TP3 - Threads et concurrence

Introduction au projet Plazza

Uriel CORFA uriel@corfa.fr
David GIRON thor@epitech.net

Résumé: Les threads sont un outil extrêmement puissant permettant à un programme d'exécuter plusieurs actions, liées ou non, simultanément. Ce TP a pour but de vous familiariser avec leur maniement, et avec les nombreuses contraintes qui les entourent.

Table des matières

Chapitre I

Préambule au TP

I.1 Cadre et esprit de ce TP

Ce TP peut s'avérer relativement long. Et pour cause, il n'est pas prévu pour tenir dans les 3 heures de créneau. A vous de vous organiser pour en tirer le meilleur parti.

De même, bon nombre d'étapes peuvent se faire très rapidement en recopiant le code fourni dans le cours. Il serait fort dommage pour vour de gâcher ce TP en le réalisant ainsi, sans prendre le temps de vous assurer d'avoir bien compris toutes les notions.

Sachez enfin que derrière ce qui peut sembler n'être qu'une API de plus, les threads cachent en réalité tout un nouveau paradigme. Ceci n'est qu'une introduction rapide à la programmation concurrente, qui cache bien des secrets et des pièges. Vous trouverez à la fin de ce document une intéressante collection de liens sur le sujet.

I.2 Remarques diverses

Nous attachons une grande importance à la qualité de nos documents. Si vous constatez dans ce sujet des erreurs, des fautes d'orthographe ou de typographie, ou si vous avez des remarques quelconques à faire, contactez-nous à koala@epitech.eu.

Chapitre II

Threads et mutexes

II.1 Exercice 1

- Afin de commencer ce TP, vous devez réaliser un cas typique démontrant l'utilité des mutexes. Créez une fonction `increment_counter` qui reçoit un pointeur sur int et incrémente l'int pointé, et ce, de nombreuses fois. On notera ce nombre `N`.
- Créez un main lançant de multiples threads, exécutant chacun `increment_counter`, sur le même int. On notera le nombre de threads `M`. Constatez, comme dans le cours, qu'à la fin de votre exécution le total contenu dans votre compteur est inférieur à `M * N`.
- Ajoutez maintenant un mutex pour protéger votre compteur. Constatez que le total est désormais cohérent.

II.2 Exercice 2

- Encapsulez maintenant vos mutexes dans des objets. Ces objets devraient respecter une interface similaire à :

```
1     class IMutex {
2       public:
3       virtual ~IMutex(void);
4       virtual void lock(void) = 0;
5       virtual void unlock(void) = 0;
6       virtual bool trylock(void) = 0;
7     };
```

- Utilisez votre nouvelle classe dans le code réalisé à l'étape précédente.

II.3 Exercice 3

- Créez maintenant une classe `ScopedLock`. Vous êtes libres de vous inspirer de celle fournie dans le cours, tant que vous la comprenez. Pour rappel, un `ScopedLock` est construit en prenant en paramètre un mutex. Il verrouille ce mutex lors de sa construction, et déverrouille ce mutex lors de sa destruction.
- Bien sur votre `ScopedLock` devra fonctionner avec votre abstraction aux mutexes, et non avec des `pthread_mutex_t`.
- Attention au vocabulaire !
  - Un **Mutex** est un objet pouvant être ou non verrouillé, servant à la synchronisation ;
  - Un **Lock** est l'action de verrouiller ce mutex.
- Utilisez votre nouveau `ScopedLock` dans le code réalisé aux étapes précédentes.

II.4 Exercice 4

- Créez maintenant une classe `Thread`, encapsulant la notion de thread. Prenez le temps de réflechir à son interface. Une base intéressante serait d'encapsuler les notions suivantes :
  - Statut du thread (pas encore démarré, en cours, mort) ;
  - Démarrage du thread
  - Attente de la mort d'un thread
- Utilisez votre nouvelle classe `Thread` dans le code réalisé aux étapes précédentes, et qui devrait maintenant commencer à ressembler à du vrai `C++`.

Chapitre III

Producteur-consommateur

III.1 Exercice 5

- Créez un objet `SafeQueue`. Cet objet encapsule une queue dans laquelle plusieurs threads viendront ajouter et retirer des éléments. Il convient donc de la mutexer. Bien sûr, vous utiliserez pour cela vos classes `Mutex` et `ScopedLock` à bon escient.
- Pour le moment, votre `SafeQueue` contiendra des `int`s. Vous la ferez évoluer plus tard vers une solution plus générique.
- Votre classe devra respecter l'interface suivante :

```
1  class ISafeQueue {
2    public:
3    virtual ~ISafeQueue(void);
4    virtual void push(int value) = 0;
5    virtual bool tryPop(int* value) = 0;
6    virtual bool isFinished(void) = 0;
7    virtual void setFinished(void) = 0;
8  };
```

- Votre méthode `tryPop` dépile un élément de la queue si elle n'est pas vide, la stocke à l'adresse pointée par `value`, et renvoie `true`. Si la queue est vide, elle renvoie `false` immédiatement.
- `setFinished` indique que la queue ne sera plus remplie. `isFinished` renvoie `true` si `setFinished` a été appelée et que la queue est vide.
- Pensez à tester votre `SafeQueue`. N'hésitez pas à faire des tests intensifs, qui pourraient mettre en lumière des race conditions ou des deadlocks. Attention, les bugs de cette catégorie sont particulièrement insidieux.

III.2 Exercice 6

- Vous allez maintenant implémenter un pattern producteur-consommateur. Celui-ci utilisera naturellement votre `SafeQueue`.
- Le consommateur devra effectuer une attente active. C'est à dire que lorsqu'il tente de dépiler une valeur et n'y parvient pas car la queue est vide, il s'endort pour un certain temps (vous pouvez utiliser `usleep` pour cela), puis retente.
- Votre consommateur pourra, pour l'exemple, réaliser une action très simple (comme afficher le nombre récupéré), et le producteur pourra créer des nombres arbitrairement. L'essentiel ici est la structure.
- Créez un main lançant un nombre paramétrable de producteurs et de consommateurs. Ici encore, testez votre programme exhaustivement.

III.3 Exercice 7

- Réalisez une classe `CondVar` encapsulant la notion de variable conditionnelle.
- Cette classe pourrait avoir une interface proche de :

```
1  class ICondVar {
2    public:
3    virtual void wait(void) = 0;
4    virtual void signal(void) = 0;
5    virtual void broadcast(void) = 0;
6  };
```

- Utilisez votre `CondVar` pour ajouter une méthode `int pop(void);` à votre `SafeQueue` . Cette méthode bloque jusqu'à pouvoir dépiler un `int` de la queue. Si la queue est `finished`, cette méthode pourrait, par exemple, sortir en jetant une exception.

Chapitre IV

Pour aller plus loin

IV.1 Exercice 8

- Votre `SafeQueue` n'est pas très utile en l'état : une queue d'ints n'est que rarement utile. Vous allez maintenant la templater pour en faire un conteneur générique.
- Attention aux copies qui peuvent être utiles ou inutiles. A vous de penser votre objet correctement pour qu'il soit utile. Les références comme les pointeurs peuvent vous aider.

IV.2 Exercice 9

- Reprenez votre objet `Thread` et templatez-le afin qu'il accepte n'importe quel type d'objet appelable. Par là on entend les pointeurs sur fonctions, mais également les objets surchargeant l'`operator()`.
- Besoin d'un rappel sur `operator()` ? Retournez-donc au d09 de la piscine vous rafraîchir la mémoire.

IV.3 Exercice 10

- Implémentez un thread pool.
- Un thread pool est tout simplement constitué d'une liste de tâches à effectuer (pourquoi ne pas la représenter par une `SafeQueue` ?) et d'un nombre fixe de threads. Ces threads dépilent depuis la queue des actions à effectuer. Ces actions pourront elles-mêmes en empiler par la suite.

IV.4 Lectures conseillées

Il est hélas difficile en un TP d'aborder plus que la base en matière de programmation concurrente. Voici plusieurs liens que l'équipe Koala vous recommande. L'ordre n'est pas forcément important. Bien sûr, tout lire serait extrêmement long, c'est pourquoi nous vous encourageons à revenir feuilleter ce TP à l'avenir.

- http://en.wikipedia.org/wiki/Thread_pool_pattern
  Une présentation plus générale des thread pools.
- http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf
  Une présentation du concept des "Active Objects", ou "Acteurs".
- http://pubs.opengroup.org/onlinepubs/009695399/functions/pthread_rwlock_init.html
  Une autre façon de mutexer ses données, les read-write locks.
- http://www.roma1.infn.it/SIC/_OLD_documentazione/unix/migr/digital-unix-doc/DOCUMENTATION/HTML/AA-Q2DPC-TKT1_html/thrd0038.html
  Une présentation de la notion de mutex récursif (voir également la manpage de `pthread_mutex_init`), ainsi que http://stackoverflow.com/questions/187761/recursive-lock-mutex-vs-non-recursive-lock-mutex et http://www.zaval.org/resources/library/butenhof1.html deux discussions sur leur utilité et celle des mutexes non récursifs.
- http://bartoszmilewski.wordpress.com/2009/06/02/race-free-multithreading-ownersh
  Une discussion avancée sur le partage des objets dans des langages impératifs.
- http://web.newsguy.com/sab123/tpopp/
  "The practice of Parallel programming" un livre entier consacré à la programmation parallèle et concurrente.
- http://drdobbs.com/go-parallel/article/showArticle.jhtml?articleID=219200099
  Pourquoi la programmation parallèle est importante, par Herb Sutter.
- http://johannes-fetz.blogspot.com/2010/02/intel-threading-building-blocks.html
  Une présentation par le Koalab d'Intel TBB, un framework d'Intel pour favoriser le calcul parallèle. La vidéo de cette présentation est disponible à l'adresse : http://koalab.epitech.net/files/inteltbb_1_0.flv
- https://computing.llnl.gov/tutorials/openMP/
  Un tutorial pour OpenMP, une solution de multithreading par le compilateur.
- http://en.wikipedia.org/wiki/Software_transactional_memory
  Une présentation du mécanisme de Mémoire Transactionnelle, une autre façon de réaliser le partage de données et le contrôle d'accès dans du code multithread.
- http://valgrind.org/docs/manual/hg-manual.html
  La documentation d'Helgrind, un outil de détection d'erreurs courantes en programmation multithread.