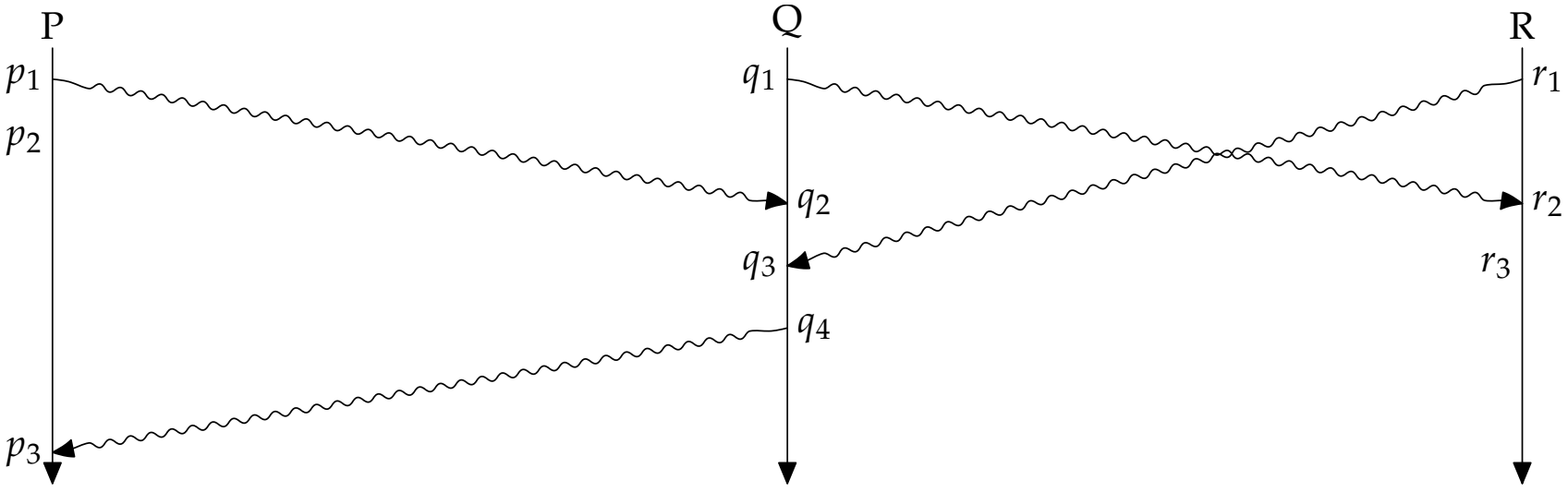
- Auf Basis von CSP lassen sich die einzelnen Threads und die Speicherkomponenten als separate Prozesse $P_1 \ldots P_n$ modellieren.
- Diese haben untereinander keine Ereignisse gemeinsam, d.h. es gelte $\alpha P_i \cap \alpha P_j = \emptyset \; \forall i \neq j$.
- Die Kommunikation zwischen den Prozessen $P_1 \ldots P_n$ erfolge über die Netzwerkkomponenten N_{ij} mit $i < j$, d.h. es gelte $\alpha P_i \cap \alpha N_{ij} \neq \emptyset \;\wedge\; \alpha P_j \cap \alpha N_{ij} \neq \emptyset$.
- Weiter gelte, dass jedes Ereignis eines Prozesses P_i nur einmal eintrete, d.h. $\mathit{trace}(P_i) \upharpoonright \langle a \rangle \leq \langle a \rangle \; \forall a \in \alpha P_i$.
- Bei sich wiederholenden Prozessen bedeutet dies, dass wir die sonst gleichen Ereignisse durchnumerieren und wegen der Endlichkeit des Alphabets damit auch die Abläufe endlich sind, d.h. wir betrachten nur einen sehr begrenzten Ablauf.



- Beispiel: Dargestellt sind die Prozesse P, Q und R, jeweils mit den zugehörigen Ereignissen in entsprechenden Zeitlinien. Gewellte Linien deuten eine Nachrichtenübertragung ein. So steht p_1 für das Senden der Nachricht, die später im Rahmen des Ereignisses q_2 empfangen wird. Dann wissen wir, dass p_1 dem Ereignis q_2 vorausgehen muss. Ebenso wissen wir, dass p_3 dem Ereignis r_1 folgt.
- Wir wissen jedoch nicht, ob p_2 oder r_3 zuerst eintreten. Oder ob p_2 vor oder nach q_4 eintritt.

Leslie Lamport führte 1978 die folgende *happens-before*-Relation ein: Die Relation $\rightarrow$ sei definiert als die kleinste Relation, die die folgenden Bedingungen erfüllt:

(1) Wenn a und b Ereignisse des gleichen Prozesses sind und a vor b eintritt, dann gelte $a \rightarrow b$.

(2) Wenn a für das Senden einer Nachricht und b für das Empfangen derselben Nachricht steht, dann gelte $a \rightarrow b$.

(3) Wenn $a \rightarrow b$ und $b \rightarrow c$ gilt, dann gelte auch $a \rightarrow c$.

Zwei Ereignisse werden als *parallel* betrachtet, falls $a \not\rightarrow b$ und $b \not\rightarrow c$.

- 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 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 es in eine Cache-Line fällt.

- 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 Wegs 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 einen 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));
}
```

- 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 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.

- *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 möglicherweise 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.

Zähler mit *memory_order_relaxed*

counter.cpp

```
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 lesenderweise 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++ 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 hier etwas freundlicher als eine nackte *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;
         std::atomic<Element*> next;
      };
      std::atomic<Element*> head;
};
```

- Einfach verkettete Liste mit nur einem Ende.

lflist.hpp

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