ulm university universität
uulm

Institut für Numerische Mathematik

Dr. Andreas F. Borchert

12. Mai 2017
Blatt 4

Parallele Programmierung mit C++ (SS 2017)

Abgabe bis zum 19. Mai 2017, 14:00 Uhr

Lernziele:

- Entwicklung neuer Synchronisierungsmechanismen auf der Basis von Bedingungsvariablen

Aufgabe 7: Flexibler warten

Die Bedingungsvariablen der Thread-Bibliothek (und gleichmaßen auch die der darunterliegenden POSIX-Threads-Bibliothek) haben die Einschränkung, dass jeweils nur auf das Eintreten einer Bedingung gewartet werden kann. Es ist aber durchaus möglich, auf Basis dieser unteren Ebene eine Schicht aufzubauen, die das Warten auf eine Menge von Bedingungsvariablen zulässt, das beendet wird, sobald eine der Bedingungen zutrifft.

Um dies umzusetzen, empfiehlt sich folgender Ansatz:

- Innerhalb der neuen Schicht gibt es eine neue Klasse für Bedingungsvariablen namens *Condition*. Anders als zuvor ist ein direktes Warten darauf nicht vorgesehen. Stattdessen können solche Bedingungsvariablen in beliebig viele Mengen aufgenommen werden.

- Eine Menge von Bedingungsvariablen wird durch die neue Klasse *ConditionSet* repräsentiert. Diese offeriert auch eine *wait*-Methode, die solange den aufrufenden Thread suspendiert, bis eine der Bedingungsvariablen aus der Menge notifiziert wird.

- Die Implementierung sieht so aus, dass jeder *ConditionSet* eine Bedingungsvariable der Thread-Bibliothek hat. Wenn *wait* aufgerufen wird, registriert sich das *ConditionSet*-Objekt bei allen in ihr enthaltenen Bedingungsvariablen und wartet dann auf ihre private Bedingungsvariable.

- Die einzelnen Bedingungsvariablen des Typs *Condition* verwalten jeweils die *ConditionSet*-Objekte, die gerade auf sie warten. Bei *notify_all* wird dann bei allen sie referenzierenden Mengen die Methode *notify* aufgerufen, bei *notify_one* nur einer.

- Die *notify*-Methode der Klasse *ConditionSet* ruft dann *notify_all* bei der privaten Bedingungsvariablen auf.

Dabei sind allerdings folgende Punkte zu beachten:

- Die Methode *wait* der Klasse *ConditionSet* benötigt genauso wie die *wait*-Methode der *std::condition_variable* einen Lock-Parameter und es ist zwingend notwendig, dass die Suspendierung und die Freigabe des Locks atomar erfolgen. Dies lässt sich nur erreichen, indem der als Parameter übergebene Lock direkt an das *std::condition_variable*-Objekt weitergegeben wird.
- Damit die *notify_one*-Methode von *Condition* korrekt funktioniert, müssen nicht nur die bereits notifizierten Mengen aus der Verwaltung herausgeworfen werden, sondern auch alle Mengen, die bereits dekonstruiert wurden. Dies lässt sich am leichtesten durch die Verwendung schwacher Zeiger (*std*::*weak_ptr*) erreichen.

So könnte eine Schnittstelle aussehen:

```
#ifndef CONDITION_HPP
#define CONDITION_HPP

#include <condition_variable>
#include <list>
#include <memory>
#include <mutex>

class Condition;
using ConditionPtr = std::shared_ptr<Condition>;

class ConditionSet;
using ConditionSetPtr = std::shared_ptr<ConditionSet>;
using WeakConditionSetPtr = std::weak_ptr<ConditionSet>;

class Condition {
    public:
        /* notify all dependent condition sets */
        void notify_all();

        /* notify just one dependent condition set */
        void notify_one();

        /* invoked by condition sets that include this condition
            _and_ wait for it to become true
        */
        void enlisten(ConditionSetPtr cset);

    private:
        std::mutex mutex;
```

```
        /* condition sets that are currently waiting for us */
        std::list<WeakConditionSetPtr> dependents;
        /* common part of notify_all and notify_one */
        void notify(bool justone);
};

class ConditionSet: public std::enable_shared_from_this<ConditionSet> {
    public:
        /* include the given condition into the set */
        void include(ConditionPtr condition);

        /* wait until one of the condition variables gets notified;
            suspension and release of the lock are atomic; the lock
            is restored upon return;
            this is a no-operation if the set is empty
        */
        void wait(std::unique_lock<std::mutex>& lock_of_caller);

        /* this method is invoked by one of the included conditions
            to wake up from a call to wait();
            this is a no-operation if wait() has not been called
        */
        void notify();

    private:
        std::mutex mutex; // assure mutual exclusion
        std::list<ConditionPtr> conditions; // set of conditions
        std::condition_variable notified;
};

#endif
```

Einige Hinweise zu der vorgestellten Schnittstelle:

- Die Schnittstelle arbeitet durchgängig mit intelligenten Zeigern. Dies automatisiert das korrekte Aufräumen.
- Da *enlisten* einen Zeiger erwartet und somit die *wait*-Methode in *ConditionSet* einen intelligenten Zeiger auf sich selbst liefern muss, ist die Basis-Klasse *std::enable_shared_from_this* hilfreich, die hierzu die Methode *shared_from_this()* anbietet.
- Da *notify_all* und *notify_one* sich weitgehend gleichen, wurde die Gemeinsamkeit in die private Methode *notify* herausfaktorisiert. Achten Sie darauf, dass *notify* die mittlerweile dekonstruierten Bedingungsmengen übergeht und aus der Datenstruktur herauswirft.

So könnte dann eine Benutzung aussehen in einer Methode, die auf die Bedingungsvariable *a* oder *b* wartet:

```
void do_sth() {
    std::unique_lock<std::mutex> lock(mutex);
    // ...
    auto cset = std::make_shared<ConditionSet>();
    cset->include(a);
    cset->include(b);
    cset->wait(lock);
    // ...
}
```

Dabei wären wie bisher die *mutex*-Variable und die Bedingungsvariablen private Variablen der Klasse:

```
private:
    std::mutex mutex;
    ConditionPtr a;
    ConditionPtr b;
    // ...
```

Sie sind aber frei, diese Schnittstelle zu ändern. Einzureichen ist die Schnittstelle, die Implementierung und ein kleines Testprogramm:

```
submit pp 7 condition.hpp condition.cpp testit.cpp
```

Viel Erfolg!