23

Oct 2025

Once Upon a Thread in the Mutex

The Road to `std::synchronized_value<T>` and Why It Matters

Bartosz Moczulski

https://bartosz.codes/

conference code::dive

Intro
MT is hard

Few help, many sorrow

MT-Safe advice

Problem statement

```
int foo, bar;              // (1.)
std::mutex m1, m2;         // (2.)

// caller MUST hold appropriate lock  (3.)
int incFoo_(int d) { return foo += d; }

// API
int incFoo(int d) {
   // MISSING LOCK !!!     (4.)
   return incFoo_(d);
}
```

We want answers!

1. Is `foo` protected by any mutex, which one?
2. Which variables does a `m1` protect?
3. Which variables are safe to access at any given place? a.k.a. “Which locks are held here?”
4. Assert locks are held at call site.
5. **C++ syntax only**, no compiler extensions (ideally).

MT challenges

Data races

- Mutex missing entirely
- 2 atomic vars ≠ 1 atomic pair
- MT-safe? Check you `man` page!

Deadlocks

- 2 or more mutexes
- ABBA (and more)
- detection
- prevention

C++ Core Guidelines

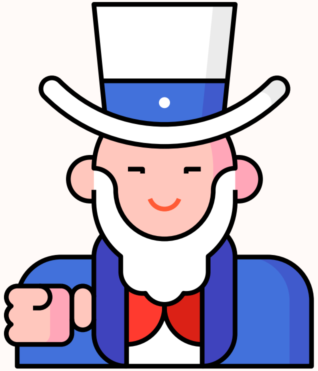
CP.1:

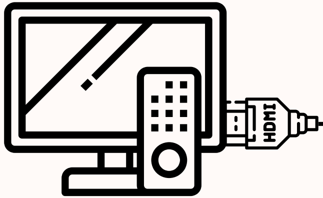
Assume that your code will run as part of a multi-threaded program

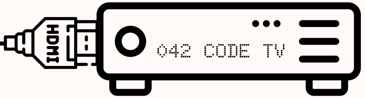
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

I am a Linux detective

I want you



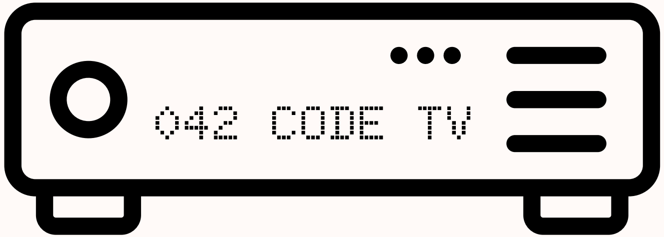




to watch TV

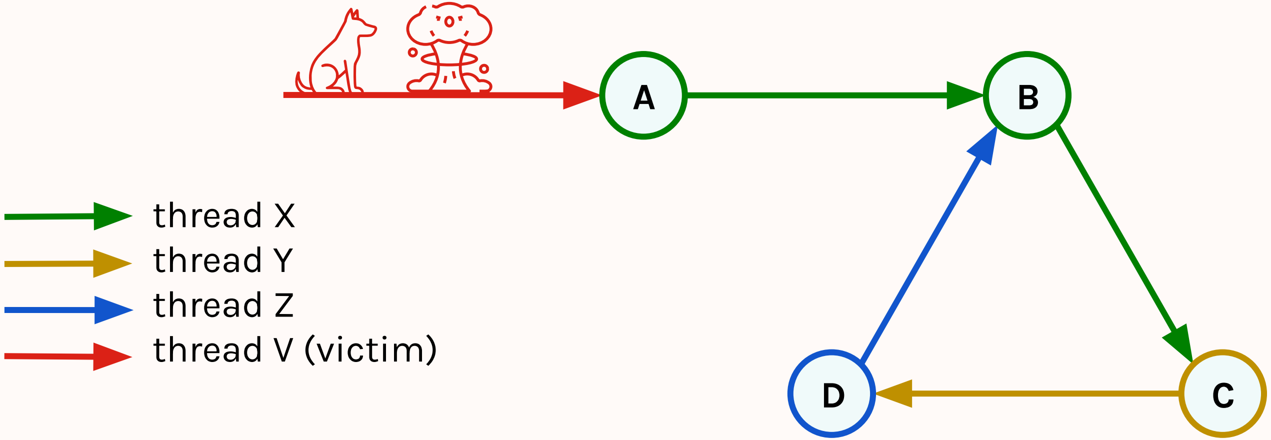
and streaming services

MT at large!



Take cover!

MT at large!



thread X

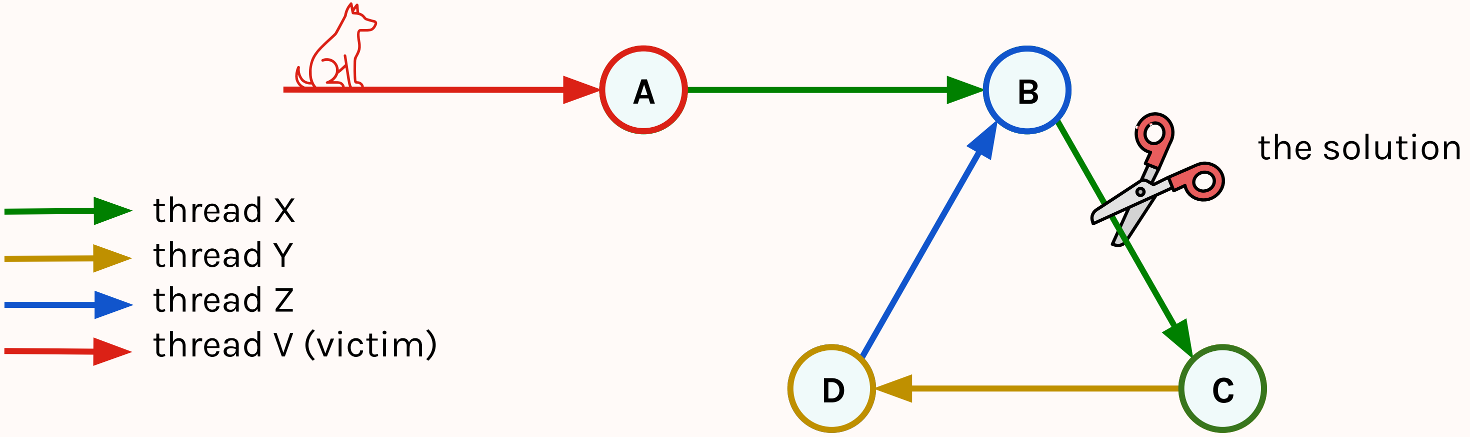
thread Y

thread Z

thread V (victim)

Take cover!

MT at large!



thread X

thread Y

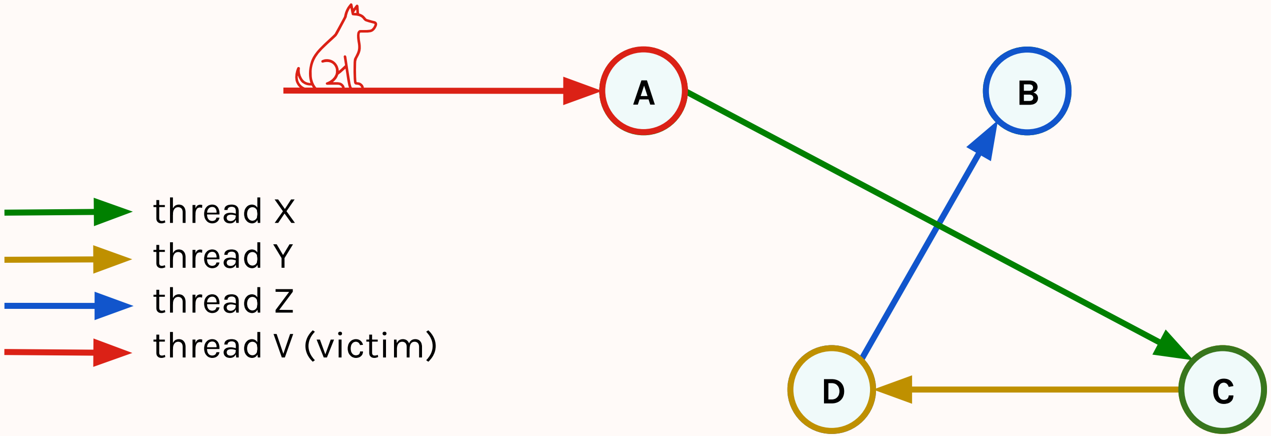
thread Z

thread V (victim)

the solution

Take cover!

MT at large!



thread X

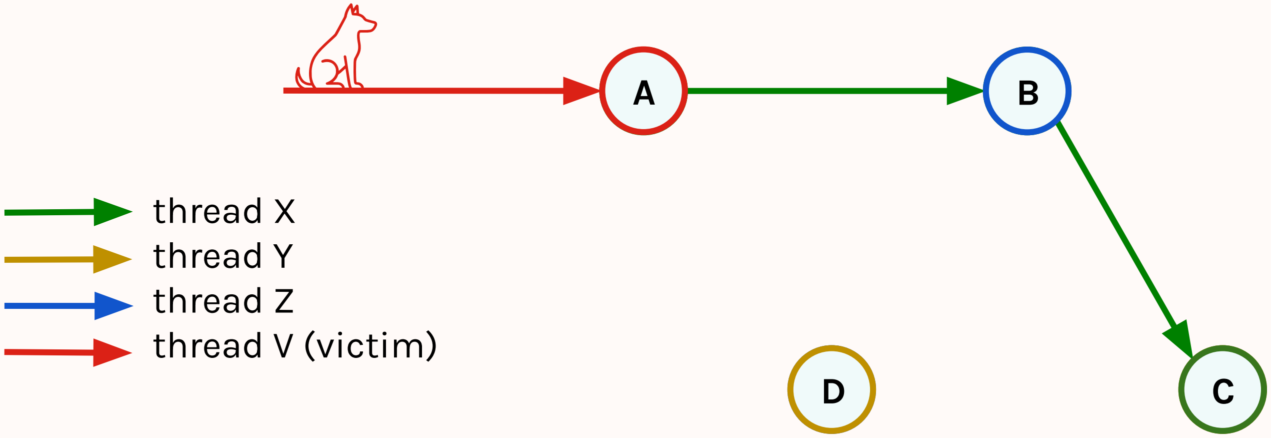
thread Y

thread Z

thread V (victim)

Take cover!

MT at large!



thread X

thread Y

thread Z

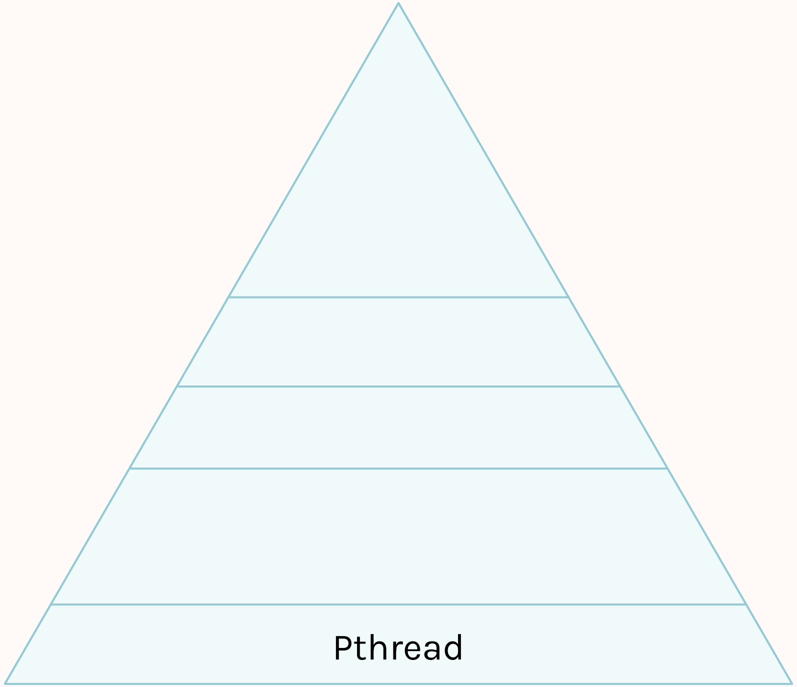
thread V (victim)

Take cover!

The Past

C++98 MT landscape

The C++ MT pyramid



MT features in C++

Meh, not my problem!

C++11, 14, 17

Mutex

and other
*Lockables

.lock()   // BasicLockable
.unlock()

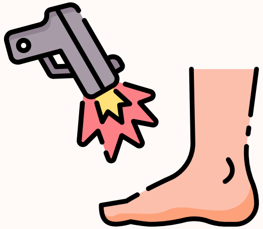
.try_lock()   // Lockable

// TimedLockable
.try_lock_for()
.try_lock_until()

std::
mutex
std::
recursive
_mutex
std::
timed
_mutex
Corporate needs you to find differences
between these pictures.
pthread_mutex_t
They're the same picture.
imgflip.com

*Lockable = RAII violation

```
#include <pthread.h>
struct PTMutex {
   pthread_mutex_t m;
   PTMutex()     { pthread_mutex_init   (&m, nullptr); }
   ~PTMutex()    { pthread_mutex_destroy(&m); }
   // BasicLockable
   void lock()   { pthread_mutex_lock   (&m); } // WHY?! 😭
   void unlock() { pthread_mutex_unlock (&m); }
};
```



C++ Core Guidelines

CP.20: Use RAII, never plain lock()/unlock()

C++11, 14, 17

Mutex

and other
*Lockables

```
.lock()         // BasicLockable
.unlock()

.try_lock()   // Lockable

// TimedLockable
.try_lock_for()
.try_lock_until()
```

Lock

optional RAII
locking

```
mutex m;

// now, defer, adopt?
unique_lock l(m);
unique_lock l(m, defer);
shared_lock l(sm, adopt);
```

Unlocking possible

Guard

mandatory RAII
locking

mutex m;

// now or adopt, no defer
scoped_lock g(m1, m2);
~~lock_guard g(m, defer);~~
lock_guard g(m, adopt);

Pure RAII, no unlocking

C++11, 14, 17

Mutex

and other
*Lockables

std::shared_**mutex**
is a
pthread_rw**lock**_t

Lock

optional RAII locking

std::**shared**_lock
is sort of **unique**
(unlike shared_ptr)

std::lock() is **not a lock**
and does **not** unlock
(needs an extra **guard**)

prefer std::scoped_lock
in C++ >= 17

Guard

mandatory RAII locking

std::scoped_**lock**
is in fact a **guard**

why not
std::scoped_**guard**?

C++11, 14, 17

Mutex

There are many
*Lockable

- std::mutex ==
  std::recursive_mutex ==
  std::timed_mutex ==
  pthread_mutex_t
- std::shared_mutex =
  pthread_rw**lock**_t
- **broken RAII**

Lock

std::unique|shared_lock
optional RAII locking

- Is *Lockable
  (just like **Mutex**)
- May or may not lock
  (defer, adapt)
- std::lock() is **not a lock** and does **not** unlock (needs an extra **guard**)
- **shared**_lock is unique
  (unlike shared_ptr)

Guard

std::lock_guard
mandatory RAII locking

- Guards any *Lockable, including Mutex (not just a **lock**)
- Is an ownership wrapper
  (just like a **lock**)
- std::scoped_lock (C++17) is a **guard**
- Can adapt
  (but not defer)

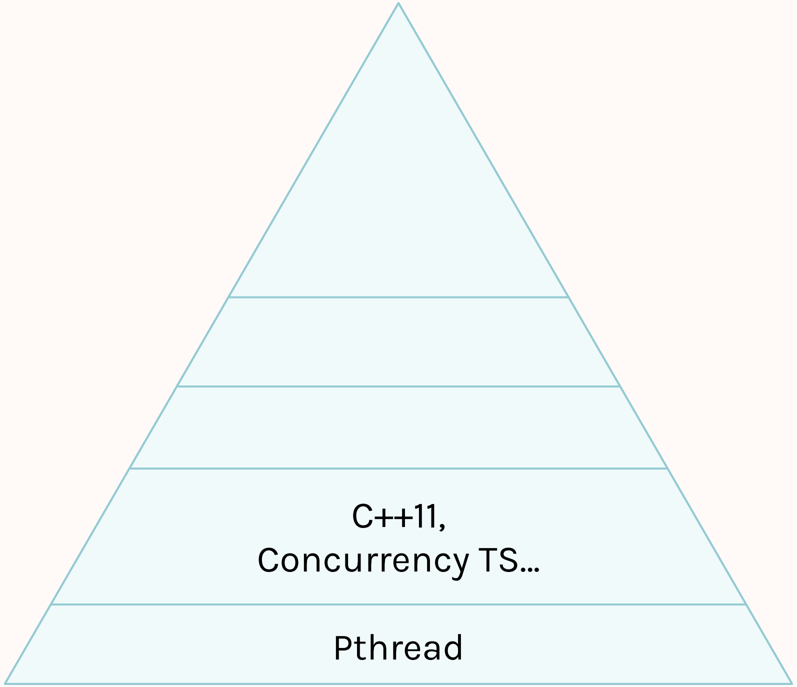
Deadlocks? Meh!

std::mutex::lock()

An implementation that can detect the invalid usage is encouraged to **throw** a **std::system_error** with error condition **resource_deadlock_would_occur** instead of deadlocking.

Yes, do it, please! 🙏

The C++ MT pyramid



C++11,
Concurrency TS…

Pthread

MT features in C++

Perhaps it *is* my problem after all?
(mutexes, locks, CVs)

Meh, not my problem!

DEMO 1

The Present

MT Quiz, is it safe?

```
#include <mutex>
class C1 {
    int foo;
public:
    int getFoo() const {
        return foo;
    }
    void setFoo(int newFoo) {
        foo = newFoo;
    }
};
```

MT Quiz, is it safe?

```cpp
#include <mutex>
class C2 {
    int foo;  std::mutex m;
public:
    int getFoo() const {
        std::scoped_lock l(m);  return foo;
    }
    void setFoo(int newFoo) {
        std::scoped_lock l(m);  foo = newFoo;
    }
};
```

MT Quiz, is it safe?

```
#include <mutex>
class C3 {
    int foo;  std::mutex m;
    int getFoo_locked() const {
        return foo; // caller MUST hold m
    }
public:
    int getFoo() const {
        // oops! lock_guard(m) missing here :(
        return getFoo_locked();
    }
};
```

MT Quiz, is it safe?

```cpp
#include <mutex>
class C4 {
    int foo, bar;  std::mutex m1, m2;
public:
    int getFoo() const {
        std::scoped_lock l(m1);  return foo;
    }
    int getBar() const {
        std::scoped_lock l(m2);  return bar;  // m2 or m1 ?!
    }
    // ... setters accordingly
};
```

Challenges

1. How to express data-mutex relationship?

2. How for force mutex ownership during data access?

This is **impossible** in current C++!

Or is it … ?

Thread Safety Analysis (TSA)

GUARDED_BY

```
std::mutex m;
int foo GUARDED_BY(m);
```

REQUIRES

```
int bar() {
  foo++; // racy 👎
}

int bar() REQUIRES(m) {
  foo++; // safe 👍
}
```

Deadlock prevention

```
std::mutex m1;
std::mutex m2
  ACQUIRED_AFTER(m1);
```

more: https://clang.llvm.org/docs/ThreadSafetyAnalysis.html

Thread Safety Analysis (TSA)

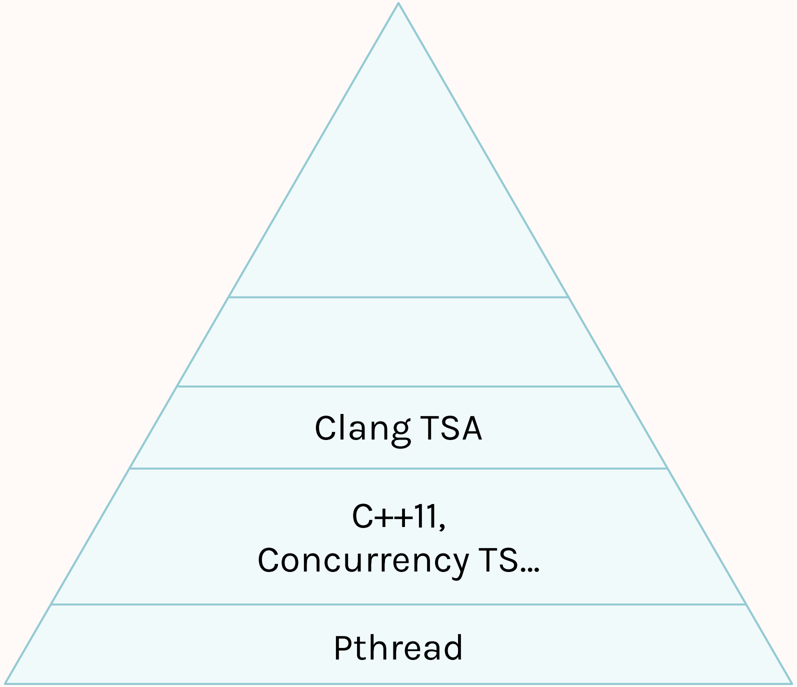
Pros

- Explicit about protected data
- Locking forced by compiler

Cons

- Only in Clang
- Annotations required
- Only libc++ (not libstdc++)
- Not everything works,
  e.g. std::unique_lock doesn’t

The C++ MT pyramid



Clang TSA

C++11,
Concurrency TS...

Pthread

MT features in C++

I can help with locking, if you need

Perhaps it *is* my problem after all?
(mutexes, locks, CVs)

Meh, not my problem!

DEMO 2

The Future

std::synchronized_value<T>

https://wg21.link/P0290

by Anthony Williams

Part of Concurrency TS2 (https://wg21.link/N4953)

synchronized_value

```
template<class T>
class synchronized_value
{
  T data;
  std::mutex m;
  // NOTE: everything is private!
}

// F is any callable:   R F(Ts... &data);
R apply(F f, SV<Ts>... &sv);
```

synchronized_value

```
// example
std::synchronized_value<int> balance;

void deposit(int amount) {
   std::apply([=](auto &b) { // b is balance.data
       b += amount;
   }, balance); // will lock balance.m for you
}
```

synchronized_value

```cpp
// example
std::synchronized_value<int> from, to;

void transfer(int amount) {
   std::apply([=](auto &f, auto &t) {
       f -= amount;
       t += amount;
   }, from, to); // lock both, deadlock avoidance
}
```

synchronized_value

```
class Account {
    struct Data {
        int balance;
    };
    std::synchronized_value<Data> data;

public:
    void transfer(Account &to, int amount) {
        std::apply([=](auto& our, auto& their) {
            our.balance -= amount;  their.balance += amount;
        }, data, to.data);
    }
};
```

synchronized_value

```
class Account {
    struct Data {
        int balance;
    };
    std::synchronized_value<Data> data;

    // Raw Data& param? Must have been already locked by the caller 👍
    void addInterest(Data &d, double interestRate) {
        d.balance *= (1.0 + interestRate / 100.0);
    }

    // ...
};
```

T + mutex = ❤️

Aye (forced)

- Explicit about protected data
- Locking forced by syntax
- Makes TSA unnecessary (if used correctly)

Nay (prevened)

- No direct data access
- Lockable is hidden
- No accessor operators (=, *, ->)

These are pros too, not cons!

Lambda vs. Handle

```
std::synchronized_value<int>
   from, to;

std::apply([=](auto &f, auto &t){
   f -= amount;
   t += amount;
}, from, to);
```

```
boost::synchronized_value<int>
   from, to;

{
   auto [f, t] =
      boost::synchronize(from, to);
   *f -= amount;
   *t += amount;
}
```

Handle misuse is easy!

```
boost::synchronized_value<int>
   from, to;

{
   auto [f, t] =
      boost::synchronize(from, to);
   *f -= amount;
   *t += amount;
}
```

```
boost::synchronized_value<int>
  from, to;

{
   auto f = from.synchronize();
   auto t = to.synchronize();
   *f -= amount;
   *t += amount;
}
```

Handle misuse is easy!

```
boost::synchronized_value<int>
   from, to;

{
   auto [f, t] =
      boost::synchronize(from, to);
   *f -= amount;
   *t += amount;
}
```

```
boost::synchronized_value<int>
  from, to;

{
   auto t = to.synchronize();
   auto f = from.synchronize();
   *f -= amount;
   *t += amount;
}
```

Handle misuse is easy!

```
class Account {
   boost::synchronized_value<int> balance;
};

Account::send(Account& other, int amount)
{
   auto from = this->balance.synchronize(); // ERROR: ABBA!
   auto to = other.balance.synchronize();
   *from -= balance;
   *to += balance;
}
```

Handle misuse is easy!

```
boost::synchronized_value<int>
   from, to;

{
   auto [f, t] =
      boost::synchronize(from, to);
   *f -= amount;
   *t += amount;
}
```

```
boost::synchronized_value<int>
  from, to;

{
   // Dreaded accessors, alas!
   *from = *from - amount;
   *to = *to + amount;
}
```

Avoid sequential locking

```
std::synchronized_value<int>
   from, to;

std::apply([=](auto &f, auto &t){
   f -= amount;
   t += amount;
}, from, to); // Do this (2-ary) 👍
```

```
std::synchronized_value<int>
   from, to;

std::apply([=](auto &f){
   std::apply([&](auto &t){
      f -= amount;
      t += amount;
   }, to); // NOT that! ABBA! 👎
}, from);
```

NEVER LEAK ACCESS TO DATA!

```
std::synchronized_value<int> foo;

int &raw = std::apply([=](auto &f){
  return f; // DON'T!
}, foo);

int *p = std::apply([=](auto &f){
  return &f; // DON'T!
}, foo);
```

C++ Core Guidelines

CP.50:

Define a mutex together with the data it guards. **Use synchronized_value<T>** where possible

https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

Current SV limitations

- Only Experimental implementation
  - Only in GCC >= 13
- Limited functionality:
  - Only `std::mutex`, not other Lockables
  - Doesn’t work with `const` methods
  - Doesn’t support shared locking
  - Doesn’t support condition variables (this is tricky!)

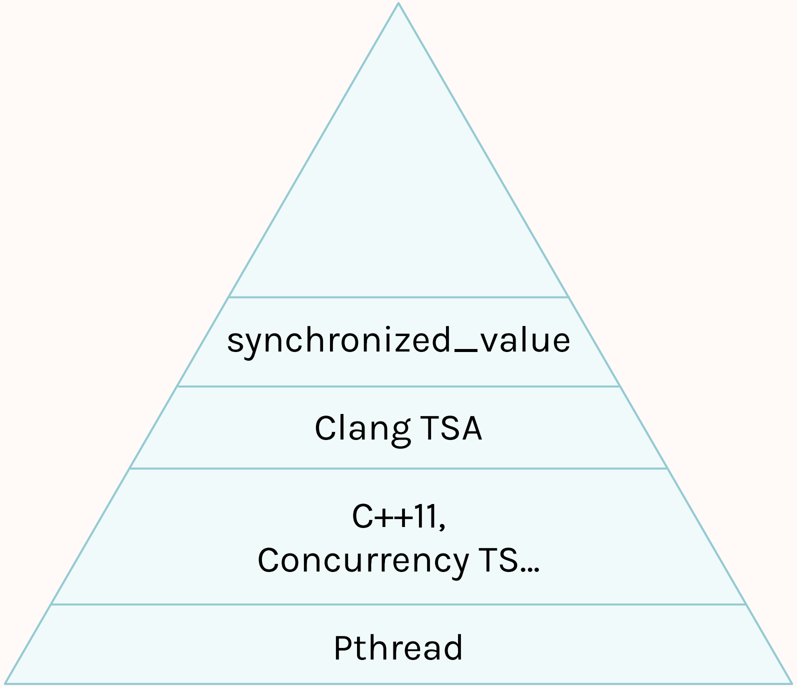
My own variant

synchronized_value by Bartosz Moczulski

https://github.com/bmoczulski/synchronized_value

- Features beyond P0290r4:
  - ✓ Works with any Lockables
  - ✓ Works in `const` methods
  - ✓ Supports shared locking (experimental)

The C++ MT pyramid



synchronized_value

Clang TSA

C++11,
Concurrency TS...

Pthread

MT features in C++

You SHALL NOT pass without a lock!

I can help with locking, if you need

Perhaps it *is* my problem after all?
(mutexes, locks, CVs)

Meh, not my problem!

DEMO 3

The Summary

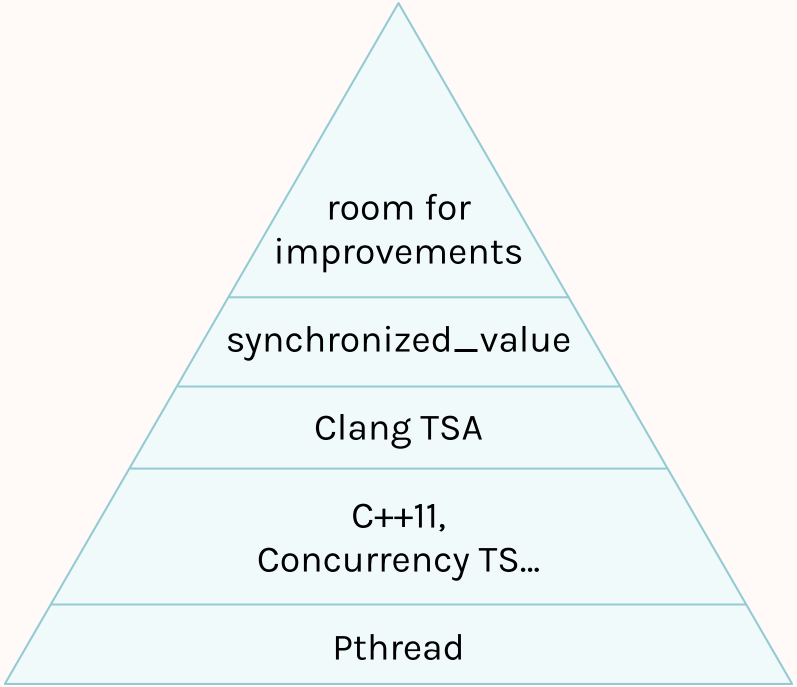
Feature matrix

	C++ Standard	You know what you are locking	Unlocked access impossible (no data races)	Deadlock detection	Deadlock prevention
pthread	❌ (POSIX)	❌	❌	❌	❌
Other libraries	❌	✅ (some)	✅ (some)	✅ (some)	❌
std::mutex	✅	❌	❌	❌	❌
Clang TSA	❌	✅	✅ (partially)	❌	❌
synchronized_value	✅ (coming)	✅	✅ (if used well)	❌	❌
(future solutions)	?	?	?	✅ (hope)	❌ (unlikely)

Alternatives

- Boost::synchronized_value<T>
  - Access through proxy object only, n-ary support
- CopperSpice CsLibGuarded
  - Access through proxy object only, single mutex only (n-ary not supported)
- Facebook Folly: Synchronized<T, Mutex>
  - Access through proxy object or lambda
- ~~Google Abseil: absl::Mutex~~
  - CV included, runtime deadlock detection in debug mode, conditional critical sections, free-standing mutex with TSA

The C++ MT pyramid



room for improvements

synchronized_value

Clang TSA

C++11, Concurrency TS…

Pthread

MT features in C++

Who knows… ¯_(ツ)_/¯

You SHALL NOT pass without a lock!

I can help with locking, if you need

Perhaps it *is* my problem after all? (mutexes, locks, CVs)

Meh, not my problem!

Valgrind

Blind to CVs!

(in some scenarios, RTFM)

Thank you

Bartosz Moczulski
will return

FEEDBACK

Q & A

https://bartosz.codes

https://github.com/bmoczulski/synchronized_value

FEEDBACK