

# Best Practices

Rainer Grimm

Training, Coaching und  
Technologieberatung

[www.ModernesCpp.de](http://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;

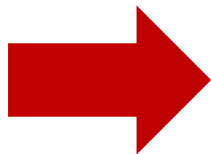
void addToTeleBook(const string& na, int tele){
    lock_guard<shared_timed_mutex> writerLock(teleBookMutex);
    cout << "\nSTARTING UPDATE " << na;
    this_thread::sleep_for(chrono::milliseconds(500));
    teleBook[na]= tele;
    cout << " ... ENDING UPDATE " << na << endl;
}

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;
}
```



```
rainer : bash - Konsole <5>
File Edit View Bookmarks Settings Help
rainer@linux:~> readerWriterLock

Scott: 1976Dijkstra: 1972
STARTING UPDATE Scott ... ENDING UPDATE Scott
Ritchie: 1983Scott: 1968
STARTING UPDATE Bjarne ... ENDING UPDATE Bjarne
Ritchie: 1983Scott: 1968Scott: 1968Bjarne: 1965

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

rainer@linux:~> █
```

# Code Reviews

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

shared_timed_mutex teleBookMutex;

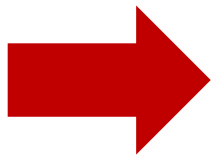
void addToTeleBook(const string& na, int tele){
    lock_guard<shared_timed_mutex> writerLock(teleBookMutex);
    cout << "\nSTARTING UPDATE " << na;
    this_thread::sleep_for(chrono::milliseconds(500));
    teleBook[na]= tele;
    cout << " ... ENDING UPDATE " << na << endl;
}

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:~$
```

# Code Reviews

```
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;

}
```

# Minimiere Teilen

- Summation eines Vektors mit 100 Millionen Elementen

```
constexpr long long size = 1000000000;

...

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

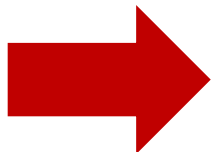
std::random_device seed;
std::mt19937 engine(seed());
std::uniform_int_distribution<> uniformDist(1, 10);
for (long long i = 0 ; i < size ; ++i)
    randValues.push_back(uniformDist(engine));

...
// 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
```



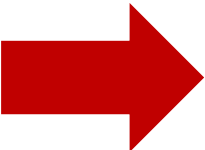
# Minimiere Teilen

- 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);
```



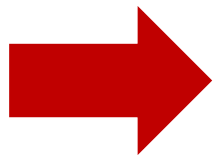
```
File Edit View Bookmarks Settings Help
rainer@suse:~> calculateWithLock
Time for addition 3.3389 seconds
Result: 549961505
rainer@suse:~> □
rainer: bash
```

# 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);
    }
}
```



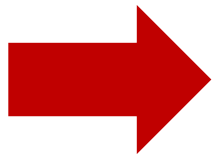
```
File Edit View Bookmarks Settings Help
rainer@suse:~> calculateWithAtomic
sum.is_lock_free(): true
Time for addition 1.33837 seconds
Result: 549992025
Time for addition 1.34625 seconds
Result: 549992025
rainer@suse:~> █
```

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

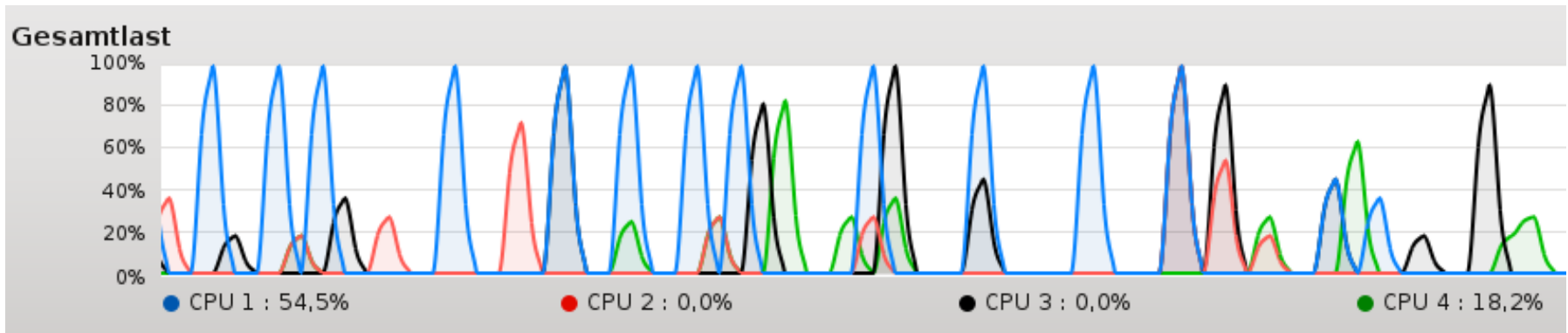
- Die Ergebnisse:

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?

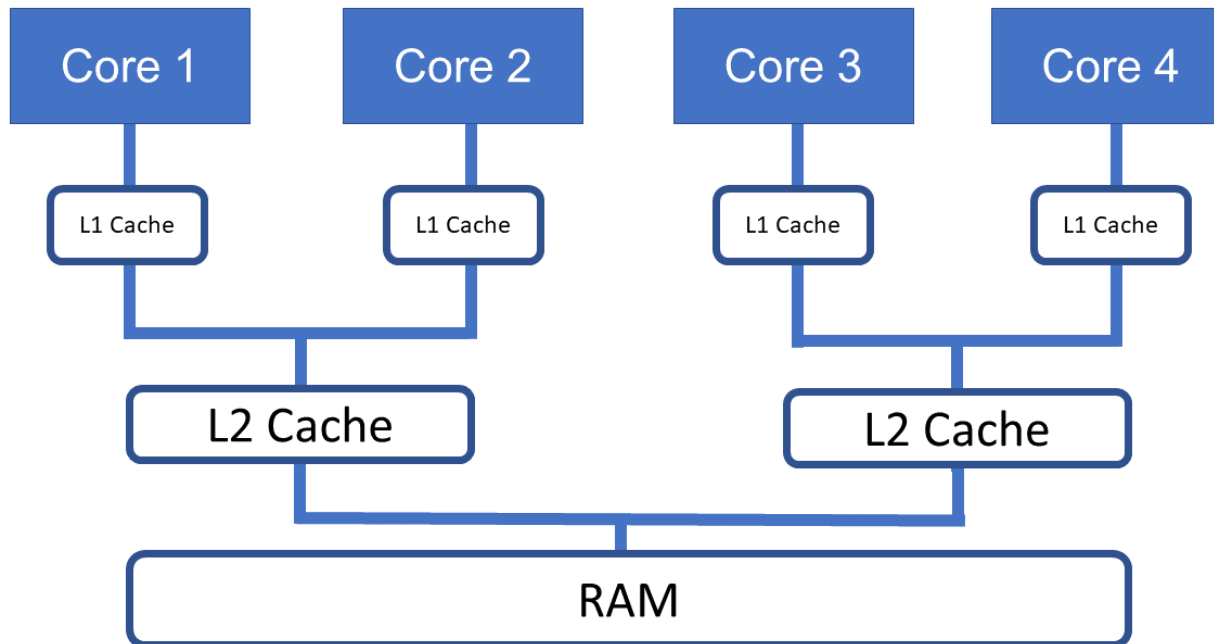
# Minimiere Teilen

- Die CPU-Auslastung



# Minimiere Teilen

- Die Memory-Wall



Der Speicherzugriff bremst die Applikation aus.

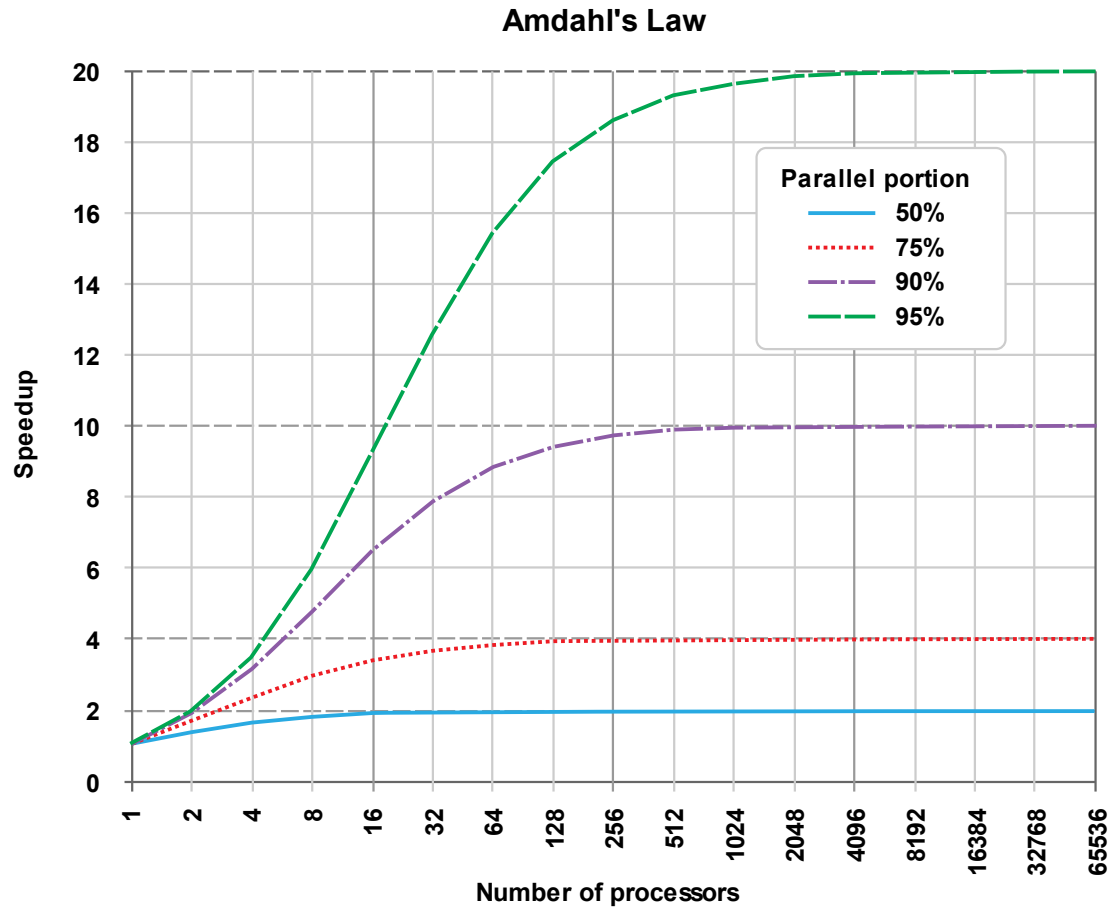
Minimiere Warten (Amdahl's Law)

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

*p: Parallele Code*

$$p = 0.5 \rightarrow 2$$

# Minimiere Warten (Amdahl's Law)





# Verwende Codeanalyse Werkzeuge (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 Werkzeuge (ThreadSanitizer)

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

```
File Edit View Bookmarks Settings Help
rainer@suse:~> dataRace
=====
WARNING: ThreadSanitizer: data race (pid=6764)
  Read of size 4 at 0x7fff031ca3bc by thread T2:
    #0 operator() /home/rainer/dataRace.cpp:10 (dataRace+0x000000400f01)
    #1 _M_invoke<> /usr/local/include/c++/6.3.0/functional:1391 (dataRace+0x000000401b3d)
    #2 operator() /usr/local/include/c++/6.3.0/functional:1380 (dataRace+0x000000401a51)
    #3 _M_run /usr/local/include/c++/6.3.0/thread:196 (dataRace+0x0000004019bc)
    #4 execute_native_thread_routine .././.././libstdc++.so.6+0x00000000c11be)

  Previous write of size 4 at 0x7fff031ca3bc by thread T1:
    #0 operator() /home/rainer/dataRace.cpp:9 (dataRace+0x000000400eb9)
    #1 _M_invoke<> /usr/local/include/c++/6.3.0/functional:1391 (dataRace+0x000000401be7)
    #2 operator() /usr/local/include/c++/6.3.0/functional:1380 (dataRace+0x000000401a8b)
    #3 _M_run /usr/local/include/c++/6.3.0/thread:196 (dataRace+0x000000401a06)
    #4 execute_native_thread_routine .././.././libstdc++.so.6+0x00000000c11be)

  Location is stack of main thread.

  Thread T2 (tid=6767, running) created by main thread at:
    #0 pthread_create .././.././libsanitizer/tsan/tsan_interceptors.cc:876 (libtsan.so.0+0x00000002aaed)
    #1 __gthread_create /home/rainer/languages/C++/gcc-6.3.0/x86_64-pc-linux-gnu/libstdc++.v3/include/x86_64-pc-linux-gnu/bits/gthr-default.h:662 (libstdc++.so.6+0x00000000c14b4)
    #2 std::thread::_M_start_thread(std::unique_ptr<std::thread::_State, std::default_delete<std::thread::_State> >, void (*)()) .././.././libstdc++.v3/src/c++11/thread.cc:163 (libstdc++.so.6+0x00000000c14b4)
    #3 main /home/rainer/dataRace.cpp:10 (dataRace+0x000000400f97)

  Thread T1 (tid=6766, finished) created by main thread at:
    #0 pthread_create .././.././libsanitizer/tsan/tsan_interceptors.cc:876 (libtsan.so.0+0x00000002aaed)
    #1 __gthread_create /home/rainer/languages/C++/gcc-6.3.0/x86_64-pc-linux-gnu/libstdc++.v3/include/x86_64-pc-linux-gnu/bits/gthr-default.h:662 (libstdc++.so.6+0x00000000c14b4)
    #2 std::thread::_M_start_thread(std::unique_ptr<std::thread::_State, std::default_delete<std::thread::_State> >, void (*)()) .././.././libstdc++.v3/src/c++11/thread.cc:163 (libstdc++.so.6+0x00000000c14b4)
    #3 main /home/rainer/dataRace.cpp:9 (dataRace+0x000000400f70)

SUMMARY: ThreadSanitizer: data race /home/rainer/dataRace.cpp:10 in operator()
=====
ThreadSanitizer: reported 1 warnings
rainer@suse:~> █
```

# Verwende Codeanalyse Werkzeuge (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 Werkzeuge (ThreadSanitizer)

```
File Edit View Bookmarks Settings Help
rainer@linux:~$ ./conditionVariablePingPong.cpp
Begin: false
true
=====
WARNING: ThreadSanitizer: data race (pid=181234)
Read of size 4 at 0x0000000604350 by thread T2:
#0 setFalse() /home/rainer/conditionVariablePingPong.cpp:30 (conditionVariablePingPong+0x000000401818)
#1 void std::_Bind_simple<void (*)()>::operator()() /usr/include/c++/6/functional:1400
#2 std::_Bind_simple<void (*)()>::operator()() /usr/include/c++/6/functional:1389 (conditionVariablePingPong+0x000000401818)
#3 std::thread::_State_impl<std::_Bind_simple<void (*)()> >::_M_run() /usr/include/c++/6/thread:196 (conditionVariablePingPong+0x000000401818)
#4 <null> <null> (libstdc++.so.6+0x0000000c22de)

Previous write of size 4 at 0x0000000604350 by thread T1 (mutexes: write M11):
#0 setTrue() /home/rainer/conditionVariablePingPong.cpp:21 (conditionVariablePingPong+0x00000040173d)
#1 void std::_Bind_simple<void (*)()>::operator()() /usr/include/c++/6/functional:1400
#2 std::_Bind_simple<void (*)()>::operator()() /usr/include/c++/6/functional:1389 (conditionVariablePingPong+0x00000040173d)
#3 std::thread::_State_impl<std::_Bind_simple<void (*)()> >::_M_run() /usr/include/c++/6/thread:196 (conditionVariablePingPong+0x00000040173d)
#4 <null> <null> (libstdc++.so.6+0x0000000c22de)

Location is global 'counter' of size 4 at 0x0000000604350 (conditionVariablePingPong+0x0000000604350)

Mutex M11 (0x00000006042a0) created at:
#0 pthread_mutex_lock <null> (libtsan.so.0+0x000000003bc0f)
#1 __gthread_mutex_lock /usr/include/c++/6/x86_64-suse-linux/bits/gthr-default.h:748 (conditionVariablePingPong+0x00000006042a0)
#2 std::mutex::lock() /usr/include/c++/6/bits/std_mutex.h:103 (conditionVariablePingPong+0x00000006042a0)
#3 std::unique_lock<std::mutex>::lock() /usr/include/c++/6/bits/std_mutex.h:267 (conditionVariablePingPong+0x00000006042a0)
#4 std::unique_lock<std::mutex>::unique_lock(std::mutex&) /usr/include/c++/6/bits/std_mutex.h:197 (conditionVariablePingPong+0x00000006042a0)
#5 setTrue() /home/rainer/conditionVariablePingPong.cpp:18 (conditionVariablePingPong+0x000000060416f4)
#6 void std::_Bind_simple<void (*)()>::operator()() /usr/include/c++/6/functional:1400
#7 std::_Bind_simple<void (*)()>::operator()() /usr/include/c++/6/functional:1389 (conditionVariablePingPong+0x000000060416f4)
#8 std::thread::_State_impl<std::_Bind_simple<void (*)()> >::_M_run() /usr/include/c++/6/thread:196 (conditionVariablePingPong+0x000000060416f4)
#9 <null> <null> (libstdc++.so.6+0x0000000c22de)

Thread T2 (tid=18140, running) created by main thread at:
#0 pthread_create <null> (libtsan.so.0+0x000000002b740)
#1 std::thread::_M_start_thread(std::unique_ptr<std::thread::_State, std::default_delete<std::thread::_State> >, ...) /usr/include/c++/6/thread:196 (conditionVariablePingPong+0x0000000604197c)
#2 main /home/rainer/conditionVariablePingPong.cpp:49 (conditionVariablePingPong+0x0000000604197c)

Thread T1 (tid=18139, running) created by main thread at:
#0 pthread_create <null> (libtsan.so.0+0x000000002b740)
#1 std::thread::_M_start_thread(std::unique_ptr<std::thread::_State, std::default_delete<std::thread::_State> >, ...) /usr/include/c++/6/thread:196 (conditionVariablePingPong+0x0000000604196b)
#2 main /home/rainer/conditionVariablePingPong.cpp:48 (conditionVariablePingPong+0x0000000604196b)

SUMMARY: ThreadSanitizer: data race /home/rainer/conditionVariablePingPong.cpp:30 in setFalse()
=====
false
true
false
true
false
```

# Verwende Codeanalyse Werkzeuge (CppMem)

## CppMem: Interactive C/C++ memory model

Model  
☐ standard ☒ preferred ☐ release\_acquire ☐ tot ☐ relaxed\_only  
Program

examples/Paper data\_race.c

☒ C ☐ Execution

```
// a data race (dr)
int main() {
    int x = 2;
    int y;
    {{{ x = 3;
      ||| y = (x==3);
      }}};
    return 0; }
```

2 reset help 2 executions; 1 consistent, not race free

Computed executions

### Display Relations

☒ sb ☐ asw ☐ dd ☐ cd  
☒ rf ☒ mo ☒ sc ☒ lo  
☐ hb ☐ vse ☐ ithb ☒ sw ☐ rs ☐ hrs ☒ dob ☐ cad  
☒ unsequenced\_races ☒ data\_races

### Display Layout

☐ dot ☐ neato\_par ☒ neato\_par\_init ☐ neato\_downwards

☐ tex

edit display options

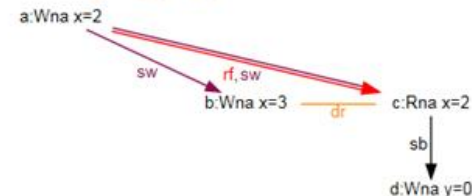
## Execution candidate no. 2 of 2

previous consistent previous candidate next candidate next consistent 2 goto

### Model Predicates

consistent\_race\_free\_execution = **false**  
☒ consistent\_execution = **true**  
☒ assumptions = **true**  
☒ well\_formed\_threads = **true**  
☒ well\_formed\_rf = **true**  
☒ locks\_only\_consistent\_locks = **true**  
☒ locks\_only\_consistent\_lo = **true**  
☒ consistent\_mo = **true**  
☒ sc\_accesses\_consistent\_sc = **true**  
☒ sc\_fenced\_sc\_fences\_heeded = **true**  
☒ consistent\_hb = **true**  
☒ consistent\_rf = **true**  
☒ det\_read = **true**  
☒ consistent\_non\_atomic\_rf = **true**  
☒ consistent\_atomic\_rf = **true**  
☒ coherent\_memory\_use = **true**  
☒ rmw\_atomicity = **true**  
☒ sc\_accesses\_sc\_reads\_restricted = **true**  
unsequenced\_races are **absent**  
data\_races are **present**  
indeterminate\_reads are **absent**  
locks\_only\_bad\_mutexes are **absent**

4



Files: out.exc, out.dot, out.dsp, out.tex

# Verwende Codeanalyse Werkzeuge (CppMem)

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

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

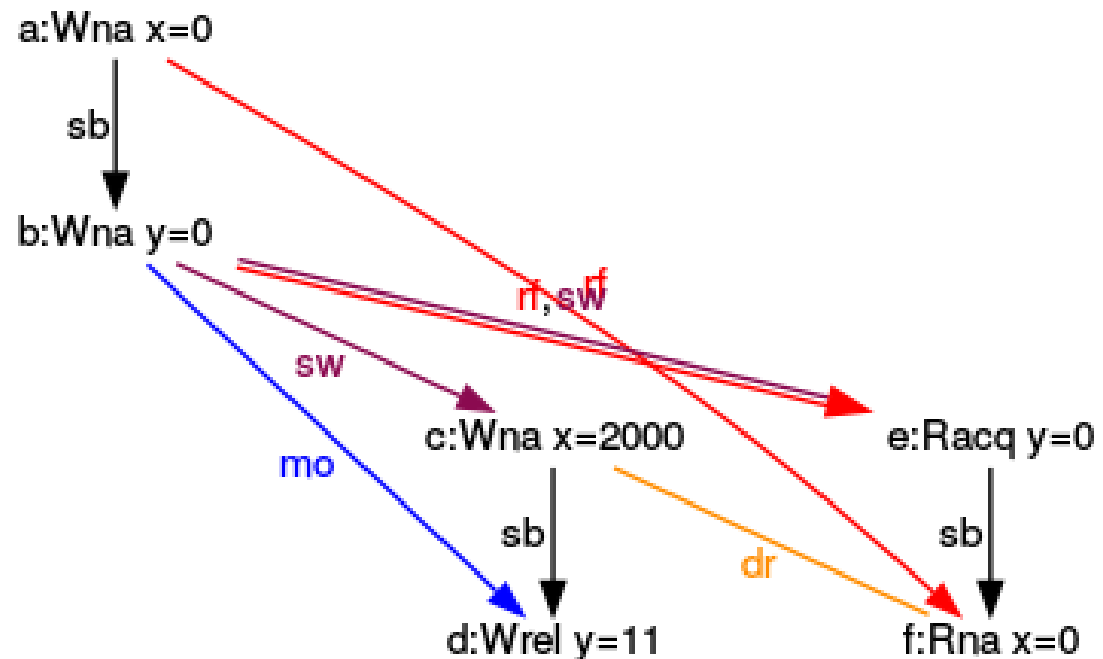
```
void reading(){  
    std::cout << y.load(std::memory_order_acquire) << " ";  
    std::cout << x << std::endl;  
}
```

```
...
```

```
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.

## Veränderlich?

**Geteilt?**

	nein	ja
nein	OK	Ok
ja	OK	<b>Data Race</b>



# Verwende unveränderliche Daten

**Das verbleibende Problem:** Die Daten müssen thread-sicher initialisiert werden.

## 1. Frühzeitiges Initialisieren

```
const int val = 2011;  
thread t1([&val]{ .... });  
thread t2([&val]{ .... });
```

## 2. Konstante Ausdrücke

```
constexpr auto doub = 5.1;
```

## 3. `call_once` und `once_flag`

```
void onlyOnceFunc{ .... };  
call_once(onceFlag, onlyOnceFunc);  
thread t3{ onlyOnceFunc() };  
thread t4{ onlyOnceFunc() };
```

## 4. Statische Variablen mit Blockgültigkeit

```
void func(){  
    .... static int val 2011; ....  
}  
thread t5{ func() };  
thread t6{ func() };
```

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

# Verwende reine Funktionen

- Funktionen

```
int powFunc(int m, int n){  
    if (n == 0) return 1;  
    return m * powFunc(m, n-1);  
}
```

- Metafunktionen

```
template<int m, int n>  
struct PowMeta{  
    static int const value = m * PowMeta<m, n-1>::value;  
};
```

```
template<int m>  
struct PowMeta<m, 0>{  
    static int const value = 1;  
};
```

# Verwende reine Funktionen

- `constexpr` Funktionen
  - sind fast reine Funktionen

```
constexpr int powConst(int m, int n){  
    int r = 1;  
    for(int k = 1; k <= n; ++k) r *= m;  
    return r;  
}
```

```
auto res = powConst(2, 10);
```

```
auto constexpr res2 = powConst(2, 10);
```

# 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

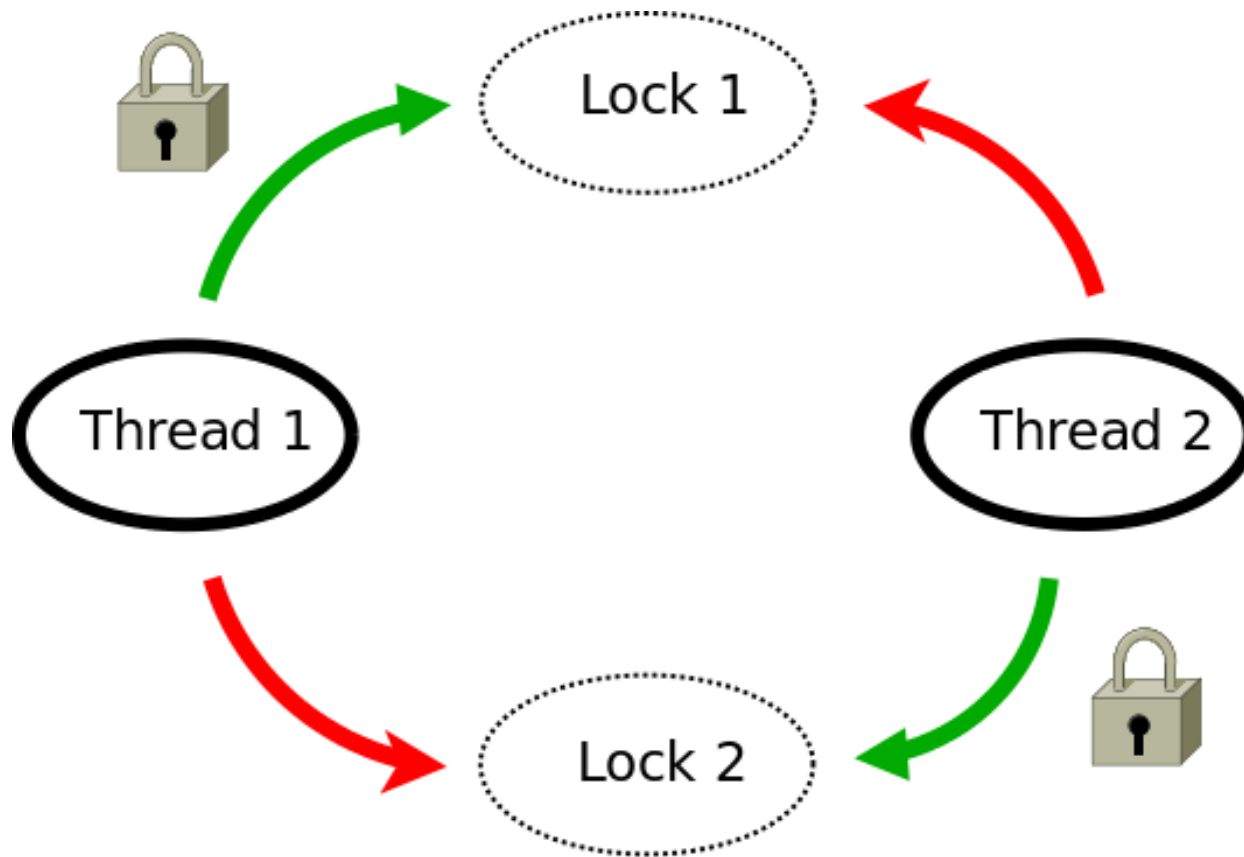
```
mutex m;  
  
{  
    m.lock();  
    shrVar = getVar();  
    m.unlock();  
}
```

- lock\_guard

```
mutex m;  
  
{  
    lock_guard<mutex> myLock(m);  
    shaVar = getVar();  
}  
  
{  
    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> guard2 (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.*

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

Anthony Williams



- Writing lock-free programs is hard **Fedor Pikus**
- Writing correct lock-free programs is even harder

## The ugly side of weakly ordered atomics

Extreme complexity.

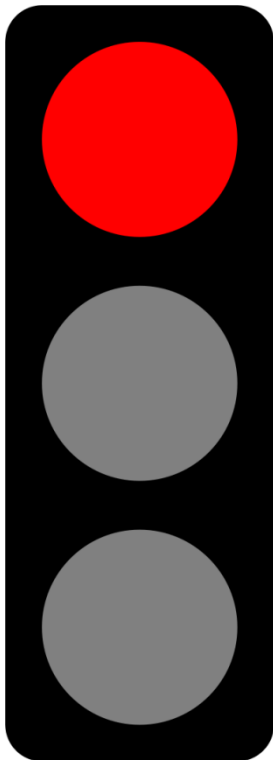
- The rules are not obvious.
- They're often downright surprising.
- And not even well understood.
- The committee still hasn't figured out how to define `memory_order_relaxed`.  
... and I'm not even going to talk about `memory_order_consume`.

Hans Böhm

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

# Programmiere nicht lock-frei: ABA

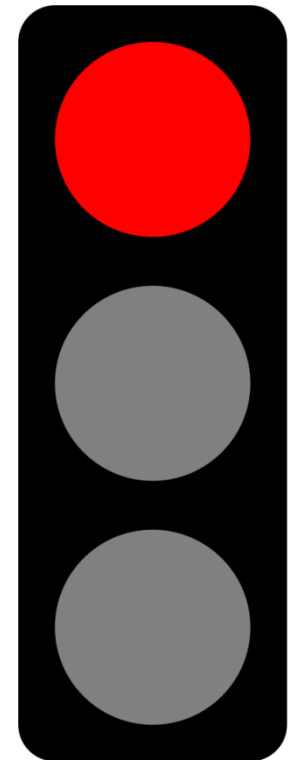
A



B



A



# Programmiere nicht lock-frei: ABA

```
template <typename T>
T fetch_mult(std::atomic<T>& shar, T mul){
  T old = shar.load();
  while (!shar.compare_exchange_strong(old, old * mul));
  return old;
}
```

```
int main(){
  std::atomic<int> myInt{5};
  std::cout << myInt << std::endl;
  fetch_mult(myInt, 5);
  std::cout << myInt << std::endl;
}
```



ABA ist eine große Herausforderung in lock-freien Datenstrukturen.

# 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



# Programmiere nicht lock-frei: ABA

- Problem der einfache verketteten Liste
  - B wird gelöscht, obwohl es noch verwendet wird
- Lösungen:
  - Tagged State:
    - Jeder Knoten bekommt einen Tag, der anzeigt, wie oft er verändert wurde
  - Verzögertes Löschen:
    - Garbage Collection
    - Hazard Pointers: Garbage Collection auf Thread-Ebene
    - RCU:
      - Modifikation werden nur auf Kopien vollzogen
      - Wenn es keinen Zugriff auf die Daten mehr gibt, werden diese ausgetauscht

# Verwende bewährte Muster

## Warten mit Sequenzieller Konsistenz

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

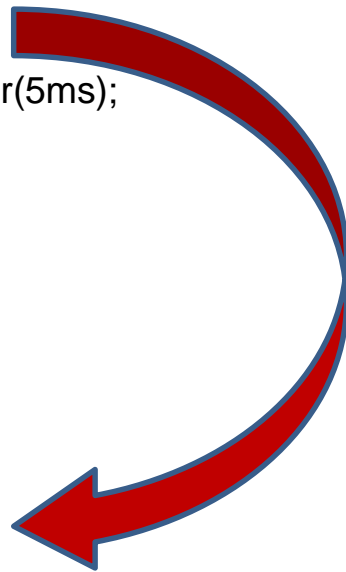
```
void waitingForWork(){  
    while ( !dataReady.load() ){  
        std::this_thread::sleep_for(5ms);  
    }  
    mySharedWork[1] = 2;  
}
```

```
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

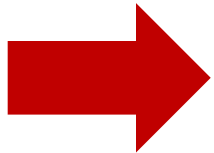
```
#include <vector>
#include <iostream>
#include <thread>
#include<atomic>

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

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

Rainer Grimm

Training, Coaching und  
Technologieberatung

[www.ModernesCpp.de](http://www.ModernesCpp.de)