

- Gegeben seien die Prozessoren CPU_1, \dots, CPU_n , die die Prozesse P_1, \dots, P_n ausführen.
- Jede der Prozesse P_i führt sequentiell Instruktionen aus, zu denen auch Speicher-Instruktionen gehören.
- Die Speicher-Instruktionen lassen sich vereinfacht einteilen:
 - ▶ Schreib-Instruktionen, die den Inhalt einer Speicherzelle an einer gegebenen Adresse ersetzen und als abgeschlossen gelten, sobald der neue Inhalt für alle anderen Prozessoren sichtbar ist.
 - ▶ Lade-Instruktionen, die den Inhalt einer Speicherzelle an einer gegebenen Adresse auslesen und als abgeschlossen gelten, sobald der geladene Inhalt durch Schreib-Instruktionen anderer Prozessoren nicht mehr beeinflusst werden kann.
 - ▶ Atomare Lade/Schreib-Instruktionen, die beides miteinander integrieren und sicherstellen, dass die adressierte Speicherzelle nicht mittendrin durch andere Instruktionen verändert wird.
- (Diese und die folgenden Definitionen und Notationen wurden dem Kapitel D des *SPARC Architecture Manual* entnommen, siehe <https://sparc.org/technical-documents/#V9.>)

$$X_i <_p Y_i$$

- X_i und Y_i seien beliebige Speicher-Operationen des Prozesses P_i .
- Dann gilt die auf den Prozess P_i bezogene Relation $X_i <_p Y_i$, wenn die Instruktion, die zu X_i führt, vor der Instruktion ausgeführt wird, die zu Y_i führt.
- Diese Relation reflektiert die Reihenfolge, die durch das Programm gegeben ist, das durch den Prozess P_i ausgeführt wird. Dies ist nicht notwendigerweise die Reihenfolge, in der die entsprechenden Operationen tatsächlich ausgeführt werden.
- Die Relation $<_p$ ist eine Totalordnung, d.h. es gilt entweder $X_i <_p Y_i$ oder $Y_i <_p X_i$, falls $X_i \neq Y_i$.

$$X_i <_d Y_i$$

- Es gilt $X_i <_d Y_i$ genau dann, wenn $X_i <_p Y_i$ und mindestens eine der folgenden Bedingungen erfüllt ist:
 - ▶ Y_i ist oder enthält eine Schreib-Instruktion, die einer bedingten Sprunganweisung folgt, die von X_i abhängt.
 - ▶ Y_i liest ein Register, das von X_i abhängt.
 - ▶ Die Schreib-Instruktion X_i und die Lade-Instruktion Y_i greifen auf die gleiche Speicherzelle zu.
- $<_d$ ist eine partielle Ordnung, die transitiv ist.

$$X <_m Y$$

- Die totale Speicher-Ordnung reflektiert die Reihenfolge, in der die Speicher-Operationen zur Laufzeit umgesetzt werden.
- Die Reihenfolge ist im allgemeinen nicht deterministisch.
- Stattdessen gibt es in Abhängigkeit des zur Verfügung stehenden Speichermodells einige Zusicherungen.
- Ferner können auch gewisse Ordnungen erzwungen werden.
- Die Relation $<_m$ reflektiert die Restriktion, dass X vor Y erfolgen muss.
- Das Speichersystem ist in der Wahl der Ausführungsreihenfolge frei, sofern die durch $<_m$ definierte partielle Ordnung respektiert wird.

$$M(X, Y)$$

- Alle gängigen Prozessorarchitekturen bieten Instruktionen an, die eine Ordnung erzwingen. Diese werden *memory barriers* bzw. bei den Intel/AMD-Architekturen *fences* genannt.
- Eine Barrier-Instruktion wirkt sich zunächst unmittelbar auf die Instruktionen aus, zwischen denen sie steht.
- Hinzu kommt, dass sie über alle Prozessoren hinweg erzwingt, dass bestimmte Operationen abgeschlossen werden müssen, bevor bestimmte Operationen begonnen werden können.
- Eine Barrier-Instruktion spezifiziert, welche Sequenz von Speicher-Instruktionen geordnet wird. Prinzipiell gibt es vier Varianten bzw. Kombinationen davon: Lesen-Lesen, Lesen-Schreiben, Schreiben-Lesen und Schreiben-Schreiben.
- $M(X, Y)$ gilt dann, wenn wegen einer dazwischenliegende Barrier-Instruktion zuerst X und dann Y ausgeführt werden muss.

- Lesen-Lesen: Alle Lese-Operationen vor der Barrier-Instruktion müssen beendet sein, bevor folgende Lese-Operationen durchgeführt werden können.
- Lesen-Schreiben: Alle Lese-Operationen vor der Barrier-Instruktion müssen beendet sein, bevor die folgenden Schreib-Operationen für irgendeinen Prozessor sichtbar werden.
- Schreiben-Lesen: Die Schreib-Operationen vor der Barrier-Instruktion müssen abgeschlossen sein, bevor die folgenden Lese-Operationen durchgeführt werden.
- Schreiben-Schreiben: Die Schreib-Operationen vor der Barrier-Instruktion müssen alle beendet sein, bevor die folgenden Schreib-Operationen ausgeführt werden.

- Auf der x86-Architektur:

LFENCE Lesen-Lesen

SFENCE Schreiben-Schreiben

MFENCE Lesen/Schreiben-Lesen/Schreiben

All diese Operationen serialisieren global.

- Auf der SPARC-Architektur:

MEMBAR *membar_mask* mit $membar_mask = cmask \mid mmask$

Mögliche kombinierbare Bits für *mmask*: **#StoreStore**, **#LoadStore**, **#StoreLoad** und **#LoadLoad**.

Mögliche kombinierbare Bits für *cmask*: **#Sync**, **#MemIssue** und **#Lookaside**. (Per Voreinstellung wirkt sich **MEMBAR** nur auf die Operationen eines Prozessors aus, bei **#Sync** wirkt sich das global auf alle Prozesse aus.)

Eine Speicherordnung $<_m$ ist RMO genau dann, wenn

- ▶ $X <_d Y \wedge L(X) \Rightarrow X <_m Y$
- ▶ $M(X, Y) \Rightarrow X <_m Y$
- ▶ $Xa <_p Ya \wedge S(Y) \Rightarrow X <_m Y$

Xa steht für eine Speicher-Instruktion auf der Speicherzelle an der Adresse a ; die Prädikate $L(X)$ und $S(Y)$ treffen zu, wenn X eine Lese-Instruktion ist oder umfasst bzw. Y eine Schreib-Instruktion ist oder umfasst.

Die drei Punkte bedeuten im Einzelnen:

- ▶ RMO sichert zu, dass die Abhängigkeitsrelation beachtet wird, wenn die vorangehende Instruktion eine Lade-Operation ist. Darauf folgende Speicher-Instruktionen können sich verzögern und ihre Ausführungsreihenfolge ist zunächst nicht festgelegt.
- ▶ Memory-Barrier werden respektiert.
- ▶ Bei Speicher-Instruktionen an die gleiche Adresse wird die durch das Programm gegebene Ordnung respektiert.

- *Partial Store Order* (PSO) erfüllt alle Bedingungen der RMO. Hinzu kommt, dass allen ladenden Speicher-Instruktionen implizit eine Barrier-Instruktion für Lesen-Lesen und Lesen-Schreiben folgt.
- *Total Store Order* (TSO) erfüllt alle Bedingungen der PSO. Hinzu kommt, dass allen schreibenden Speicher-Instruktionen implizit eine Barrier-Instruktion für Schreiben-Schreiben folgt.
- *Sequential Consistency* (SQ) erfüllt alle Bedingungen der TSO. Hinzu kommt, dass allen schreibenden Speicher-Instruktionen eine Barrier-Instruktion für Schreiben-Lesen folgt.
- SQ ist das einfachste Modell, bei dem die Reihenfolge der Instruktionen eines einzelnen Prozesses sich direkt aus der Programmordnung ergibt. Andererseits bedeutet SQ, dass bei einem Schreibzugriff sämtliche Speicherzugriffe blockiert werden, bis die Cache-Invalidierungen abgeschlossen sind. Das ist sehr zeitaufwendig.

- Gegeben seien die beiden globalen Variablen a und b mit einem Initialwert von 0 und drei Prozesse mit den folgenden Instruktionen:

P_1	P_2	P_3
$a = 1;$	int $r1 = a;$	int $r1 = b;$
$b = 1;$	int $r2 = b;$	int $r2 = a;$

- Mögliche Szenarien (bei SQ und TSO nicht vollständig):

	$P_2: r1$	$P_2: r2$	$P_3: r1$	$P_3: r2$
Unter SQ:	0	0	0	0
	1	1	1	1
Unter TSO, !SQ:	1	0	1	1
Unter PSO, !TSO:	0	0	1	0
	0	1	1	0
	1	1	1	0
Unter RMO, !PSO:	1	0	1	0

- Jede Prozessorarchitektur bietet einige Datentypen an, die atomar geladen oder geschrieben werden.
- Atomizität bedeutet hier, dass bei einer Lade-Instruktion, die einer Speicher-Instruktion folgt, entweder einer der früheren Werte oder der neue Wert zu sehen ist.
- Wenn beispielsweise ein 32-Bit-Wort auf einer 4-Byte-Kante sich in diesem Sinne atomar verhält, kann es nicht passieren, dass beim Lesen die ersten zwei Bytes noch den alten Wert aufweisen, die letzten zwei Bytes den neuen.
- Entscheidend für die Eigenschaft der Atomizität eines Datentyps ist die Größe, das Alignment bzw. der Punkt, ob die Daten vollständig in einer Cache-Line liegen.

- Bei den x86-Architekturen von Intel wird Atomizität für Lese- und Schreibzugriffe folgender Datentypen zugesichert:
 - ▶ einzelne Bytes (also etwa **char**),
 - ▶ 16-Bit-Worte auf 2-Byte-Kanten (also etwa **short**),
 - ▶ 32-Bit-Worte auf 4-Byte-Kanten (also etwa **int**),
 - ▶ 64-Bit-Worte auf 8-Byte-Kanten ab der Pentium-Architektur (also etwa **long long int** oder **double**),
 - ▶ 16-Bit-Worte, die in ein 4-Byte-Wort auf einer 4-Byte-Kante fallen ab der Pentium-Architektur und
 - ▶ 16-, 32- und 64-Bit-Worte, die in beliebiger Weise in eine Cache-Line fallen ab der Pentium-Pro-Architektur (P6).
- Bei der SPARCv9-Architektur wird die Atomizität für alle 64-Bit-Worte auf 8-Byte-Kanten (und alles, was kleiner ist) zugesichert.

- Alle gängigen Prozessorarchitekturen bieten atomare Instruktionen auf Maschinenebene an.
- Jede dieser Instruktionen lädt und speichert (in dieser Reihenfolge) von und in den Hauptspeicher in einer Weise, die sicherstellt, dass keine andere Lade- oder Speicherinstruktion für die gleiche Speicherlokation dazwischen ausgeführt wird.
- Ebenso werden fremde Traps während der Ausführung der Instruktion unterdrückt.
- Die Sichtbarkeit des neu gespeicherten Werts kann sich aber durchaus verzögern; d.h. unter Umständen ist es notwendig, auf eine atomare Instruktion noch eine Barrier-Instruktion folgen zu lassen.

- Die einfachste atomare Lade- und Speicheroperation lädt den alten Wert eines atomaren Datentyps aus dem Speicher in ein Register und schreibt den Inhalt eines anderen Registers in die gleiche Speicherzelle hinein.
- Auf der x86-Architektur, jeweils $tmp = r$; $r = m$; $m = tmp$; wobei r ein Register und m eine entsprechende Speicherzelle ist.

XCHG $r8, m8$ 8-Bit-Operation

XCHG $r16, m16$ 16-Bit-Operation

XCHG $r32, m32$ 32-Bit-Operation

XCHG $r64, m64$ 64-Bit-Operation (nur im 64-Bit-Modus)

- Auf der SPARC-Architektur geht dies mit

LDSTUB $m8, r8$ 8-Bit-Operation

SWAP $m32, r32$ 32-Bit-Operation

Die SWAP-Instruktionen können für einfache Locks verwendet werden, die einen gegenseitigen Ausschluss ermöglichen. Gegeben sei eine Datenstruktur und eine **bool**-Variable, die mit einem Byte repräsentiert wird und nur die Werte 0 (Lock ist frei) und 1 (Lock ist belegt) unterstützt:

```
r = 1;
XCHG r,lock;
if (r == 0) {
    /* lock was free and is now set to 1 */
    /* critical region with exclusive access */
    /* memory barrier: write-read */
    XCHG r,lock;
} else {
    /* no access this time */
}
```

Wenn sichergestellt ist, dass im kritischen Bereich nur wenige Instruktionen ausgeführt werden, deren Ausführungszeit sehr beschränkt ist, können die anderen ggf. ungeduldig mit einer Schleife auf die Freigabe des Locks warten:

```
r = 1;
do {
    XCHG r,lock;
} while (r == 1);
/* lock was free and is now set to 1 */
/* critical region with exclusive access */
/* memory barrier: write-read */
XCHG r,lock;
```

Achtung: Wenn die kritische Region durch einen TRAP unterbrochen wird, dann hängen die anderen Threads u.U. sehr lange. Wenn innerhalb einer Signalbehandlungsfunktion ein Versuch unternommen wird, den Lock zu gewinnen, haben wir möglicherweise einen Deadlock.

- Neben der SWAP-Instruktion wird auf allen gängigen Architekturen auch eine bedingte Instruktion angeboten, die atomar den alten Wert lädt, ihn mit einem vorgegebenen Wert vergleicht und im Falle der Gleichheit den Inhalt eines anderen Registers in die gleiche Speicherzelle schreibt. Typischerweise nennt die Instruktion sich CAS (*compare and set*).
- Auf der x86-Architektur, jeweils implizit mit dem Vergleichswert im Register AL, EAX oder RAX, je nach Wortbreite:
 - CMPXCHG** *m8,r8* 8-Bit Operation
 - CMPXCHG** *m16,r16* 16-Bit Operation
 - CMPXCHG** *m32,r32* 32-Bit Operation
 - CMPXCHG** *m64,r64* 64-Bit Operation
 - CMPXCHG8B** *m64* 64-Bit Operation mit EDX:EAX und ECX:EBX
 - CMPXCHG16B** *m128* 128-Bit Operation mit RDX:RAX und RCX:RBX
- Auf der SPARC-Architektur:
 - CAS** *m32,r32,r32* 32-Bit Operation
 - CASX** *m64,r64,r64* 64-Bit Operation

Die 32-Bit-Variante der atomaren CMPXCHG-Operation in Pseudo-Code als **bool**-wertige Funktion:

```
atomic bool CMPXCHG(&mem, &eax, r) {  
    if (eax == mem) {  
        mem = r; return true;  
    } else {  
        eax = mem; return false;  
    }  
}
```

Mit Hilfe einer atomaren CAS-Instruktion ist es möglich, ein neues Element in eine einfache lineare Liste einzufügen:

```
/* initialization */
struct Element { T* info; Element* next; };
Element* head = 0;

/* adding an element in front of the list pointed to by head */
void add_element(Element* head, Element* new_element) {
    r = new_element; eax = head;
    do {
        r->next = eax;
    } while (!CMPXCHG(head, eax, r));
}
```

Vorteile dieser Lösung:

- ▶ Die Schleife muss nur wiederholt werden, wenn **CMPXCHG** scheitert.
- ▶ Dies passiert aber nur, wenn ein konkurrierender Prozess aktiv ist und das gleiche probiert.
- ▶ Ein inaktiver Prozess kann (anders als beim Spin-Lock) hier niemanden aufhalten.
- ▶ Das Verfahren ist somit frei von Deadlocks und längeren Verzögerungen.
- ▶ Es gilt nur, dass ABA-Problem zu beachten (dazu später mehr).

- Das Speichermodell in C++ berücksichtigt zunächst nicht mehrere Threads. Stattdessen wird zunächst nur sichergestellt, dass der Speicher sich aus der Sicht eines Threads intuitiv korrekt verhält, d.h. SQ (*sequential consistency*) erfüllt ist.
- Wenn mehrere Threads auf die gleiche Datenstruktur zugreifen und dies nicht durch die Verwendung von `std::mutex`-Variablen geregelt wird, dann liegt ein *data race* vor und der Effekt ist undefiniert.
- Die Verwendung von **volatile** hilft hier nicht (anders als in Java).
- Da (wie präsentiert) die gängigen Prozessorarchitekturen Barriers, atomare Datentypen und atomare Instruktionen anbieten, lag es auch nahe, dies in C++ zu unterstützen.
- Die große Herausforderung lag hier darin, dass die zu unterstützenden Architekturen sehr unterschiedlich sind und dass andererseits eine Vereinfachung (wie etwa in Java) Optimierungsmöglichkeiten verschenkt.

- Wenn auf gemeinsame Datenstrukturen konkurrierend zugegriffen werden soll, ohne dies durch `std::mutex`-Variablen zu regeln, dann ist der Einsatz atomarer Datentypen zwingend notwendig.
- Atomare Datentypen werden mit Hilfe der `std::atomic`-Template-Klasse deklariert:

```
#include <atomic>
// ...
std::atomic<unsigned int> counter {0};
```

- Die **bool**-wertige Methode `is_lock_free` teilt mit, ob das Lesen und Schreiben des gesamten Objekts atomar ohne Locks möglich sind. Bei elementaren Datentypen ist dies normalerweise möglich, bei zusammengesetzten Datentypen eher nicht.
- Die Klasse bietet `load`- und `store`-Methoden, die eine Spezifikation der gewünschten `memory_order` ermöglichen.

Grundsätzlich kann auch `std::atomic` für selbstdefinierte Typen unter Voraussetzungen verwendet werden:

- ▶ Trivialer Default-Konstruktor
- ▶ Trivialer Dekonstruktor
- ▶ Standard-Layout
- ▶ Initialisierbar als Aggregat

Zulässig sind somit traditionelle Typen aus C, d.h. Tupel aus elementaren Datentypen einschließlich Zeigern. Entsprechend können solche Datentypen sicher mit `memcpy` kopiert und mit `memcmp` verglichen werden. Virtuelle Methoden sind nicht zulässig, Vergleichs-Operatoren werden nicht honoriert.

Neuere Übersetzer (wie `g++ 6.x`) unterstützen atomare Aktualisierungen solcher Typen ohne Locks, wenn das die Architektur unterstützt (bis zu 16 Bytes bei neueren x86-Architekturen).

- `std::memory_order` ist in C++ wie folgt definiert:

```
enum memory_order {  
    memory_order_relaxed,  
    memory_order_consume,  
    memory_order_acquire,  
    memory_order_release,  
    memory_order_acq_ret,  
    memory_order_seq_cst  
};
```

- Zu beachten ist hier, dass dies nicht nur die Prozessorarchitektur betrifft, sondern auch die Freiheit des Übersetzers, die Anordnung von Lade- und Speicherinstruktionen umzuordnen (*instruction scheduling*).
- Wenn nichts explizit angegeben wird, dann kommt `memory_order_seq_cst` zum Einsatz (*sequential consistency*), die dem SQ-Modell entspricht. Dies lässt sich ggf. noch nachvollziehen, kostet aber den Einsatz entsprechender Barriers.
- Die anderen Varianten erschweren sehr den Nachweis, kommen aber teilweise ohne Barriers aus.

- *memory_order_relaxed* steht dafür, dass es keinerlei Zusicherung über die Ausführungsreihenfolge gibt.
- Bjarne Stroustrup nennt dafür folgendes Beispiel mit den atomaren ganzzahligen Variablen *x* und *y*, jeweils mit 0 initialisiert:

```
// thread 1:  
    int r1 = y.load(memory_order_relaxed);  
    x.store(r1, memory_order_relaxed);  
// thread 2:  
    int r2 = x.load(memory_order_relaxed);  
    y.store(42, memory_order_relaxed);
```

- Dann ist es hier möglich, dass der zweite Thread danach den Wert 42 in der Variable *r2* vorfindet. Das liegt daran, dass sogar die Anordnung der Anweisungen des gleichen Threads sogar dann flexibilisiert werden, wenn dem Abhängigkeiten entgegenstehen. Denkbar ist also:

```
y.store(42, memory_order_relaxed); // thread 2  
int r1 = y.load(memory_order_relaxed); // thread 1  
x.store(r1, memory_order_relaxed); // thread 1  
int r2 = x.load(memory_order_relaxed); // thread 2
```

- *memory_order_relaxed* erfüllt nicht die Bedingungen, die an die RMO (*relaxed memory order*) im SPARC-Architecture-Manual gestellt werden.
- Bei der x86-Architektur und SPARC kann dies auf der Maschinenebene nicht vorkommen. Eine der Ausnahmen ist die DEC-Alpha-Architektur.
- Aber prinzipiell gibt *memory_order_relaxed* dem Übersetzer auch auf anderen Architekturen die Freiheit, entsprechende Anweisungen zu vertauschen.

```
std::atomic<unsigned int> counter {0};

void count() {
    for (unsigned int i = 0; i < 1000; ++i) {
        counter.fetch_add(1, std::memory_order_relaxed);
    }
}

int main() {
    std::vector<std::thread> threads;
    for (unsigned int i = 0; i < 10; ++i) {
        threads.push_back(std::thread(count));
    }
    for (auto& t: threads) t.join();
    std::cout << "counter = " << counter << std::endl;
}
```

- *fetch_add* ist eine atomare Lade- und Schreiboperation, die den Wert inkrementiert.
- Die Reihenfolge der Additionen ist hier nicht relevant, deswegen ist *memory_order_relaxed* ausreichend. Der Haupt-Thread kann das Resultat korrekt ausgeben, da er sich mit *join* synchronisiert.

- Häufig ist eine atomare Variable mit einer Datenstruktur verbunden. Wenn die Datenstruktur und danach die atomare Variable aktualisiert wird, soll beim Feststellen des neuen Werts der atomaren Variable auch die Datenstruktur aktualisiert vorgefunden werden.
- Zwei Threads P_1 und P_2 gelten als synchronisiert, wenn
 - ▶ Prozess P_1 die atomare Variable a mit *memory_order_release* aktualisiert und
 - ▶ Prozess P_2 leserweise auf die atomare Variable a mit *memory_order_acquire* zugreift und dabei den von P_1 geschriebenen Wert vorfindet.
- Für die entsprechende Schreiboperation X des Prozesses P_1 und den aktualisierten Wert vorfindenden Lese-Operation Y des Prozesses P_2 gilt dann die Relation $X <_s Y$.

Die *happens-before-Relation* $<_{hb}$ in C++ für einen Thread sei dann die minimale Relation, die folgende Bedingungen erfüllt:

- ▶ $X <_p Y \Rightarrow X <_{hb} Y$
- ▶ $X <_s Y \Rightarrow X <_{hb} Y$
- ▶ $X <_{hb} Y \wedge Y <_{hb} Z \Rightarrow X <_{hb} Z$

rel-acq.cpp

```
std::atomic<bool> done {false};
unsigned int result_of_overlong_computation = 0;

void compute() {
    result_of_overlong_computation = 42;
    done.store(true, std::memory_order_release);
}

void print_result() {
    /* busy loop that waits for done to become true */
    while (!done.load(std::memory_order_acquire)) {
        std::this_thread::yield();
    }
    std::cout << result_of_overlong_computation << std::endl;
}

int main() {
    std::thread t1(compute); std::thread t2(print_result);
    t1.join(); t2.join();
}
```

rel-acq.cpp

```
void compute() {
    result_of_overlong_computation = 42;
    done.store(true, std::memory_order_release);
}

void print_result() {
    /* busy loop that waits for done to become true */
    while (!done.load(std::memory_order_acquire)) {
        std::this_thread::yield();
    }
    std::cout << result_of_overlong_computation << std::endl;
}
```

- Für die Berechnung X und die Ausgabe Y gilt hier $X <_{hb} Y$, da *done.store* und *done.load* zur Synchronisierung führen, sobald *done.load* den von *done.store* abgespeicherten Wert zu sehen bekommt.
- *std::this_thread::yield()* weist den Scheduler an, dass andere Threads jetzt eher zum Zuge kommen sollten. Das ist etwas freundlicher als eine reine *busy*-Loop.

memory_order_acq_rel

Vereinigt *memory_order_acquire* mit *memory_order_release* und ist insbesondere sinnvoll für atomare Lese-und-Schreiboperationen analog zu den SWAP- und CAS-Instruktionen.

memory_order_consume

Ist die billigere Ausführung von *memory_order_acquire*, bei der die Sequentialisierung nur für die Ausdrücke gewährleistet ist, die direkt oder indirekt von der zu ladenden atomaren Variable abhängen.

memory_order_seq_cst

Ist die Voreinstellung, die global SQ erzwingt, d.h. mit *memory_order_seq_cst* versehene Lese- und Schreib-Operationen sind global über alle Threads hinweg total geordnet. Das ist vergleichsweise teuer.

lflist.hpp

```
template <typename T>
class LFList {
private:
    struct Element;
public:
    LFList() : head(nullptr) { }
    /* ... */
private:
    struct Element {
        Element() : next(nullptr) {
        }
        Element(const T& item) : item(item), next(nullptr) {
        }
        T item;
        Element* next;
    };
    std::atomic<Element*> head;
};
```

- Einfach verkettete Liste mit nur einem Ende.

```
void push(const T& item) {
    Element* element = new Element(item);
    Element* p = head.load();
    element->next = p;
    while (!head.compare_exchange_weak(p, element)) {
        element->next = p;
    }
}
```

- `head.compare_exchange_weak(p, element)` entspricht folgender atomaren Operation:

```
if (head == p && /* some good fortune */) {
    head = element;
    return true;
} else {
    p = head;
    return false;
}
```

lflist.hpp

```
void push(const T& item) {
    Element* element = new Element(item);
    Element* p = head.load();
    element->next = p;
    while (!head.compare_exchange_weak(p, element)) {
        element->next = p;
    }
}
```

- Alternativ gibt es *compare_exchange_strong*, da fällt die Nebenbedingung weg. Auf einigen Architekturen sind die beiden unterschiedlich.
- Schleifen mit *compare_exchange_strong* sind keine sinnlosen *busy*-Loops, da im Wiederholungsfall ein anderer Prozess nachweislich vorangekommen ist. Wenn kein Fortschritt andersweitig erfolgt, ist die Schleife in der nächsten Iteration erfolgreich.

lflist.hpp

```
bool pop(T& item) {
    Element* p = head.load();
    while (p && !head.compare_exchange_weak(p, p->next))
        ;
    if (p) {
        item = p->item;
        return true;
    } else {
        return false;
    }
}
```

- Alle Zugriffsoperationen erfolgen hier implizit mit *memory_order_seq_cst*. Das ist teuer, aber leichter nachzuvollziehen.

lflist.hpp

```
element->next = p;
while (!head.compare_exchange_weak(p, element)) {
    element->next = p;
}
```

Verwendung der Register im folgenden durch *g++* erzeugten
Assembler-Text:

```
%esi   element
%ecx   &head
%eax   &p
```

```
    movl    %eax, 8(%esi)
    mfence
.L352:
    movl    -64(%ebp), %eax
    lock cmpxchgl  %esi, (%ecx)
    je     .L343
    movl    %eax, -64(%ebp)
    movl    %eax, (%edx)
    mfence
    jmp    .L352
```

lflist.hpp

```
bool pop(T& item) {
    Element* p = head.load();
    // p can be a dangling reference here:
    while (p && !head.compare_exchange_weak(p, p->next))
        ;
    if (p) {
        item = p->item;
        delete p; // now with delete
        return true;
    } else {
        return false;
    }
}
```

- Die vorherige Fassung von *pop* verzichtete auf **delete**. Das ließe sich natürlich wie hier hinzufügen.
- Wenn aber mehrere *pop*-Operationen parallel laufen, kann es passieren, dass *p* auf ein gerade eben gelöscht Element zeigt.
- Der Zugriff auf *p->next* ist dann nicht mehr wohldefiniert.

```
std::atomic<Element*> head;  
std::atomic<Element*> free;
```

- Idee: Nicht mehr benötigte Elemente werden nicht mit **delete** freigegeben, sondern in eine andere Liste eingefügt.
- Dann wird zuerst überprüft, ob es noch wiederbenutzbare Elemente in der *free*-Liste gibt, bevor **new** aufgerufen wird.
- Erst bei dem Abbau der Liste werden die verbliebenen Elemente freigegeben:

```
~LFList() {  
    Element* next;  
    for (Element* p = head.load(); p; p = next) {  
        next = p->next;  
        delete p;  
    }  
    for (Element* p = free.load(); p; p = next) {  
        next = p->next;  
        delete p;  
    }  
}
```

lflist.hpp

```
void push_element(std::atomic<Element*>& head, Element* element) {
    Element* p = head.load();
    element->next = p;
    while (!head.compare_exchange_weak(p, element)) {
        element->next = p;
    }
}

Element* pop_element(std::atomic<Element*>& head) {
    Element* p = head.load();
    while (p && !head.compare_exchange_weak(p, p->next))
        ;
    return p;
}
```

- Da wir nun zwei Listen haben, lohnt es sich entsprechend verallgemeinerte private Methoden zu haben.

```
void push(const T& item) {
    Element* element = pop_element(free);
    if (element) {
        element->item = item;
    } else {
        element = new Element(item);
    }
    push_element(head, element);
}

bool pop(T& item) {
    Element* p = pop_element(head);
    if (p) {
        item = p->item;
        push_element(free, p);
        return true;
    } else {
        return false;
    }
}
```

- Die öffentlichen Methoden *push* und *pop* berücksichtigen nun die *free*-Liste.

```
Element* pop_element(std::atomic<Element*>& head) {
    Element* p = head.load();
    while (p && !head.compare_exchange_weak(p, p->next))
        ;
    return p;
}
```

- Der Aufruf von *head.compare_exchange_weak* erfolgt in drei Schritten:
 1. *p* laden,
 2. *p->next* laden und die
 3. Methode *compare_exchange_weak* aufrufen.
- Zwischen dem zweiten und dritten Schritt könnte eine parallel laufende *pop*-Operation das Element entfernen, wodurch es in die *free*-Liste eingefügt wird. Wenn weitere Änderungen erfolgen einschließlich einem *push*, dann könnte aus der *free*-Liste das zuvor gelöschte Element wieder eingefügt werden.
- Dann hat möglicherweise der erste Parameter den gleichen Wert von *head*, allerdings ist es dann gut möglich, dass der zweite Parameter nicht mehr *p->next* entspricht.

- Bei *compare_exchange_weak* aktualisieren wir atomar einen Wert (hier *head*) unter der impliziten Annahme, dass während des Updates der zweite Parameter noch den Wert von $p \rightarrow head$ hat.
- Dies muss aber nicht der Fall sein.
- Das ist das ABA-Problem, d.h. wir „sehen“ *A* und übergeben die beiden Parameter, dann gibt es zwischendurch *B*, das nicht beobachtet wird und danach kehrt *A* zurück (jedoch mit einem anderen *next*-Zeiger), worauf mit dem Update die Datenstruktur korrumpiert wird.

thread #1	thread # 2	thread # 3	Status
			head \rightarrow A \rightarrow B \rightarrow C free \rightarrow
<pre> pop pop_element(head) p = head.load() // p points to A load p->next // p->next points to B </pre>			
	<pre> pop </pre>		head \rightarrow B \rightarrow C free \rightarrow A
	<pre> pop p = pop_element(head) </pre>		head \rightarrow C free \rightarrow A
		<pre> push </pre>	head \rightarrow A \rightarrow C free \rightarrow
<pre> head.compare_exchange_weak (A, B) // succeeds </pre>			head \rightarrow B \rightarrow C free \rightarrow

- A, B und C repräsentieren die Adressen von Elementen und nicht deren Inhalt, der sich hier in diesem Szenario bei A durch die *push*-Operation bei thread # 3 verändert.

Der prinzipielle Lösungsansatz sieht vor, dass A nicht unerwartet auftaucht, nachdem es durch B ersetzt wurde und während noch ein Thread in einer entsprechenden Ersetzungsschleife sich aufhält.

Dazu gibt es folgende Lösungsansätze:

- ▶ Aus A wird ein Tupel, bestehend aus dem eigentlichen zu aktualisierenden Wert und einem zusätzlichen einmaligen Wert. Letzteres kann beispielsweise durch einen atomaren Zähler erreicht werden. Auf diese Weise wird ABA durch ABA' ersetzt, d.h. der zweite Wert des Tupels unterscheidet sich.
- ▶ Da das Problem bei Zeigern durch Recycling entsteht, könnte das Recycling zeitlich auf einen Moment verschoben werden, in dem sich kein Thread in einer Ersetzungsschleife befindet.

Beide Ansätze verursachen zusätzliche Kosten, da weitere atomare Operationen anfallen. Nur neuere Übersetzer (wie etwa `g++ 6.x`) sind in der Lage, Tupel atomar ohne die Verwendung von Locks zu aktualisieren. Dies geht auf moderneren Architekturen, die atomare 128-Bit-Operationen unterstützen.

- Bei einer Datenbank verändern Transaktionen den Zustand in einer Weise, die
 - ▶ atomar (Ausführung ganz oder überhaupt nicht),
 - ▶ konsistent (wenn die Datenbank zuvor konsistent war, bleibt sie es),
 - ▶ isoliert (konkurrierende Transaktionen beeinflussen sich nicht gegenseitig) und
 - ▶ dauerhaftist (ACID-Prinzip).
- Dieses Konzept wird auch gerne auf den Hauptspeicher übernommen, wobei hier die Dauerhaftigkeit wegfällt.
- Die prinzipielle Idee ist, dass vom optimistischen Fall ausgegangen wird, dass kein Konflikt vorliegt. Wird dennoch ein solcher festgestellt, dann wird eine der betroffenen Transaktionen zurückgerollt und neu gestartet, sobald die andere abgeschlossen ist.
- Von der Nutzerseite ist ein besonderer Vorteil darin zu sehen, dass die Verwendung expliziter Locks wegfällt.

- Prinzipiell sind jetzt bereits einige Architekturen wie die modernen x86-Prozessoren in der Lage, im Rahmen des Pipelinings im Umgang mit bedingten Sprüngen mehrere Varianten parallel spekulativ zu verfolgen und dann mit Verzögerung sich für eine der Varianten zu entscheiden.
- Das lässt sich auch ausdehnen auf Speicherzugriffe, wenn die Cache-Hierarchie einen spekulativen Zustand unterstützt, der im Erfolgsfalle atomar in den Hauptspeicher abgesichert werden kann oder der im Falle eines Abbruchs der Transaktion wieder zurückgerollt werden kann.
- Der Cache kann dann ähnlich wie die Register zumindest partiell als zum lokalen Zustand eines Cores gehörig betrachtet werden.

Sun Microsystems plante dies für den 2005 bis 2009 entwickelten Rock-Prozessor, der die SPARC-Architektur weiterführen sollte:

- ▶ Jeder Core unterstützte 2 reguläre Threads und je nach Konfiguration ein oder zwei sogenannte Scout-Threads, die die spekulative Ausführung einer Transaktion übernahmen.
- ▶ Im Rahmen einer Transaktion markierte ein Scout-Thread im eigenen L1-Cache veränderte Cache-Lines und führte in einem Commit-Buffer eine Liste der veränderten Cache-Lines.
- ▶ Im Erfolgsfalle wurde der Commit-Buffer genutzt, um die veränderten Cache-Lines aus dem L1 atomar im Hauptspeicher zu sichern.
- ▶ Wenn ein eine veränderte Cache-Line durch einen anderen Thread invalidiert wurde, kam es zum Abbruch der Transaktion.
- ▶ Die Kapazitäten waren wegen der Beschränkung auf den L1 mit 32 KiB begrenzt.

In seiner Supercomputer-Serie Blue Gene führte IBM 2011 bei den Blue-Gene/Q-Systemen ebenfalls Transactional Memory ein:

- ▶ Statt dem L1 wird hier der L2 mit 32 MiB verwendet.
- ▶ Es werden keine Scout-Threads benötigt. Stattdessen unterstützt der L2-Cache mehrere Versionen für eine Cache-Line und entsprechende atomare Operationen.
- ▶ Seit 2015 wurde von IBM die weitere Entwicklung der Blue-Gene-Familie eingestellt.

TSX sind ein Erweiterung der x86-Architektur, die zunächst auf der Haswell- und Broadwell-Architektur eingeführt wurde, dort aber fehlerhaft implementiert war. Benutzbar ist TSX somit ab der 2015 eingeführten Skylake-Architektur.

- ▶ Zum Einsatz kommt hier der L1-Cache mit 32 KiB (auf Haswell).
- ▶ Wie bei Rock führen Cache-Konflikte zum Abbruch der Transaktion.
- ▶ Wenn eine spekulativ veränderte Cache-Line ersetzt werden muss (*cache line eviction*), dann wird die Transaktion ebenfalls abgebrochen.
- ▶ Umgesetzt wird dies über x86-Instruktionspräfixe **XACQUIRE** und **XRELEASE** (sind NOPs bei älteren Architekturen, in Kombination mit dem **LOCK**-Präfix, Ausführung zunächst ohne **LOCK**, im Erfolgsfalle Wiederholung mit **LOCK**) bzw. **XBEGIN** und **XEND**, bei denen explizit spezifiziert wird, wie im Falle einer abgebrochenen Transaktion zu reagieren ist, und die mit der **XABORT**-Instruktion einen Abbruch ermöglichen.

- Mit N4514 gibt es einen Vorschlag für eine entsprechende C++-Erweiterung.
- Momentan ist nicht abzusehen, ob und wann das in den C++-Standard aufgenommen wird. Für C++20 ist dies bislang nicht geplant.
- g++ bietet ab 6.1 eine entsprechende Unterstützung mit der Option „-fgnu-tm“. Nach meiner Erfahrung führt diese aber noch häufig zu internen Übersetzerfehlern und ist daher noch nicht ausgereift.
- Angeboten werden Synchronisationsblöcke (keine Transaktionen, aber gegenseitiger Ausschluss ohne explizite Locks) und atomare Blöcke, die auch den Abbruch einer Transaktion vorsehen. Letztere haben das Potential mit entsprechenden Hardware-Erweiterungen umgesetzt zu werden, kommen daher aber auch mit einer Reihe von Restriktionen.