Überblick über das Speichermodell von C/C++

Benjamin Valentin

18. Juni 2010

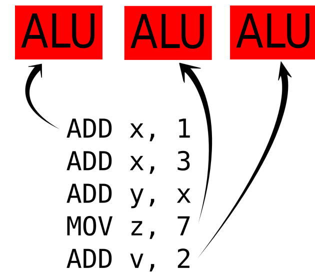
1 Einleitung

Nebenläufige Anwendungen sind heute die Regel, Systeme mit mehren Rechenkernen sind inzwischen sehr verbreitet. Dennoch wird für sie hauptsächlich in Programmiersprachen wie C/C++ entwickelt, die ursprünglich keine Threads vorsahen. Dies wirft bei der am häufigsten verwendeten, da direktesten Synchronisationsmethode, dem gemeinsamen Speicher Fragen auf, auf die der C/C++ Standard bisher keine Antworten gab: Was geschieht beim gleichzeitigen Zugriff auf eine Variable? Wie kann Synchronisation stattfinden?
Bisherige Ansätze, die den Versuch unternahmen die Unterstützung für nichtsequentielle Programme über Bibliotheken zu realisieren stoßen auf das Problem, dass der Compiler oder die CPU immer noch von einem nichtsequentiellen Programm ausgeht, und daher Fehler durch unsynchronisierte Caches oder in der Reihenfolge vertauschte Operationen entstehen.
Um dieses Problem klar zu definieren und eine einheitliche Lösung bereitzustellen, bedarf es eines Speichermodells.
In Abschnitt 2 werden wir uns daher genauer mit der Notwendigkeit eines Speichermodells von der Hardwareseite aus befassen und zwei mögliche Lösungen vorstellen. In 3. wird anhand von `pThreads` ein etablierter Ansatz von Multithreading als Bibliothek vorgestellt sowie die diesem Konzept inherenten Schwächen aufgezeigt. Abschnitt 4 behandelt schließlich das vorgeschlagene Speichermodell für C/C++ und stellt ausgewählte Details davon vor.

2 Überblick Speichermodelle

2.1 Warum benötigt man ein Speichermodell?

Ein Speichermodell oder genauer, ein Speicherkonsistenzmodell, gibt Regeln vor, nach denen ein Programm 'umgeformt' werden darf. Diese Umformungen nimmt dabei nicht nur explizit der Compiler vor (etwa um mehrere zu effektiveren Spezialanweisungen wie SSE oder MMX zusammenzufassen) Sonder auch die CPU. So wird eine out-of-order CPU versuchen, durch Umschichtung voneinander unabhängigen Operationen ihre Rechenwerke möglichst stark auszulasten (bei einer in-order CPU kann dies auch der Compiler vornehmen). Dies ist für ein sequentielles Programm völlig transparent, würde sich jedoch ein ein zweiter Thread auf die Ordnung der Zuweisungen des ersten Threads verlassen würde dies zu einem unbestimmten Ergebnis führen, da die Ordnung der Anweisungen nicht festgelegt ist.



1: Befehle dürfen von CPU und Compiler umgeordnet werden um z.B. die ALUs besser auszulasten

Doch nicht nur Optimierungen, die eine bessere Auslastung der Rechenwerke zum Ziel zeigt den Bedarf eines Speichermodells. Caches stellen ein wichtiges Element für die Leistung aktueller Prozessoren dar. In der Regel besitzt bei Mehrkernsystemen jeder Rechenkern mindestens einen eigenen, lokalen Cache, zusätzlich existiert ein gemeinsamer Cache, der wiederum Daten aus dem Hauptspeicher vorhält. Würde man bei jedem Schreib/Lesevorgang auf die Synchronizität der Caches achten, wären diese obsolet. Tut man es nicht, werden womöglich Variablen inkonsistent, Schreibzugriffe gehen verloren, Funktionen operieren auf veralteten Werten.

Erst durch ein Speichermodell wird definiert, wann welches Synchronisationsverhalten angebracht ist, wann der Compiler eine Optimierung vornehmen darf, oder wann er die CPU davon abhalten muss, diese vorzunehmen. Damit ergeben sich für den Programmierer grundlegende Regeln, wie er auf Speicherwerte zugreifen kann, und welches Verhalten zu erwarten ist.

Dabei gilt es abzuwägen zwischen einem Speichermodell, dass nur wenige Garantien macht und daher mit sehr performanten Primitiven arbeiten kann, dabei aber schwer zu programmieren oder zu erlernen ist und im Endeffekt zu komplexen, fehleranfälligen Programmen führt. Und einem Speichermodell das starke Garantien bereitstellt um eine intuitive Programmierung zu gewährleisten, dabei aber hohe Synchronisationskosten hat, was schließlich langsame Programmen zur Folge hat. Dabei ist auch die bereits vorhandene Hardware in die Designentscheidungen einzubeziehen.

Im vorgeschlagenen Speichermodell für C/C++ wurde ein Kompromiss gewählt, der bei Bedarf mit `low level atomics` sehr schwache Garantien zulässt, sich ansonsten aber zu großen teilen wie gewohnt und erwartet verhält.

2.2 Intuitiver Ansatz - Sequentielle Konsistenz

Geht man an das Problem des Multithreadings heran und betrachtet z.B. den Ablauf mehrerer Threads auf einer CPU, so könnte man zu dem Schluss kommen, dass dies wie folgt umgesetzt werden kann:

- Auswahl eines Threads
- Aktuellen Maschinenbefehl im Thread ausführen
- Nächster Ausführungsschritt machen oder Thread wechseln

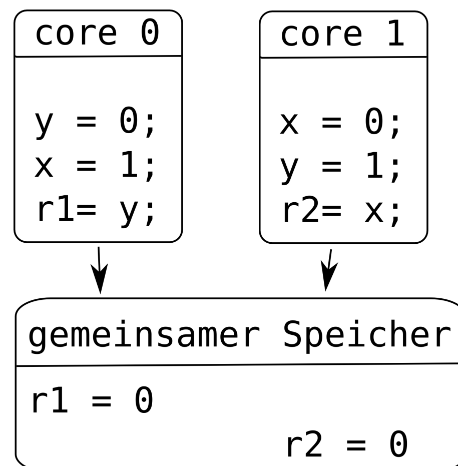
Dabei werden die Instruktionen alle in der Reihenfolge ausgeführt, in der programmiert wurde, Änderungen an Variablen sind stets sofort für alle Threads sichtbar.
Ein solches Modell heißt *Sequentiell Konsistent*.

```
x = 0;
y = 0;

x = 1;    | y = 1;
r1= y;    | r2= x;
```

Listing 1: Bei sequentieller Konsistenz kann z.B. nicht $r1 = r2 = 0$ gelten.

In der Realität ist ein solches Modell jedoch nicht umsetzbar. Abgesehen von der ungenauen Granularität der Ausführungsschritte ist es in diesem Modell nicht möglich, die Reihenfolge der Instruktionen zu ändern um die Recheneinheiten besser auszulasten. Ebenfalls sind die Anforderungen an Speicherkohärenz utopisch, da CPU Caches nicht nach jedem write synchronisiert werden - dies würde eine extreme Performanceeinbußen bedeuten, da alle Kopien im gemeinsamen Speicher aktualisiert werden müssten, der lokale Speicher also im Endeffekt auf die Geschwindigkeit des gemeinsamen Speichers abgebremst würde.
Dies wäre zwar nur bei gemeinsamen Variablen nötig und es existieren Möglichkeiten, diese zu erkennen, Compiller wissen aber in der Regel zu wenig, um dies stets zweifelsfrei feststelen zu können, daher garantieren aktuelle Compiler in der Regel keine sequentielle konsitenz.



2: CPU Caches werden nicht nach jedem write synchronisiert

Zwar schient dieses Modell auf den erste Blick logisch und erstrebenswert, so zeigen sich doch eklatante Performanceprobleme beim Versuch es zu implementieren.

Doch glücklicherweise ist garantierte Sequentielle Konsistenz nicht so vorteilhaft wie man dies anfangs annehmen würde: Bei gleichzeitigen Schreib- (und Lesezugriffen) auf

die gleiche Variable treten auch hier meist unvorhersehbare Ergebnisse auf, so dass man derartige Data Races in der Regel ohnehin vermeiden möchte. Diese Einschränkung führt zu einem deutlich praktikableren Speichermodell, dass von *Data-Race Freiheit* ausgeht.

2.3 Data-Race Freiheit

Die diesem Model zu Grunde liegende Annahme besteht darin, dass die Korrektheit eines Programms nur dann garantiert wird, wenn kein *Data Race*, d.h. keine parallelen Zugriffe auf ein Speicherwort stattfinden, von denen mindestens einer schreibend ist.
Ziel ist es, Operationen mit `lock` und `unlock` in eine definierte Ordnung zu bringen, und so ein Data Race zu verhindern. Enthält ein Programm keine Data Races, verhält es sich sequentiell konsistent.[3] (Im Gegensatz zu *Listing 1*, bei dem z.B. durch das geleichzeitige Lesen `r1 = y` und Schreiben `y = 1` ein Data Race entsteht)
Im Nachfolgenden werfen wir einen Blick auf eine verbreitete Implementierung, die sich diesem Modell bedient. Dabei wird deutlich, warum sich ein Speichermodell nicht orthogonal implementieren lässt, sondern Änderungen in Compiler und der Definition der Programmiersprache erforderlich macht.

3 Traditioneller Ansatz - Beispiel pThreads

Bei der Spezifikation von C/C++ wurde einer möglichen parallelen Ausführung des Programms keine Beachtung geschenkt, folglich wurde auch kein Speichermodell spezifiziert. Der Bedarf, dennoch parallele Anwendungen mit C/C++ zu entwickeln führte zur Entwicklung diverser Threading Bibliotheken, unter anderem den *POSIX Threads*. Diese orientieren sich zwar am *Data Race Free* Modell, spezifizieren es aber nicht genau, was dazu führt, dass sich Programmierer in Fragen des Verhaltens des Speichermodells auf die Implementierung verlassen müssen.
Doch das eigentliche Problem, das allen bibliotheksgebundenen Lösungen immanent ist, besteht darin, dass der Compiler nichts über Synchronisationsmechanismen weiß und diese daher 'kaputt optimieren' kann bzw. der Prozessor nicht davor bewahrt wird, dies zu tun. Damit kann die Korrektheit des Programms nicht garantiert werden.

3.1 Ansatz

Das Speichermodell verhält sich schwächer als *sequentiell konsistent*, data races führen zu echt undefiniertem Verhalten. (es kann also eine Variable 'mitten im Schreiben' gelesen werden, womit weder der neue noch der alte Wert gelesen wird, theoretisch können auch noch schlimmere Dinge passieren) Funktionen wie `pthread_mutex_lock` führen Anweisungen aus, die die Hardware vor Speicherumsortierung abhält bzw. zu Synchronisation zwingt. Zusätzlich stellen sie für den Compiler *opake* Funktionen dar, die potentiell auf jede Globale oder nachfolgende Variable zugreifen können, was dafür sorgt, dass keine Anweisungen vom Compiler aus dem zu lockenden Bereich heraus verschoben werden.

3.2 Probleme

Das Problem besteht nun darin, dass der Compiler oder die CPU nichts von einem Speichermodell wissen und daher Optimierungen vornehmen, die Data Races in völlig korrekte Programme einführen. Die drei möglichen Situationen werden im nachfolgenden anhand von Beispielen aus [4] erläutert.

3.2.1 Umformulieren von Anweisungen

Betrachten wir folgendes Programm:

```
Initial: x = y = 0
if (x == 1)              |          if (y == 1)
        ++y;             |                  ++x;
```

Da weder x noch y geschrieben wird findet kein Data Race statt, das einzig valide Ergebnis wäre $x = y = 0$.
Nichts hielte einen Compiler davon ab, dies in die (für einen Thread) äquivalente Anweisung

```
++y;                |        ++x;
if ( x != 1 )       |        if ( y != 1 )
        --y;        |                --x;
```

zu transformieren. Nun findet sehr wohl ein data race statt, da z.B. x in einem Thread gelesen und in einem anderen geschrieben wird.
Diese 'Optimierung' ist in der Realität zwar eher unwahrscheinlich, jedoch aus Compilersicht völlig legal.

3.2.2 Zusammenfassen von Variablen

Realistischer sind da schon Probleme, die durch das Zusammenfassen von benachbarten Variablen entstehen. So könnte sich ein Compiler für eine 64bit Architektur dafür entscheiden, zwei 32bit Werte zusammenzufassen, etwa um Speicherplatz zu sparen:

```
struct { u32 a; u32 b; } x;
x.a = 23;           |        x.b = 42;
```

Während auf einer 32bit Maschine beide Variablen getrennt in verschiedenen Threads ohne Data Races benutzt werden können, würde die Modifikation einer der beiden Variablen mit der 64bit Optimierung zwangsläufig auch einen Schreibvorgang der anderen auslösen:

```
u64 x;

u64 tmp = x;        |        u64 tmp = x;
tmp &= ~0xffff;     |        tmp &= ~0xffff0000;
tmp |= 23;          |        tmp |= 42 << 8;
```

```
x       = tmp;          |           x           = tmp;
```

Ein offensichtliches Data Race ist eingetreten.
Solche Optimierungen sind für beliebige Variablen legal und bergen ein enormes Fehlerpotential, da sie plötzlich bei der Übersetzung auf einer anderen Architektur auftreten können, womit der Fehler nur schwer auszumachen ist.

3.2.3 Variable wird nicht mehr synchronisiert

Ein weiteres Problem kann Auftreten, wenn der Compiler oder die CPU sich entscheiden, eine häufig benutzte Variable in einem Register vorzuhalten, welches nicht mehr mit dem gemeinsamen Speicher synchronisiert wird.
Betrachten wir folgendes Muster: Wir schützen die gemeinsame Variable `x`, die von mehren Threads in einer `for` Schleife für eine Berechnung verwendet wird durch eine Mutex. Um die Performance zu optimieren führen wir die `lock` Operationen nur aus, wenn auch andere Threads laufen (`mt` ist ungleich 0).
Durch die pthread Operationen wird eine Synchronisation von `x` mit dem gemeinsamen Speicher erzwungen.

```
for( ... ) {
        ...
        if(mt)
                pthread_mutex_lock(mtx);
        x = calc_something(x);
        if(mt)
                pthread_mutex_unlock(mtx);
}
```

Der Compiler oder die CPU könnte sich aber für folgende 'Optimierung' entscheiden, `x` wird aufgrund der häufigen Verwendung im lokalen Register `DX` vorgehalten.

```
DX = x;
for( ... ) {
        ...
        if(mt) {
                x = DX;          // Race contition
                pthread_mutex_lock(mtx);
                DX = x; }
        DX = calc_something(DX);
        if(mt) {
                x = DX;
                pthread_mutex_unlock(mtx);
                DX = x; }
}
x = DX;
```

Nun wird `x` zwar immer noch wie gefordert bei Aufrufen von `pthread_mutex_*` mit dem gemeinsamen Speicher synchronisiert, die Zuweisung `x = DX` geschieht jedoch völlig ungeschützt, ein Data Race ist entstanden.

3.3 Performance Betrachtungen

Die Verwendung von

```
pthread_mutex_lock();
shared_val = 23;
pthread_mutex_unlock()
```

zur Absicherung von gemeinsamen Variablen ist äußerst langsam, da zwei Speicherbarrieren nötig sind (die bei allen `pthread_mutex_*` Funktionen vorhanden sein müssen) und zusätzlich der Verwaltungsaufwand für die Mutex hinzukommt. Während dies auf den ersten Blick als akzeptable Einschränkung empfunden wird, verhindert sie in der Praxis doch die Implementation einiger effizienter Algorithmen mit pThreads. Hierzu ein greifen wir das Beispiel eines einfachen Sieb des Eratosthenes aus [4] auf:

```
for(my_prime = start; my_prime < INITAL_PRIMES; ++my_prime)
 if(!primes[my_prime]) {
  for(multiple = my_prime; multiple < ARRAY_SIZE; multiple += my_prime)
   if(!primes[multiple])
    primes[multiple] = TRUE;
 }
```

Wir gehen davon aus, dass alle nicht-Primzahlen im Bereich 0 bis `INITIAL_PRIMES` in dem booleschen Array `primes` mit `TRUE` initialisiert sind, Primzahlen mit `FALSE`. Nach Ablauf des Algorithmus gillt diese Bedingung für alle Elemente des Array.
Das interessante an diesem Algorithmus besteht nun darin, dass, ausgehend von atomaren Schreib- und Leseoperationen, der Algorithmus auch dann korrekt bleibt, wenn er parallel auf dem selben Array ausgeführt wird. Dabei spielt es keine Rolle, ob die Änderungen für alle Threads gleichzeitig sichtbar werden, im schlimmsten Fall wird das Ergebnis ein weiteres mal berechnet und mit dem gleichen Wert überschrieben. Hat aber bereits ein anderer Thread die entsprechende Berechnung getätigt, kann die Ausführungszeit des Algorithmus dadurch reduziert werden.
Das Problem besteht darin, dass pThreads keinen adäquaten Mechanismus bereitstellt, dies Performanz zu realisieren. Das Array über eine Mutex zu sichern würde an der Idee des Algorithmus vorbei gehen, da dieser eben gerade ohne locks auskommen möchte. Ein einfaches, ungeschütztes Array garantiert hingegen keine Atomaren Zugriffe - im schlimmsten Fall könnte der Compiler die booleschen Variablen zu einem Bit array zusammenfassen. Nun würde ein Thread gleich mehre Elemente aus dem Array lesen, eines ändern und alle zurückschreiben - wobei auch benachbarte Elemente, die in der Zwischenzeit schon geupdatet wurden mit dem alten Wert überschrieben werden, der Algorithmus funktioniert nicht mehr.
Um für all diese Probleme eine adäquate Lösung anzubieten ist es erforderlich den

C/C++ Standard um ein Speichermodell und entsprechende Thread Operationen zu erweitern. Gerade für letztgenanntes Problem gab es in der Vergangenheit diverse Compiler und Hardwarespezifische Primitive, deren Benutzung jedoch keine Portabilität gewährte. Das Speichermodell stellt in dieser Hinsicht mit *low level atomics* nun auch eine Abstraktion dar, um zu signalisieren, dass man mit schwächeren Bedingungen, die weniger Optimierungen verhindern, auskommt.

4 Das C/C++ Speichermodell

4.1 Wege zu aktuellem Vorschlag

Da die bisherige Situation ohne ein definiertes Speichermodell nicht nur unbefriedigend, sondern auch fehleranfällig ist, wurde das Bedürfnis nach einem Standard für das Verhalten von gemeinsamen Speicher und Synchronisierung immer dringender. Hardwarehersteller produzieren bereits seit längerem Mehrkernprozessoren, die verschiedene Synchronisationsprimitive anbieten. Diese sind aufgrund der unklaren Anforderungen von Seiten der Software jedoch nicht einheitlich, es hat sich jedoch als sinnvoll erwiesen, dass die meisten Hardwarespeichermodelle schwächere Annahmen als *Sequentielle Konsistenz* garantieren.
2005 begann schließlich die Entwicklung eines Speichermodells für C/C++, das auch in den bevorstehenden C++0x Standard einzuhalten wird. Beim Design stützte man sich dabei vor allem auf folgende Überlegungen:

- Schwächere Modelle als data-race free sind schwer zu benutzen
- Auf x86 und anderen Architekturen ist data-race free relativ performant zu implementieren, hauptsächlich `atomic` erweist sich als teuer
- Das Verhalten eines Programms mit data races ist undefiniert. Dies ist aber akzeptabel für eine Sprache wie C/C++. (Im Gegensatz zu Java)

4.2 Umsetzung

Die Grundlage für das Speichermodell bilden die Bedingungen von *Data Race Freiheit* mit synchronen `atomic` Variablen. Diese lassen sich bei Bedarf abschwächen, was zu geringeren Synchronisationskosten führt. Speicheroperationen operieren auf (abstrakten) Speicherbereichen, das heißt primitive/skalare Variablen belegen dedizierte Speicherbereiche und lassen sich getrennt voneinander schreiben/lesen. Eine Ausnahme stellen Elemente in Bitfeldern dar, da diese auf den selben Speicherbereich abgebildet werden. Ein paralleler Schreib/Lesezugriff auf einen solchen unsynchronisierten Speicherbereich bedeutet ein Data Race, dass zu unvorhersehbaren Ergebnissen führt:

```
long long x = 0;
x = -1; | r1 = x;
```

Bei sequentieller Konsistenz könnte man davon ausgehen, dass r1 entweder 0 oder -1 ist. Auf einer 32bit Maschine kann aber, wenn diese nicht garantiert wird, auch ein anderer Zwischenwert auftreten, da die 64bit Variable nicht in einem Schritt geschrieben wurde.

4.2.1 erlaubte Optimierungen

Ein Speichermodell definiert vor allem auch legale Code Transformationen, Adve nennt und beweist diese in seinen Arbeiten[5][2]. Dabei werden die Aktionen in **Synchronisationsoperationen** (`lock`, `unlock`, `atomic load`, `atomic store`, `atomic read-modify-write` (z.B. compare&swap)) und **Datenoperationen** (`load`, `store`) eingeteilt. Die unabhängigen Speicheroperationen `M1` und `M2` dürfen von Compiler oder Hardware nun unter folgenden Bedienungen in ihrer Ordnung vertauscht werden:

1. M1 ist eine Datenoperation und M2 eine synchronisierende Leseoperation
2. M1 ist eine synchronisierende Schreiboperation und M2 ist Datenoperation
3. M1 und M2 sind Datenoperationen
4. M1 ist Datenoperation und M2 das Schreiben eines Locks
5. M1 ist Unlock und M2 ist entweder das Schreiben oder Lesen eines Locks

Normalerweise ist dies völlig transparent für Data-Race freie Programme, es führt jedoch zu einem Problem bei einer nicht vorgesehenen Verwendung von trylock.

```
x = 42;           | while (trylock(l) == succsess)
lock(l);          |         unlock(l);
                  | assert(x == 42);
```

Die Anweisung im ersten Thread darf nach (4) umsortiert werden, dies stellt in der Regel kein Problem dar, eine Verhinderung dieser Optimierung würde eine zusätzliche Speicherbarriere vor dem lock erfordern und so Implementationen benachteiligen, die `trylock` wie vorgesehen nutzen.
Die pragmatische Lösung besteht darin, die Definition von `trylock()` dahingehend zu ändern, dass es nun auch 'unberechtigt' fehlschlagen darf, so kann es nicht mehr über den Status des locks Auskunft geben. Dies bedeutet trylock schlägt stets fehl, wenn das lock bereits genommen wurde, kann aber nun auch fehlschlagen, wenn das lock frei ist. Dadurch ist es legitim die while Schleife zu verlassen, auch wenn x != 42.

4.2.2 Atomare Variablen und Low Level Atomics

Zusätzlich zu den locking Mechanismen werden `atomic` Variablen (`volatile` in Java) angeboten, die sich stets sequentiell konsistent verhalten.

```
int x = 0;
atomic_int y = 0;
x = 17;           | while(y.load() != 1);
y.store(1);       | assert(x == 17);
```

Diese Variablen müssen dabei von der Implementierung nicht nur über alle Threads synchron gehalten werden, sondern dürfen auch nicht umsortiert werden, was Speicherbarrieren erforderlich macht. Damit ist die Verwendung von `atomic` Variablen in relativ teuer.

Nicht zu verwechseln ist dieses neue Schlüsselwort mit dem bereits existierenden `volatile` in C/C++, dass eine etwas andere Bedeutung hat. Zwar verbietet auch `volatile` die Verschiebung der Variable durch den Compiler, ist aber z.B. eigentlich Abbildung von Hardwarestrukturen vorgesehen, so deklarierte Variablen können sich ohne den Einfluss des Programms ändern und müssen daher stets aus dem Hauptspeicher eingelesen werden. Im Gegensatz zu `atomic` Variablen ist jedoch nicht garantiert, dass der Zugriff auf diese Variablen tatsächlich atomar erfolgt. Von einer Erweiterung des `volatile` Begriffes um die zusätzlichen Eigenschaften von `atomic` wurde aufgrund der Effekte, vor allem der Performancebeeinträchtigung bestehender Programme abgesehen.[6]

Es existieren jedoch sehr performante lock free Algorithmen die unter Benutzung von atomaren Speicherzugriffen Locks vermeiden. (siehe 3.3 oder [7] für eine praktische Anwendung dieses Prinzips) Um diese auch in C/C++ performant umzusetzen existiert die Möglichkeit, die Anforderungen, die an die atomics gestellt werden zu entspannen - während dies richtig angewandt zu einer höheren Performance führen kann, da Speicherbarrieren wegfallen, so gleich ist die Verwendung von `low level atomics` komplex und fehleranfällig.

Dabei existieren die folgenden Optionen:

- **memory_order_seq_cst** - die Atomare Variable verhält sich sequentiell konsistent (Standard)
- **memory_order_release** - kann umsortiert werden, ein Schreiben dieser Variable geschieht jedoch vor einem Lesen mit dem Parameter `memory_order_acquire`
- **memory_order_acquire** - kann umsortiert werden, ein Lesen dieser Variable geschieht jedoch nach einem Schreiben mit dem Parameter `memory_order_release`
- **memory_order_relaxed** - einzig der atomare Zugriff wird garantiert

Der obige Code würde z.B. Unter der Verwendung von

```
int x = 0;
atomic_int y = 0;
x = 17;            | while(y.load(memory_order_relaxed) != 1);
y.store(1, memory_order_relaxed);        | assert(x == 17);
```

Nicht mehr auf das assert garantieren, da die beiden Variablen im ersten Thread nun unabhängig voneinander betrachtet werden und ihre Zuweisungsreihenfolge nicht garantiert ist.

4.3 Probleme

Das größte Problem des Modells stellt die Undefiniertheit von Data Races dar. Da es theoretisch sogar möglich ist, dass sich bei einem Data Race Rücksprungadressen ändern oder fremde Variablen verfügbar werden, stellt dies ein großes Risiko bei der Ausführung von fremden Code dar.
Weiger als tatschliches Problem ist Performance der atomaren Variablen auf aktueller Hardware anzusehen, dies liese sich jedoch durch eine Erweiterung der ISA um synchrone Versionen von `load` und `store`) verbessern. Zudem kann man zu *low level Atomics* greifen, wenn man nicht alle Garantiene benötigt, die einem `atomic` bietet.

4.4 Mögliche Gegenmaßnahmen

Eine Möglichkeit zumindest das Debugging zu erleichtern wäre die Erkennung von Data Races zur Compilezeit. Da das zugrunde liegende Speichermodell jedoch erst kürzlich definiert wurde und auch der entsprechende Standard sich noch im Ratifizierungsprozess befindet, gibt es auf diesem Gebiet bisher noch wenig Forschungsarbeit. Unklar ist, ob es überhaupt in allen Fällen möglich ist, ein Data Race im Vorraus zu erkennen.
Bestehen bliebe damit nur noch das Problem von fremdem Code, der im eigenen Programmkontext ausgeführt wird.

5 Fazit

Ein Speichermodell, wie es für den kommenden C++ Standard vorgeschlagen und auch von C adaptiert wird, löst viele bisher vorhandene Probleme. Erstmals ist es möglich, hoch optimierten, portablen nichtsequentiellen Code in C/C++ zu schreiben. Dabei stellt das gefundene Modell einen guten Kompromiss zwischen Programmierbarkeit und Performance dar und zeigt sich dabei möglichst kompatibel mit bestehendem Code.

Literatur

[1] Adve, Sarita V.: Memory models: a case for rethinking parallel languages and hardware. In: *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing.* New York, NY, USA : ACM, 2009. – ISBN 978–1–60558–396–9, S. 2–2

[2] Adve, Sarita V. ; Adve, Sarita V.: Designing Memory Consistency Models for SharedMemory Multiprocessors. 1993. – Forschungsbericht

[3] Boehm, Hans-J.: *Sequential Consistency for Race-Free Programs.* `http://www.hpl.hp.com/personal/Hans_Boehm/c++mm/seq_con.html`

[4] Boehm, Hans-J.: Threads cannot be implemented as a library. In: *SIGPLAN Not.* 40 (2005), Nr. 6, S. 261–268. `http://dx.doi.org/http://doi.acm.org/10.1145/`

1064978.1065042. – DOI http://doi.acm.org/10.1145/1064978.1065042. – ISSN 0362–1340

[5] Boehm, Hans-J. ; Adve, Sarita V.: Foundations of the C++ concurrency memory model. In: *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA : ACM, 2008. – ISBN 978–1–59593–860–2, S. 68–78

[6] Hans Boehm, Nick M.: *Should volatile Acquire Atomicity and Thread Visibility Semantics?* http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2016.html. Version: 2006

[7] Michael, Maged M.: Scalable Lock-Free Dynamic Memory Allocation. In: *SIGPLAN Not.* 39 (2004), Nr. 6, 35–46. http://dx.doi.org/http://doi.acm.org/10.1145/996893.996848. – DOI http://doi.acm.org/10.1145/996893.996848. – ISSN 0362–1340