Best Practices

Rainer Grimm

Training, Coaching und Technologieberatung

www.ModernesCpp.de

Best Practices

- Allgemein
- Multithreading
- ~~Parallelität~~
- Speichermodell

Best Practices

Allgemein

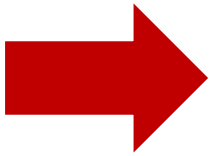
Multithreading

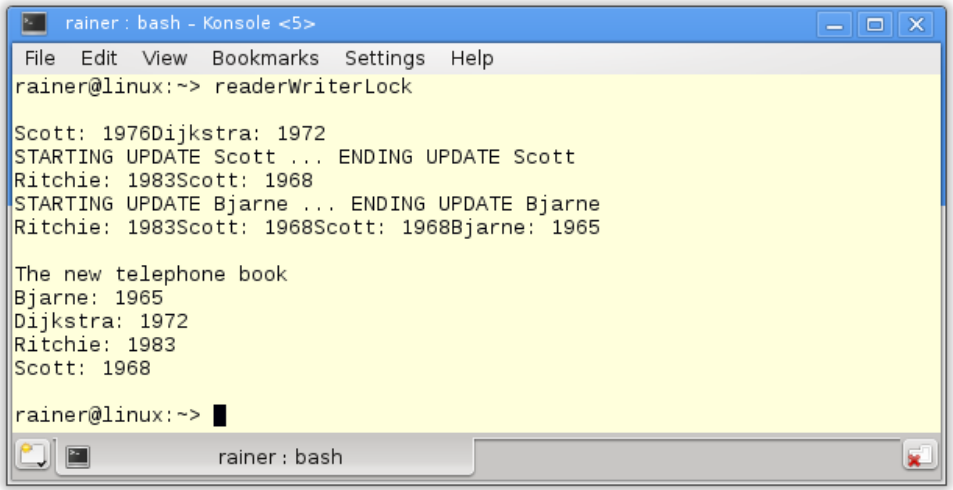
Speichermodell

Code Reviews

```
map<string,int> teleBook{{"Dijkstra", 1972},
                         {"Scott", 1976}, {"Ritchie", 1983}};

shared_timed_mutex teleBookMutex;

void addToTeleBook(const string& na, int tele){
  lock_guard<shared_timed_mutex> writerLock(teleBookMutex);
  cout << "\nSTARTING UPDATE " << na;
  this_thread::sleep_for(chrono::milliseconds(500));
  teleBook[na]= tele;
  cout << " ... ENDING UPDATE " << na << endl;
}

void printNumber(const string& na){
  shared_lock<shared_timed_mutex> readerLock(teleBookMutex);
  cout << na << ": " << teleBook[na] << endl;
}
```

```
thread reader1([]{ printNumber("Scott"); });
thread reader2([]{ printNumber("Ritchie"); });
thread w1([]{ addToTeleBook("Scott",1968); });
thread reader3([]{ printNumber("Dijkstra"); });
thread reader4([]{ printNumber("Scott"); });
thread w2([]{ addToTeleBook("Bjarne",1965); });
thread reader5([]{ printNumber("Scott"); });
thread reader6([]{ printNumber("Ritchie"); });
thread reader7([]{ printNumber("Scott"); });
thread reader8([]{ printNumber("Bjarne"); });

reader1.join(), reader2.join();
reader3.join(), reader4.join();
reader5.join(), reader6.join();
reader7.join(), reader8.join();
w1.join(), w2.join();

cout << "\nThe new telephone book" << endl;
for (auto teleIt: teleBook){
  cout << teleIt.first << ": " << teleIt.second << endl;
}
```

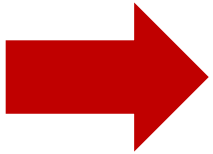




Code Reviews

```
map<string,int> teleBook{{"Dijkstra", 1972},
                         {"Scott", 1976}, {"Ritchie", 1983}};

shared_timed_mutex teleBookMutex;

void addToTeleBook(const string& na, int tele){
  lock_guard<shared_timed_mutex> writerLock(teleBookMutex);
  cout << "\nSTARTING UPDATE " << na;
  this_thread::sleep_for(chrono::milliseconds(500));
  teleBook[na]= tele;
  cout << " ... ENDING UPDATE " << na << endl;
}

void printNumber(const string& na){
  shared_lock<shared_timed_mutex> readerLock(teleBookMutex);
  cout << na << ": " << teleBook[na] << endl;
}
```

```
thread reader1([]{ printNumber("Scott"); });
thread reader2([]{ printNumber("Ritchie"); });
thread w1([]{ addToTeleBook("Scott",1968); });
thread reader3([]{ printNumber("Dijkstra"); });
thread reader4([]{ printNumber("Scott"); });
thread w2([]{ addToTeleBook("Bjarne",1965); });
thread reader5([]{ printNumber("Scott"); });
thread reader6([]{ printNumber("Ritchie"); });
thread reader7([]{ printNumber("Scott"); });
thread reader8([]{ printNumber("Bjarne"); });

reader1.join(), reader2.join();
reader3.join(), reader4.join();
reader5.join(), reader6.join();
reader7.join(), reader8.join();
w1.join(), w2.join();

cout << "\nThe new telephone book" << endl;
for (auto teleIt: teleBook){
  cout << teleIt.first << ": " << teleIt.second << endl;
}
```



rainer : bash — Konsole

File Edit View Bookmarks Settings Help

```
rainer@seminar:~> readerWriterLocks

Bjarne: 0Ritchie: 1983
STARTING UPDATE Scott ... ENDING UPDATE Scott

STARTING UPDATE Bjarne ... ENDING UPDATE Bjarne
Ritchie: 1983Scott: 1968Scott: 1968Scott: 1968Dijkstra: 1972Scott: 1968

The new telephone book
Bjarne: 1965
Dijkstra: 1972
Ritchie: 1983
Scott: 1968

rainer@seminar:~>
```

rainer : bash

Minimiere Teilen

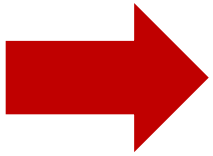
- Summation eines Vektors mit 100 Millionen Elementen

```
constexpr long long size = 100000000;

...

// random values
std::vector<int> randValues;
randValues.reserve(size);

std::random_device seed;
std::mt19937 engine(seed());
std::uniform_int_distribution<> uniformDist(1, 10);
for (long long i = 0 ; i < size ; ++i)
       randValues.push_back(uniformDist(engine));

...
// calculate sum
...
```

Minimiere Teilen

- Single-Threaded in zwei Variationen

```
unsigned long long sum {};
for (auto n: randValues) sum += n;

const unsigned long long sum = accumulate(randValues.begin(),
                                          randValues.end(), 0ll);
```

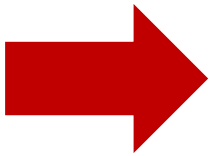


```
File  Edit  View  Bookmarks  Settings  Help
rainer@suse:~> calculateWithStd

Time for addition 0.0651712 seconds
Result: 550030112

rainer@suse:~>
rainer : bash
```

Minimiere Teilen

- Vier Threads mit einer geteilten Summationsvariable

```
void sumUp(unsigned long long& sum, const vector<int>& val,
           unsigned long long beg, unsigned long long end){
    for (auto it = beg; it < end; ++it){
        lock_guard<mutex> myLock(myMutex);
        sum += val[it];
    }
}

...
unsigned long long sum{};

thread t1(sumUp, ref(sum), ref(randValues), 0, fir);
thread t2(sumUp, ref(sum), ref(randValues), fir, sec);
thread t3(sumUp, ref(sum), ref(randValues), sec, thi);
thread t4(sumUp, ref(sum), ref(randValues), thi, fou);
```

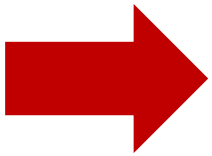


```
File  Edit  View  Bookmarks  Settings  Help
rainer@suse:~> calculateWithLock

Time for addition 3.3389 seconds
Result: 549961505

rainer@suse:~>
rainer : bash
```

Minimiere Teilen

- Vier Threads mit einer einer geteilten, atomaren Summationsvariable

```
void sumUp(atomic<unsigned long long>& sum, const vector<int>& val,
           unsigned long long beg, unsigned long long end){
    for (auto it = beg; it < end; ++it){
        sum += val[it];
    }
}

void sumUp(atomic<unsigned long long>& sum, const vector<int>& val,
           unsigned long long beg, unsigned long long end){
    for (auto it = beg; it < end; ++it){
        sum.fetch_add(val[it], memory_order_relaxed);
    }
}
```

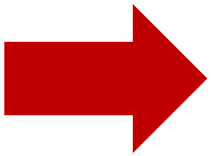


```
File  Edit  View  Bookmarks  Settings  Help
rainer@suse:~> calculateWithAtomic

sum.is_lock_free(): true

Time for addition 1.33837 seconds
Result: 549992025

Time for addition 1.34625 seconds
Result: 549992025

rainer@suse:~>
rainer : bash
```

Minimiere Teilen

- Vier Threads mit einer lokalen Summationsvariable

```
void sumUp(unsigned long long& sum, const vector<int>& val,
           unsigned long long beg, unsigned long long end){
    unsigned long long tmpSum{};
    for (auto i = beg; i < end; ++i){
        tmpSum += val[i];
    }
    lock_guard<mutex> lockGuard(myMutex);
    sum += tmpSum;
}
```



```
File  Edit  View  Bookmarks  Settings  Help
rainer@suse:~> localVariable

Time for addition 0.0284271 seconds
Result: 549996948

rainer@suse:~>
rainer : bash
```

Minimiere Teilen

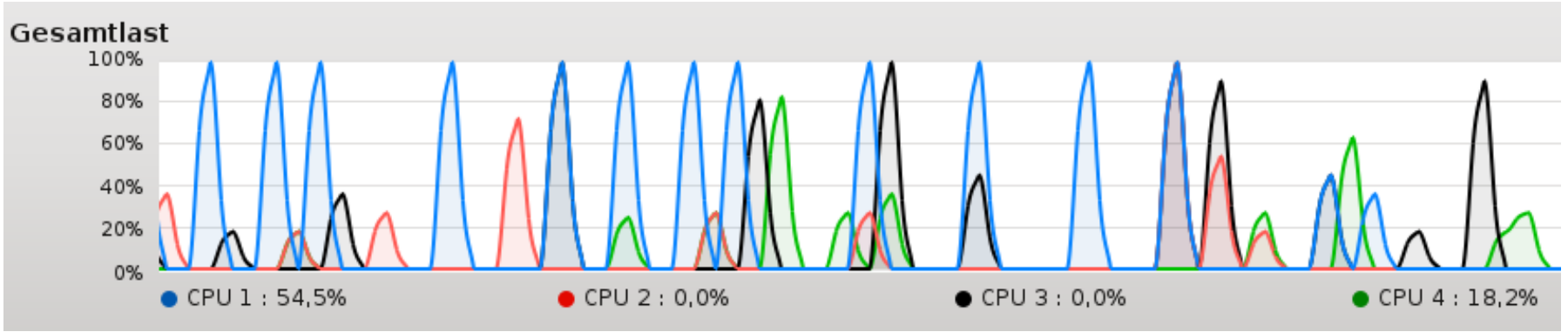
- Die Ergebnisse:

Single-Threaded	4 Threads mit Lock	4 Threads mit atomarer Variable	4 Threads mit lokaler Variable
0.07 sec	3.34 sec	1.34 sec	0.03 sec

Alles gut?

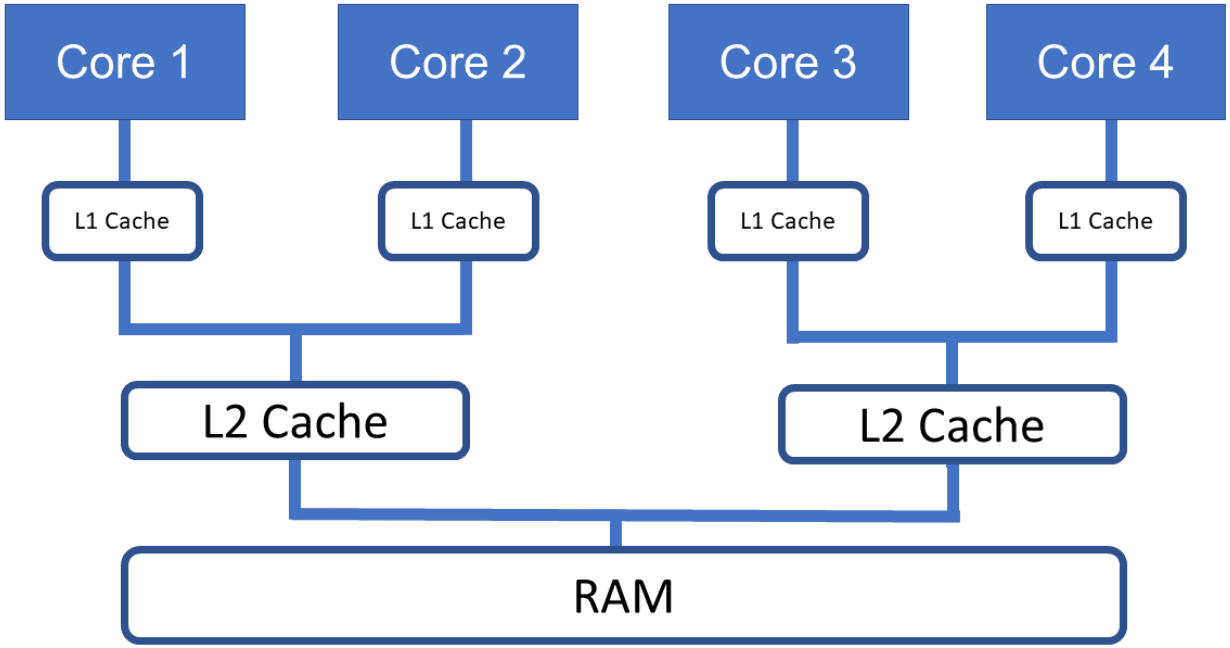
Minimiere Teilen

- Die CPU-Auslastung



Minimiere Teilen

- Die Memory-Wall

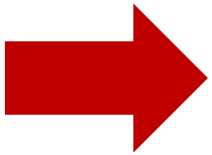


Der Speicherzugriff bremst die Applikation aus.

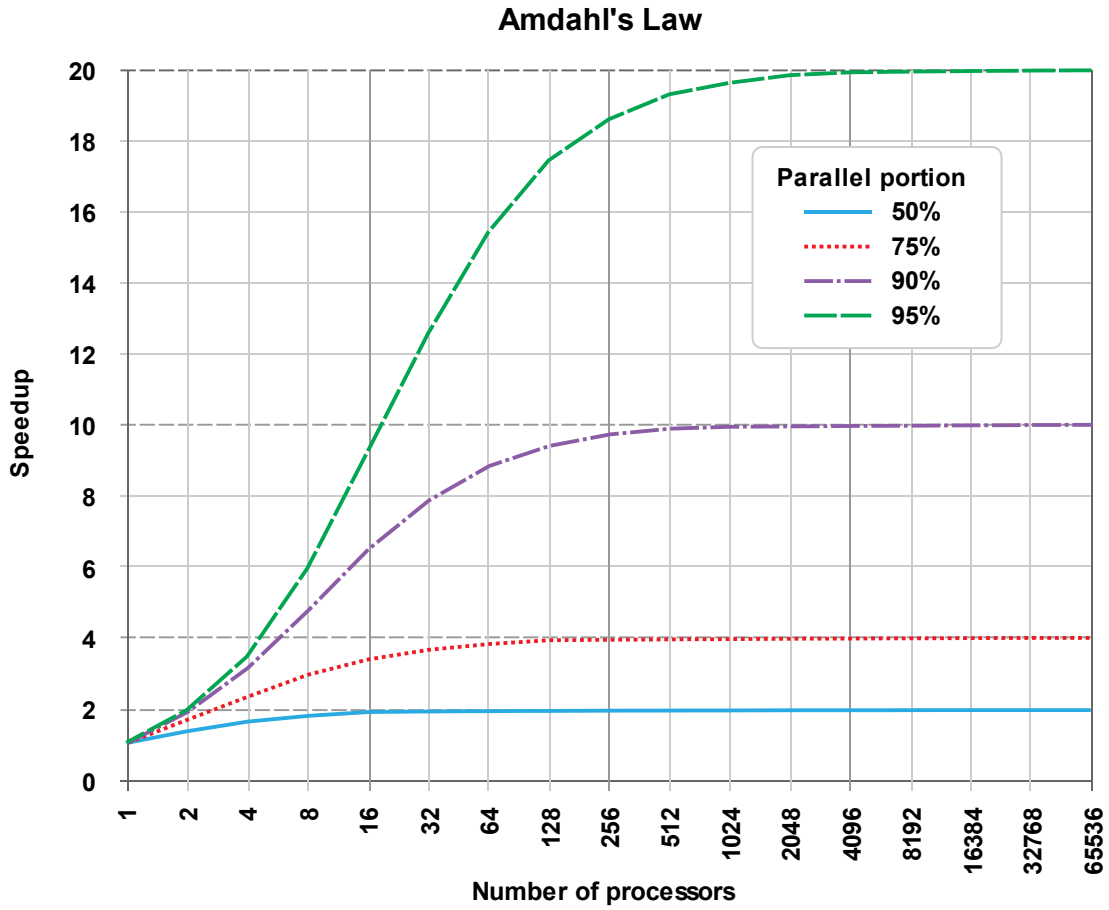
Minimiere Warten (Amdahl's Law)

$$\frac{1}{1 - p}$$

p: $Parallele\ Code$

$p = 0.5$  2

Mininimiere Warten (Amdahl‘s Law)



Verwende Codeanalyse Werzeuge (ThreadSanitizer)

- Findet Data Races
- Speicher- und Performanzoverhead: 10x

```
#include <thread>

int main(){

  int globalVar{};
  std::thread t1([&globalVar]{ ++globalVar; });
  std::thread t2([&globalVar]{ ++globalVar; });
  t1.join(), t2.join();

}
```

Verwende Codeanalyse Werzeuge (ThreadSanitizer)

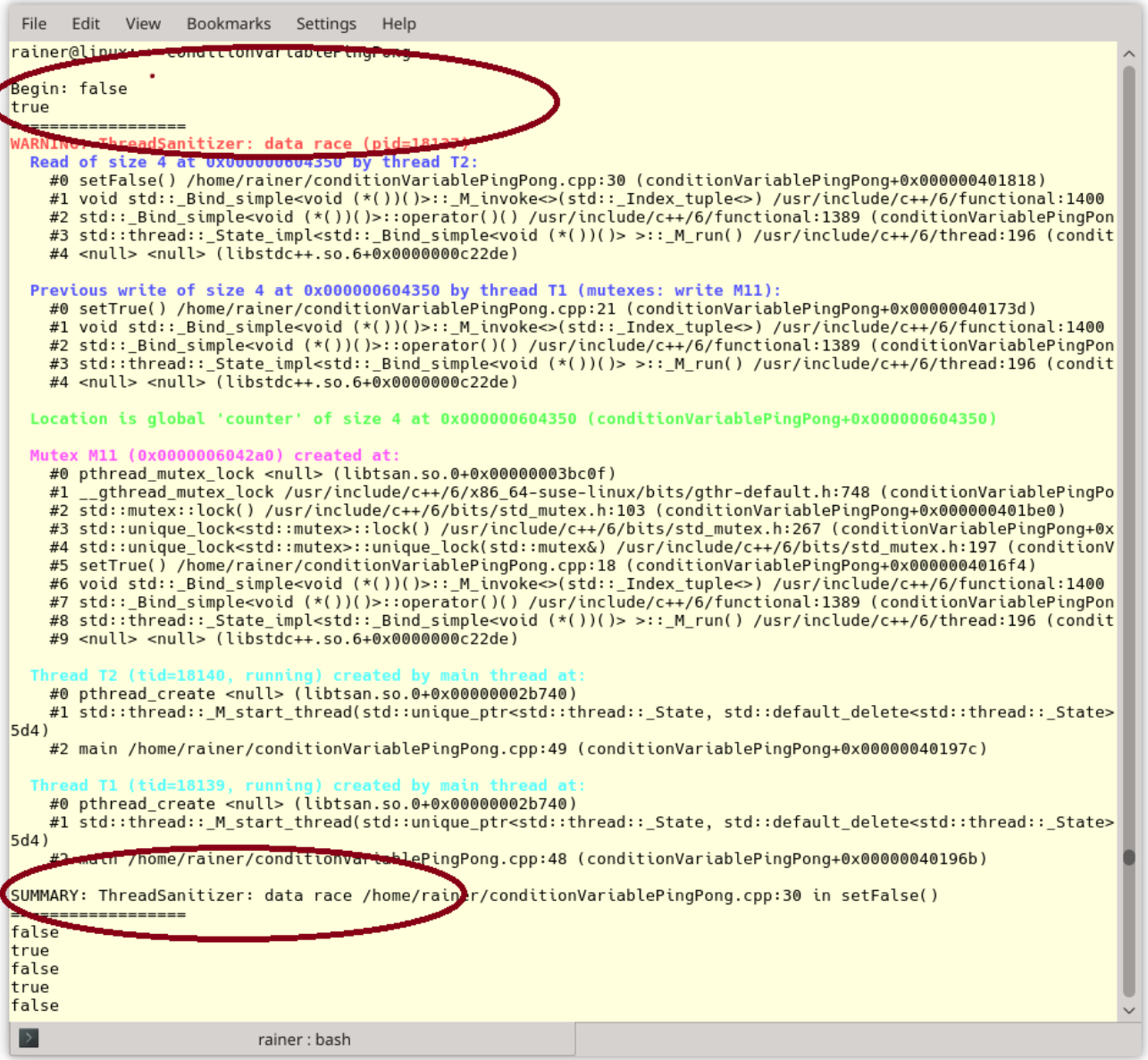
```
g++ dataRace.cpp -fsanitize=thread -pthread -g -o dataRace
```

Verwende Codeanalyse Werzeuge (ThreadSanitizer)

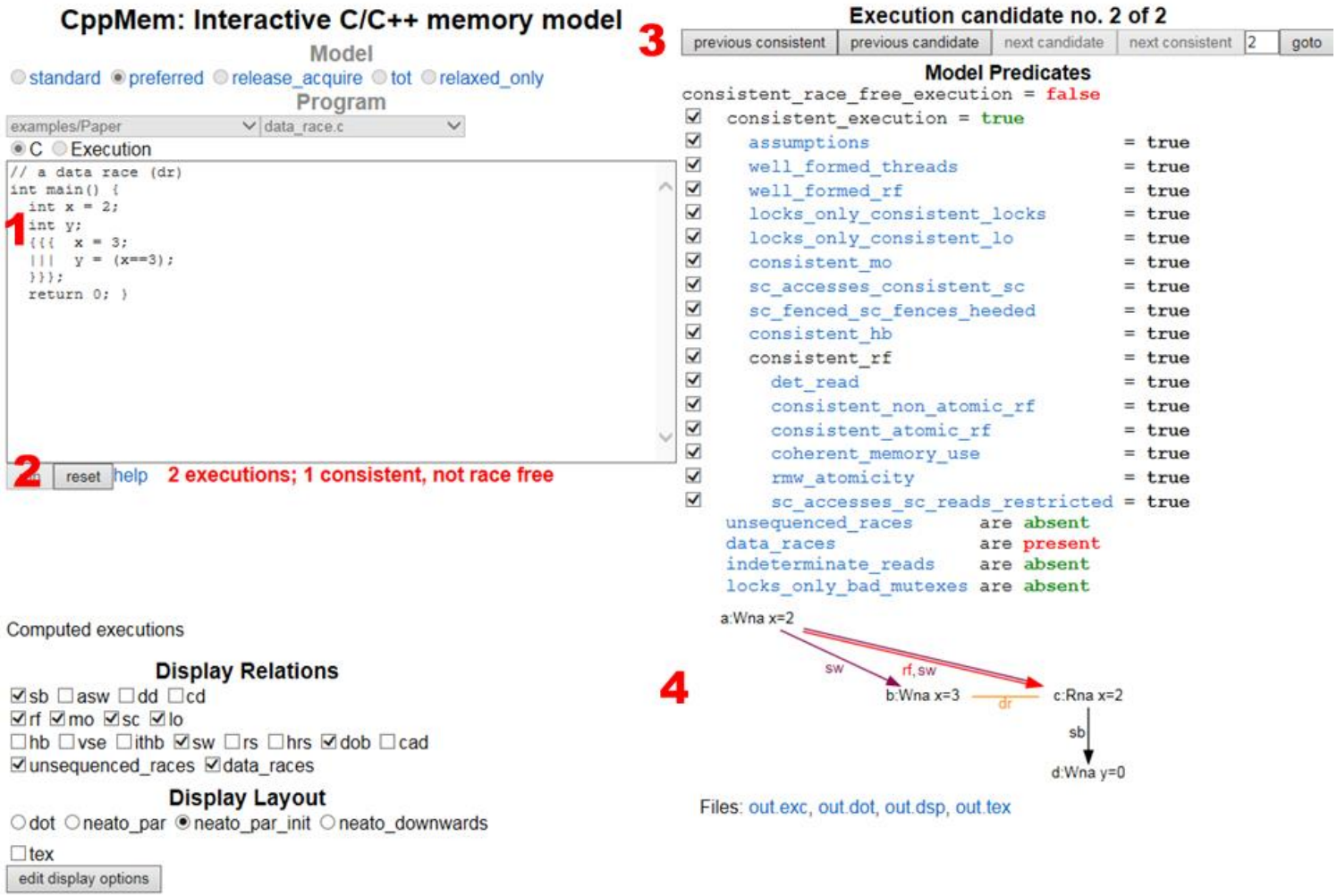
```
bool dataReady= false;

std::mutex mutex_;
std::condition_variable condVar1;
std::condition_variable condVar2;

int counter=0;
int COUNTLIMIT=50;

void setTrue(){

  while(counter <= COUNTLIMIT){

    std::unique_lock<std::mutex> lck(mutex_);
    condVar1.wait(lck,[]{return dataReady == false;});
    dataReady= true;
    ++counter;
    std::cout << dataReady << std::endl;
    condVar2.notify_one();

  }
}
```

```
void setFalse(){

  while(counter < COUNTLIMIT){

    std::unique_lock<std::mutex> lck(mutex_);
    condVar2.wait(lck,[]{return dataReady == true;});
    dataReady= false;
    std::cout << dataReady << std::endl;
    condVar1.notify_one();

  }

}

int main(){

  std::cout << std::boolalpha << std::endl;

  std::cout << "Begin: " << dataReady << std::endl;

  std::thread t1(setTrue);
  std::thread t2(setFalse);

  t1.join();
  t2.join();

  dataReady= false;
  std::cout << "End: " << dataReady << std::endl;

  std::cout << std::endl;
}
```

Verwende Codeanalyse Werzeuge (ThreadSanitizer)



Verwende Codeanalyse Werkzeuge (CppMem)

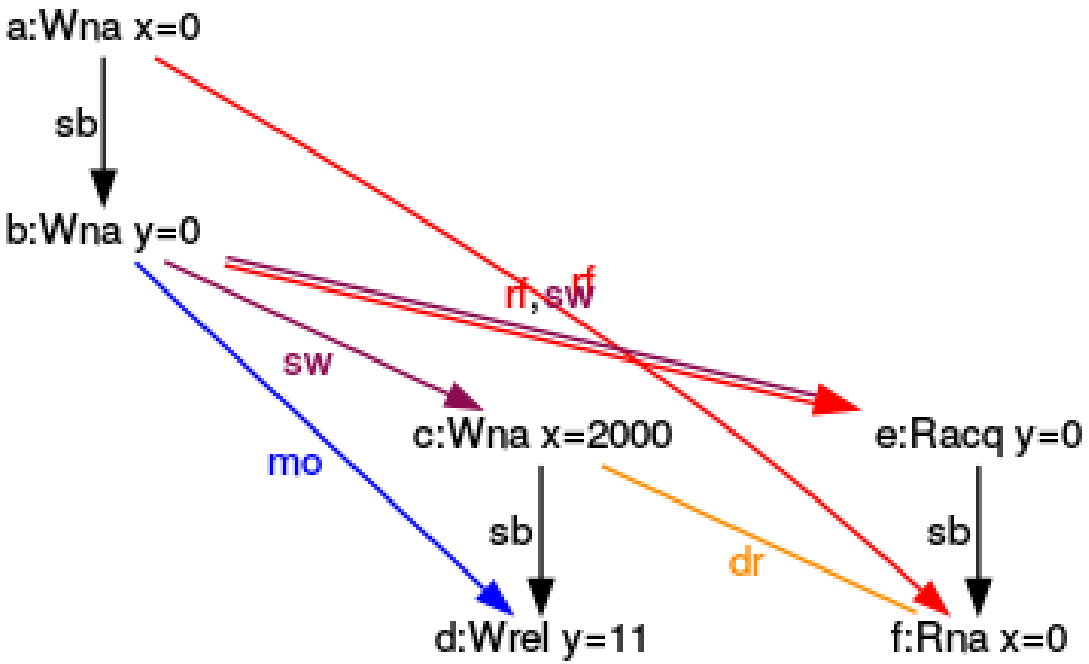


Verwende Codeanalyse Werkzeuge (CppMem)

```
int x = 0, std::atomic<int> y{0};

void writing(){
  x = 2000;
  y.store(11, std::memory_order_release);
}

void reading(){
  std::cout << y.load(std::memory_order_acquire) << " ";
  std::cout << x << std::endl;
}
...
std::thread thread1(writing);
std::thread thread2(reading);
thread1.join(), thread2.join();
```

Verwende Codeanalyse Werkzeuge (CppMem)

```
int x = 0, atomic_int y = 0;
{{{ {
  x = 2000;
  y.store(11, memory_order_release);
    }
||| {
  y.load(memory_order_acquire);
  x;
    }
}}}
```



Verwende unveränderliche Daten

Data Race: Mindestens zwei Threads greifen zu einem Zeitpunkt auf gemeinsame Daten zu. Zumindestens ein Thread versucht diese zu verändern.

Geteilt? \ Veränderlich?	nein	ja
nein	OK	Ok
ja	OK	**Data Race**

Verwende reine Funktionen

Reine Funktionen
Erzeugen immer dasselbe Ergebnis, wenn sie die gleichen Argumente erhalten.
Besitzen keine Seiteneffekte.
Verändern nie den globalen Zustand des Programms.

Vorteile

- Korrektheitsbeweise sind einfacher durchzuführen.
- Refaktoring und Tests sind einfacher möglich.
- Ergebnisse von Funktionsaufrufe können gespeichert werden.
- **Die Ausführungsreihenfolge der Funktion kann (automatisch) umgeordnet oder parallelisiert werden.**

Best Practices

Allgemein

Multithreading

Speichermodell

Tasks statt Threads

Thread

```
int res;
thread t([&]{ res = 3 + 4; });
t.join();
cout << res << endl;
```

Task

```
auto fut = async([]{ return 3 + 4; });
cout << fut.get() << endl;
```

Kriterium	Thread	Task
Beteiligten	Erzeuger- und Kinderthread	Promise und Future
Kommunikation	gemeinsame Variable	Kommunikationskanal
Threaderzeugung	verbindlich	optional
Synchronisation	`join`-Aufruf wartet	`get`-Aufruf blockiert
Ausnahme im Kinderthread	Kinder- und Erzeugerthread enden	Rückgabewert des `get` -Aufrufs
Formen der Kommunikation	Werte	Werte, Benachrichtigungen und Ausnahmen

Tasks statt Threads

C++20: Erweiterte Future werden die Komposition unterstützen

- **`then`**: Führe den Future aus, sobald der vorherige Future fertig ist.
- **`when_any:`** Führe den Future aus, sobald einer der Futures fertig ist.
- **`when_all:`** Führe den Future aus, sobald alle Futures fertig sind.

Tasks statt Bedingungsvariablen

Thread 1

```
{
  lock_guard<mutex> lck(mut);
  ready = true;
}
condVar.notify_one();
```

Thread 2

```
{
  unique_lock<mutex>lck(mut);
  condVar.wait(lck, []{ return ready; });
}
```

```
prom.set_value();
```

```
fut.wait();
```

Kriterium	Bedingungsvariablen	Tasks
Kritischer Bereich	Ja	Nein
Spurious Wakeup	Ja	Nein
Lost Wakeup	Ja	Nein
Mehrmalige Synchronisation möglich	Ja	Nein

Mutexe in Locks verpacken

Keine Freigabe des Mutex

- mutex

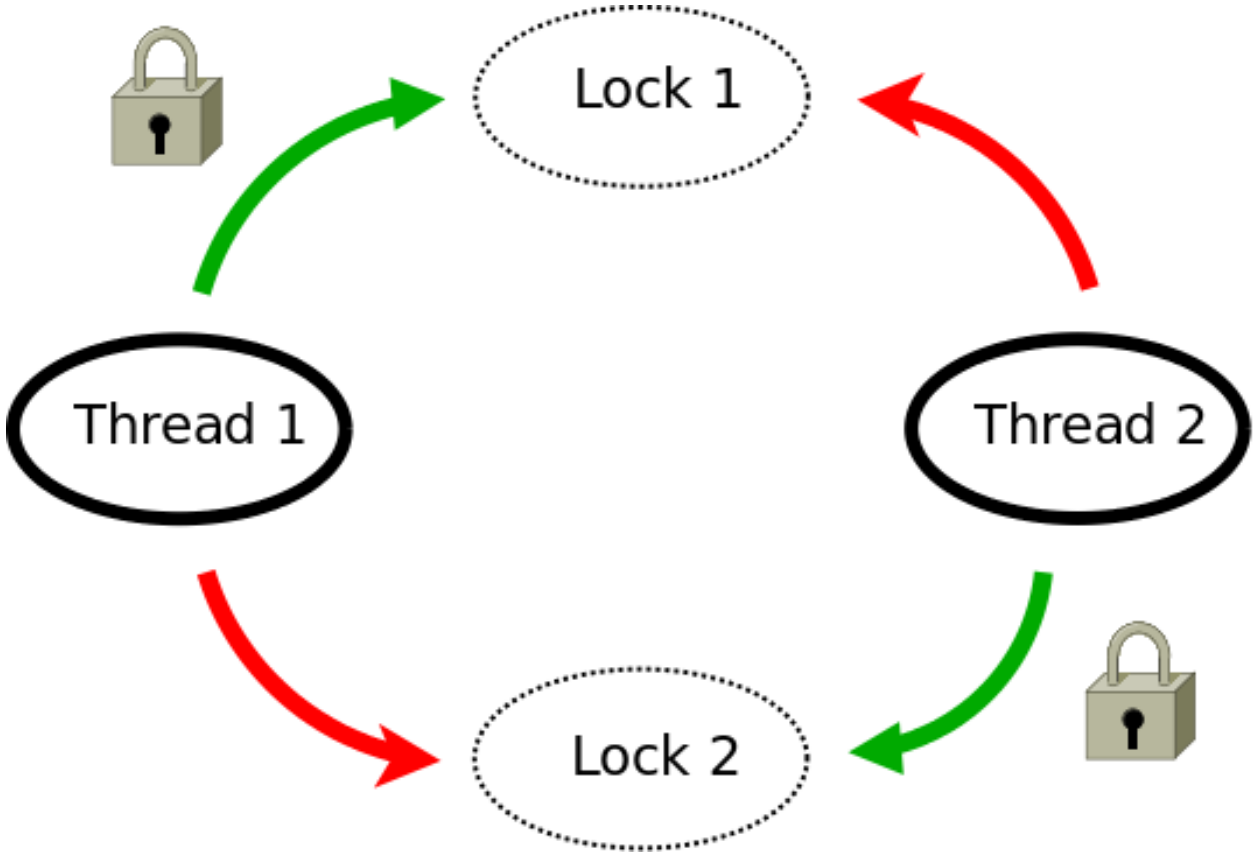
```
mutex m;
{
    m.lock();
    shrVar = getVar();
    m.unlock();
}
```

- lock_guard

```
mutex m;
{   // still bad
    lock_guard<mutex> myLock(m);
    shaVar = getVar();
}
{   // good
    auto temp = getVar();
    lock_guard<mutex> myLock(m);
    shaVar = temp;
}
```

Mutexe in Locks verpacken

Locken der Mutexe in verschiedener Reihenfolge



Mutexe in Locks verpacken

Atomares Locken der Mutexe

- unique_lock

```
{
    unique_lock<mutex> guard1(mut1, defer_lock);
    unique_lock<mutex> guard1(mut2, defer_lock);
    lock(guard1, guard2);
}
```

- lock_guard (C++17)

```
{
    std::scoped_lock(mut1, mut2);
}
```

Best Practices

Allgemein

Multithreading

Speichermodell

Programmiere nicht lock-frei

Herb Sutter

Lock-Free Programming

Herb Sutter

Guide to Threaded Coding

1. Forget what you learned in Kindergarten
   (ie stop Sharing)
2. Use Locks
3. Measure
4. Measure
5. Change your Algorithm
6. GOTO 1

Tony Van Eerd

∞. **Lock-free**

Lock-free coding is the last thing you want to do.

- Writing lock-free programs is hard
- Writing correct lock-free programs is even harder

Fedor Pikus

Safety: off
How not to shoot yourself in the foot with C++ atomics

Anthony Williams

The ugly side of weakly ordered atomics

Extreme complexity.

- The rules are not obvious.
- They're often downright surprising.
- And not even well understood.
- The committee still hasn't figured out how to define `memory_order_relaxed`.

… and I'm not even going to talk about `memory_order_consume`.

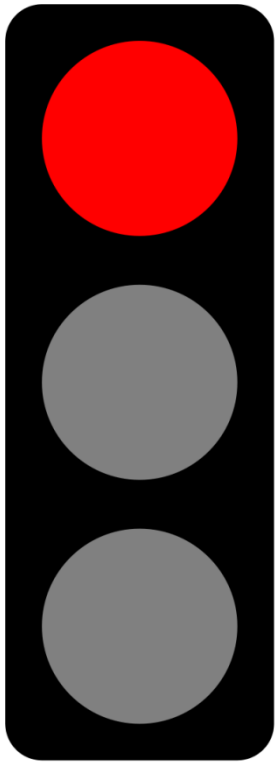
Hans Böhm

The specification of release-consume ordering is being revised, and the use of `memory_order_consume` is temporarily discouraged. (since C++17)

en.cppreference.com

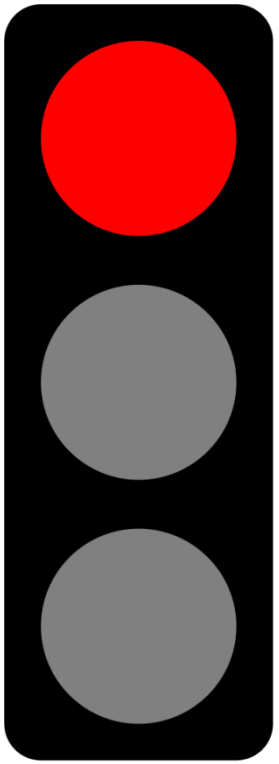
Programmiere nicht lock-frei: ABA

A



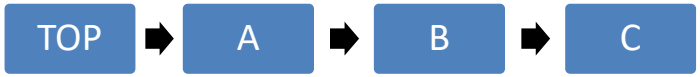
B

A



Programmiere nicht lock-frei: ABA

Eine lock-freie, einfach verkettete Liste (Stack)



- Thread 1
  - möchte `A` entfernen
  - speichert
    - head = A
    - next = B

- Thread 2
  - entfernt `A`

  - entfernt `B` und löscht `B`

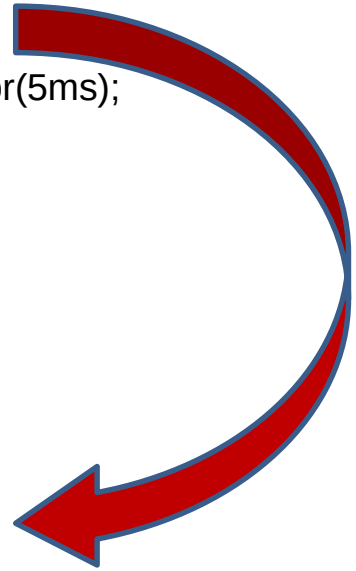
  - schiebt `A` auf Liste zurück

- prüft, ob `A == head`
- macht `B` zum neuen `head`
- `B` wurde bereits durch Thread 2 gelöscht

Verwende bewährte Muster

Warten mit Sequenzieller Konsistenz

```
std::vector<int> mySharedWork;
std::atomic<bool> dataReady(false);

void waitingForWork(){
    while ( !dataReady.load() ){
        std::this_thread::sleep_for(5ms);
    }
    mySharedWork[1] = 2;
}

void setDataReady(){
    mySharedWork = {1, 0, 3};
    dataReady.store(true);
}
```

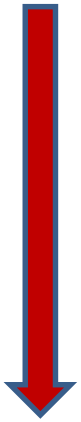


```
int main(){

    std::thread t1(waitingForWork);
    std::thread t2(setDataReady);
    t1.join();
    t2.join();
    for (auto v: mySharedWork){
        std::cout << v << " ";              // 1 2 3

};
```

Verwende bewährte Muster

Warten mit Acquire-Release Semantik

```
std::vector<int> mySharedWork;
std::atomic<bool> dataReady(false);

void waitingForWork(){
    while ( !dataReady.load(std::memory_order_acquire) ){
        std::this_thread::sleep_for(5ms);
    }
    mySharedWork[1] = 2;
}

void setDataReady(){
    mySharedWork = {1, 0, 3};
    dataReady.store(true, std::memory_order_release);
}
```



```
int main(){

    std::thread t1(waitingForWork);
    std::thread t2(setDataReady);
    t1.join();
    t2.join();
    for (auto v: mySharedWork){
        std::cout << v << " ";        // 1 2 3

};
```

Verwende bewährte Muster

Atomare Zähler

```
#include <vector>
#include <iostream>
#include <thread>
#include<atomic>

std::atomic<int> count{0};
void add(){
  for (int n = 0; n < 1000; ++n){
    count.fetch_add(1, std::memory_order_relaxed);
  }
}
```

```
int main(){
  std::vector<std::thread> v;
  for (int n = 0; n < 10; ++n){
    v.emplace_back(add);
  }
  for (auto& t : v) { t.join(); }
  std::cout << count;      // 10000
}
```

Erfinde das Rad nicht neu

Boost.Lockfree
CDS (Concurrent Data Structures)

Erfinde das Rad nicht neu

- Boost.Lockfree
  - Queue
    - eine lock-freie, mehrere Erzeuger-Verbraucher (producer-consumer) Queue
  - Stack
    - eine lock-freie, mehrere Erzeuger-Verbraucher (producer-consumer) Stack
  - spsc_queue
    - eine wait-freie, ein Erzeuger-Verbraucher (producer-consumer) Queue (Ringpuffer)

Erfinde das Rad nicht neu

- Concurrent Data Structures (CDS)
  - Enthält viele intrusive und nicht-intrusive Container
  - Stacks (lock-frei)
  - Queues und Priority-Queues (lock-frei)
  - Ordered lists
  - Ordered sets und maps (lock-frei und mit Lock)
  - Unordered sets und maps (lock-frei und mit Lock)

Blogs

[www.grimm-jaud.de [De]](http://www.grimm-jaud.de)

[www.ModernesCpp.com [En]](http://www.ModernesCpp.com)

Rainer Grimm

Training, Coaching und Technologieberatung

www.ModernesCpp.de