous mettrons comme d'habitude à *NULL*, le second, plus interressant précise si l'event sera remis à l'état "*non signalé*" ou pas après la fonction d'attente (qui équivaut à notre boucle...), ensuite on donne l'état initial, et finalement un nom à notre event, ce nom servira à l'identifier, ce n'est utile que si l'on cherche à obtenir l'état d'un event d'un autre process... (via *OpenEvent*) La fonction renvoie un HANDLE qui servira à identifier l'event.

Quand à la fonction *WaitForSingleObject* elle est très simple à comprendre, on lui donne notre *HANDLE* et le temps maximum d'attente (que l'on peut définir à *INFINITE* dans ce cas, la

[5] En fait il existe plusieurs fonctions permettant de faire cela : *WaitForSingleObject(), WaitForMultipleObjects(), SignalObjectAndWait(), WaitForMultipleObjectsEx(), MsgWaitForMultipleObjects(), MsgWaitForMultipleObjectsEx, MsgWaitForMultipleObjectsEx(), SignalObjectAndWait(), WaitForMultipleObjectsEx(), WaitForSingleObjectEx() et RegisterWaitForSingleObject()*. La plupart étant réservé à Windows 2000 et Windows XP, vous devriez, après avoir lu cette article, être capable de les utiliser.

fonction ne renverra que si l'event est signalé... ce qui correspond exactement au comportement de notre boucle).

Il ne nous manque qu'une chose : comment définir l'état d'un event ? il y'a pour cela les deux fonctions :

```
56 BOOL SetEvent(
57   HANDLE hEvent   // handle to event
58 );
59
60 BOOL ResetEvent(
61   HANDLE hEvent   // handle to event
62 );
```

La première définissant l'event a l'état signalé, la seconde se chargeant de l'opération inverse.

3.1.1 Exemple de code

Nous allons, puisque nous savons maintenant demander à un thread d'attendre, "corriger" notre exemple de façon à éviter ce stupe *Sleep(1000) ;*. Ce qui donne :

```
63 HANDLE g_event;
64
65 DWORD WINAPI Thread1(LPVOID lpParameter)
66 {
67     printf("Ceci est ma thread\n");
68     SetEvent(g_event);
69 }
70
71 int main()
72 {
73     g_event=CreateEvent(NULL, 0, 0, NULL);
74
75     DWORD threadID;
76     CreateThread(NULL, 0, Thread1, NULL, 0, &threadID);
77
78     WaitForSingleObject(g_event, INFINITE);
79
80     printf("fin du programme de test\n");
81     return 0;
82 }
```

Voilà ! Le thread principale lance notre thread, elle s'exécute, pendant que la première attend l'évènement *g_event*, dès que la seconde a fini, elle signale cette event, le thread principal continue alors, et se termine.

3.2 Les mutex

Les mutex sont d'autre objets de synchronisation bien pratique, ils sont très proche des event, en fait on peut les voir comme une spécialisation des events. La différence étant dans le fait qu'un event est automatiquement remis à l'état non-signalé dès qu'une fonction d'attente a retourné du fait de son signalement : ça permet d'éviter d'avoir plusieurs threads qui se ruent sur la même resource suite au signalement d'un event. Je ne les détaillerais pas plus, la lecture de MSDN devrait être suffisante.

3.3 Accéder "atomiquement" au variables

Win32 nous fourni 5 fonctions permettant de modifier la valeur d'une variable en étant sûr qu'aucun autre thread ne lira la variable pendant ce temps (on appelle cela une opération "atomique", du au fait qu'elle est exécuté en une seul fois, sans que le thread soit schédulé). Elles permettent de faire des choses assez interressantes, si par exemple on doit maintenir un compteur.

```
83 LONG InterlockedIncrement(
84   LPLONG lpAddend   // variable to increment
85 );
86
87 LONG InterlockedDecrement(
88   LPLONG lpAddend   // variable address
89 );
```

Permettant d'incrémenter (ou décrémenter) un nombre en passant un pointer vers ce nombre.

```
90 LONG InterlockedExchange(
91   LPLONG Target, // value to exchange
92   LONG Value     // new value
93 );
```

Permettant de définir la valeur de **Target* à *Value* , et renvoie l'ancienne valeur de **Target* .

```
94 LONG InterlockedExchangeAdd (
95   LPLONG Addend,  // addend
96   LONG Increment  // increment value
97 );
```

Permettant de définir la valeur de **Target* à **Target* + *Value* , et renvoie l'ancienne valeur de **Target* .

```
98 PVOID InterlockedExchangePointer(
99   PVOID *Target,  // value to exchange
100  PVOID Value     // new value
101 );
```

Permettant de définir la valeur de **Target* à *Value* , et renvoie l'ancienne valeur de **Target* .

3.4 Les autres mécanismes de synchronisations

3.4.1 Les sections critiques

On peut les voir comme des mutex, mais elles présentent certaines particularités dans un environnement multiprocesseur. Elles offrent aussi des api plus concises (une fonction reprenant plusieurs opérations).

3.4.2 Les sempahores

Comme les events, ils peuvent être signalés ou non. Ils ont la particularité de maintenir un compteur : lorsqu'un thread appelle une fonction d'attente, le compteur est décrémenté, quand le compteur atteint 0, l'objet est défini comme non signalé. Un appel à *ReleaseSemaphore(...)* incrémente le compteur.

3.4.3 Les autres

Il existe encore d'autre objets de synchronisation, je ne vais faire que les citer, car ils ne sont disponible que dans des versions de Win32 supérieure à Windows 95 (càd : Windows 2000

ou Windows 98), il s'agit : des timer-queue (permettant de gérer une série d'opération devant s'exécuter à un moment donné), des waitable-timer (qui sont des "events" qui seront signalés au bout d'un certain moment, on peut aussi y associer une fonction qui sera exécutée à ce moment là). Tous ces objets sont en fait parfaitement évitable : on peut les recréer à partir des objets de bases (qui sont les events).

Un autre moyen consiste à utiliser... les messages de Windows, personnellement, je n'aime pas trop cette technique, sauf en ce qui concerne la syncrhonisation montante vers le thread principal... si par exemple vous devez demander à la thread principale de désactiver un controle de son ui, le plus simple est alors d'envoyer un message Win32.

4 Conclusion

Vous savez maintenant créer des threads, et vous avez tous les objets de bases pour les synchroniser, vous devriez pouvoir créer de grand (ou petit) programme utilisant les threads.

Si vous avez besoin de plus de renseignement, n'hésiter pas à me mailer[6] ou mieux, venez poster sur le newsgroup usenet fr.comp.os.ms-windows.programmation.

[6] dev@graffproject.com