ulm university universität
uulm

Institut für Numerische Mathematik

Dr. Andreas F. Borchert

2. Juni 2017
Blatt 6

Parallele Programmierung mit C++ (SS 2017)

Abgabe bis zum 9. Juni 2017, 14:00 Uhr

Lernziele:

- Gestaltung von Containern mit konkurrierenden Zugriffen
- Umgang mit Mutex- und Lock-Variablen, die zwischen Lese- und Schreibzugriffe unterscheiden

Aufgabe 9: Hash-Tabelle mit konkurrierenden Zugriffen

Nicht jede klassische Datenstruktur für Container eignet sich gleichermaßen für den konkurrierenden Zugriff durch mehrere Threads. Natürlich ist es trivialerweise möglich, einen der klassischen Container der STL einzupacken in eine Klasse, die sämtliche Aufrufe mit Hilfe eines *std::mutex*-Objekts und eines Locks absichert. Bei Datenstrukturen mit konkurrierenden Zugriffen wäre es jedoch attraktiv, wenn

- mehrere Threads gleichzeitig daraus lesen können, wenn gerade kein Thread darauf schreiben möchte, und wenn
- der gegenseitige Ausschluss feiner granuliert wird, so dass nicht die gesamte Datenstruktur durch einen Thread blockiert wird.

Hierfür bieten sich Hash-Verfahren an, weil die Bucket-Tabelle eine ideale Parallelisierungsmöglichkeit bietet, d.h. es muss zu einem Zeitpunkt für einen Thread nur jeweils für eine der linearen Listen ein exklusiver Zugang (bzw. ein gemeinsamer Lesezugriff) zugesichert werden.

Im nächsten geplanten C++-Standard, der vorläufig C++17 genannt wird, wird die Einführung von *std::shared_mutex* geplant analog zum bereits jetzt existierenden *boost::shared_mutex*. So sieht die typische Anwendung aus:

```
#include <mutex>
#include <boost/thread/locks.hpp>
#include <boost/thread/shared_mutex.hpp>
```

```
class Container {
    public:
        /* ... */
        void access(...) const {
            boost::shared_lock<boost::shared_mutex> lock(mutex);
            /* read−only access of the private data */
        }
        void update(...) {
            std::unique_lock<boost::shared_mutex> lock(mutex);
            /* exclusive write access of the private data */
        }
    private:
        mutable boost::shared_mutex mutex;
        /* ... */
};
```

Wenn Sie auf der Thales mit der Boost-Bibliothek arbeiten, muss Ihre Lösung so übersetzt werden:

```
g++ ... -pthreads -lboost_thread -lboost_system
```

Bei g++ ab der Version 6.1 wird der erwartete Standard C++17 vorausschauend unterstützt. Die Anwendung würde dann so aussehen:

```
#include <mutex>
#include <shared_mutex>

class Container {
    public:
        /* ... */
        void access(...) const {
            std::shared_lock<std::shared_mutex> lock(mutex);
            /* read−only access of the private data */
        }
        void update(...) {
            std::unique_lock<std::shared_mutex> lock(mutex);
            /* exclusive write access of the private data */
        }
    private:
        mutable std::shared_mutex mutex;
        /* ... */
};
```

Beim Hash-Verfahren lassen sich diese Locks jedoch granuliert anwenden, indem eine solche *mutex*-Variable für jede lineare Liste eines Bucket-Eintrags verwaltet wird. Auf unserem

Server Theon steht experimentell bereits g++ in der Version 6.3.0 zur Verfügung. Damit dieser im *PATH* erscheint, muss vorher die Datei *~/.options* um den Eintrag „athenry“ ergänzt werden. Beim Übersetzen benötigen Sie dann die Option „-std=c++17“.

Im Rahmen dieser Aufgabe ist eine entsprechende Hash-Verwaltung für konkurrierende Zugriffe zu entwickeln, die nur die elementaren Zugriffsmethoden unterstützen muss, die etwa folgendermaßen aussehen könnten:

void *add_or_update*(const *Key*& *key*, const *Value*& *value*)
: Falls es bereits einen Eintrag mit dem Schlüssel *key* existiert, wird der zugehörige Wert aktualisiert. Andernfalls wird ein neuer Eintrag mit dem Paar $(key, value)$ in die Tabelle eingefügt.

void *remove*(const *Key*& *key*)
: Entfernt den Eintrag mit dem Schlüssel *key*, falls existent.

bool *lookup*(const *Key*& *key*, *Value*& *value*)
: Wenn ein Eintrag mit dem Schlüssel *key* existiert, wird *value* auf dessen Wert gesetzt und *true* zurückgeliefert, andernfalls *false*.

Wenn Sie das als Template-Klasse konzipieren, kann diese von den Datentypen *Key* und *Value* abhängen, wobei auch die jeweils passende Hash-Funktion berücksichtigt werden kann:

```
template<typename Key, typename Value, typename Hash = std::hash<Key>>
class ConcurrentHash {
    public:
        /* ... */
        void add_or_update(const Key& key, const Value& value) {
            std::size_t index = hash(key) % nofbuckets;
            /* ... */
        }
    private:
        const Hash hash;
        const std::size_t nofbuckets;
        /* ... */
};
```

Eine Hash-Funktion ist ein Funktionsobjekt, das einen Wert des Schlüsseltyps entgegennimmt und *std*::*size_t* zurückliefert. Die Template-Klasse *std*::*hash* dient dazu, die Standard-Implementierungen der Hash-Funktionen insbesondere für die vordefinierten Datentypen elegant zugänglich zu machen. Die Template-Klasse kommt mit zwei Template-Parametern aus und kann dann die passende voreingestellte Hash-Funktion auswählen. Es ist aber auch möglich, Alternativen explizit zu benennen.
Fügen Sie noch ein kleines Testprogramm hinzu, verpacken Sie all Ihre Quellen mit *tar* in ein Archiv und reichen dies ein:

```
tar cvf chash.tar *.?pp [mM]akefile
submit pp 9 chash.tar
```

Viel Erfolg!