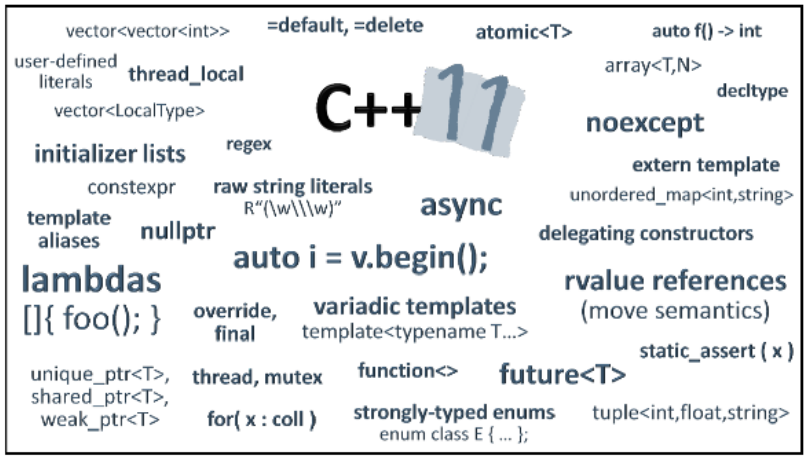
Freie Universität Berlin



Threads und Shared Variables in C++11
Memory Model

Kevin Funk
Freie Universität Berlin

Seminar Programmiersprachen 2012

Inhalt

Outline

Motivation

Motivation

Historie

- C und C++ wurden entwickelt bevor Multithreading populär wurde
- Weder der C98, noch der C++98-Standard erwähnen *threads*
- Multithreading ist nicht Bestandteil der Sprachspezifikation

Heute: C11/C++11 (vorher: C++0x)

- Interessanteste Neuerung im Standard: Multithreading
- Erstmals spezifiziert der Standard das Verhalten bei nebenläufigem Code
- Ohne auf Plattform-/Compiler-spezifische Erweiterungen zurückgreifen zu müssen
- Weitere wichtige Bestandteile: Memory Model und Atomics

(Quelle Titelbild: http://www.dotnetrangers.net/2012/01/05/speaker-for-c-11-session-during-techdays-2012-in-paris/)

Outline

Outline

Was ist ein Prozess?

In seiner einfachsten Form ist ein Prozess ein einfaches Programm

- Ein Prozess stellt Resourcen für ein Programm bereit
- Resourcen wie zum Beispiel...
  - Virtueller Addressraum für Speicherallokationen
  - File handles
  - Eindeutige Prozess-ID (PID)
  - Umgebungsvariablen
  - Eine Priorität (für Scheduling)
  - (...)

Jeder Prozess wird mit einem einzigen Thread gestarted, dem *Hauptthread* (main thread).

Was sind Threads?

- Ein Thread ist ein Teil eines Prozesses
- Ein Prozess kann wiederum mehrere Threads beinhalten
- Jeder Thread hat eine eindeutige ID
- Alle Threads haben folgendes gemeinsam:
  - Den virtuellen Addressraum des Vater-Prozesses
  - Globale Variablen
  - System-Resourcen (z.B. File handles)
- Jeder Thread verbraucht aber noch eigene Resourcen
  - Thread-local storage
  - Stack
  - Registerzustände
  - (...)
- Der Zustand des Threads wird im *thread context* festgehalten (Wichtig um *context switching* später zu realisieren)

Zusammenspiel von Prozessen und Threads

Beispiel: Prozess mit zwei Threads

- Abfolge der Ausführungen auf der y-Achse
- (hier: Single-Core-CPU)

Abfolge:

- Thread 1 wird bearbeitet
- Thread 1 wird schlafen gelegt (suspend)
- Context switch
- Thread 2 wird bearbeitet
- Thread 2 wird schlafen gelegt / beendet
- Context switch
- Thread 1 wird weiter ausgeführt

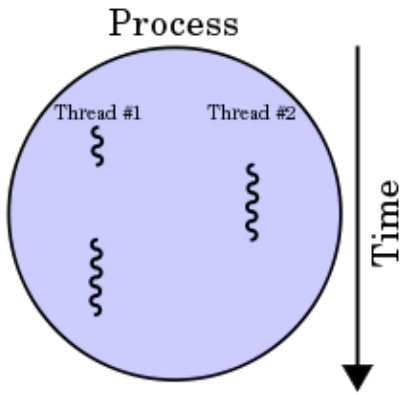


Abbildung: Prozess mit Threads

Bildquelle: http://en.wikipedia.org/wiki/Thread_%28computing%29

Outline

Nebenläufigkeit in Computersystemen

Nebenläufigkeit im Allgemeinen:

- Ein Subjekt führt mehrere Aktivitäten parallel aus
- Beispiel: Mensch kann gleichzeitig reden und laufen

Nebenläufigkeit in Computersystemen

Nebenläufigkeit im Allgemeinen:

- Ein Subjekt führt mehrere Aktivitäten parallel aus
- Beispiel: Mensch kann gleichzeitig reden und laufen

Nebenläufigkeit in Computersystemen:

- Mehrere Tasks können parallel ausgeführt werden
- Beispiel: E-Mail-Programm und Instant-Messenger (beides bestimmte Tasks) laufen gleichzeitig
- Aber: Unterschiede beim *Task scheduling*

Task scheduling

Unterschiede zwischen Einfach- (Single-) und Mehrprozessorsystemen (Multi-Core)

- Single-Core-CPUs
  - Tasks laufen nur *scheinbar* parallel ab
  - Es kann jeweils nur ein Task zu einem Zeitpunkt bearbeitet werden
  - ⇒ *Task scheduling* mit *context switches*
- Multi-Core-CPUs
  - Tasks können echt-parallel ausgeführt werden (*true concurrency*)
  - ⇒ *hardware concurrency*

Single-Core vs. Multi-Core

Beispiel: Single-Core- vs. Multi-Core (hier: Dual-Core)

- Zwei Tasks
- Je Task jeweils 10 gleichgroße *Chunks*

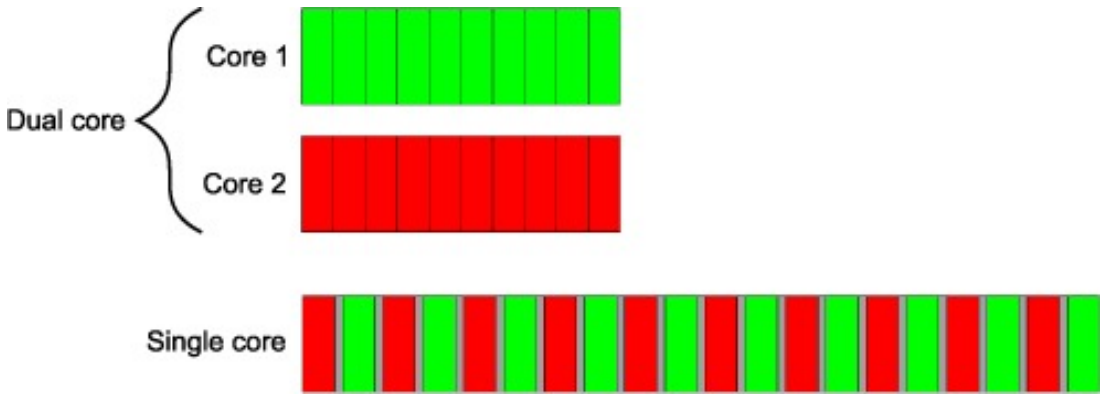


Abbildung: Parallel Ausführung auf Multi-Core-System, Context switching auf Single-Core-System

Bildquelle: [Wil11, Chapter 1]

Single-Core vs. Multi-Core

Beispiel: Single-Core- vs. Multi-Core (hier: Dual-Core)

- Zwei Tasks
- Je Task jeweils 10 gleichgroße *Chunks*

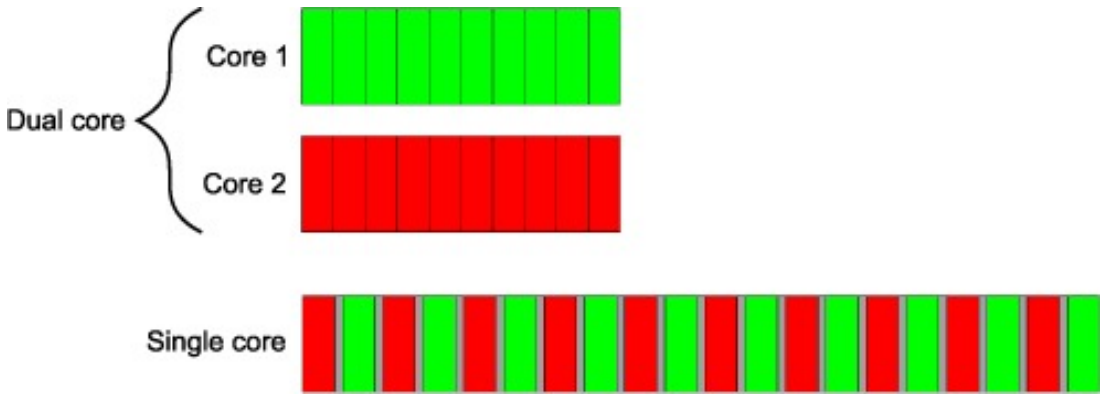


Abbildung: Parallel Ausführung auf Multi-Core-System, Context switching auf Single-Core-System

Achtung: Auf dem Single-Core muss bei jedem Taskwechsel ein *Context switch* vollzogen werden (kostet Zeit!)

Bildquelle: [Wil11, Chapter 1]

Wie ist Nebenläufigkeit implementiert?

Es gibt grundsätzlich zwei verschiedene Möglichkeiten:

- Nebenläufigkeit mit mehreren Prozessen
- Nebenläufigkeit mit mehreren Threads

Beide Möglichkeiten haben jeweils ihre Vor- und Nachteile

Nebenläufigkeit mit mehreren Prozessen

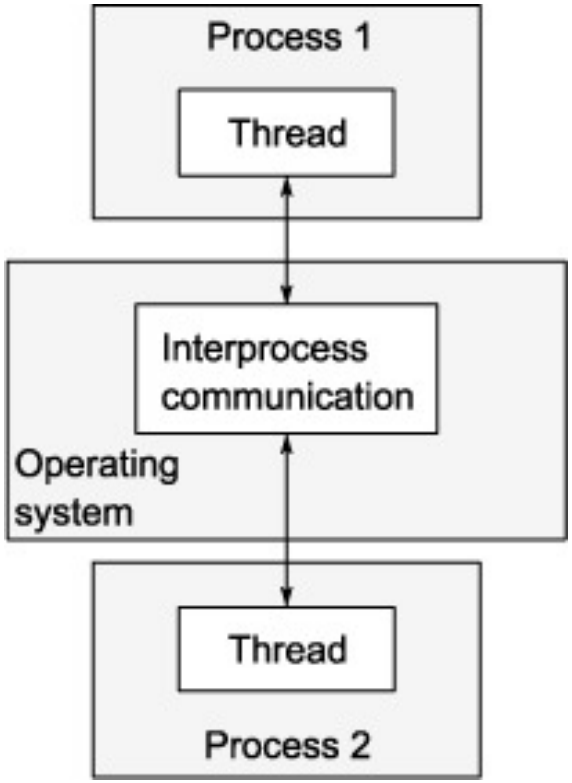


Abbildung: Zwei Prozesse mit jeweils einem Hauptthread. Kommunikation erfolgt über die Betriebssystemsschicht (OS layer)

Bildquelle: [Wil11, Chapter 1]

Nebenläufigkeit mit mehreren Prozessen (1)

- Eigenschaften
  - Einfach zu implementieren
  - Jeder Prozess hat eigenen Adressraum ⇒ sicher
  - Kommunikation erfolgt über vordefinierte Schnittstellen (Message queues, Sockets, lokale Dateien) ⇒ (relativ) sicher
- Nachteile
  - Schwergewichtig, benötigen Ressourcen vom Betriebssystem
  - Interprozesskommunikation ist teuer

Nebenläufigkeit mit mehreren Threads

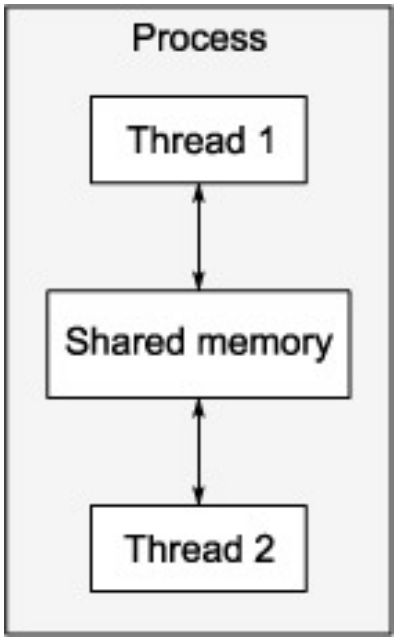


Abbildung: Ein Prozess mit mehreren Threads. Threads kommunizieren im Prozess über *shared memory* (im Normalfall)

Bildquelle: [Wil11, Chapter 1]

Nebenläufigkeit mit mehreren Threads (1)

- Eigenschaften
  - Leichtgewichtiger als Prozesse
  - Benötigen weniger Ressourcen (da kein neuer Prozesskontext)
  - Teilt Speicher mit Vaterprozess (*shared memory*)
  - Kommunikation mit anderen Threads über sog. *Synchronisationsvariablen* (werden später näher erläutert)
- Nachteile
  - Weniger bis gar kein Schutz zwischen Threads
  - Programmierer / Compiler muss auf *Speicherkonflikte* aufpassen (werden ebenfalls später behandelt)

Warum wollen wir überhaupt Nebenläufigkeit?

- Trennung von Zuständigkeiten (*Separation of concerns*)
  - Beispiel: DVD-Player-Applikation
  - Ein Thread zum Reagieren auf Benutzerevents
  - Ein anderer Thread dekodiert/rendert die DVD-Daten
  - Ziel: Die GUI soll ansprechbar bleiben

Warum wollen wir überhaupt Nebenläufigkeit?

- Trennung von Zuständigkeiten (*Separation of concerns*)
  - Beispiel: DVD-Player-Applikation
  - Ein Thread zum Reagieren auf Benutzerevents
  - Ein anderer Thread dekodiert/rendert die DVD-Daten
  - Ziel: Die GUI soll ansprechbar bleiben
- Nebenläufigkeit für bessere Performance
  - Vorher: Single-threaded Programme wurden ’automatisch’ mit jeder neuen Prozessorgeneration schneller
  - Heute: Immer mehr Multi-Core-System auf dem Markt
  - ⇒ Programme müssen für Nebenläufigkeit angepasst werden
  - ⇒ Algorithmen, die für parallele Berechnungen geeignet sind

Outline

- Vorher
  - Nebenläufigkeit ist bis dato (C++98) nicht im Standard erwähnt
  - Programmierer muss sich auf Plattform-/Compilererweiterungen stützen
  - Plattformerweitungen (Auszug): Microsoft Windows API, POSIX C Standard, ...
  - Plattformübergreifende Frameworks: Boost, ACE, Qt, ...
- Heute
  - C++11 bietet eine Threads API und entsprechende Sprachformalisierungen
  - D.h. es ist/wird möglich, ohne Erweiterungen nebenläufigen Code zu schreiben

Nebenläufigkeit im neuen Standard

Neuerungen in C++11:

- Threads API (Thread-Erstellung, Locks, etc.)
- Thread-aware memory model (formalisiert)
- Atomics (*shared variables*)

Nebenläufigkeit im neuen Standard

Neuerungen in C++11:

- Threads API (Thread-Erstellung, Locks, etc.)
- Thread-aware memory model (formalisiert)
- Atomics (*shared variables*)

(Bezieht sich natürlich nur auf Nebenläufigkeit, C++11 selbst hat einige mehr neue Features)

Outline

Outline

Credits

Kernentwickler am neuen C++11 memory model:

- Hans J. Boehm (HP)
  - Bekannt durch *Boehm garbage collector* (C++)
  - Veröffentlichungen bzgl. *C++ memory model* und *atomic operations*
- Herb Sutter (Microsoft)
  - Software-Architekt, Buchautor von *Exceptional C++*
  - Vorsitzender des ISO-C++-Standardisierungskomitees
- Und viele andere

Abbildung: Hans J. Boehm (http://www.hpl.hp.com/personal/Hans_Boehm/)

Formale Definition: Memory Model

The memory model, or memory consistency model, is at the heart of the concurrency semantics of a shared-memory program or system. It defines the set of values that a read in a program is allowed to return, thereby defining the basic semantics of shared variables

(Adve, Boehm, von [AB10, Introduction]).

Outline

Sequential Consistency: Ein einfaches Modell

- Sequential Consistency ist ein simples *memory consistency model*
- Am besten zu vergleichen mit der Ausführung von verschiedenen Threads auf einem Uniprozessor, also Single-Core, wo keine *echte* Parallelität herscht

Sequential Consistency: Beispiel

Dekker's Example:

```
1 // Thread 1
2
3 x = 1;
4 r1 = y;
```

```
1 // Thread 2
2
3 y = 1;
4 r2 = x;
```

Annahmen:

- x und y (Integertypen), sowie r1 und r2 sind zu Beginn auf 0 initialisiert
- Einsilbige Variablen sind Shared Variables, Variablen wie r1 oder r2 sind Thread-lokal

Die Frage ist nun: In welcher Reihenfolge werden die Statements unter diesem Modell durchgeführt?

Sequential Consistency: Beispiel (1)

Die Idee ist folgende:

- Unter Annahme von Sequential Consistency wird die Menge von Instruktionen zusammengeführt und zufällig verschachtelt (*interleaved*)

Die Ausgabe kann deshalb wie folgt aussehen (Auszug):

```
1 // Execution 1
2
3 x = 1;
4 r1 = y;
5 y = 1;
6 r2 = x;
7 // result:
8 // r1 == 0 and r2 == 1
```

```
1 // Execution 2
2
3 y = 1;
4 r2 = x;
5 x = 1;
6 r1 = y;
7 // result:
8 // r1 == 1 and r2 == 0
```

```
1 // Execution 3
2
3 x = 1;
4 y = 1;
5 r1 = y;
6 r2 = x;
7 // result:
8 // r1 == 1 and r2 == 1
```

Sequential Consistency: Beispiel (1)

Die Idee ist folgende:

- Unter Annahme von Sequential Consistency wird die Menge von Instruktionen zusammengeführt und zufällig verschachtelt (*interleaved*)

Die Ausgabe kann deshalb wie folgt aussehen (Auszug):

```
1 // Execution 1
2
3 x = 1;
4 r1 = y;
5 y = 1;
6 r2 = x;
7 // result:
8 // r1 == 0 and r2 == 1
```

```
1 // Execution 2
2
3 y = 1;
4 r2 = x;
5 x = 1;
6 r1 = y;
7 // result:
8 // r1 == 1 and r2 == 0
```

```
1 // Execution 3
2
3 x = 1;
4 y = 1;
5 r1 = y;
6 r2 = x;
7 // result:
8 // r1 == 1 and r2 == 1
```

Interessanterweise kann unter Model eine Ausgabe nicht möglich sein: r1 und r2 können nicht beide 0 sein, da immer entweder x oder y zunächst einen Wert zugewiesen bekommt und damit eine If-Bedingung erfüllt

Das Problem mit Sequential Consistency

- Modell ist in vielen Fällen zu einfach
- Verhindert Optimierungen, die eigentlich gewollt sind, wie etwa
  - Hardware-Optimierungen
  - Compiler-Optimierungen

Das Problem mit Sequential Consistency (1)

Beispiel: *Store-buffer forwarding* (hardware optimization)

- Die meisten Plattformen besitzen Caches (z.B. L1-Cache)
- Damit wird bspw. bei $x = 1$ nur ein *store buffer* aktualisiert
- Dieser *store buffer* ist aber nur in diesem Thread sichtbar
- Der andere Thread bekommt davon nichts mit

Beispiel: *Early-load* (compiler transformation)

- Compiler zieht *load*-Befehl in $r1 = y$ nach vorne
- Liefert mehr Zeit bis der Wert in $r1$ gebraucht wird

Das Problem mit Sequential Consistency (1)

Beispiel: *Store-buffer forwarding* (hardware optimization)

- Die meisten Plattformen besitzen Caches (z.B. L1-Cache)
- Damit wird bspw. bei $x = 1$ nur ein *store buffer* aktualisiert
- Dieser *store buffer* ist aber nur in diesem Thread sichtbar
- Der andere Thread bekommt davon nichts mit

Beispiel: *Early-load* (compiler transformation)

- Compiler zieht *load*-Befehl in $r1 = y$ nach vorne
- Liefert mehr Zeit bis der Wert in *r1* gebraucht wird

Beide Optimierungen führen zu den gleichen Auswirkungen: Es kann sein, dass wir $r1 == r2 == 0$ erhalten.

Das Problem mit Sequential Consistency (2)

Weiteres Beispiel: Speichergranularität:

```
1 // Thread 1
2
3 x = 300;
```

```
1 // Thread 2
2
3 x = 100;
```

Annahme:

- Speicher kann nur Byte-weise adressiert werden

Führt möglicherweise zu folgender Ausführung:

```
1 // Thread 1
2    // Thread 2
3
4    x_high = 0;
5 x_high = 1; // x = 256
6 x_low = 44; // x = 300;
7    x_low = 100; // x = 356;
```

Das Problem mit Sequential Consistency (2)

Weiteres Beispiel: Speichergranularität:

```
1 // Thread 1
2
3 x = 300;
```

```
1 // Thread 2
2
3 x = 100;
```

Annahme:

- Speicher kann nur Byte-weise adressiert werden

Führt möglicherweise zu folgender Ausführung:

```
1 // Thread 1
2    // Thread 2
3
4    x_high = 0;
5 x_high = 1; // x = 256
6 x_low = 44; // x = 300;
7    x_low = 100; // x = 356;
```

x == 356 ist kein gültiges Ergebnis bzgl. Sequential Consistency!

Das Problem mit Sequential Consistency (3)

- Probleme
  - Compiler, bzw. Hardware kann nicht feststellen ob die Variablen in Beziehung zu einander stehen
- Schlussfolgerung
  - Optimierungen in Hardware und Software führen zu Problemen bzw. unterschiedlichem Verhalten in nebenläufigen Code, da sie *memory operations* umordnen.

Outline

Memory model in C++11

Erkenntnis:

- Sequential Consistency alleine nicht sinnvoll

Weitere Erkenntnisse:

- ’Probleme’ bei Sequential Consistency treten nur auf, wenn auf *unabhängige* Variablen parallel zugegriffen wird
- Mindestens ein Thread greift *schreibend* auf die Daten zu
- Und generell: Simultaner Zugriff auf Shared Variables bringt Probleme mit sich, da sich das Programm dann unter Umständen nicht-deterministisch verhält
  (siehe verschiedene Ausführungsreihenfolgen vom initialen Beispiel)

Outline

Data races und Konflikte

Die Definition von *Data races* ist wie folgt:

We say that a program allows a data race on a particular set of inputs if there is a sequentially consistent execution, that is an interleaving of operations of the individual threads, in which two conflicting operations can be executed /simultaneously/.

Von [Boe08, Data races and a more restrictive model].

Data races und Konflikte (1)

- Programmiersprachen stellen üblicherweise Mechanismen zum Verhindern von unkontrollierten simultanen Zugriff auf Shared Variables bereit.
- *Sequential consistency* wird nur garantiert, falls das Programm keine *data-races* erlaubt.
- Diese Garantie entspricht dem Kern des Threads-Programming Models von Java und C++11!

⇒ **Sequential Consistency for Race-Free Programs**[1].

[1] A Memory Model for C++: Sequential Consistency for Race-Free Programs: http://www.hpl.hp.com/personal/Hans_Boehm/c++mm/seq_con.html

Data races und Konflikte (2)

Zurück zu Dekker's Example:

```
1 // Thread 1
2
3 x = 1;
4 r1 = y;
```

```
1 // Thread 2
2
3 y = 1;
4 r2 = x;
```

Data race?

Data races und Konflikte (2)

Zurück zu Dekker's Example:

```
1 // Thread 1
2
3 x = 1;
4 r1 = y;
```

```
1 // Thread 2
2
3 y = 1;
4 r2 = x;
```

Data race?
Ja!

Outline

Outline

Mutex

Mutex (*mut*ual *ex*clusion)[2]

- Verhindert Zugriff auf kritische Abschnitte
- Nebenläufige Threads/Prozesse werden wechselseitig ausgeschlossen

Listing 1: C++11 mutex

```
1  class mutex {
2  public:
3    mutex();
4    ~mutex();
5    mutex(const mutex&) = delete;
6    mutex& operator=(const mutex&) = delete;
7    void lock();
8    bool try_lock(); // may fail even if lock available!
9    void unlock();
10   ...
11 };
```

Wenn ein Thread A *lock()* aufruft, blocken andere Aufrufe solange, bis Thread A wieder den mutex mit *unlock()* freigibt.

[2] http://en.cppreference.com/w/cpp/thread/mutex

Beispiel: Counter

Listing 2: Counter with a mutex

```
1 mutex m;
2
3 void increment() {
4   m.lock();
5   x = x + 1;
6   m.unlock();
7 }
```

Problem hier:

- Es ist möglich das Code innerhalb der kritischen Sektion eine *Exception* wirft
- Hier: Krische Sektion lediglich Zeile 5
- Wenn dieser Fall eintritt, wird der Lock nicht wieder gelöst ⇒ Dead-lock!

Outline

Lock guards

Lock guards[3] sind simple Wrapper um Mutexes

- Mutex wird im Konstruktur übergeben und dort gelockt
- Mutex wird wiederum im Destruktor von dem lock guard gelöst
- Sobald der lock guard aufgeräumt wird, wird auch der Mutex wieder gelöst.

Listing 3: C++ lock guard

```
1  template <class Mutex>
2  class lock_guard {
3  public:
4    typedef Mutex mutex_type;
5    explicit lock_guard(mutex_type& m);
6    lock_guard(mutex_type& m, adopt_lock_t);
7    ~lock_guard();
8    lock_guard(lock_guard const&) = delete;
9    lock_guard& operator=(lock_guard const&) = delete;
10 private:
11   mutex_type& pm; // for exposition only
12 };
```

[3] http://en.cppreference.com/w/cpp/thread/lock_guard

Example: Counter 2

Listing 4: Counter with lock guard

```
1 mutex m;
2
3 void increment() {
4   lock_guard<mutex> _(m);
5   x = x + 1;
6 }
```

- Lock wird angefordert in Zeile 4
- Lock wird wieder gelöst am Ende des *Scopes*, da das Objekt aufgeräumt wird
- Funktioniert immer noch, wenn die kritische Sektion eine Exception wirft

⇒ Idiom: Resource Acquisition Is Initialization (RAII)

Unique lock

Außerdem:

Unique lock[4]

- Generalisierte Version von *lock_guard*
- Hat mehr Features als *lock_guard*
- Eigenschaften
  - Übergabe der Ownership
  - Verspätetes Locking
  - Recursive Locking
  - (...)

[4] http://en.cppreference.com/w/cpp/thread/unique_lock

Outline

Atomics

- Problem
  - Lock-basierte Synchronisation ist zu langsam/schwergewichtig in manchen Situationen
  - Eine feinere Kontrolle über die Synchronisation ist gewollt
- Lösung
  - C++11 führt die sog. *atomic objects* ein
  - Atomic objects sind ähnlich zu Java Objects deklariert mit *volatile* keyword
    (Hat nichts mit C++'s *volatile* keyword zu tun)

Atomics im Detail

- Atomic[5] ist eine Template-Klasse
- Atomic objects erlauben Lock-freie Implementierung
- Jede Operation auf einem Atomic object ist atomar
  - (vgl. *atomic commits* in Datenbanksystemen)
  - "Änderungen werden voll, oder gar nicht sichtbar"
- Nebenläufige Zugriffe auf atomic objects stellen somit keine *data races* dar

[5] http://en.cppreference.com/w/cpp/atomic

Klassendeklaration von atomic

Listing 5: C++11 atomic

```
1  template< T > struct atomic {
2    // GREATLY simplified
3    constexpr atomic( T ) noexcept;
4    atomic( const atomic& ) = delete;
5    atomic& operator =( const atomic& ) = delete;
6    void store( T ) noexcept;
7    T load( ) noexcept;
8    T operator =( T ) noexcept; // similar to store()
9    T operator T () noexcept; // equivalent to load()
10   T exchange( T ) noexcept;
11   bool compare_exchange_weak( T&, T) noexcept;
12   bool compare_exchange_strong( T&, T) noexcept;
13   bool is_lock_free() const noexcept;
14 };
```

Beispiel: Counter 3

Durch Spezialisierung für Integral- und Zeigertypen ist es möglich auch Inkrementation und Dekrementation atomar auszuführen.
Dies eignet sich beispielsweise für nebenläufigen Code mit Countern.

Listing 6: Counter mit C++11 atomic

```
1 atomic<int> x;
2
3 void increment() {
4   x++; // not x = x + 1
5 }
```

Beispiel: Counter 3

Durch Spezialisierung für Integral- und Zeigertypen ist es möglich auch Inkrementation und Dekrementation atomar auszuführen.
Dies eignet sich beispielsweise für nebenläufigen Code mit Countern.

Listing 7: Counter mit C++11 atomic

```
1 atomic<int> x;
2
3 void increment() {
4   x++; // not x = x + 1
5 }
```

- Kein *data race* hier, da *x* als *atomic* deklariert ist
- Die Operation *x*++ ist sequentiell konsistent

Beispiel: Dekker's Example (erw.)

Listing 8: Dekker's example mit C++11 atomic

```
1 atomic<int> x,y; // initially zero
2
3 // Thread 1
4 x = 1;
5 r1 = y;
6
7 // Thread 2
8 y = 1;
9 r2 = x;
```

Beispiel: Dekker's Example (erw.)

Listing 9: Dekker's example mit C++11 atomic

```
1 atomic<int> x,y; // initially zero
2
3 // Thread 1
4 x = 1;
5 r1 = y;
6
7 // Thread 2
8 y = 1;
9 r2 = x;
```

- Kein *data race*
- Statement 1 von Thread 1 erhält SC
- Statement 1 von Thread 2 erhält SC ebenso
- Ergebnis *r1 == r2 == 0* wiederum nicht möglich

Outline

C++98 vs. C++11

Nochmals Dekker's Example:

```
1 // Thread 1
2
3 x = 1;
4 r1 = y;
```

```
1 // Thread 2
2
3 y = 1;
4 r2 = x;
```

Bedeutung unter C++98:

- Keine, nicht einmal *undefined behavior*
- Threads sind einfach nicht Teil der Spezifikation

C++98 vs. C++11

Nochmals Dekker's Example:

```
1 // Thread 1
2
3 x = 1;
4 r1 = y;
```

```
1 // Thread 2
2
3 y = 1;
4 r2 = x;
```

Bedeutung unter C++98:

- Keine, nicht einmal *undefined behavior*
- Threads sind einfach nicht Teil der Spezifikation

Bedeutung unter C++11:

- *Undefined behavior* aufgrund eines *data races*

C++98 vs. C++11

Nochmals Dekker's Example:

```
1 // Thread 1
2
3 x = 1;
4 r1 = y;
```

```
1 // Thread 2
2
3 y = 1;
4 r2 = x;
```

Bedeutung unter C++98:

- Keine, nicht einmal *undefined behavior*
- Threads sind einfach nicht Teil der Spezifikation

Bedeutung unter C++11:

- *Undefined behavior* aufgrund eines *data races*

Im Grunde genommen keine Verbesserung bzgl. des Programmablaufs, aber das Verhalten ist formalisiert!

Zusammenfassung

- Mit dem neuen C++11-Standard hat C++ endlich ein klar definiertes *Memory Model*
- Der neue Standard stellt eine umfassende Threads API bereit
- Es ist nicht mehr nötig Plattform-/Compiler-spezifische Erweiterungen zu benutzen
- Nebenläufiger Zugriff auf Shared Variables (Atomics) ist klar definiert
- Mit Atomics ist es möglich, Lock-freie Algorithmen zu implementieren
- Atomics geben größtmögliche Kontrolle über Low-Level Operationen des Betriebssystems (für Performance-kritische Anwendungen)

Quellen I


Sarita V. Adve and Hans-J. Boehm.
Memory models: A case for rethinking parallel languages and hardware.
COMMUNICATIONS OF THE ACM, 53(8), 2010.


Hans Boehm.
Threads basics.
2008.


A. Williams.
C++ Concurrency in Action: Practical Multithreading.
Manning Pubs Co Series. Manning Publications, 2011.

Fragen?

Fragen?

Danke für die Aufmerksamkeit!