Best Practices for Concurrency

Rainer Grimm

Training, Coaching, and Technology Consulting

www.ModernesCpp.de

Best Practices for Concurrency

General

Multithreading

~~Parallel~~

Memory Model

Best Practices for Concurrency

General

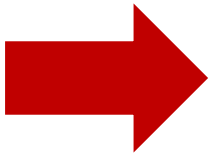
Multithreading

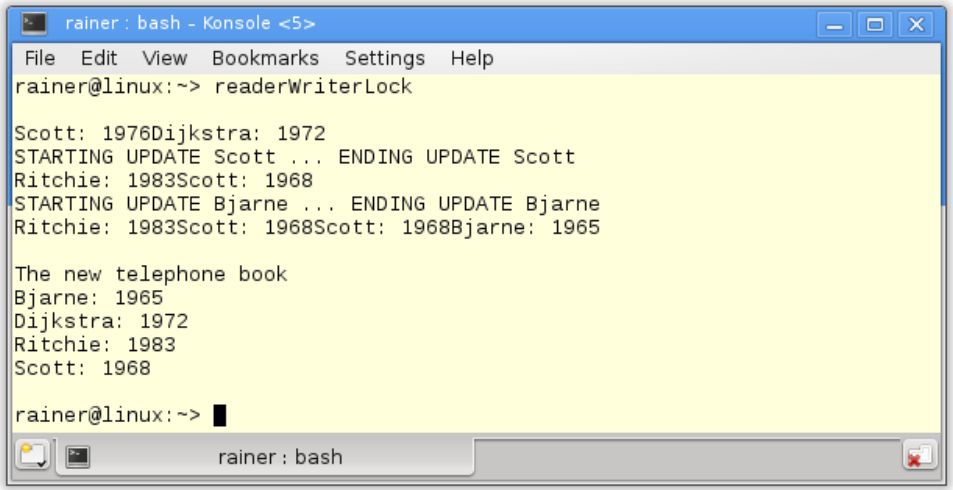
Memory Model

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;
}
```

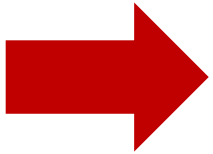




Code Reviews

```
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

Minimise Sharing

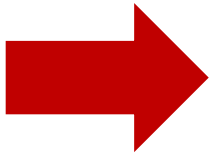
- Summation of a vector with 100 000 000 elements

```
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
...
```

Minimise Sharing

- Single-threaded in two variations

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

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

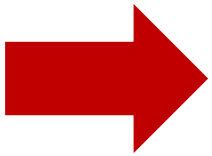


```
File  Edit  View  Bookmarks  Settings  Help
rainer@suse:~> calculateWithStd

Time for addition 0.0651712 seconds
Result: 550030112

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

Minimise Sharing

- Four threads with a shared summation variable

```
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);
```

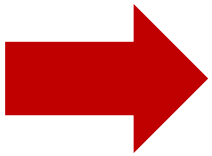


```
File  Edit  View  Bookmarks  Settings  Help
rainer@suse:~> calculateWithLock

Time for addition 3.3389 seconds
Result: 549961505

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

Minimise Sharing

- Four threads with a shared, atomic summation variable

```
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);
    }
}
```

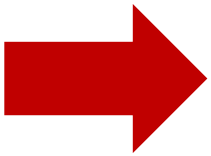


```
File  Edit  View  Bookmarks  Settings  Help
rainer@suse:~> calculateWithAtomic

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
```

Minimise Sharing

- Four threads with a local summation variable

```
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
```

Minimise Sharing

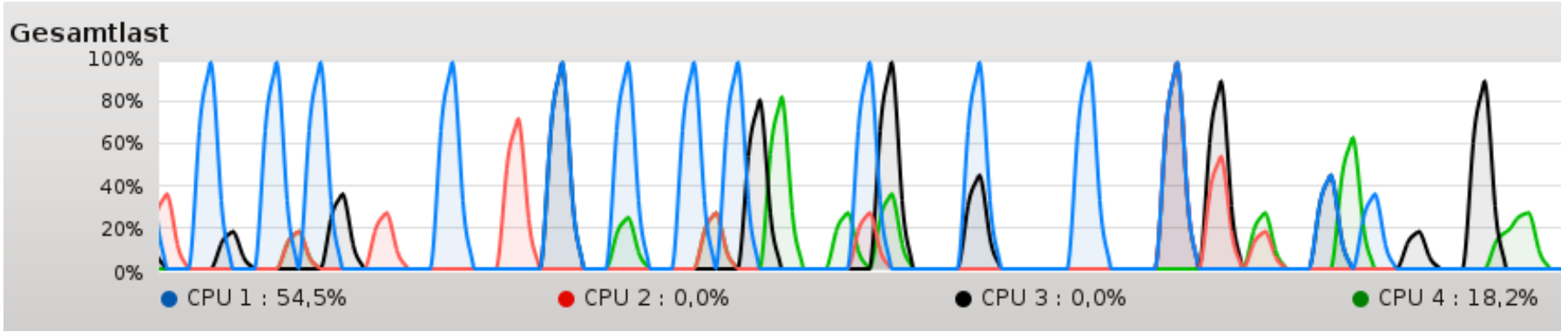
- Results

Single-threaded	4 threads with a lock	4 threads with an atomic	4 threads with a local variable
0.07 sec	3.34 sec	1.34 sec	0.03 sec

All Fine?

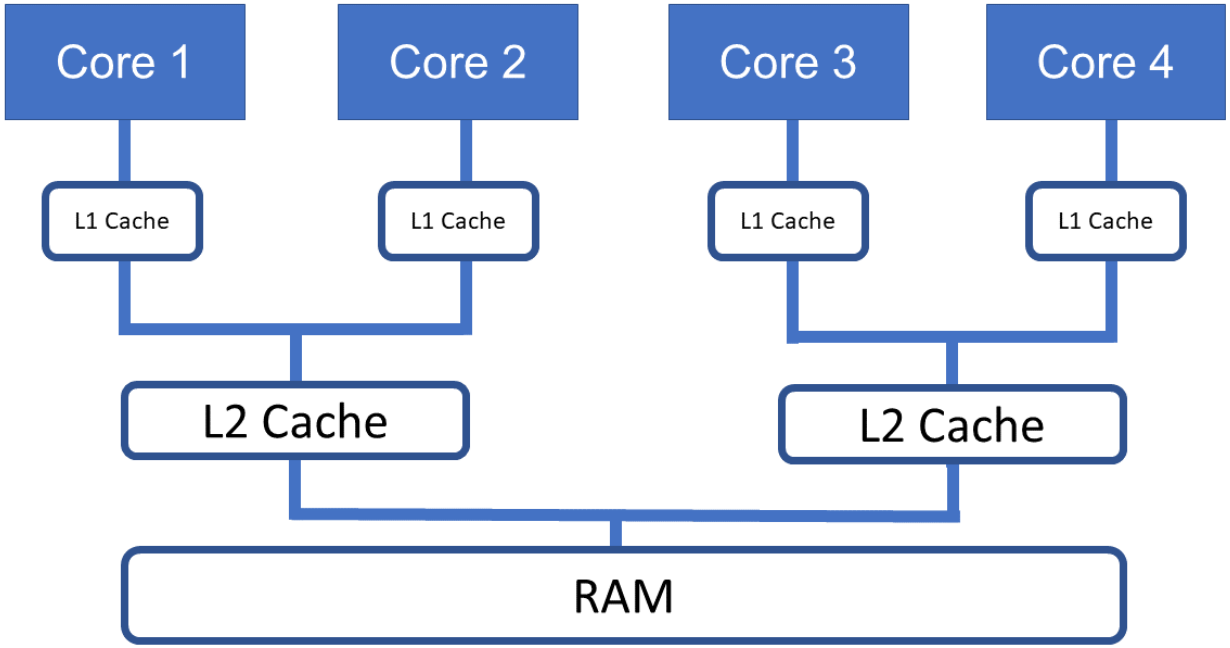
Minimise Sharing

- CPU Utilisation



Minimise Sharing

- Memory Wall

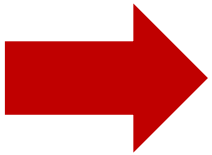


The RAM is the bottleneck.

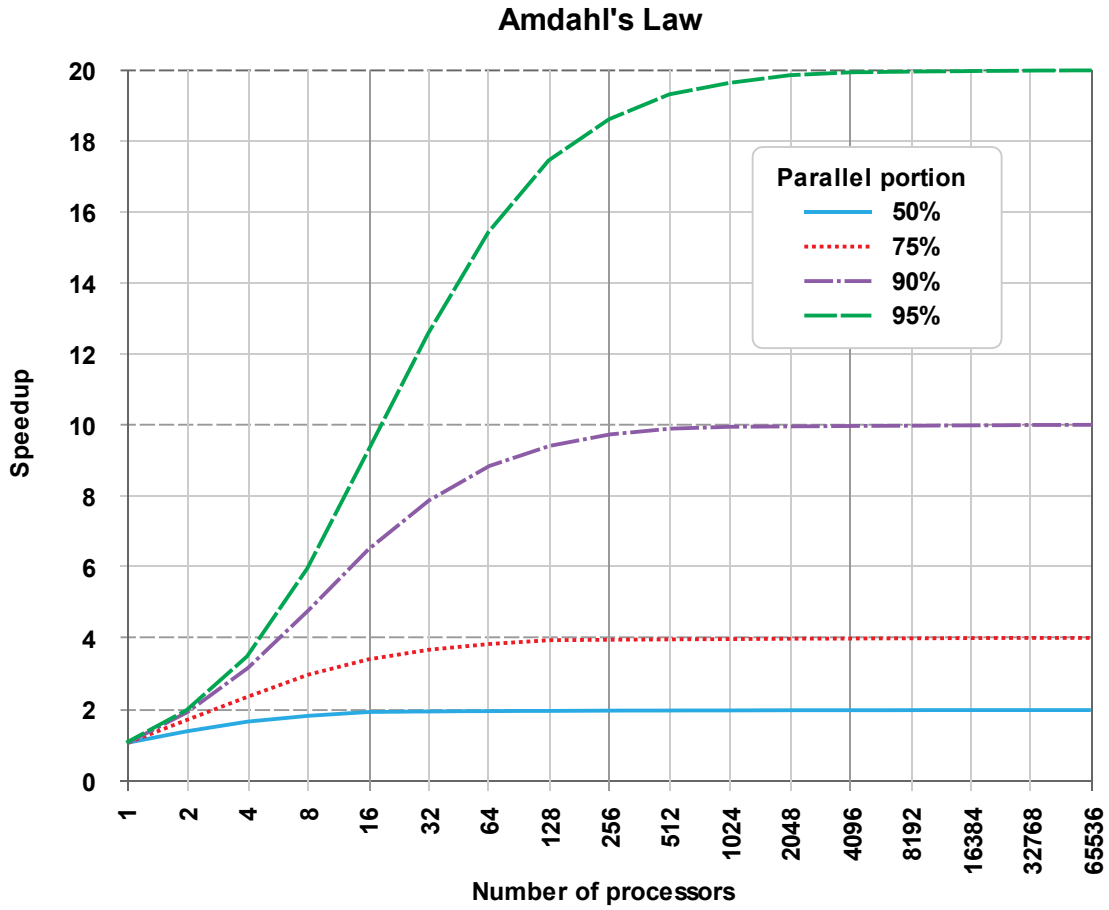
Minimise Waiting (Amdahl's Law)

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

p: $Parallele\ Code$

$p = 0.5$  2

Minimise Waiting (Amdahl‘s Law)



Use Code Analysis Tools (ThreadSanitizer)

Thread Sanitizer detects data races at runtime.

- Memory and performance overhead: 10x
- Available since GCC 4.8 and Clang 3.2

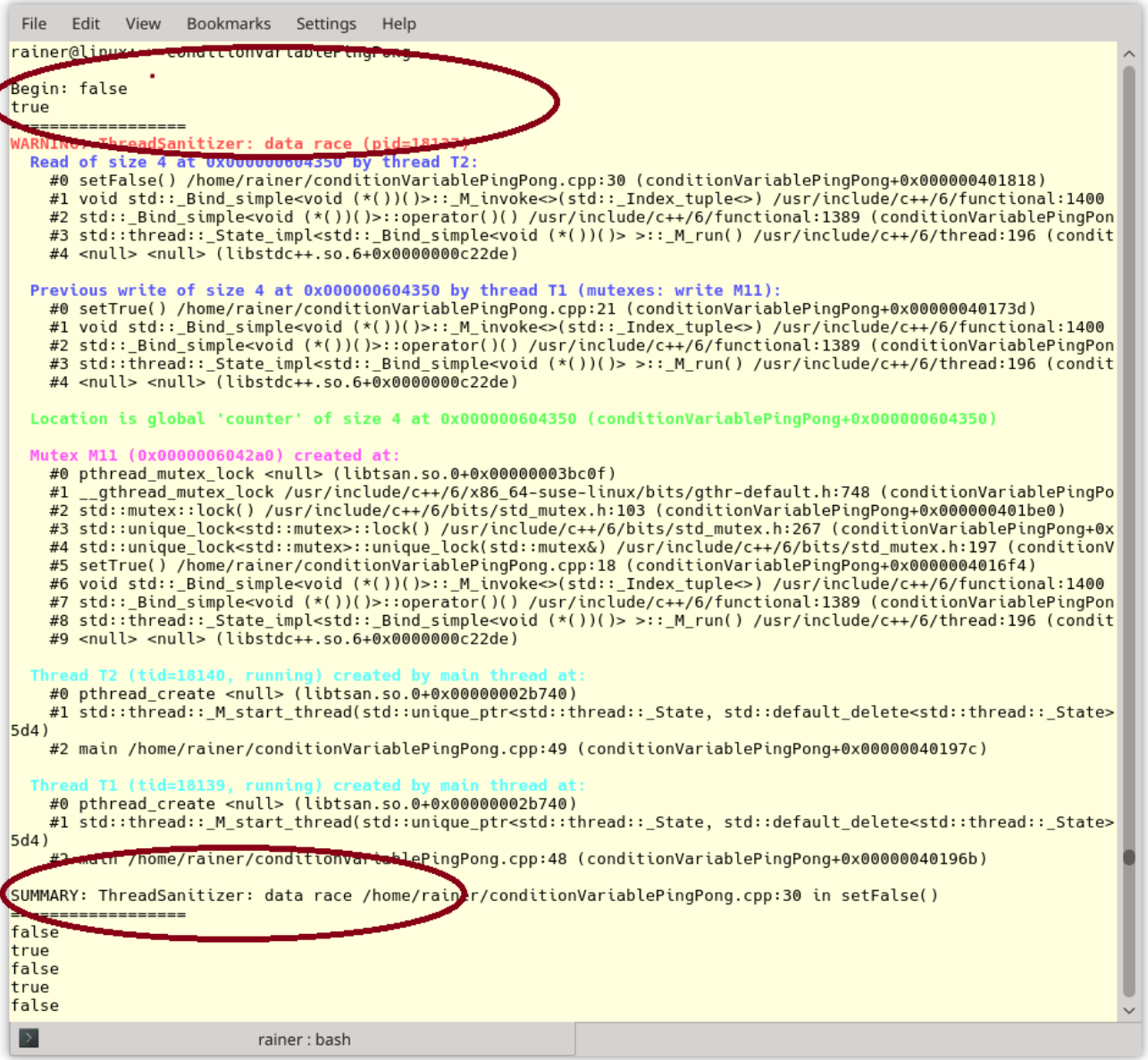
```
g++ conditionVariablePingPong.cpp -fsanitize=thread -g
    -o conditionVariablePingPong -pthread
```

Use Code Analysis Tools (ThreadSanitizer)

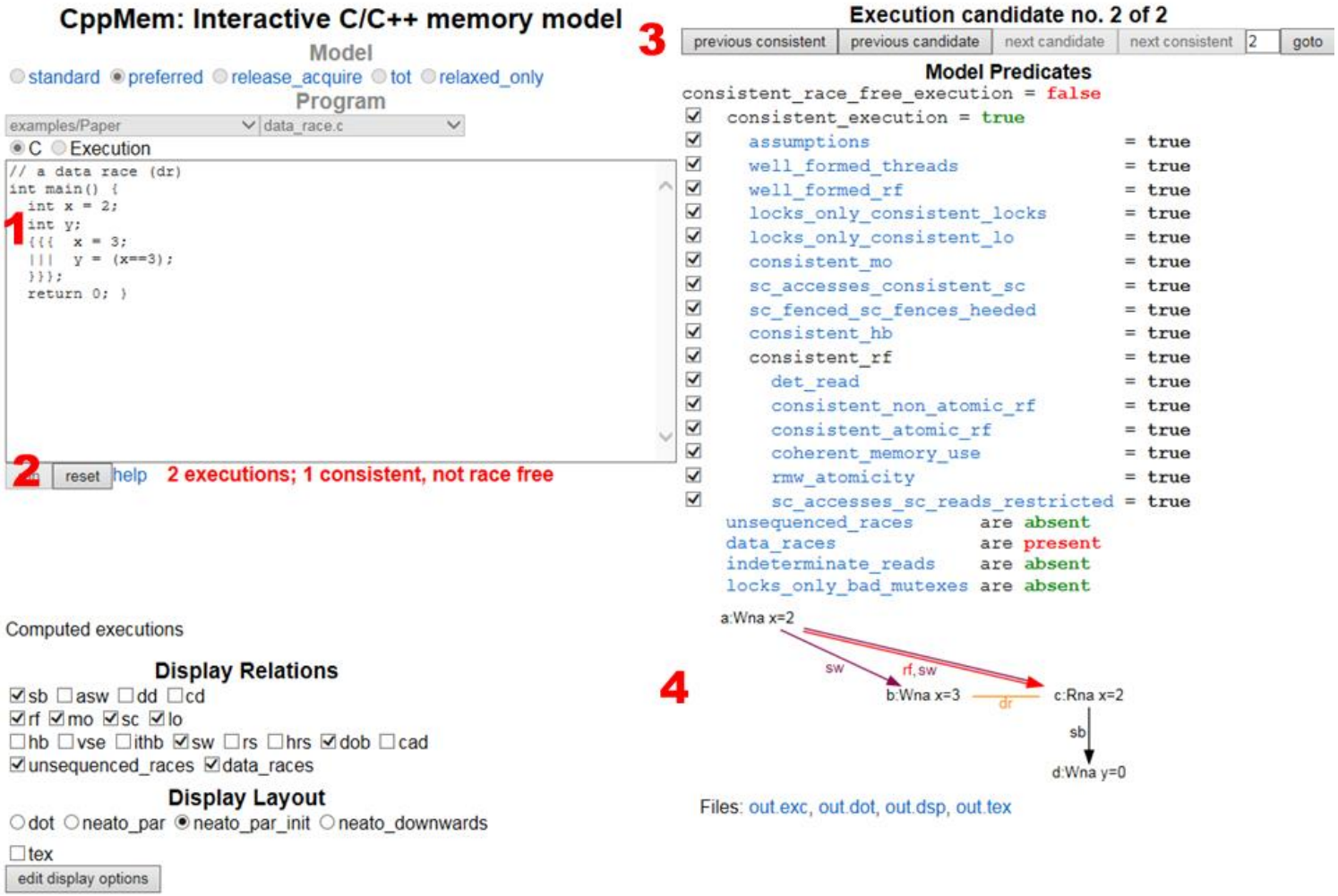
```
bool dataReady= false;

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;
}
```

Use Code Analysis Tools (ThreadSanitizer)



Use Code Analysis Tools (CppMem)

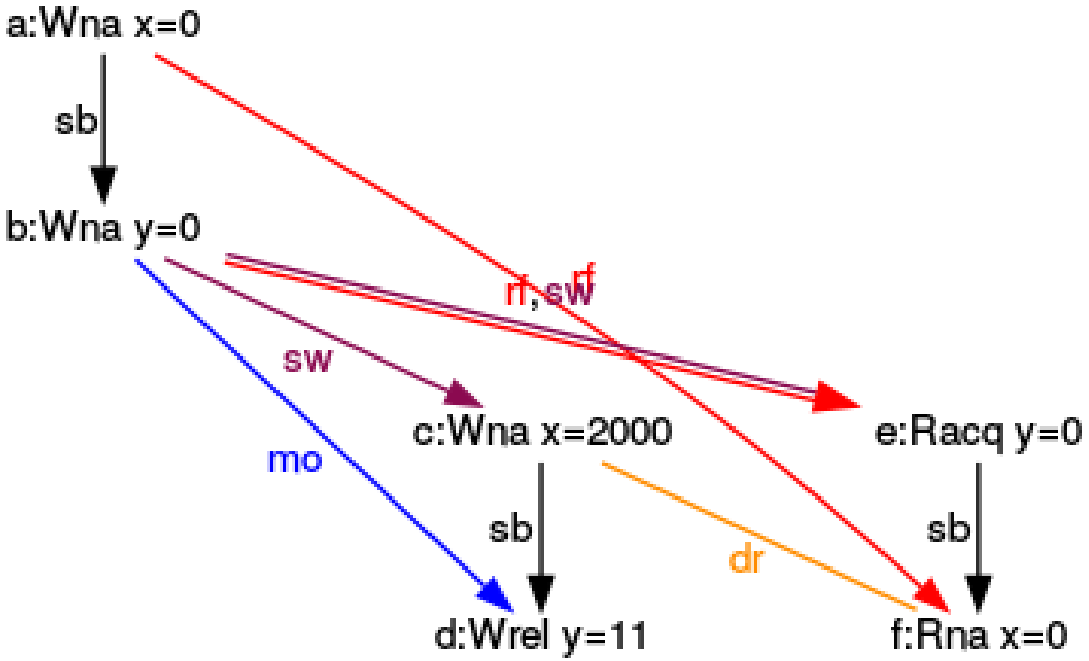


Use Code Analysis Tools (CppMem)

```
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();
```

Use Code Analysis Tools (CppMem)

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



Use Immutable Data

Data Race: At least two threads access a shared variable at the same time. At least on thread tries to modify it.

Mutable?

Shared?	No	Yes
No	OK	Ok
Yes	OK	**Data Race**

Use Immutable Data

Remaining Problem: initialise the data thread-safe

1. Early initialisation

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

2. Constant expressions

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

3. `call_once` and `once_flag`

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

4. Static variables in a scope

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

Use Pure Functions

Pure Functions
Produce always the same result when given the same arguments
Have no side effect
Don't change the global state of the program

Added Value

- Easier to make correctness proofs
- Refactoring and testing is easier
- Results from previous function calls can be memorised
- **The function invocations can automatically be reordered or parallelised.**

Use Pure Functions

- Function

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

- Metafunction

```
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;
};
```

Use Pure Functions

- `constexpr` Function
  - almost pure functions

```
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 res = PowMeta<2, 10>::value;
constexpr auto res = powConst(2, 10);
```

Best Practices for Concurrency

General

Multithreading

Memory Model

Use Tasks instead of Threads

Threads

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

Tasks

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

Criteria	Thread	Task
Parties Involved	creator thread and child thread	promise and future
Communication	shared variable	communication channel
Thread Creation	obligatory	optional
Synchronisation	`join` call blocks	`get` call blocks
Exception in Child	creator thread and child thread die	return value of the promise
Forms of Communication	values	values, notifications, and exceptions

Use Tasks instead of Threads

C++20/23: Extend futures will support composition

- **`then`**: Execute the future if the previous future is done
- **`when_any`**: Execute the future if any of the previous future is done
- **`when_all`**: Execute the future if all of the previous futures are done

Don‘t use Fire and Forget Futures

- The future created by `std::async` blocks until ist associated promise is done.

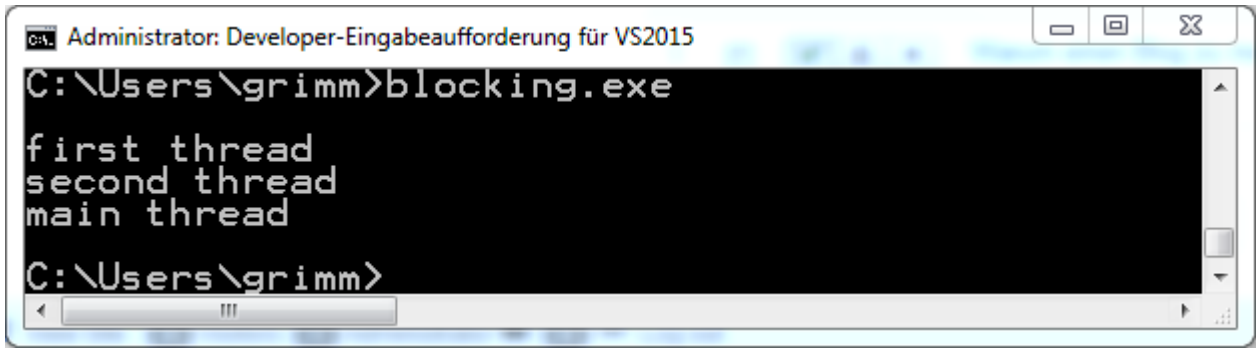
```
#include <iostream>
#include <future>
int main() {
    std::cout << std::endl;
    std::async([]{ std::cout << "fire and forget" << std::endl; });
    std::cout << "main done " << std::endl;
}
```

No `join` or `detach` call is required.

The asynchronous job is done synchronously.

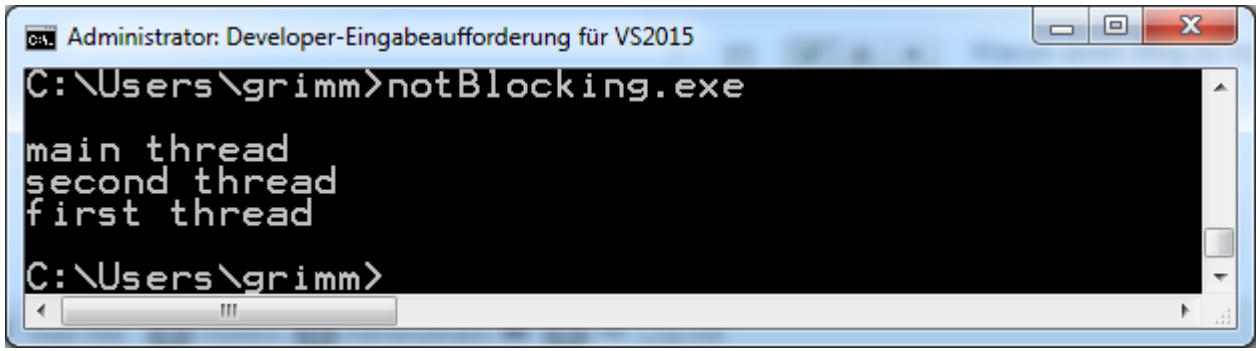
Don‘t use Fire and Forget Futures

```
int main(){
    std::async(std::launch::async,[]{
        std::this_thread::sleep_for(std::chrono::seconds(2));
        std::cout << "first thread" << std::endl;
    });
    std::async(std::launch::async,[]{
        std::this_thread::sleep_for(std::chrono::seconds(1));
        std::cout << "second thread" << std::endl;}
    );
    std::cout << "main thread" << std::endl;
}
```



Don‘t use Fire and Forget Futures

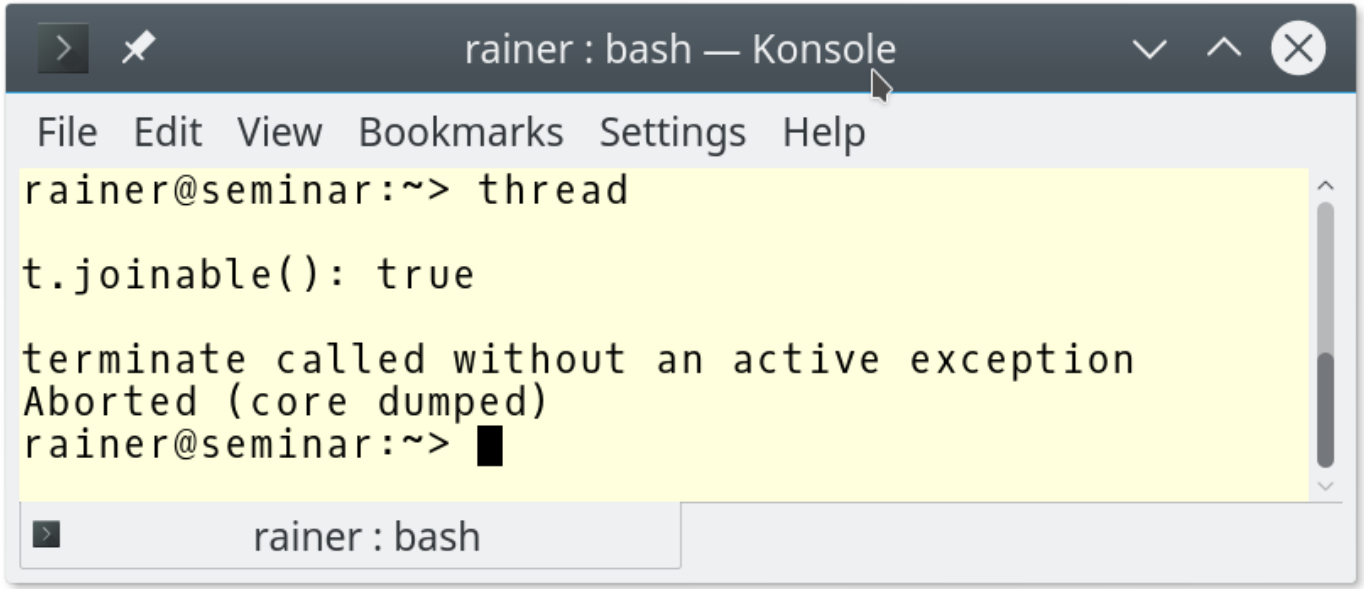
```
int main(){
    auto first = std::async(std::launch::async,[]{
        std::this_thread::sleep_for(std::chrono::seconds(2));
        std::cout << "first thread" << std::endl;
    });
    auto second = std::async(std::launch::async,[]{
        std::this_thread::sleep_for(std::chrono::seconds(1));
        std::cout << "second thread" << std::endl;}
    );
  std::cout << "main thread" << std::endl;
}
```



std::jthread

Problem: `std::thread` throws `std::terminate` in its destructor if still joinable.

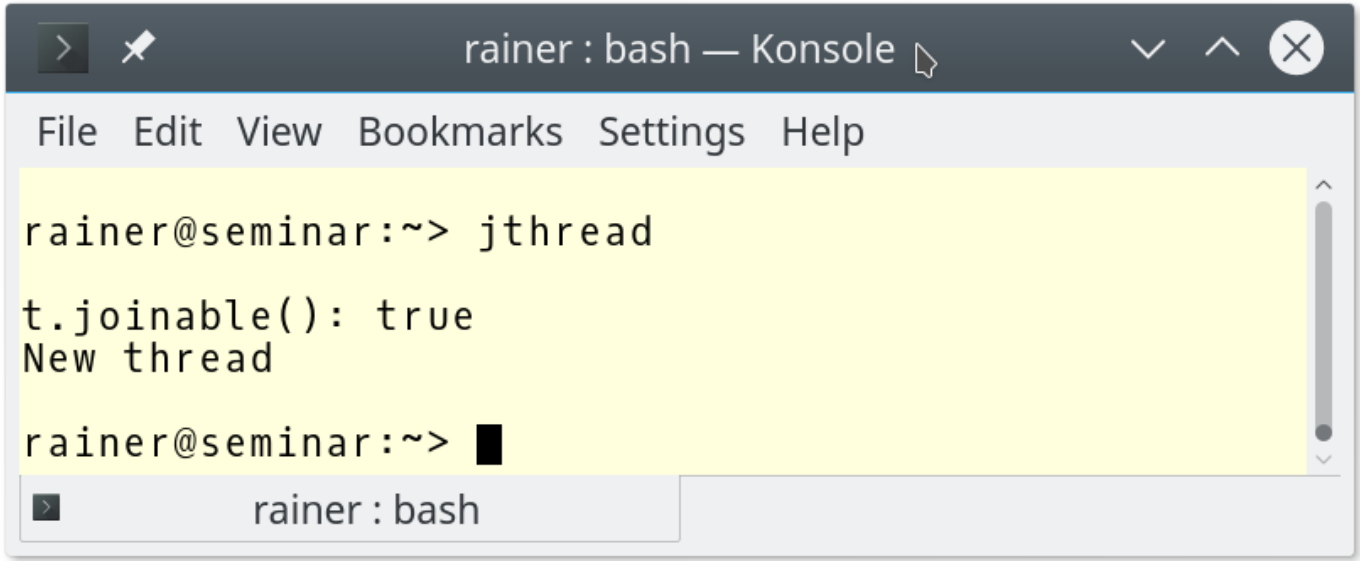
```
std::thread t{[]{ std::cout << "New thread"; }};
std::cout << "t.joinable(): " << t.joinable();
```



`std::jthread`

Solution: `std::jthread` (C++20) joins automatically at the end of its scope.

```
std::jthread t{[]{ std::cout << "New thread"; }};
std::cout << "t.joinable(): " << t.joinable();
```



Cooperative Interruption of `std::jthread`

- Instances of `std::jthread` can be interrupted

Receiver

- Explicit check:
  - **`is_interrupted`**: yields, when an interrupt was signalled
  - **`wait`** variations with predicate of `std::condition_variable`

Sender

- **interrupt:** signals an interrupt (and returns whether an interrupt was signaled before)

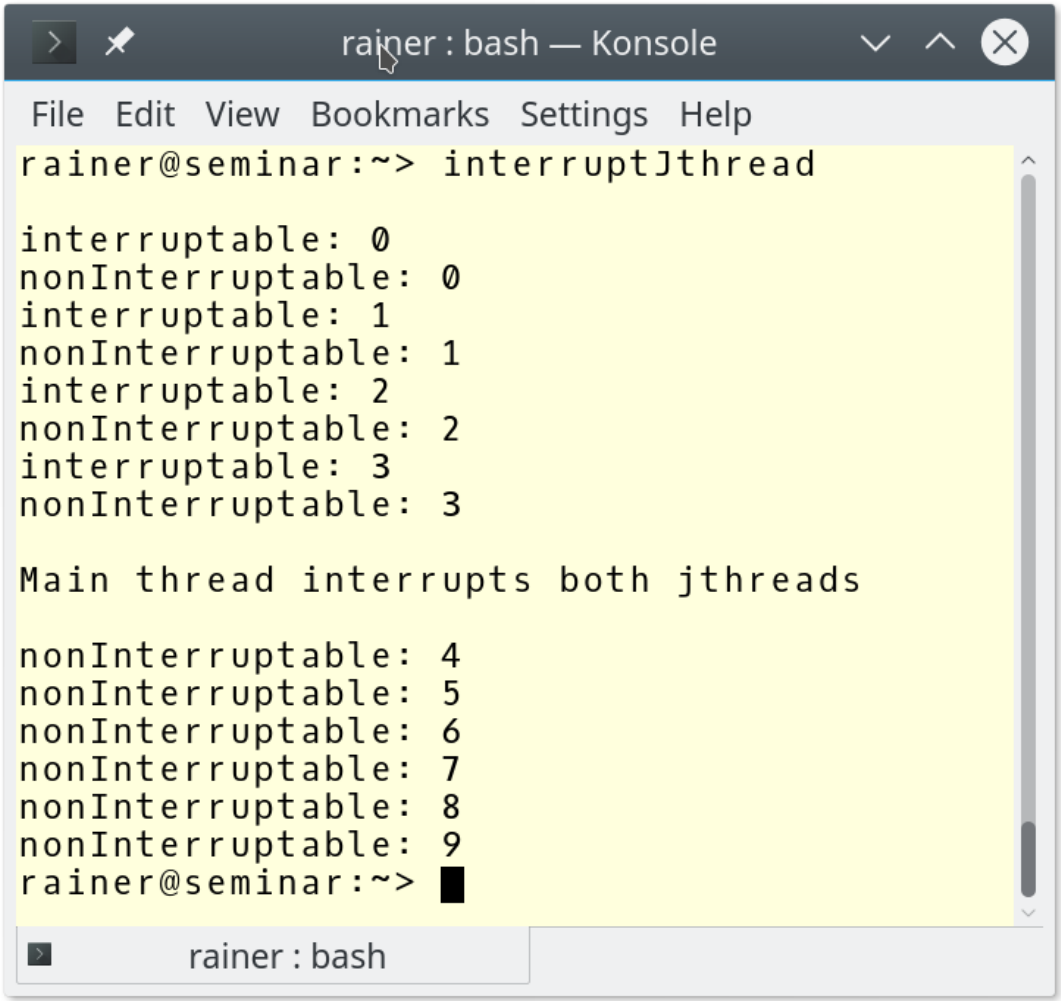
Cooperative Interruption of `std::jthread`

```
jthread nonInterruptable([]{
  int counter{0};
  while (counter < 10){
    this_thread::sleep_for(0.2s);
    cerr << "nonInterruptable: "
         << counter << endl;
    ++counter;
  }
});
```

```
jthread interruptable([](interrupt_token itoken){
  int counter{0};
  while (counter < 10){
    this_thread::sleep_for(0.2s);
     if (itoken.is_interrupted()) return;
    cerr << "interruptable: "
         << counter << endl;
    ++counter;
  }
});
```

```
this_thread::sleep_for(1s);
cerr << endl;
cerr << "Main thread interrupts both jthreads" << endl;

nonInterruptable.interrupt();
interruptable.interrupt();
```

Cooperative Interruption of `std::jthread`



Use Tasks instead of Condition Variables

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();
```

Criteria	Condition Variable	Tasks
Critical Region	Yes	No
Spurious Wakeup	Yes	No
Lost Wakeup	Yes	No
Repeatedly Synchronisation	Yes	No

Condition Variables are hard to use

Save Variant

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

 ...

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

Wrong Opimisation

```
std::atomic<bool>  ready{false};

...

ready = true;
condVar.notify_one();

 ...

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

Condition Variables are hard to use

Wrong Opimisation

```
std::atomic<bool>  ready{false};

...

ready = true;
condVar.notify_one();
```

Wrong Opimisation

```
std::unique_lock<std::mutex> lck(mut);
while ( ![]{ return dataReady.load(); }()
{
   // time window
   condVar.wait(lck);
}
```

Condition variables are synchronisation mechanism at the **same** time.

Pack Mutexes into Locks

No Release of the lock

- `std::mutex`

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

- `std::lock_guard`

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

Minimal Locking

Minimal locking or

Never call unknown code while holding a lock

- `std::lock_guard`

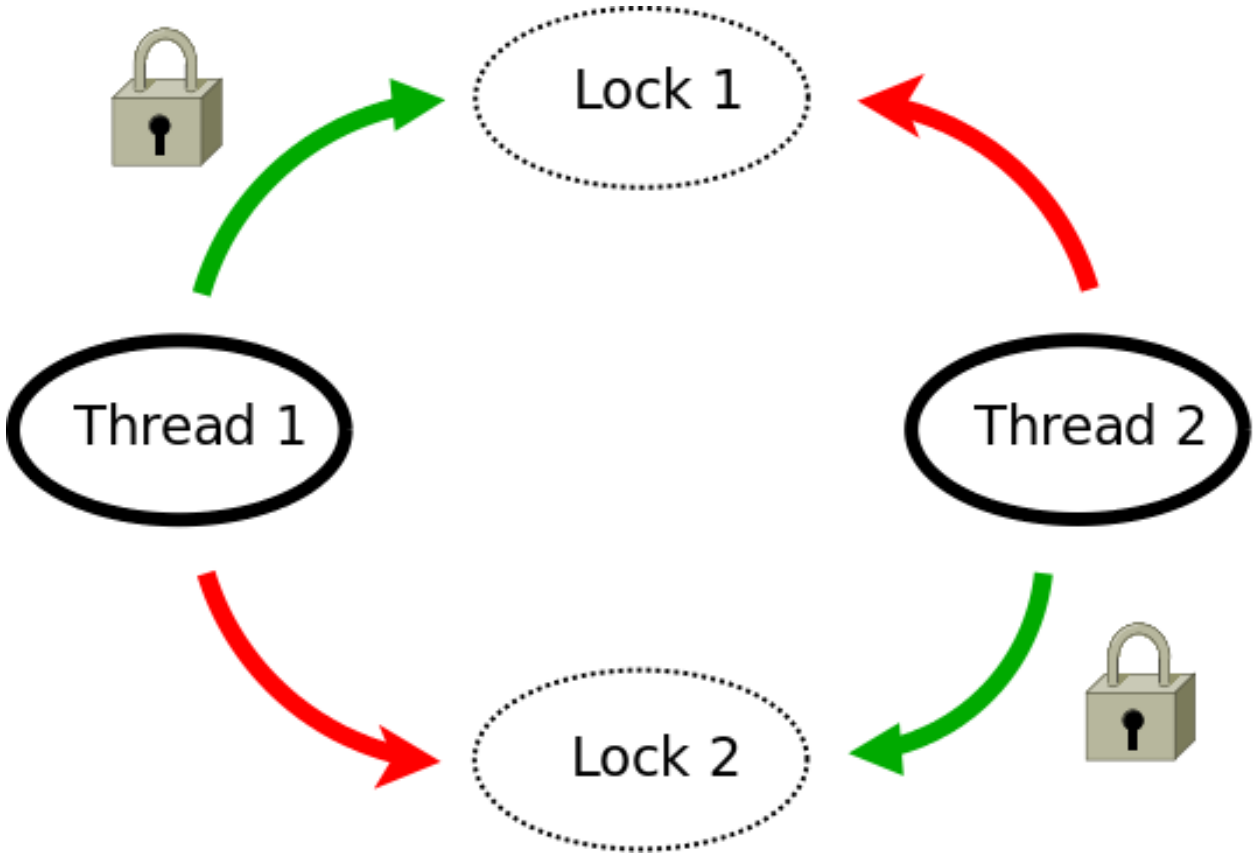
```
mutex m;
{
  lock_guard<mutex> myLock(m);
  shaVar = getVar();
}
```

- `std::lock_guard`

```
mutex m;
auto localVar = getVar();
{
  lock_guard<mutex> myLock(m);
  shaVar = localVar;
}
```

Pack Mutexes into Locks

Locking of the mutexes is different order



Pack Mutexes into Locks

Atomic lock of the mutex

- `std::unique_lock`

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

- `std::scoped_lock` (C++17)

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

Best Practices for Concurrency

General

Multithreading

Memory Model

Don't Program lock-free

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`.

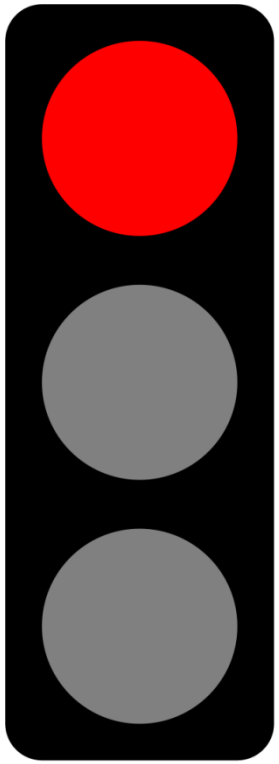
Hans Böhm

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

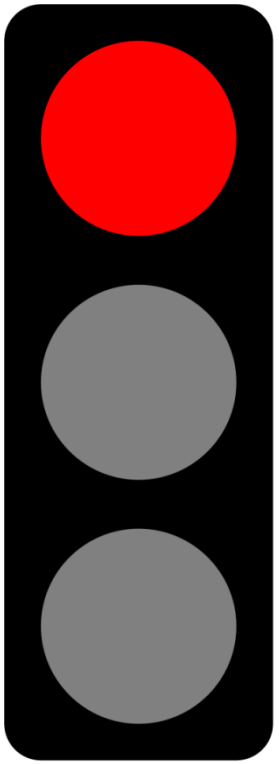
Don't Program lock-free: ABA

A



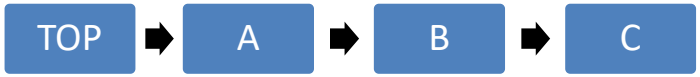
B

A



Don't Program lock-free: ABA

A lock-free, singly linked list (stack)



- Thread 1
  - wants to remove `A`
  - stores
    - head = A
    - next = B
  - checks if `A == head`
  - make `B` to the new `head`
  - `B` is already deleted by Thread 2
- Thread 2
  - removes `A`

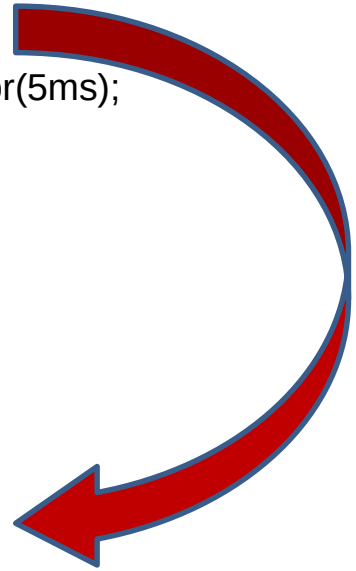
  - removes `B` and deletes `B`

  - pushes `A` back

Use Proven Patterns

Wait with sequentiel consistency

```
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);
}
```

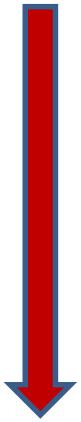


```
int main(){

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

};
```

Use Proven Patterns

Wait with acquire-release semantic

```
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

};
```

Use Proven Patterns

Atomic counter

```
#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
}
```

Don't Reinvent the Wheel

[Boost.Lockfree]
[CDS (Concurrent Data Structures)]

Don't Reinvent the Wheel

- Boost.Lockfree
  - Queue
    - A lock-free multi-producer/multi-consumer queue
  - Stack
    - A lock-free multi-producer/multi-consumer stack
  - spsc_queue
    - A wait-free single-producer/single-consumer queue (ringbuffer)

Don't Reinvent the Wheel

- Concurrent Data Structures (CDS)
  - Contains a lot of containers
  - Stacks (lock-free)
  - Queues and Priority-Queues (lock-free)
  - Ordered lists
  - Ordered sets and maps (lock-free and lock-based)
  - Unordered sets and maps (lock-free and lock-based)

Best Practices for Concurrency

General

Multithreading

~~Parallel~~

Memory Model

Blogs

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

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

Rainer Grimm

Training, Coaching, and Technology Consulting

www.ModernesCpp.de