# C++11/14 Multithreading: Überblick, Highlights und Fallstricke

Karl Nieratschker, SKT Nieratschker

**Seit der Einführung von C++11 bietet die Standardbibliothek von C++ auch Unterstützung für die Entwicklung von Multithread-Applikationen. Im neuesten Standard C++14 wurde diese Funktionalität sogar noch erweitert. Die Verwendung des C++-Multithread-APIs vereinfacht zwar die Portierung derartiger Anwendungen, führt aber gleichzeitig auch dazu, dass man sich auf die Möglichkeiten der Standardbibliothek beschränken muss, wenn man davon profitieren möchte. Nicht nur bei der Entwicklung von neuen Applikationen, sondern auch für existierende Anwendungen, die noch auf plattformspezifischen Multithread-Lösungen basieren, stellt sich deshalb die Frage, ob es sinnvoll ist, dieses API einzusetzen bzw. darauf umzustellen. Der Vortrag gibt einen Überblick über den Leistungsumfang des C++-Multithread-APIs und zeigt, was bei der Portierung von Applikationen zu beachten ist.**

Beim Entwurf der Multithread-Standardbibliothek von C++11 wurden viele Elemente aus der weit verbreiteten C++-Boost-Bibliothek übernommen und weitere Funktionalitäten hinzugefügt. Kenner von Boost sollten allerdings darauf achten, dass nicht alles übernommen wurde, und dass die übernommen Dinge nicht immer exakt gleich umgesetzt wurden. Bei der Implementierung der Bibliothek wurde intensiv von den neuen C++11-Spracheigenschaften wie z.B. Variadic Templates, Rvalue-Referenzen und Lambda-Funktionen Gebrauch gemacht.

## Die Klasse thread und ihre Möglichkeiten

Ein Thread wird repräsentiert durch eine Instanz der Klasse thread, bei deren Erzeugung der vom Thread auszuführende Code in Form einer globalen Funktion, einer Instanz- oder Klassenmethode, eines Funktors oder einer Lambda-Funktion angegeben werden kann. Der Konstruktor sorgt für die Erzeugung eines Laufzeit-Threads, der den Code unmittelbar ausführen kann. Der Destruktor der Thread-Klasse prüft, ob der zugehörige Laufzeitthread noch läuft und wirft gegebenenfalls eine Exception. Um dies zu verhindern, kann entweder mithilfe der Methode join() auf das Ende des Threads gewartet, oder der Laufzeitthread vom C++-Threadobjekt mithilfe der Methode detach() entkoppelt werden. Die auszuführende Threadfunktion kann beliebig viele Parameter beliebigen Typs haben, die immer kopiert oder transferiert werden, um eine ausreichend lange Lebensdauer zu gewährleisten (**Bild 1**).

```cpp
#include <thread>
#include <iostream>
using namespace std;

//////////////////////////////////////////////////////////

void thread_code(const char* text, int thread_nr)
{
    cout << text << ' ' << thread_nr << endl;
}

//////////////////////////////////////////////////////////

int main()
{
    thread t1(thread_code, "Hello from thread", 1);
    thread t2(thread_code, "This is thread", 2);
    thread t3(thread_code, "My thread number is:", 3);

    ...
}
```

**Bild 1:** Drei unterschiedlich parametrierte Threads mit identischer Threadfunktion

Jeder Thread besitzt zu seiner Identifikation eine ID vom Typ thread::id. Da die ID plattformspezifisch ist, unterstützt der Typ nur wenige Operationen, wie z.B. get_id() als Methode der Klasse thread und als Funktion des Namensraums this_thread, sowie Vergleichsoperatoren. Weitere Funktionen des Namensraums this_thread sind sleep_for() und sleep_until(), mit denen ein Thread für eine relative bzw. absolute Zeit schlafen kann, sowie die Funktion yield(), durch deren Ausführung ein Thread den Rest seiner Zeitscheibe abgibt. Auch threadlokaler Speicher wird von C++11 unterstützt, allerdings wesentlich intuitiver als dies in Boost der Fall war. Schließlich liefert die statische Methode hardware_concurrency() die Anzahl der aktuell zur Verfügung stehenden Hardware-Ausführungseinheiten (Cores). Allerdings darf diese Funktion auch Null liefern, so dass der von ihr gelieferte Wert nur als Hinweis verstanden werden darf.

**Synchronisation**

C++11 unterstützt zum Schutz von Ressourcen neben „normalen“ Mutexen vom Typ mutex auch solche, die von einem Thread mehrfach (nichtblockierend) angefordert werden können (recursiv_mutex), sowie solche, die mit Fehler zurückkehren, wenn das Mutex nicht innerhalb einer gewissen Zeit angefordert werden kann (timed_mutex, recursive_timed_mutex). Mit C++14 wurde für Multiple-Reader/Single-Writer-Anwendungen auch das shared_mutex von Boost als shared_timed_mutex in den Standard aufgenommen. Mit den Klassen lock_guard, unique_lock und shared_lock (C++14) können Deadlock-Probleme aufgrund von fehlenden Mutexfreigaben zuverlässig verhindert werden (**Bild 2**).

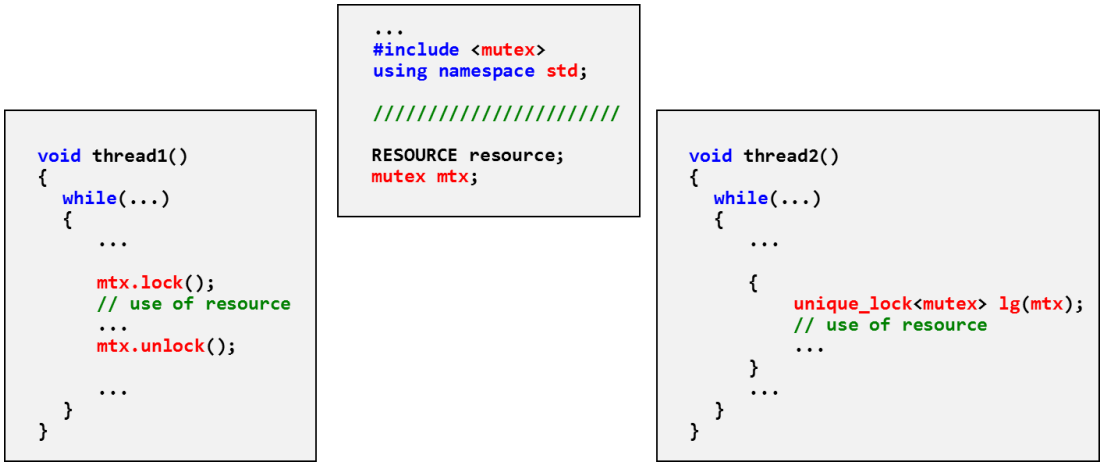


**Bild 2:** Geschützter Zugriff auf eine gemeinsame Ressource
mit mutex und unique_lock

Eines der größten Highlights der C++11-Multithread-Bibliothek sind die Atomics. Mit ihrer Hilfe können Standarddatentypen wie z.B. int oder float als atomar deklariert werden. Die Operationen einer Variablen vom Typ atomic<int> (bzw. atomic_int für C-Programme) sind unteilbar und müssen somit nicht mehr explizit z.B. durch ein Mutex geschützt werden (**Bild 3**).

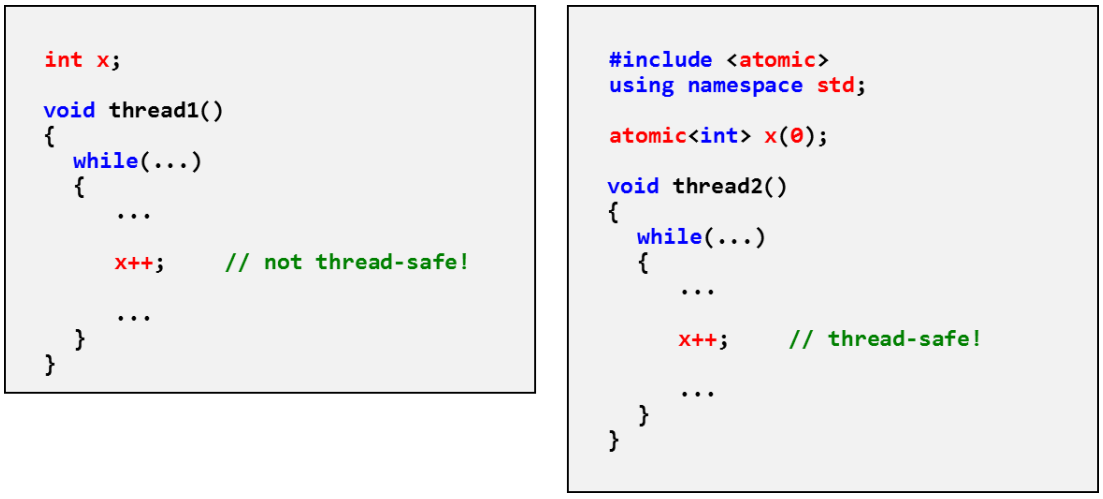


**Bild 3:** Sicheres Inkrementieren einer Variablen mit atomic

Außerdem wurde ein neues Speichermodell eingeführt, mit dessen Hilfe klassische Multithreading-Probleme, ausgelöst durch Compiler- oder Prozessoroptimierungen, mithilfe von Speicherbarrieren und Atomic-Variablen zuverlässig gelöst werden können. Darüber hinaus schaffen die Atomics mithilfe ihrer compare_exchange-Methoden die Voraussetzungen für lockfreie Programmierung. Mit Ausnahme des

Datentyps atomic_flag garantiert der Standard nicht, dass die Operationen eines Atomic-Datentyps lockfrei sind. Dies kann aber mithilfe der Methode is_lock_free() geprüft werden. Grundsätzlich ist es sogar möglich, eigene atomare Datentypen zu definieren.

Für die Ereignissynchronisation stellt C++11 Condition-Variable zur Verfügung. Mithilfe der wait-Methode der Condition-Variable kann ein Thread darauf warten, dass eine Bedingung erfüllt ist. Führt ein anderer Thread Code aus, durch den die Bedingung erfüllt wird, dann muss nach der Ausführung dieses Codes die Methode notify_one der Condition-Variable aufgerufen werden, um den wartenden Thread aufzuwecken. Mit notify_all können sogar mehrere wartende Threads gleichzeitig aufgeweckt werden. In jedem Fall muss ein Thread nach der Rückkehr aus dem Wait die Bedingung erneut prüfen, da das Laufzeitsystem den Thread auch aus anderen Gründen aufwecken kann.

Schließlich kann mit der Funktion call_once dafür gesorgt werden, dass eine Funktion garantiert nur ein einziges Mal ausgeführt wird, unabhängig davon, wie oft und von wie vielen Threads sie aufgerufen wird. Normalerweise wird dies im Zusammenhang mit Initialisierungen benötigt.

**Futures**

Der Returnwert einer C++11-Threadfunktion kann nicht zum Liefern eines Ergebnisses verwendet werden, wie dies bei vielen andern Multithreading-Plattformen der Fall ist. Stattdessen muss ein Objekt vom Typ future<T> verwendet werden. Es dient zum Zwischenspeichern des Ergebnisses, wenn der Empfänger-thread noch nicht bereit ist, bzw. zum Blockieren des Empfängerthreads, wenn das Ergebnis noch nicht zur Verfügung steht. Es gibt 3 verschiede Anwendungs-möglichkeiten.

Die Anwendungsvariante mit dem niedrigsten Abstraktionsgrad besteht darin, ein promise<T>-Objekt zu erzeugen und dem Thread verfügbar zu machen, der das Ergebnis ermittelt, das er dann mit der Methode set_value() an das Promise-Objekt übergibt. Der Empfänger besorgt sich mit der Methode get_future() des Promise-Objektes ein Future-Objekt. Nach Abschluss eventueller Parallelarbeiten kann er dann auf das Ergebnis mit der Methode get() des Future-Objektes zugreifen, wobei er bei Bedarf blockiert wird, bis das Ergebnis verfügbar ist.

Bei der nächsthöheren Abstraktionsebene wird ein packaged_task-Objekt erzeugt und mit einer Funktion parametriert, die das Ergebnis als normalen Returnwert liefert. Die get_future-Methode des Task-Objektes liefert wieder das zugehörige Future-Objekt. Um Nebenläufigkeit zu erzielen, muss wieder ein Thread erzeugt und mit dem Task-Objekt parametriert werden. Die Ergebnisübergabe erfolgt wie bei der ersten Lösung.

Bei der höchsten Abstraktionsebene wird nur noch mit der Funktion async der Code der Funktion angegeben, die das ermittelte Ergebnis mithilfe eines normalen

Returnwerts liefert. Ob die Funktion asynchron in einem eigenen Thread oder synchron im Kontext des Threads, der das Ergebnis anfordert, ausgeführt wird, kann dem Laufzeitsystem überlassen oder mithilfe eines Parameters festgelegt werden. async liefert das Future-Objekt unmittelbar als Returnwert (**Bild 4**).

```
int async_operation(int multiplier)
{
   sleep_for(seconds(2));     // simulate work
   return 42*multiplier;
}
```

```
#include <future>
...
int main()
{
   future<int> future = async(launch::async,async_operation,3);
   ...     // potentially concurrent work
   cout << future.get() << endl;

}
```

**Bild 4:** Nebenläufige Ausführung einer Funktion mit async und Übergabe des Ergebnisses mithilfe von future<int>

Das Ergebnis eines future<T>-Objektes darf nur ein einziges Mal gelesen werden. Benötigen mehrere Threads dasselbe Ergebnis, dann muss mit shared_future<T>-Objekten gearbeitet werden.

**Dinge, die der Standard nicht unterstützt**

Anders als z.B. beim Posix/Pthread-Standard bietet das C++11/14-Multithread-API keine unmittelbare Möglichkeit, den Threads unterschiedliche Prioritäten zu geben, oder Mutexe mit Prioritätsvererbung oder Ceiling-Priorität zu erzeugen. Dies ist insbesondere für die Anwendung in Echtzeitsystemen eine gravierende Einschränkung. Zwar kann man sich mit der Methode native_handle() der Klasse thread das Handle des zugehörigen Laufzeitthreads besorgen und mit dessen Hilfe die Priorität des Threads auf Betriebssystemebene festlegen. Aber dieser Lösungsansatz geht zu Lasten der Portierbarkeit.

Darüber hinaus bietet C++11/14 auch keine Unterstützung dafür, einen Thread asynchron unterbrechen oder beenden zu können. Aufgrund der damit verbundenen Komplexität sollte man zwar generell versuchen, ohne diese Dinge auszukommen, in der Praxis ist dies aber oft schwierig. Auch in diesem Bereich bietet der Posix/Pthread-Standard eine gute Unterstützung.

Weitere Dinge, die Kenner anderer Multithreadinglösungen bei C++11/14 vermissen könnten, sind z.B. allgemeine (zählende) Semaphore, sowie effiziente Eventflag-Mechanismen zur Signalisierung von reinen Ereignissen, bei denen keine Datenzugriffe synchronisiert werden müssen.

**Dinge, die bei der Portierung beachtet werden müssen**

Generell muss man sich beim Portieren von Multithread-Applikationen bewusst sein, dass die Schedulingmechanismen der verschiedenen Plattformen im Detail sehr unterschiedlich sein können. Das betrifft nicht nur das Threadzuteilungsverhalten des Schedulers selbst, sondern z.B. auch das Fairness-Verhalten von Synchronisationsobjekten. Abgesehen davon, dass manche Dinge vielleicht zu langsam sind, was speziell für Echtzeitsysteme problematisch ist, sollte eine portierte Applikation aber ohne Änderung prinzipiell auf jeder unterstützten Plattform funktionieren. Wenn dies nicht so ist, dann liegt das oft daran, dass das Design spezifische Eigenschaften der ursprünglichen Plattform implizit als „immer vorhanden“ vorausgesetzt hat.

Verhaltensunterschiede basieren mitunter auch darauf, dass der Standard von den Compilerherstellern unterschiedlich implementiert wurde. Abgesehen von echten Implementierungsfehlern kann das auch daran liegen, dass der Standard nicht alles bis ins letzte Detail festlegt. Wird z.B. ein Mutex von einem Thread freigegeben, der das Mutex gar nicht besitzt, dann hängt es von der Implementierung ab, wie auf diesen Programmfehler reagiert wird. Sowohl bei GCC 4.9.2 unter Linux, als auch bei der Releaseversion bei Visual Studio 2015 unter Windows wird das Mutex z.B. trotzdem einfach freigegeben, während der von der Debugversion von Visual Studio generierte Code eine Exception auslöst (**Bild 5**).

```
mutex m;
int main()
{
   m.lock();
   thread t([]{ m.unlock(); }); // thread is not owner!!
   t.join();
}
```

**Bild 5:** Programmbeispiel für Verhaltensunterschiede bei verschiedenen Implementierungen

**Literaturhinweis:**

**„C++ Concurrency in Action”**, Anthony Williams

**Autor**

Basierend auf einem Studium der technischen Informatik verfügt Karl Nieratschker über eine mehr als 30-jährige Erfahrung im Bereich der Embedded-Programmierung und betriebssystemnahen Softwareentwicklung. Als selbständiger Trainer, Softwareberater und Coach unterstützt er seit 1999 Kunden bei der Einführung neuer Softwaretechnologien. Besondere Schwerpunkte dabei sind „Objektorientierte Programmierung in Embedded Systemen“, sowie „Multithreading-“ und „Multicore-Programmierung“.

Internet: www.skt-nieratschker.de
Email: office@skt-nieratschker.de