Safer multithreading programming with C++

Sébastien Gonzalve

CPPP 2021 PARIS

MUREX

CODE RECKONS

Science to the CORE

A long time ago in a galaxy piece of code far, far away….

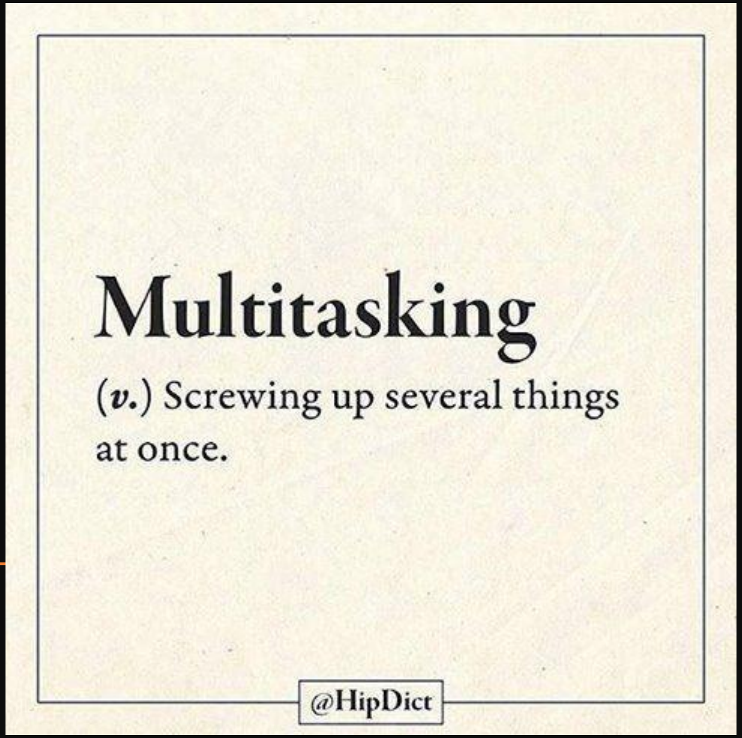
```
std::string Config::getVersion() const { return _version; }

void out_there(Config &config) {
  // [ ... a lot of code ]

  // Block startup until service version is available
  while (config.getVersion().empty()) {
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
  }
  // [ ... a lot of code again ]
}
```

Threads

Why threads?

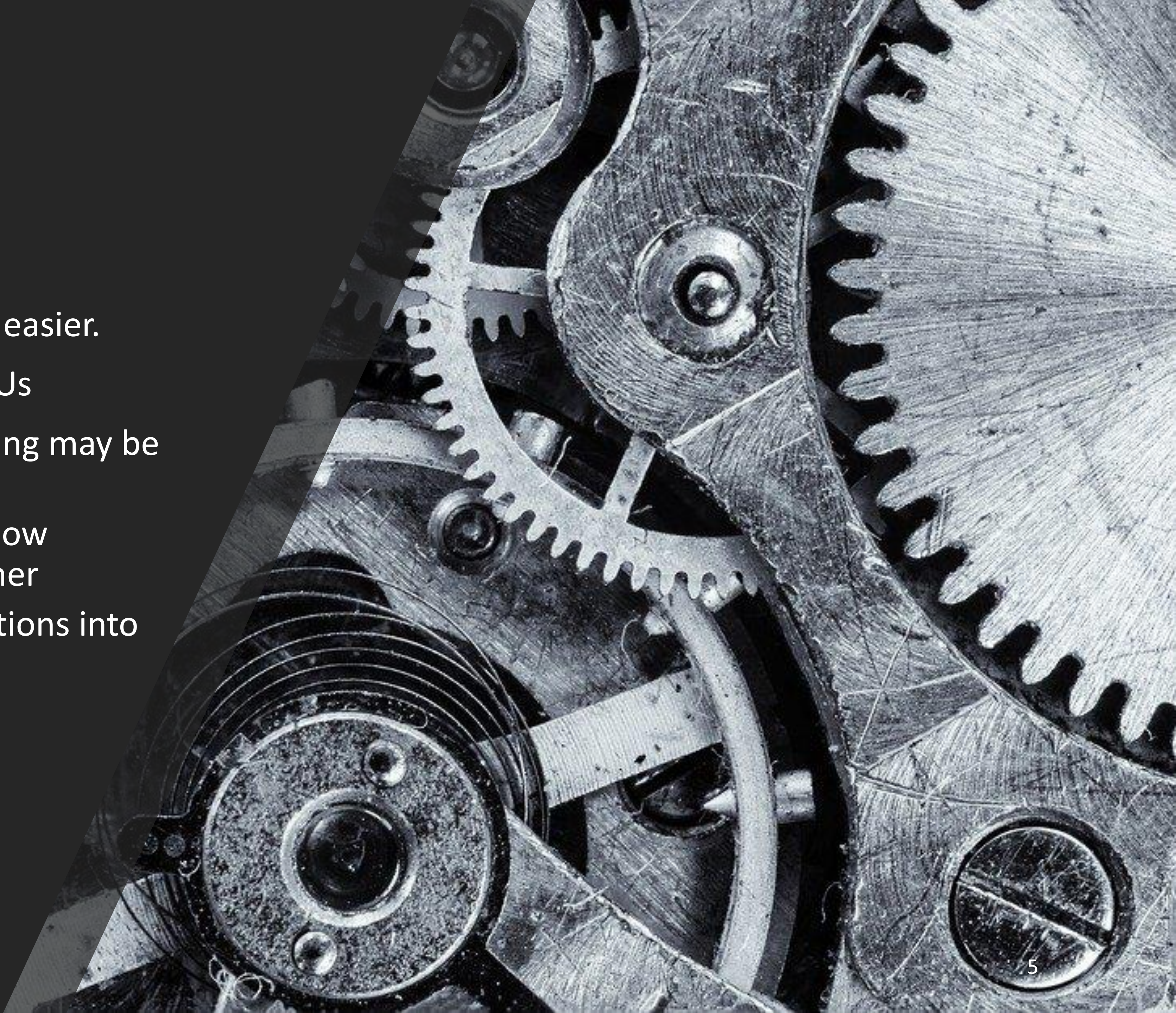
"A Computer is a state machine. Threads are for people who can't program state machines".

Alan Cox



Why threads?

- Using a state machine may be easier.
- We are all using multi core CPUs
- In some situation multithreading may be necessary/interesting:
  - Separate flows that have low interactions with each other
  - Transform blocking operations into asynchronous operations
  - need for low latency
  - Parallelization of code



Threads and C++

- Until C++11 threads were not part of the language.

« The C++ specification […] makes reference to an *abstract machine* that is a generalization of actual systems. […]

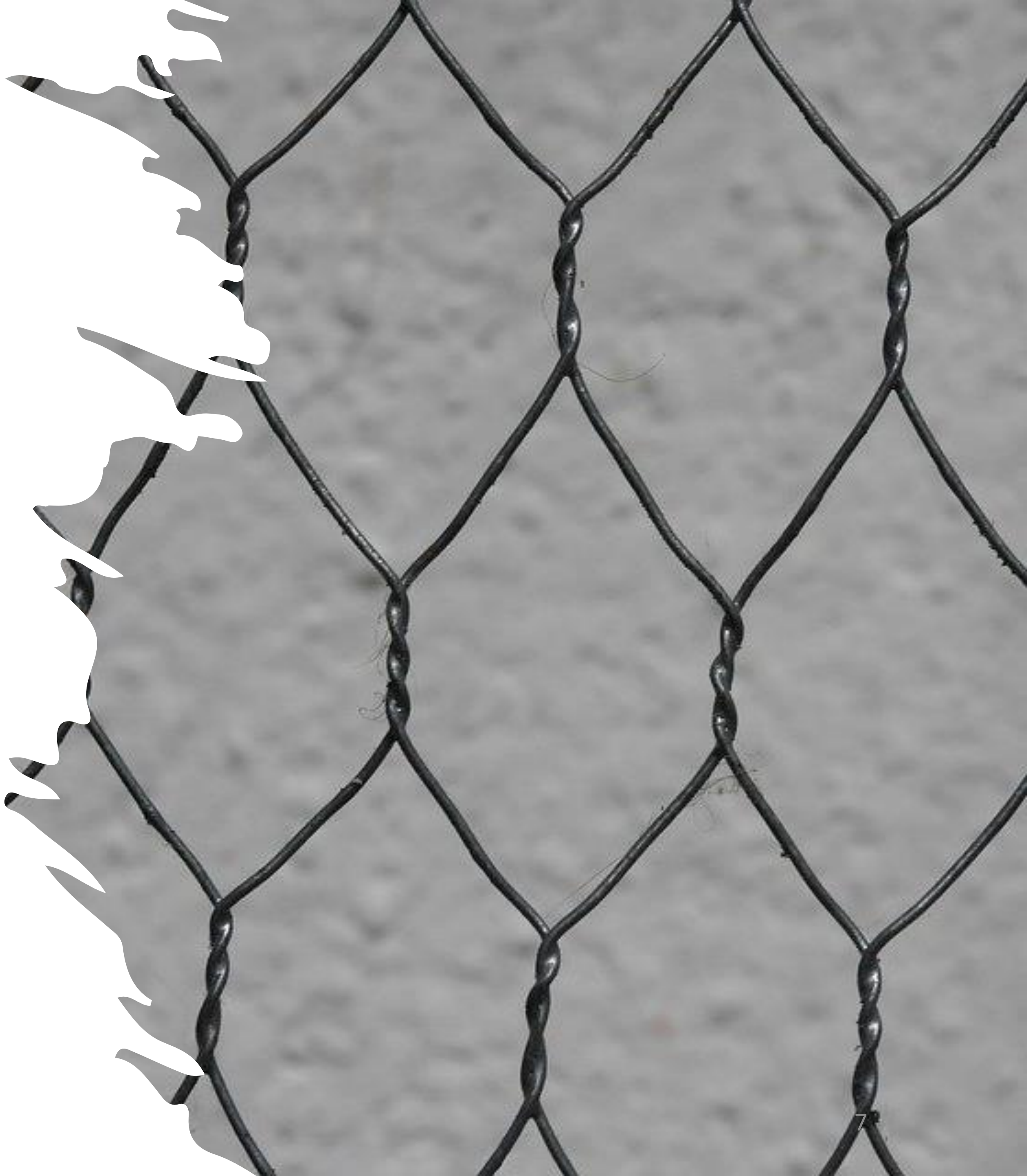
The abstract machine in the **C++98/C++03 specification is fundamentally single-threaded**, […]

The abstract machine in **C++11 is multi-threaded by design**. It also has a well-defined ***memory model***; that is, it says what the compiler may and may not do when it comes to accessing memory.”

https://stackoverflow.com/questions/6319146/c11-introduced-a-standardized-memory-model-what-does-it-mean-and-how-is-it-g

How to create thread

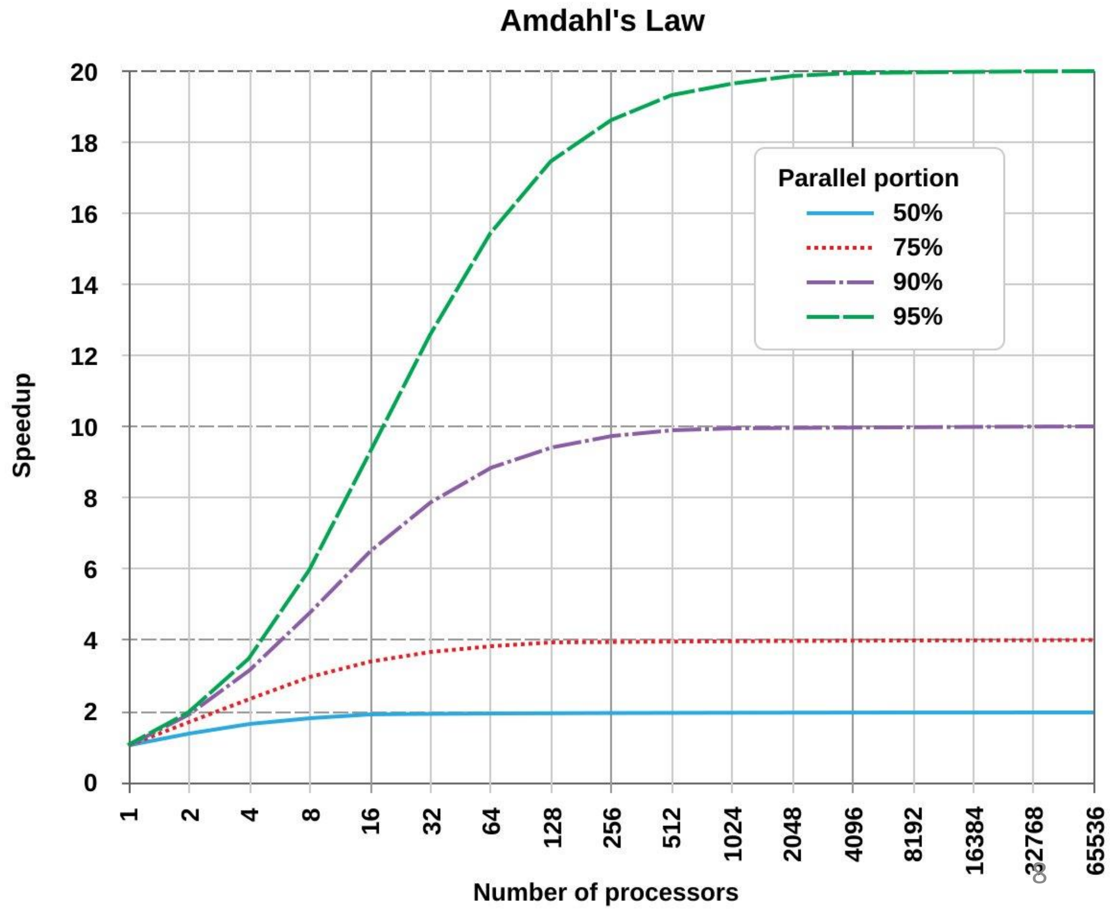
- Use std::thread/std::jthread(C++20)
- Use std::async(std::launch::async, …)
- But you may use other ways: pthreads_create, CreateThread or openMP for code parallelization



Multithreading challenges

- Distributing data is not difficult; synchronization is.
- Law of Demeter
- Parallelization has limits (Amdahl's law)

Make sure the motivation for creating thread is well known.



Concurrency

Races

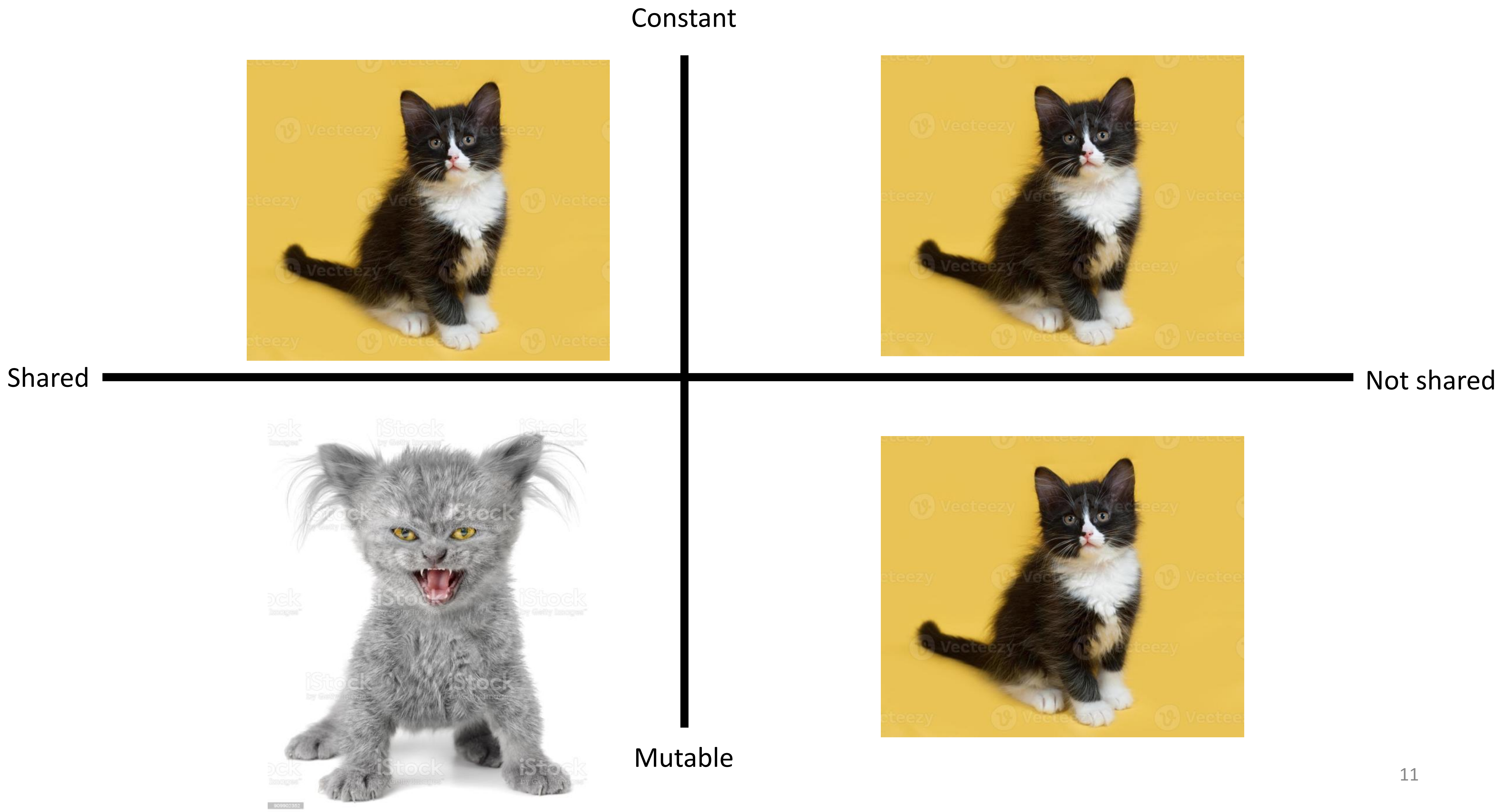
- A race condition arises when a program behaviour depends on the sequence or timing of the program's processes or threads.

- Race condition on data are calle data-races;

=>The memory model defined in C++11 specifies that a program containing a data race has undefined behavior.

It is possible to have races in a program without data races. It may not be a problem per se.

https://en.wikipedia.org/wiki/Race_condition


Constant
Shared
Not shared
Mutable

How do we know they are here?

Evidences:

- Random behaviour of you tests and/or software.
- Data corruptions

Tools:

- Static code analysers (Coverity, code sonar, klocwork, etc…)
- Valgrind with helgrind tool
- Lib sanitizer (-fsanitize=thread)

Example of data race

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

int score{0};
constexpr size_t nb_thread = 2;

void race() {
  while (score < 170000)
     score++;

  if (score == 170000)
     std::cout << "I'm the winner\n";
}

int main() {
  std::array<std::jthread, nb_thread > threads;

  for (auto &t : threads)
        t = std::jthread(race);
}
```

- When nb_thread == 2 => both threads wins
- When above, no more winner.

Can you spot issue(s)?

What could be done to fix?

Note that even when using atomic (no data race), the result is random.

Thread sanitizer report:

```
test@localhost:/tmp$ ./sample
I'm the winner
==================
WARNING: ThreadSanitizer: data race (pid=2022142)
 Read of size 4 at 0x0000004061f4 by thread T2:
  #0 race() /tmp/sample.cpp:10 (sample+0x401284)
  #1 void std::__invoke_impl<void, void (*)()>(std::__invoke_other, void (*&&)()) /usr/include/c++/9/bits/invoke.
  #2 std::__invoke_result<void (*)()>::type std::__invoke<void (*)()>(void (*&&)()) /usr/include/c++/9/bits/invoke
  #3 void std::thread::_Invoker<std::tuple<void (*)()> >::_M_invoke<0ul>(std::_Index_tuple<0ul>) /usr/include/c-
  #4 std::thread::_Invoker<std::tuple<void (*)()> >::operator()() /usr/include/c++/9/thread:251 (sample+0x4022d
  #5 std::thread::_State_impl<std::thread::_Invoker<std::tuple<void (*)()> > >::_M_run() /usr/include/c++/9/thre
  #6 execute_native_thread_routine ../../../../../libstdc++-v3/src/c++11/thread.cc:80 (libstdc++.so.6+0xd73d3)

 Previous write of size 4 at 0x0000004061f4 by thread T1:
  #0 race() /tmp/sample.cpp:11 (sample+0x4012ae)
  #1 void std::__invoke_impl<void, void (*)()>(std::__invoke_other, void (*&&)()) /usr/include/c++/9/bits/invoke.
  #2 std::__invoke_result<void (*)()>::type std::__invoke<void (*)()>(void (*&&)()) /usr/include/c++/9/bits/invoke
  #3 void std::thread::_Invoker<std::tuple<void (*)()> >::_M_invoke<0ul>(std::_Index_tuple<0ul>) /usr/include/c-
  #4 std::thread::_Invoker<std::tuple<void (*)()> >::operator()() /usr/include/c++/9/thread:251 (sample+0x4022d
  #5 std::thread::_State_impl<std::thread::_Invoker<std::tuple<void (*)()> > >::_M_run() /usr/include/c++/9/thre
  #6 execute_native_thread_routine ../../../../../libstdc++-v3/src/c++11/thread.cc:80 (libstdc++.so.6+0xd73d3)

 Location is global 'score' of size 4 at 0x0000004061f4 (sample+0x0000004061f4)

 Thread T2 (tid=2022145, running) created by main thread at:
  #0 pthread_create <null> (libtsan.so.0+0x2de12)
  #1 std::thread::_M_start_thread(std::unique_ptr<std::thread::_State, std::default_delete<std::thread::_State>
  #2 main /tmp/sample.cpp:21 (sample+0x401369)

 Thread T1 (tid=2022144, running) created by main thread at:
  #0 pthread_create <null> (libtsan.so.0+0x2de12)
  #1 std::thread::_M_start_thread(std::unique_ptr<std::thread::_State, std::default_delete<std::thread::_State>
  #2 main /tmp/sample.cpp:21 (sample+0x401369)

SUMMARY: ThreadSanitizer: data race /tmp/sample.cpp:10 in race()
==================
I'm the winner
ThreadSanitizer: reported 1 warnings
```

```
01  #include <array>
02  #include <atomic>
03  #include <iostream>
04  #include <thread>

06  int score{0};
07  constexpr size_t nb_thread = 2;

09  void race() {
10    while (score < 170000)
11      score++;

13    if (score == 170000)
14      std::cout << "I'm the winner\n";
15  }

17  int main() {
18    std::array<std::jthread, nb_thread > threads;

20    for (auto &t : threads)
21          t = std::jthread(race);
22  }
```

Synchronization primitives

- When you need concurrent access on data, you need synchronization (unless atomics are sufficient for your use case)
- There are several classical synchronization mechanisms (on *nix)
  - Semaphores (SysV semaphores, pthread semaphores, etc…)
  - Messages (messages, fifo, sockets, …)
  - Signals (SysV signals, RT signals etc…)
  - …
- Modern C++ (>11) recommend use of higher-level concepts (ex: future) not handle synchronization “by hand”

C++11 std::future

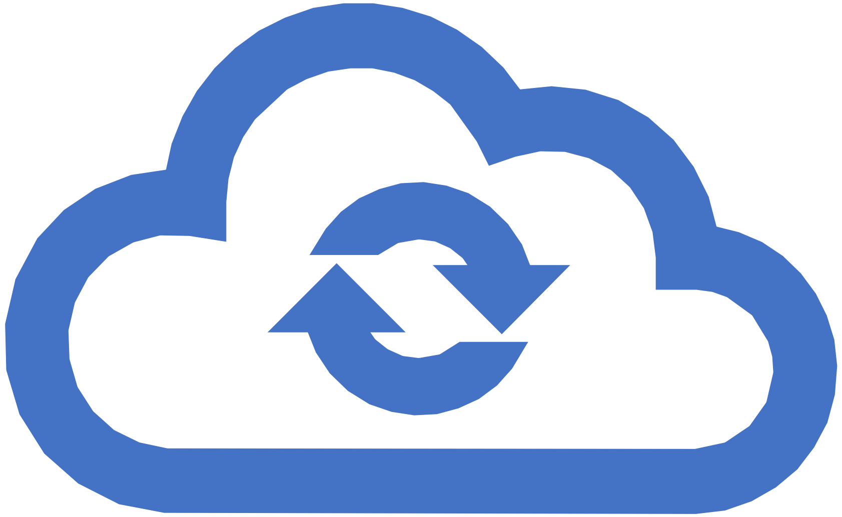
- provides a mechanism to access the result of asynchronous operations
- Easy to use when a single asynchronous result is awaited
- Result may be used from many places (std::shared_future)

```
std::promise<Matrix> multiplication;
std::future<Matrix> multiplication_result =
                      multiplication.get_future();

// In one thread
multiplication_result.wait(); // waits for computation to finish
auto result = multiplication_result.get();

// In another
auto result_matrix = make_big_matrices_compitation();
multiplication.set_value(result_matrix); // trigs a one-shot event
```

Future limitations

- Cannot be used in a signal handler
- Future cannot be reused: need to create a new one (and its pair promise)
- Future created with launch::async are blocking on destructor if promise is not set yet.
- Cannot (in c++14/17) wait on many futures at the same time (std::when_any() std::when_all() were postponed (dropped?))
- Careful: std::async is not always executed in parallel (depends on the std::launch param; async can be lazy evaluation)

=> sometimes one need to use lower-level tools to perform synchronization.



Synchronization

Volatile & threads synchronization

- ***That does _not_ help***

Tl;dr

- ***volatile object*** - an object whose type is volatile-qualified, or a subobject of a volatile object, or a mutable subobject of a const-volatile object. Every access (read or write operation, member function call, etc.) made through a glvalue expression of volatile-qualified type is treated as a visible side-effect for the purposes of optimization (that is, within a single thread of execution, volatile accesses cannot be optimized out or reordered with another visible side effect that is sequenced-before or sequenced-after the volatile access. This makes volatile objects suitable for communication with a signal handler, but not with another thread of execution, see std::memory_order
- Within a thread of execution, [volatile] accesses [...] cannot be reordered [...] within the same thread, but this order is not guaranteed to be observed by another thread, since volatile access does not establish inter-thread synchronization.

  In addition, volatile accesses are not atomic (concurrent read and write is a data race) and do not order memory (non-volatile memory accesses may be freely reordered around the volatile access). (ref: std::memory_order)

Atomic operations library

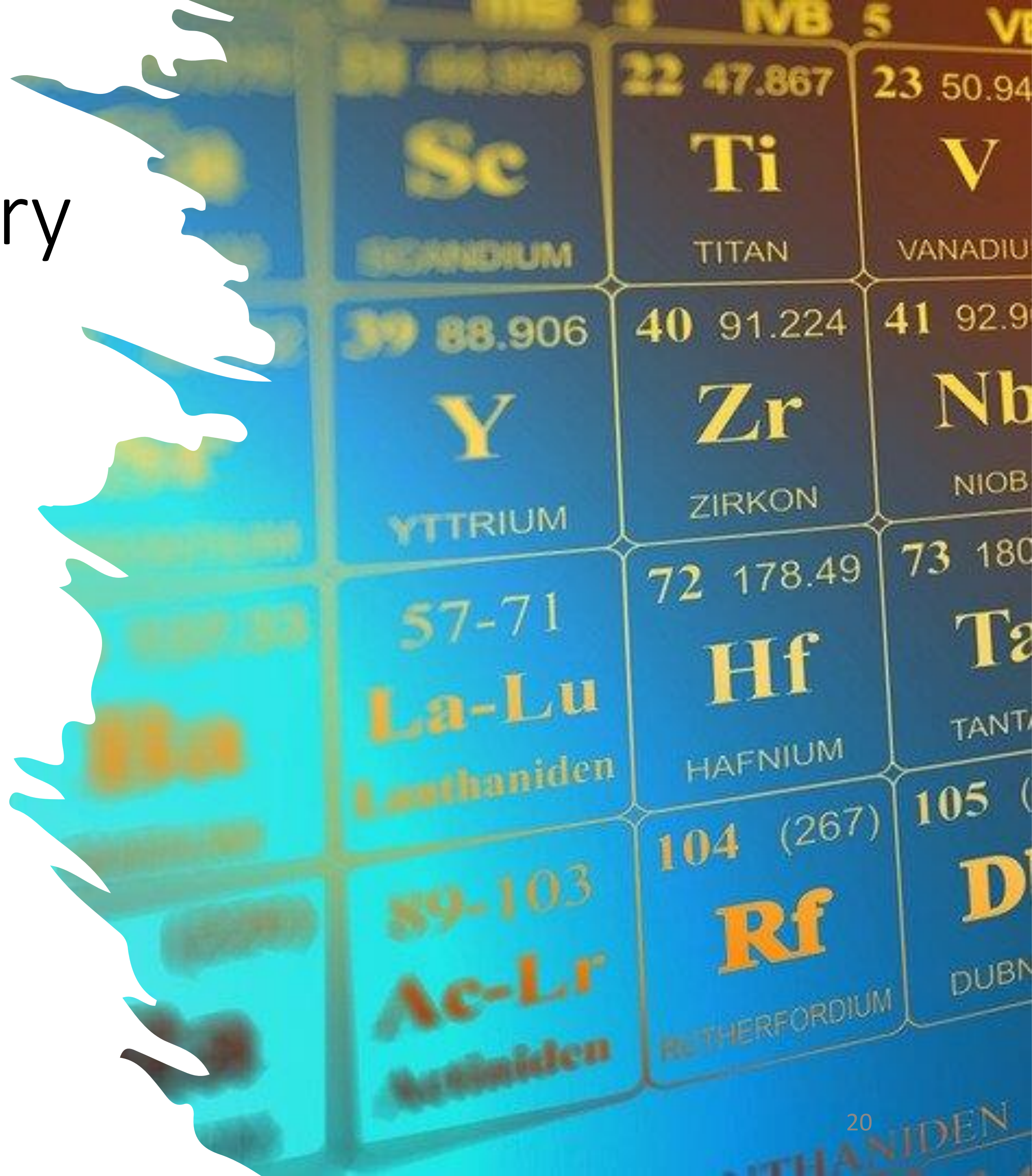
- C++ 11 provides a standard way to use atomic operations

See **Filipe Mulonde** talk on atomic (cppp 2021)

But (!):

- Hard to write correct code using only atomic variables
- Cannot wait/notify on them (prior to C++20)
- A bunch of atomic variables is not atomic; need to create an atomic bunch
- Efficiency mainly applies for native types.

Operation are all atomic but the whole behaviour may be inconsistent (see the race example: no winner)



Locks

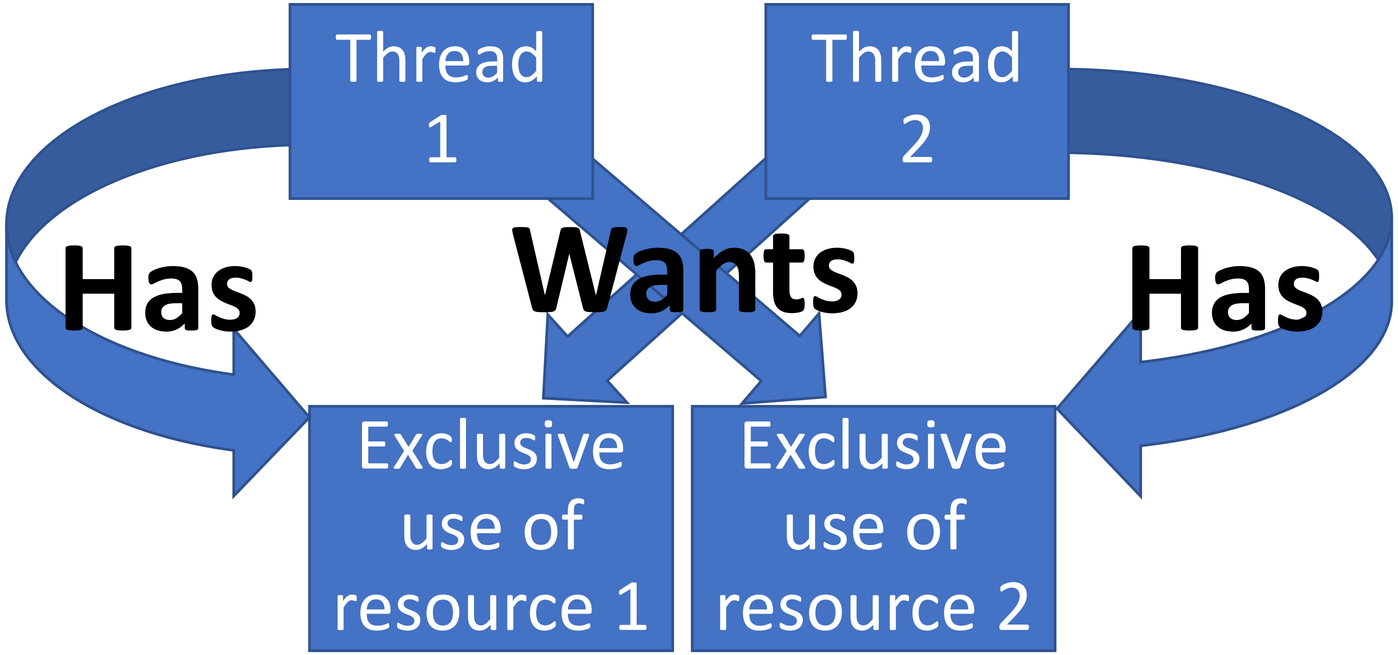
- Correct locking allows safe concurrent access on data (lock data, *not* code)
- Locks take advantage of **memory ordering.**

=> Memory ordering allows to make sure that all previous write are "visible" for any thread accessing the memory. It is some kind of 'commit'.

- One may wrap the "critical data" using a pattern like Herb Sutter monitor pattern (41:00)

Deadlock

- When some processes/threads are waiting each other at some execution point; for example, using locks:



RAII and locking

- C++ core guideline strongly encourage to use RAII for resources management.
- The STL provides RAII mechanisms:
  - std::lock_guard
  - std::unique_lock
  - std::scoped_lock
- boost provides reverse_lock; when need to temporarily release a lock (fe: call a blocking function)

Fishy locking (1)

```
void registerCb(Callback cb) {
    {
        std::lock_guard<std::mutex> lock(mtx);
        mCb = cb;
        mRun = true;
    }
    mFuture = std::async(std::launch::async, [this]() {
                                while (mRun) {
                                    auto payload = waitPayload();
                                    if (payload)
                                        mCb(*payload);

                                }
                    });
}
```

Fishy locking (2)

```
void MyApp::setContext(std::unique_ptr<Context> ctx) {
    std::lock_guard<std::mutex> _(this->mtx);
    this->context = std::move(ctx);
}

Context& MyApp::getContext() {
    std::lock_guard<std::mutex> _(this->mtx);
    return *this->context;
}
```

Conditional variable

- A conditional variable is a synchronization mechanism to wait for a particular shared states between many threads.
- Correct use involves 3 things:
  - Some data we want to monitor
  - A lock associated with this data
  - A condvar refering to the data and using the data's lock

Any missing premise is very likely a bug

« Even if the shared variable is atomic, it must be modified under the mutex in order to correctly publish the modification to the waiting thread."
https://en.cppreference.com/w/cpp/thread/condition_variable

Use predicate with your condvar

```
void wait(std::unique_lock<std::mutex>& lock );

template< class Predicate >
void wait( std::unique_lock<std::mutex>& lock, Predicate stop_waiting );
```

- When using a condvar, you are waiting for a state to be reached.

⇒You can pass a predicate to discriminate whether the event happened or not (usually a lambda)
  - this ease understanding what is waited
  - this protects against spurious wakeups

Consequences:

- Any data used to compute the predicate **MUST** be immutable or protected by THE mutex used by the condvar
- Any change done on data used in predicate should be followed by a notify()

Condition_variable (1) bad usage

```
bool called{ false };
std::mutex called_mutex;

std::mutex cv_mutex;
std::condition_variable cv;

// In one thread
{
    std::lock_guard<std::mutex> lock(called_mutex);
    called = true;
    cv.notify_all();
}

// In another
{
   std::unique_lock<std::mutex> lock(cv_mutex);
   cv.wait(lock, [this] { return called; });
}
```

Condition_variable (2) bad usage

```
#include <atomic>
#include <future>
#include <iostream>
#include <thread>
#include <condition_variable>

std::atomic<bool> stop{false};
std::condition_variable cv;
std::mutex mtx;

void job() {
    std::unique_lock<std::mutex> _(mtx);
    while (!stop)
        cv.wait(_, [] {
            bool res = stop.load();
            std::this_thread::sleep_for(std::chrono::seconds{3}); // Force race to happen
            return res;
        });
}

int main() {
   auto f = std::async(std::launch::async, job);

   cv.notify_one(); // simulate a previous wakeup for whatever reason

   std::this_thread::sleep_for(std::chrono::seconds{1}); // simulate some work

   stop = true;
   cv.notify_one();

   f.wait();
}
```

This does not work ([see it live]())

Hellgrind report

```
==2022421==
==2022421==  Lock at 0x40F240 was first observed
==2022421==    at 0x483DB78: mutex_lock_WRK (hg_intercepts.c:907)
==2022421==    by 0x4841A75: pthread_mutex_lock (hg_intercepts.c:923)
==2022421==    by 0x4028E1: __gthread_mutex_lock(pthread_mutex_t*) (gthr-default.h:749)
==2022421==    by 0x402C67: std::mutex::lock() (std_mutex.h:100)
==2022421==    by 0x404860: std::unique_lock<std::mutex>::lock() (unique_lock.h:141)
==2022421==    by 0x403DAE: std::unique_lock<std::mutex>::unique_lock(std::mutex&) (unique_lock.h:71)
==2022421==    by 0x402501: job() (sample2.cpp:12)
==2022421==    by 0x4073FD: void std::__invoke_impl<void, void (*)()>(std::__invoke_other, void (*&&)()) (invoke.h:60)
==2022421==    by 0x40735A: std::__invoke_result<void (*)()>::type std::__invoke<void (*)()>(void (*&&)()) (invoke.h:95)
==2022421==    by 0x4072A9: void std::thread::_Invoker<std::tuple<void (*)()> >::_M_invoke<0ul>(std::_Index_tuple<0ul>) (thread:244)
==2022421==    by 0x4071E5: std::thread::_Invoker<std::tuple<void (*)()> >::operator()() (thread:251)
==2022421==    by 0x406F0C: std::__future_base::_Task_setter<std::unique_ptr<std::__future_base::_Result<void>, std::__future_base::_Result_base::_Deleter>, std::thread::_Invoker<std::tuple<void (*)()> >, void>::operator()() const (future:1362)
==2022421==  Address 0x40f240 is 0 bytes inside data symbol "mtx"
==2022421==
==2022421== Possible data race during write of size 1 at 0x40F1E0 by thread #1
==2022421== Locks held: none
==2022421==    at 0x40355F: store (atomic_base.h:397)
==2022421==    by 0x40355F: std::__atomic_base<bool>::operator=(bool) (atomic_base.h:290)
==2022421==    by 0x402728: std::atomic<bool>::operator=(bool) (atomic:81)
==2022421==    by 0x4025B2: main (sample2.cpp:28)
==2022421==
==2022421== This conflicts with a previous read of size 1 by thread #2
==2022421== Locks held: 1, at address 0x40F240
==2022421==    at 0x40275D: load (atomic_base.h:419)
==2022421==    by 0x40275D: std::atomic<bool>::operator bool() const (atomic:88)
==2022421==    by 0x40250B: job() (sample2.cpp:13)
==2022421==    by 0x4073FD: void std::__invoke_impl<void, void (*)()>(std::__invoke_other, void (*&&)()) (invoke.h:60)
==2022421==    by 0x40735A: std::__invoke_result<void (*)()>::type std::__invoke<void (*)()>(void (*&&)()) (invoke.h:95)
==2022421==    by 0x4072A9: void std::thread::_Invoker<std::tuple<void (*)()> >::_M_invoke<0ul>(std::_Index_tuple<0ul>) (thread:244)
==2022421==    by 0x4071E5: std::thread::_Invoker<std::tuple<void (*)()> >::operator()() (thread:251)
==2022421==    by 0x406F0C: std::__future_base::_Task_setter<std::unique_ptr<std::__future_base::_Result<void>, std::__future_base::_Result_base::_Deleter>, std::thread::_Invoker<std::tuple<void (*)()> >, void>::operator()() const (future:1362)
==2022421==    by 0x406BF9: std::_Function_handler<std::unique_ptr<std::__future_base::_Result_base, std::__future_base::_Result_base::_Deleter> (), std::__future_base::_Task_setter<std::unique_ptr<std::__future_base::_Result<void>, std::__future_base::_Result_base::_Deleter>, std::thread::_Invoker<std::tuple<void (*)()> >, void> >::_M_invoke(std::_Any_data const&) (std_function.h:286)
==2022421==  Address 0x40f1e0 is 0 bytes inside data symbol "stop"
==2022421==
```

Condition_variable (3) bad usage

```
bool called{ false }, quit{ false };  // Now 2 booleans
std::mutex called_mutex;
std::mutex quit_mutex;
std::condition_variable cv;

// In one thread
std::unique_lock<std::mutex> lock(called_mutex);
called = true;
cv.notify_one();

// In another
std::unique_lock<std::mutex> lock(called_mutex);
cv.wait(lock, [this] { return called; });

// yet another
std::unique_lock<std::mutex> lock(quit_mutex);
cv.wait(lock, [this] { return quit; });
```

Condition_variable correct use

```
std::mutex mutex;
std::condition_variable cv;
bool called{ false };

// In one thread
std::unique_lock<std::mutex> lock(mutex);
called = true;
cv.notify_all();

// In another
std::unique_lock<std::mutex> lock(mutex);
cv.wait(lock, [this] { return called; });
```

Managing contention

- If you hold the lock only to access the protected data, contentions should be low.

⇒**Never call a (potentially) blocking function with a lock held (especially I/O functions)**. Work on a copy of the data if you need this call.

- Make data used is different locations use different locks
- Keep in mind that operators may allocate memory thus may be blocking. (fe operator=)
- If you think you have contention, **measure it**. (maybe using things like this)
- If you know you have a **many read/few write** pattern you may use std::**shared_mutex** (c++17)

Shared_mutex example

```
class ThreadSafeCounter {
 public:
  ThreadSafeCounter() = default;

  // Multiple threads/readers can read the counter's value at the same time.
  unsigned int get() const {
    std::shared_lock lock(mutex_);
    return value_;
  }

  // Only one thread/writer can increment/write the counter's value.
  void increment() {
    std::unique_lock lock(mutex_);
    value_++;
  }

 private:
  mutable std::shared_mutex mutex_;
  unsigned int value_ = 0;
};
// https://en.cppreference.com/w/cpp/thread/shared_mutex
```

Challenge your code

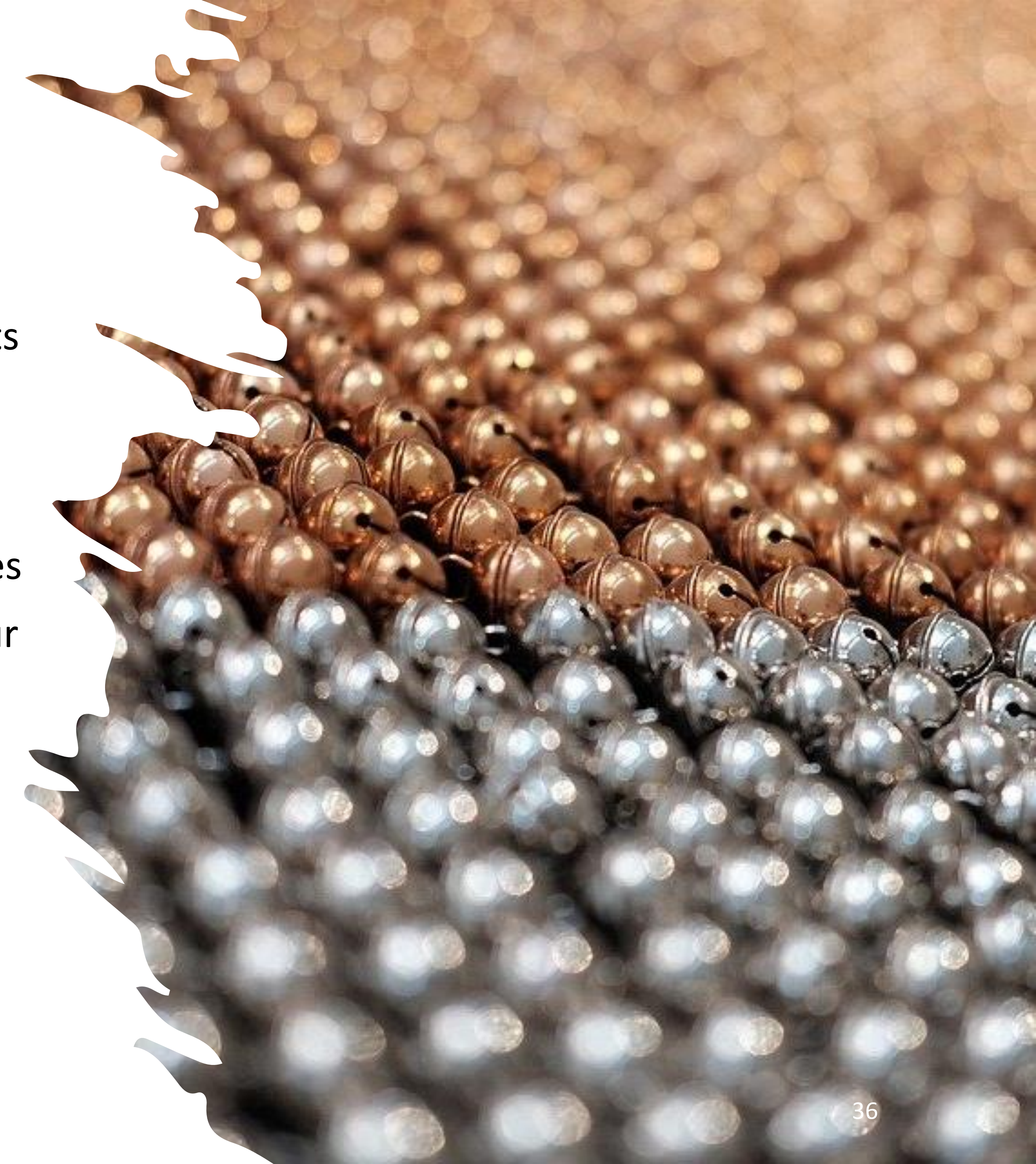
Imagine that someone can insert sleep(10s) before every single line of your software (and it should not crash nor deadlock)

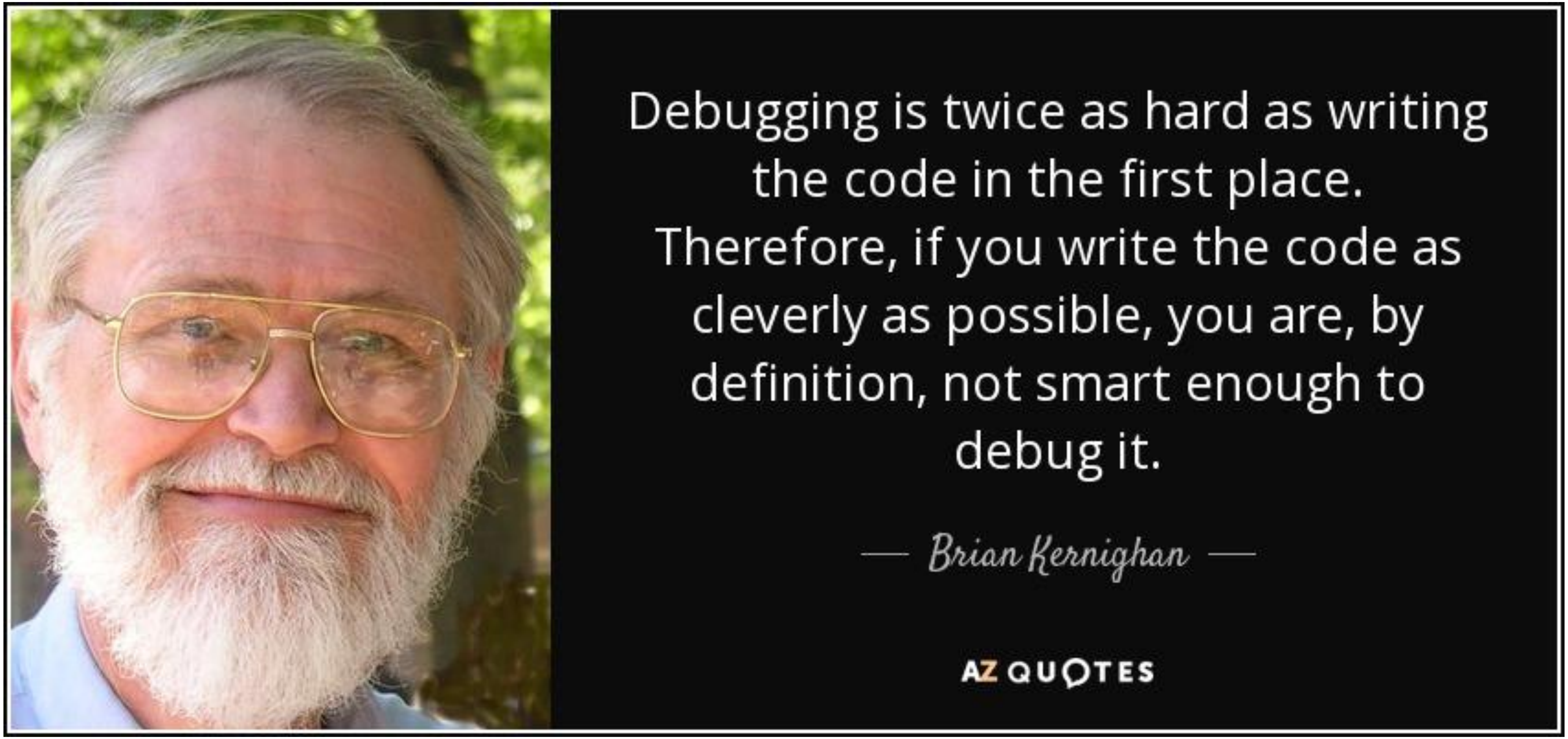
Between to locked-sections, the protected data may have changed 100 times.

Lock free code (using only atomics) is **really** hard to write and debug (99% chances the code is wrong)

Some takeaways

- Prefer future and async whenever it is possible.
- The more threads you have, the more complex it gets
- “Hide” your threads in objects that provide a thread safe interface
- Data races almost always lead to data corruption, including memory corruption that can lead to crashes
- Use tools to detect races efficiently, add them in your CI for early problem detection.
- Use locks on data, not code => any data accessed from many threads should have an identified lock, and this lock should always be taken on access (R or W)
- Keep you lock private with your data inside synchronized objects




Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.
— Brian Kernighan —
AZ QUOTES

Questions?

Thank you!

Any feedback is greatly welcome (sebastien.gonzalve *at* aliceadsl.fr)

Backup slides

Synchronization and data sharing in modern C++

Get reliable multithreaded production code.

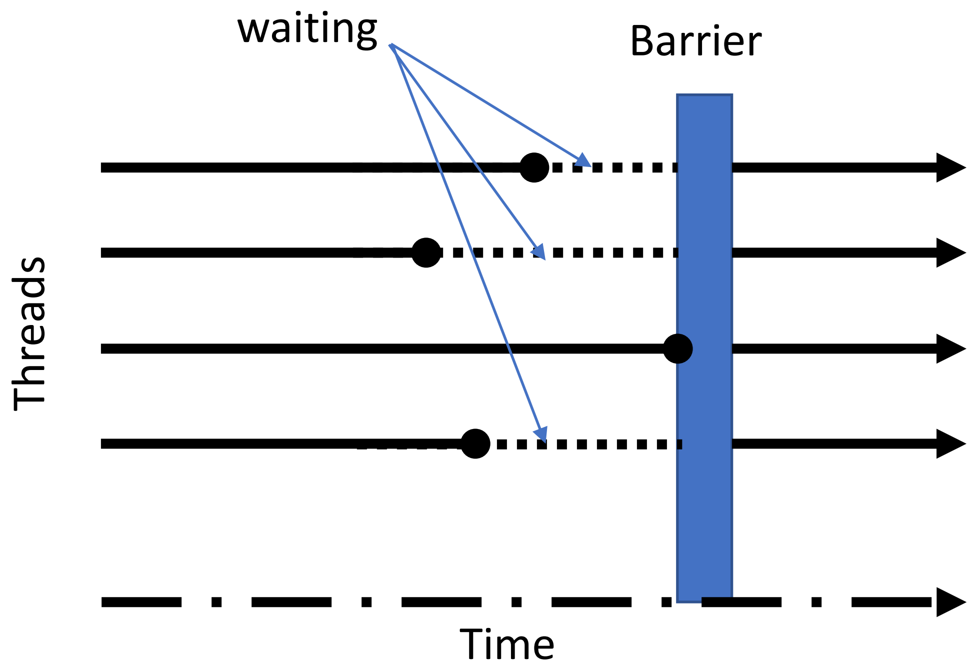
Example of bad atomic usage

```
std::atomic<int>  ai{3}; // My atomic int


int main() {
   SomeRandomThreadModifyingAi t();

   // Careful, next test is wrong!
   if (ai == 3)
      ai = 5;
}
```

```
std::atomic<int>  ai{3}; // My atomic int

int main() {
   SomeRandomThreadModifyingAi t();

   int  tst_val= 4,  new_val= 5;
   bool exchanged = false;

   // tst_val != ai   ==>  tst_val is modified
   exchanged = ai.compare_exchange_strong( tst_val, new_val );
   if (exchanged) {
     // may happen IIF value was 4 and set was done; then ai is 5
   }
}
```

Barriers/latches

- New to C++20
- Allow to create points where threads wait for each other
- Don’t confuse with memory barriers
- When all (or given number) threads reach the barrier/latch, the barrier release and thread can continue their execution.



https://godbolt.org/z/6faMer49E

```cpp
#include <condition_variable>
#include <mutex>
#include <future>
#include <list>
#include <iostream>


using Task = std::packaged_task<void(void)>;

struct Worker {
 void submit(Task t) {
   {
     std::lock_guard _(_mtx);
     _pendingTasks.emplace_back(std::move(t));
   }
   _cv.notify_one();
 }


 ~Worker() {
     submit(Task([this] { _running = false; }));
 }


 void run() {
    std::unique_lock lock(_mtx);
    while (_running || !_pendingTasks.empty()) {
      _cv.wait(lock, [&] { return !_pendingTasks.empty(); });
      if (_pendingTasks.empty())
        continue;
      auto task = std::move(_pendingTasks.front());
      _pendingTasks.pop_front();
      // release lock while processing message
      lock.unlock();
      task();
      lock.lock();
    }
 }
 private:
   std::mutex _mtx;
   std::list<Task> _pendingTasks;
   bool _running{true};
   std::condition_variable _cv;
   std::future<void> messageProcessor = std::async(std::launch::async, &Worker::run, this);
};


int main() {
  Worker w;
  for (auto i = 0u; i < 10; i++) {
     auto t = Task([i] {
                    std::cout << "Performing task " << i << '\n';
                   });

     auto f = t.get_future();
     w.submit(std::move(t));
     if (i == 9) {
         std::cout << "Waiting...";
         f.wait();
         std::cout << "done\n";
     }
  }
}
```

Shared_future

- May be used to wait for an event from several threads/contexts

Need an example on reverse lock in backup

Condition_variable (bad?) use

```
std::mutex mutex;
std::condition_variable cv;
std::atomic_bool called{ false };

// In one thread
called = true;
cv.notify_all();

// In another
std::unique_lock<std::mutex> lock(mutex);
cv.wait(lock, [this] { return called; });
```

- Does this work?
- Why?

Spurious wakeups

std::condition_variable

```
template< class Rep, class Period >
std::cv_status wait_for( std::unique_lock<std::mutex>& lock,
                         const std::chrono::duration<Rep, Period>& rel_time);
```

Atomically releases lock, blocks the current executing thread, **and** adds it to the list of threads waiting on ***this.** The thread will be unblocked when notify_all() **or** notify_one() is executed, **or** when the relative timeout rel_time expires. **It may also be unblocked spuriously**. When unblocked, regardless of the reason, lock is reacquired **and** wait_for() exits. If **this** function exits via exception, lock is also reacquired.

Return value

1) std::cv_status::timeout if the relative timeout specified by rel_time expired, **std::cv_status::no_timeout otherwise**.

semaphores

- since c++20 **std::counting_semaphore, std::binary_semaphore**

Each post()/up()/release() increase semaphore’s internal counter.

Each successful wait()/down()/acquire() decreases the counter; blocks if counter was 0.