Niveau 3

TP
S4 - Développement logiciel
S4.9 - Programmation événementielle
Environnement multitâche : traitements parallèles (thread...)

Multitâche

TABLE DES MATIÈRES

Répartition des tâches !

1 - PRISE EN MAIN DE L'API

On ne travaillera dans ce TP que dans un seul et unique fichier : `main.cpp` !

1) Au moyen de l'EDI QtCreator, créez un nouveau projet « Application console » nommé `MultiThread`. Dans le fichier `Main.cpp`, ajoutez une fonction respectant les conditions suivantes :

   - elle s'appelle `MethodeThread` ;
   - elle doit retourner un élément de type `DWORD` : la valeur figée `0L` dans l'immédiat ;
   - elle n'accepte qu'un seul paramètre de type `void *` (un pointeur sur tout type de données) dont le nom sera `lpParameter` ;
   - elle affiche à l'écran le contenu de l'objet pointé par le paramètre.

   Attention, il sera nécessaire de transtyper le paramètre de façon à retrouver la donnée initiale, par exemple un entier de valeur 8, lors de son utilisation.

   Bien évidemment, vous testerez votre méthode en l'appelant dans le `main()`.

2) Juste avant la fin du `main()`, affichez le message « `Fin du main !!!` ». Et faites pauser le programme en attente de l'appui d'une touche.

1.1 - CRÉATION DU THREAD

Il vous faudra include le fichier d'en-tête `windows.h` pour pouvoir utiliser ces fonctions !

3) D'après les informations fournies par l'aide en ligne, créez les variables nécessaires à la bonne exécution de la méthode de l'API win32, `CreateThread()`, permettant la création et l'initialisation par le système d'un thread donné.

Remarque : En cas d'incertitude sur les valeurs à passer aux diverses méthodes de l'API win32, utilisez celles par défaut proposées dans l'aide en ligne. Par exemple, le premier paramètre `lpThreadAttributes` de la méthode `CreateThread`, n'est pas simple à configurer... Pour se simplifier la vie, dans ce TP, nous pouvons utiliser la valeur `NULL` (`nullptr`) indiquée dans l'aide en ligne !

4) Toujours d'après l'aide en ligne, appelez la méthode `CreateThread`. Pour cela, vous devrez lui fournir tout un tas de paramètres que vous devrez définir par vous-même. Le thread créé devra appeler la méthode `MethodeThread` implémentée peu avant et ne devra pas être exécuté lors de sa création. La valeur de retour de cette fonction devra être testée et afficher « Handle du thread valide » si elle a réussi !

5) À partir du moment où un objet système a été créé, c'est-à-dire que la méthode `CreateThread()` nous a bien retourné un handle valide, il sera nécessaire de libérer toute la mémoire qui lui aura été allouée. Pour cela, on demande au SE de détruire l'objet en bonne et due forme grâce à la méthode `CloseHandle()` et ceci, dès que l'objet système n'est plus requis dans notre application. N'oubliez pas d'appeler cette méthode pour chaque handle que vous aurez créé par la suite et au plus tard à la fin du programme principal.

6) Initialisez la priorité de ce thread à la valeur par défaut du système. Pour cela, vous utiliserez la méthode `SetThreadPriority()` disponible. La valeur de retour de cette méthode devra être testée et validée.

1.2 - UTILISATION DU THREAD

7) À ce stade, l'exécution de l'application ne devrait pas permettre d'affichage par le thread puisqu'il n'est que créé et non lancé. On va donc s'intéresser maintenant à son lancement qui doit être réalisé par la méthode `ResumeThread()`. La valeur de retour de cette méthode devra être testée et validée.

8) Modifiez la méthode `MethodeThread()` de façon à ce qu'elle affiche la valeur passée en paramètre sur 10 lignes différentes à raison d'une ligne toutes les 500 ms. Que peut-on constater ?

9) Il est possible d'imposer au programme `main()` d'attendre la fin normale du thread pour proposer l'appui sur la touche pour quitter. Quelle est la méthode de l'API Win32 à utiliser pour attendre la fin d'un objet simple tel qu'un thread ? Implémentez-là et testez votre modification.

Remarque : Un objet thread du système est considéré comme étant « **signalé** » lorsqu'il est terminé normalement. S'il ne se termine pas dans une certaine durée, un système de time-out est alors « signalé » à la place de l'objet thread.

10) Modifiez à nouveau la méthode `MethodeThread()` de façon à ce qu'elle affiche toutes les 500 ms et sur une nouvelle ligne la donnée reçue en paramètre mais de façon continue, à l'infini. Que se passe-t-il alors pour le `main()` ? Trouvez une méthode de l'API win32 permettant de suspendre momentanément un thread un peu trop actif. Mettez-la en œuvre de façon à ce que le programme principal suspende le thread après 5 s d'activité et propose l'appui d'une touche pour continuer. Le thread, en pause sera relancé pour un maximum de 3 s dès que l'opérateur aura actionné une touche.

11) Plusieurs techniques peuvent être mises en œuvre pour stopper le thread. À la question précédente, la méthode `WaitForSingleObject()` a été bien utile avec la programmation de son time-out. Cette technique est à privilégier dès que l'on connaît des durées maximales d'exécution. Plus radicalement, il est encore possible de tuer le thread au moyen d'une des deux méthodes de l'API suivantes :

- `TerminateThread()` ;
- `ExitThread()`.

Testez ces deux méthodes et choisissez d'implémenter la plus adaptée à notre situation. Quelle est la principale raison de votre choix ?

1.3 - ÉVÉNEMENT

12) L'aide en ligne est formelle sur l'utilisation de la méthode `TerminateThread()` : elle est à utiliser seulement lorsqu'il n'y a plus aucune autre technique de programmation fonctionnant pour stopper le thread. Mais il est un procédé très largement répandu dans Windows permettant de stopper le thread sans avoir recours à cette aberration... Il s'agit des évènements. Il est en effet possible de modifier la boucle infinie dans la méthode du thread de façon à ce qu'elle se termine dès l'arrivée d'un événement spécifique. En fait le programme principal va signaler un événement et détecter ce signalement. Pour cela, on va procéder en trois phases :

- la création de l'objet système événement par la méthode de l'API `CreateEvent()`. Implémentez toutes les variables nécessaires à l'exécution correcte de cette méthode en utilisant les valeurs par défaut en cas de doutes et en sachant que l'événement n'est pas signalé à la création mais qu'il le sera, plus tard, manuellement dans le programme principal. Créez ensuite l'objet système événement ;
- la modification de la condition de la boucle infinie de façon à ce qu'on la quitte dès que l'événement est signalé. Comme pour détecter la fin d'un thread (qui est alors signalé), la méthode de l'API, `WaitForSingleObject()` fonctionne aussi pour détecter le signalement d'un événement ;
- le signalement de l'événement par le programme principal après avoir laissé une seconde chance au thread de se terminer tout seul (attente de 3s). Le programme principal attend après cela indéfiniment que le thread signale sa clôture pour terminer complètement l'application.

13) Modifiez le programme principal de façon à avoir maintenant deux threads différents exécutant la même méthode `MethodeThread()`, dont l'un permettra d'afficher la valeur entière 8 tandis que l'autre affichera la valeur 4. Le programme principal réalisera les mêmes opérations que précédemment (questions 10 et suivantes) sur le thread affichant la valeur 8 (lancer le thread 5 s, le stopper et attendre une touche pour le relancer 3 s de plus puis lui demander de se terminer tranquillement !) alors qu'il laissera le second thread, affichant la valeur 4, tranquille à l'infini. Que constate-t-on par rapport à l'exécution et à la fin des threads ? Proposez une solution simple, rapide et efficace.

1.4 - SÉMAPHORES

De la question précédente, on peut observer un dysfonctionnement de l'affichage. Il s'agit d'une ressource unique que trois tâches doivent se partager : le `main()` et les deux threads et il est bien compréhensible que tout ce petit monde peut accéder quand il veut à l'écran. Par contre, quel méli-mélo !

Un objet système peut dans ce cas nous aider fortement : le sémaphore. Il s'agit d'un compteur permettant d'accéder ou non à la/les ressource(s) qu'il protège : tant que sa valeur est supérieure à `0`, il peut nous fournir un jeton (sa valeur se décrémente alors d'une unité) et nous autoriser la ressource.

Tant qu'il reste des jetons au sémaphore, n'importe quelle tâche peut en faire la demande, l'obtenir et ainsi accéder à la ressource. Un maximum de tâches pouvant accéder simultanément à la ressource est défini lors de la création du sémaphore. Dès que l'on n'a plus besoin de cette ressource, on rend le jeton au sémaphore (dont la valeur s'incrémente alors d'une unité) qui peut le redistribuer à une autre tâche qui souhaite accéder à cette ressource.

Dans notre cas, pour éviter tout mélange d'informations sur l'écran, il convient de bloquer définitivement l'accès aux autres tâches à partir du moment où une seule et unique tâche y accède : le sémaphore ne possède qu'un seul et unique jeton !

14) Un sémaphore est créé au moyen de la méthode de l'API `CreateSemaphore()`. Implémentez toutes les variables utiles à son exécution et demandez au système d'imposer des valeurs par défaut si besoin.

15) La méthode `WaitForSingleObject()` est décidément fort utile dans ces histoires de multi-threading... Ici, elle peut être utilisée pour faire une demande de sémaphore : le sémaphore n'est pas signalé s'il ne peut pas délivrer de jeton et il se signale dès qu'un jeton est donné. Modifiez le programme de façon à demander, attendre à l'infini et récupérer un jeton avant chaque accès à l'écran.

16) Le jeton que l'on a récupéré et que l'on garde tant qu'on veut écrire à l'écran doit être libéré dès que l'écran ne nous est plus nécessaire, et ceci, le plus rapidement possible afin de ne pas léser les autres tâches qui seront bloquées par le manque de jeton (1 seul dans notre cas). Pour cela la méthode `ReleaseSemaphore()` est à utiliser. Compétez votre code pour permettre aux autres tâches d'accéder à l'écran.

2 - BONUS

17) Modifiez la priorité des threads de façon à ce que celui qui affiche la valeur 4 soit bien plus rapide que l'autre. Que constatez-vous ? Pourquoi le programme fonctionne comme ça ?

18) Ajoutez trois threads de plus, affichant respectivement les valeurs 100, 1000 et 10000. Validez le fonctionnement de votre programme, notamment par rapport à l'affichage !

19) Affichez, pour chaque retour de fonctions testé, le code erreur et la chaîne de caractères correspondante, tous deux fournis par le système. La dernière erreur peut être récupérée par `GetLastError()`. À vous de l'implémenter, et de trouver les autres fonctions de l'API à utiliser.