

Boost.Threads : les threads de Boost

par Matthieu Brucher (<http://matthieu-brucher.developpez.com/>) (Blog)

Date de publication : 08/08/2006

Dernière mise à jour : 25/08/2007

Utiliser des threads dans ses programmes nécessite généralement une connaissance de chaque plateforme utilisée, les interfaces pour les threads ne sont pas standardisées en C++. Des bibliothèques dites portables existent, permettant de faire les liens manquants. Par exemple Boost.Threads.

Introduction

I - Démarrer avec Boost.Threads

II - L'architecture de la bibliothèque

II-A - Les mutex

II-A-1 - Les stratégies de blocage

II-A-1-a - La stratégie récursive

II-A-1-b - La stratégie vérifiée

II-A-1-c - La stratégie non vérifiée

II-A-1-d - La stratégie non spécifiée

II-A-2 - Les politiques

II-A-2-a - La politique FIFO

II-A-2-b - La politique de priorité

II-A-2-c - La politique non spécifiée

II-A-3 - Les concepts de mutex

II-A-3-a - Le concept standard

II-A-3-b - Le concept d'essai

II-A-3-c - Le concept de temp

II-A-4 - Résumé des mutex

II-B - Les lock

II-B-1 - Qu'est ce qu'un lock ?

II-B-2 - Le concept de ScopedLock

II-B-3 - Le concept de TryLock

II-B-4 - Le concept de ScopedTryLock

II-B-5 - Le concept de TimedLock

II-B-6 - Le concept de ScopedTimeLock

II-B-7 - Résumé des lock

II-C - Les threads

II-C-1 - La classe thread

II-C-1-a - Constructeurs et destructeur

II-C-1-b - Les comparaisons

II-C-1-c - La méthode join()

II-C-1-d - Quelques fonctions statiques

II-C-2 - La classe thread_group

II-C-3 - Une classe bien particulière : thread_specific_ptr

II-D - Les autres classes et fonctions

II-D-1 - La classe xtime

II-D-2 - La fonction call_once

II-D-3 - La classe barrier

II-D-4 - La classe condition

III - Exemples

III-A - Un exemple simple

III-B - Un exemple plus complexe

V - Conclusion

Introduction

Les threads permettent d'exécuter du code en parallèle, de manière indépendante et asynchrone. Chaque thread appartient à un processus qui l'a créé, mais ils utilisent chacun le processeur comme s'ils étaient seuls. Les programmes utilisant les threads sont appelés **multithreadés**, les autres programmes qualifiés de **monothreadés**.

Les avantages des programmes utilisant les threads sont nombreux. Par exemple, l'attente d'un événement externe peut bloquer un programme et la déporter dans un thread peut permettre à l'application de répondre. Si on possède plusieurs cœurs, ou plusieurs processeurs/unités d'exécutions, paralléliser son application peut permettre d'accélérer un programme.

En revanche, les inconvénients sont tout aussi nombreux, et plus sournois. Par exemple, lorsque 2 threads veulent accéder en écriture à une ressource, on ne peut garantir le résultat après les accès. On peut aussi rencontrer des problèmes d'accès aux ressources - 2 threads désirant obtenir un maximum de mémoire et occupant toute mémoire disponible, mais pas suffisamment sans pouvoir la rendre -. Ces erreurs ne sont pas particulières ou rares, elles sont courantes si on ne fait pas attention. Boost.Thread essaie de remédier aux erreurs d'inattention de base.

Parmi les dernières considérations avant de se lancer, il faut noter le problème des tests et du débogage. Un programme simple, monothreadé, est simple à déboguer, tout se déroule ligne après ligne, instruction après instruction. Un programme multithreadé n'a pas cette faculté. Pendant une exécution, une instruction peut s'exécuter après une autre, puis à la suivante, c'est l'inverse. On ne peut donc pas répéter les erreurs, les versions debug n'ont pas le même comportement que les versions release, ...

Cette bibliothèque nécessite une version compilée de Boost.

I - Démarrer avec Boost.Threads

Utiliser Boost.Thread implique l'utilisation des versions multithreadées des bibliothèques runtime, sans quoi on retombe dans les travers exposés précédemment.

Dans le même registre, certaines fonctions de la bibliothèques standard du C++ héritées du C doivent être évitées, comme `rand()` - voir alors **Boost.Random**.

Les fonctions et classes de Boost.Thread répondent à des caractéristiques précises. Les destructeurs ne lèvent pas d'exception, les méthodes ou fonctions lèvent des exceptions de type **`std::bad_alloc`** lors d'un défaut d'allocation mémoire, **`boost::thread_resource_error`** lors de défaut d'autres allocations ou **`boost::lock_error`** pour des erreurs de verrouillage.


Dans le même style, les classes héritent de manière privée de **`boost::noncopyable`** pour indiquer que leurs instances ne peuvent être copiées.

II - L'architecture de la bibliothèque

L'architecture de la bibliothèque est constituée de 2 parties, même si d'autres sont en projet. La première contient les primitives de base pour la synchronisation, la deuxième contient les threads eux-mêmes.

II-A - Les mutex

Les mutex et leur implémentation sont actuellement sujet à discussion par le comité Boost, mais pour l'instant, ils restent tel quel.

Un mutex a 2 états : **locked** et **unlocked**. Cet outil permet d'accéder à une ressource commune à plusieurs threads de manière synchrone. Avant chaque accès, on bloque le mutex de protection, et après l'accès, on débloquent le mutex. Afin d'empêcher l'oubli du déblocage, fréquent et qui résoud en un blocage définitif, Boost.Threads utilise un  **pattern** appelé *Scope Locking*. En fait, un objet créé sur la pile bloque l'accès et lors de sa destruction automatique à la sortie du *scope*, il débloquent l'accès à la ressource. C'est le seul moyen utilisable avec Boost.Threads. Excessif, mais compréhensible car totalement sûr par rapport aux exceptions.

II-A-1 - Les stratégies de blocage

Plusieurs stratégies face à l'accès à un mutex par un même thread existent pour utiliser les mutex.

II-A-1-a - La stratégie récursive

Lorsqu'un thread essaie de bloquer un mutex qu'il a déjà bloqué, l'opération réussit. Cela peut être utile si une fonction d'un thread bloque le mutex et qu'une fonction appelée par cette même fonction fait de même. Les classes **boost::recursive_mutex**, **boost::recursive_try_mutex** et **boost::recursive_timed_mutex** utilisent cette stratégie.

II-A-1-b - La stratégie vérifiée

Lorsqu'un thread essaie de bloquer un mutex qu'il a déjà bloqué, une erreur est générée. Pour Boost.Threads, il s'agit de l'exception **boost::lock_error**. En revanche, aucun mutex proposé n'utilise cette stratégie, mais ce serait prévu.

II-A-1-c - La stratégie non vérifiée

Lorsqu'un thread essaie de bloquer un mutex qu'il a déjà bloqué, le thread se bloque. Cette stratégie est moins sûre que les 2 précédentes, mais aussi plus rapide. D'ailleurs, beaucoup de bibliothèques utilisent cette stratégie. Aucun mutex proposé n'utilise cette stratégie, mais ce serait prévu.

II-A-1-d - La stratégie non spécifiée

Cette stratégie est simple, lorsqu'un thread essaie de rebloquer un mutex, on ne connaît pas le comportement du programme. C'est la stratégie la plus rapide et naturellement la moins sûre. Les classes de Boost.Threads **boost::mutex**, **boost::try_mutex** et **boost::timed_mutex** utilisent cette stratégie.

II-A-2 - Les politiques

Les politiques gèrent la manière de donner l'accès aux threads demandeurs.

II-A-2-a - La politique FIFO

Cette politique consiste à donner accès au premier demandeur. Cela permet qu'un thread de haute priorité ne bloque pas l'accès à la ressource lorsqu'un thread de basse priorité en a besoin.

II-A-2-b - La politique de priorité

Avec cette politique, le thread de plus haute priorité bloque toujours le mutex en priorité. En revanche, cela signifie que les threads de basse priorité peuvent être bloqués.

II-A-2-c - La politique non spécifiée

Dans ce cas-ci, aucun ordre spécifique n'est donné aux threads désirant accéder à une ressource. Tous les mutex de Boost.Threads suivent cette politique.

II-A-3 - Les concepts de mutex

Outre les stratégies et les politiques, il y a aussi des concepts de mutex - vous suivez toujours ? -

II-A-3-a - Le concept standard

Le concept standard, présent dans **boost::mutex** et **boost::recursive_mutex** implique simplement que le mutex possède un type **scoped_lock** qui essaie de bloquer un mutex normalement.

II-A-3-b - Le concept d'essai

Le concept d'essai, présent dans **boost::try_mutex** et **boost::recursive_try_mutex** implique que le mutex possède un type **scoped_try_lock** qui renvoie une exception de type **boost::lock_error** lorsqu'une instance essaie de bloquer un mutex ou qui tente de bloquer un mutex si une condition est vérifiée.

II-A-3-c - Le concept de temp

Le concept de temps, présent dans **boost::timed_mutex** et **boost::recursive_timed_mutex** implique que le mutex possède un type **scoped_timed_lock** qui permet d'appeler un mutex avec une durée ou qui essaie de bloquer un mutex si une condition est vérifiée.

II-A-4 - Résumé des mutex

Voici donc un résumé des mutex proposés par Boost.Threads.

Concept	Dérive de	Modèle
Mutex		boost::mutex
		boost::recursive_mutex
TryMutex	Mutex	boost::try_mutex
		boost::recursive_try_mutex
TimedMutex	TryMutex	boost::timed_mutex

boost::recursive_timed_mutex

II-B - Les lock

Le but de ces concepts est de bloquer un mutex. On a déjà vu quelques versions de ces concepts, équivalents aux concepts de mutex standards, d'essai et de temps. Maintenant, Boost.Threads a décidé d'utiliser les versions *Scoped*, donc à savoir que des objets seront utilisés pour bloquer des mutex.

II-B-1 - Qu'est ce qu'un lock ?

Pour un type de Lock **L**, une instance **lk** et une instance constante **clk**, certaines expressions doivent être valides.

Express	Effets
<code>(&lk) >~L()</code>	Si le mutex est bloqué, on le débloque.
<code>(&clk) >operator const void*()</code>	Retourne une valeur non nulle si le mutex a été bloqué par clk, 0 sinon.
<code>clk.locked()</code>	Retourne true si locked() est vrai.
<code>lk.locked()</code>	Retourne boost::lock_error si locked() . Si le mutex associé est déjà bloqué par le même thread, l'effet dépend de la stratégie du mutex. Dans le cas général, le thread est placé dans un état bloqué, puis d'attente lorsque le mutex est libéré et pourra se voir passer en état actif. Après, locked() retourne true .
<code>lk.unlock()</code>	Débloquent le mutex associé. Après, locked() retourne false .

II-B-2 - Le concept de ScopedLock

Ce concept requiert que le constructeur et le destructeur aient des effets particuliers. Ici, on considère **m** comme un mutex standard

Express	Effets
L lk (m)	Construit un objet lk , associe le mutex m et appelle lock()
L lk (m , b);	Construit un objet lk , associe le mutex m et si b , appelle lock()

II-B-3 - Le concept de TryLock

Ici, on ajoute la méthode pour tenter un blocage.

Express	Effets
lk.try_lock()	Retourne boost::lock_error si locked() . Tente de bloquer le mutex, et lève une exception si le mutex est bloqué par un autre thread. S'il s'agit du même thread, la réaction dépend de la stratégie du mutex. Retourne true si le blocage a été effectué, false sinon.

II-B-4 - Le concept de ScopedTryLock

On ajoute donc à un **ScopedLock** le concept de **TryLock**.

Express	Effets
L lk (m)	Construit un objet lk , associe le mutex m et appelle try_lock()
L lk (m , b)	Construit un objet lk , associe le mutex m et si b , appelle lock()

II-B-5 - Le concept de TimedLock

Outre les objets des autres concepts, un objet **t** de type **boost::xtime** doit être ajouté. Il spécifiera le temps pendant lequel le blocage sera tenté.

Express	Effets
<code>lk.timed_lock(boost::lock_error si locked()).</code> Tente de bloquer le mutex pendant le temps t . S'il s'agit du même thread, la réaction dépend de la stratégie du mutex. Retourne true si le blocage a été effectué, false sinon.	

II-B-6 - Le concept de ScopedTimeLock

On fusionne à nouveau les **ScopedTryLock** et **TimedLock**.

Express	Effets
<code>L lk(m, t);</code> Construit un objet lk , associe le mutex m et appelle timed_lock(t)	
<code>L lk(m, b);</code> Construit un objet lk , associe le mutex m et si b , appelle lock()	

II-B-7 - Résumé des lock

Concept	Dérive de	Modèle
Lock		
ScopedLock	Lock	boost::mutex
		boost::recursive_mutex
		boost::try_mutex
		boost::recursive_try_mutex
		boost::timed_mutex
		boost::recursive_timed_mutex
TryLock	Lock	
ScopedTryLock	TryLock	boost::try_mutex
		boost::recursive_try_mutex
		boost::timed_mutex
		boost::recursive_timed_mutex
TimedLock	TryLock	
ScopedTimedLock	TimedLock	boost::timed_mutex
		boost::recursive_timed_mutex

II-C - Les threads

La partie la plus intéressante puisque sans elle, la bibliothèque ne sert pas à grand chose. Les threads peuvent être regroupés en groupe de threads. Toutes ces classes sont non copiables.

II-C-1 - La classe thread

La classe thread propose juste ce qu'il faut.

II-C-1-a - Constructeurs et destructeur

Le constructeur par défaut crée un objet représentant le thread d'exécution courant. L'autre constructeur, le plus utile, prend en argument un **boost::function0<void>** qui encapsule en fait un pointeur vers une fonction ne prenant aucun argument. Naturellement, la première forme, représentant le thread courant, ne peut pas être "rejoint", tandis que la deuxième forme oui, logiquement. Qu'entend-on par rejoindre ? En fait, le thread courant stoppe son exécution jusqu'à ce que le thread rejoint finisse.

La destruction de l'objet n'interrompt pas son exécution, celle-ci se détache complètement de l'objet.

II-C-1-b - Les comparaisons

Deux opérateurs existent, **operator==** et **operator!=** qui retournent **true** - respectivement **false** - si les 2 threads représentent la même unité d'exécution, et **false** dans le cas contraire - respectivement **true**.

II-C-1-c - La méthode join()

Cette méthode permet à un objet thread d'être rejoint, donc le thread courant sera bloqué jusqu'à la fin du thread rejoint.

II-C-1-d - Quelques fonctions statiques

La première fonction statique est **sleep** qui prend en argument un objet de type **boost::xtime** indiquant combien de temps le thread courant sera mis en pause.

La seconde fonction statique est **yield** qui remet le thread courant dans l'état Ready, l'état dans lequel ce qu'on appelle le **scheduler** choisit le thread qui continuera son exécution.

II-C-2 - La classe thread_group

Le groupe de threads permet de gérer des threads nécessitant un traitement commun - par exemple, des threads travaillant sur les mêmes ressources -. La construction crée un groupe de threads vide tandis que la destruction détruit chaque thread géré par le groupe.

La fonction **create_thread** crée un nouveau thread à partir d'un **boost::function0<void>**, l'ajoute au groupe et renvoie un pointeur vers le thread créé. Avec **add_thread**, on peut ajouter un pointeur sur un thread, tandis qu'avec **remove_thread** on le retire. **join_all** appelle **join** sur chaque thread dans le groupe.

II-C-3 - Une classe bien particulière : thread_specific_ptr

Cette classe bien particulière permet de stocker dans un seul objet plusieurs pointeurs, en fait un par thread. Lors de la destruction d'un thread, le pointeur associé est détruit à l'aide d'une fonction que l'on peut spécifier. Elle peut servir dans le cas de portage d'une interface vivant auparavant dans un environnement monothreadé, ou si chaque thread doit accéder à des ressources physiquement uniques mais accessible à travers un point d'accès global - par exemple si on lance plusieurs fois une même fonction en parallèle -.

Un des constructeurs prend en argument un **void (*cleanup)(void*)**. Cette fonction est appelée à la destruction de l'objet pointé lors de la destruction du thread. On peut aussi passer le pointeur **NULL** permettant de ne rien effectuer.

La fonction **release** cède la gestion du pointeur et le renvoie à la fonction appelante. La fonction **reset** permet de changer le pointeur à gérer pour le thread courant tout en appelant la fonction chargée pour le nettoyage.

Les fonctions **get**, **operator->** et **operator*()** retourne le pointeur géré par l'objet pour le thread courant, sauf pour **operator*()** qui retourne en fait une référence et donc qui nécessite d'avoir un pointeur géré non nul.

II-D - Les autres classes et fonctions

II-D-1 - La classe xtime

Cette classe est dépendante de la plateforme. Pour remplir une structure xtime, on passe à la fonction **xtime_get** un pointeur ainsi qu'un entier **clock_type** qui sera retourné si la fonction réussit. Cet entier fait parti de l'enum **xtime_clock_types**.

II-D-2 - La fonction call_once

Cette fonction permet d'initialiser une ressource correctement dans un environnement multithread. Elle prend en argument un pointeur sur une fonction ainsi qu'un drapeau par référence qui indique si la fonction a été appelée ou pas.

Exemple d'utilisation

```
boost::once_flag once = BOOST_ONCE_INIT;

void init()
{
    //...
}

void thread_proc()
{
    boost::call_once(&init, once);
}
```

II-D-3 - La classe barrier

La classe **barrier** permet à plusieurs threads de se synchroniser. Le constructeur prend une taille en paramètre qui est le nombre de threads à attendre.

II-D-4 - La classe condition

Une classe importante mais difficile à exploiter.

Quatre fonctions **wait** existent. La première libère le mutex associé au lock donné en paramètre, bloque le thread en attendant l'appel à **notify_one** et rebloque le mutex. La deuxième effectue la séquence précédente si le deuxième argument peut être converti à **true**. Le troisième prend comme deuxième paramètre un temps **xtime** et retourne **true** si ce temps est atteint. La dernière est une combinaison des précédentes fonctions.

Les fonctions **notify_one** et **notify_all** permettent de libérer un ou tous les threads en attente de la condition.

III - Exemples

Quelques petits exemples de fonctionnement

III-A - Un exemple simple

Comment créer un thread qui exécute une fonction particulière ?

```
#include <boost/thread/thread.hpp>

void uneFonction()
{
    boost::thread unThread(&uneAutreFonction);
}
```

III-B - Un exemple plus complexe

On va créer une fonction qui sera lancée en parallèle plusieurs fois.

```
#include <boost/thread/thread.hpp>
#include <boost/thread/thread_group.hpp>

#include <vector>

// Une variable globale qui pourra être encapsulée au besoin
boost::thread_specific_ptr<int> value;
// Une variable globale avec son mutex, le tout devrait être dans une classe englobante
std::vector<unsigned int> traitement;
boost::mutex mutexTraitement;

void uneFonctionPlusieursThreads()
{
    {
        boost::mutex::scoped_lock lock(mutexTraitement);
        // Accède aux données de traitement en toute sécurité
    }
    value.reset(new int); // On crée des données spécifiques au thread

    // Traitement sur value
}

int main()
{
    boost::thread_group group;
    for(unsigned int i = 0; i < 10; ++i)
    {
        group.create_thread(&uneFonctionPlusieursThreads);
    }
    group.join_all();
}
```

V - Conclusion

Cette bibliothèque n'est pas encore aboutie, il manque encore beaucoup de choses comme des stratégies plus évoluées pour les mutex, mais on peut espérer que l'évolution de la bibliothèque ira dans le bon sens. Certains mutex ont été supprimés depuis quelques versions à cause de blocages - les mutex **boost::read_write_mutex** encore présent en partie dans la documentation -, la preuve qu'il est difficile de concevoir une telle bibliothèque.

