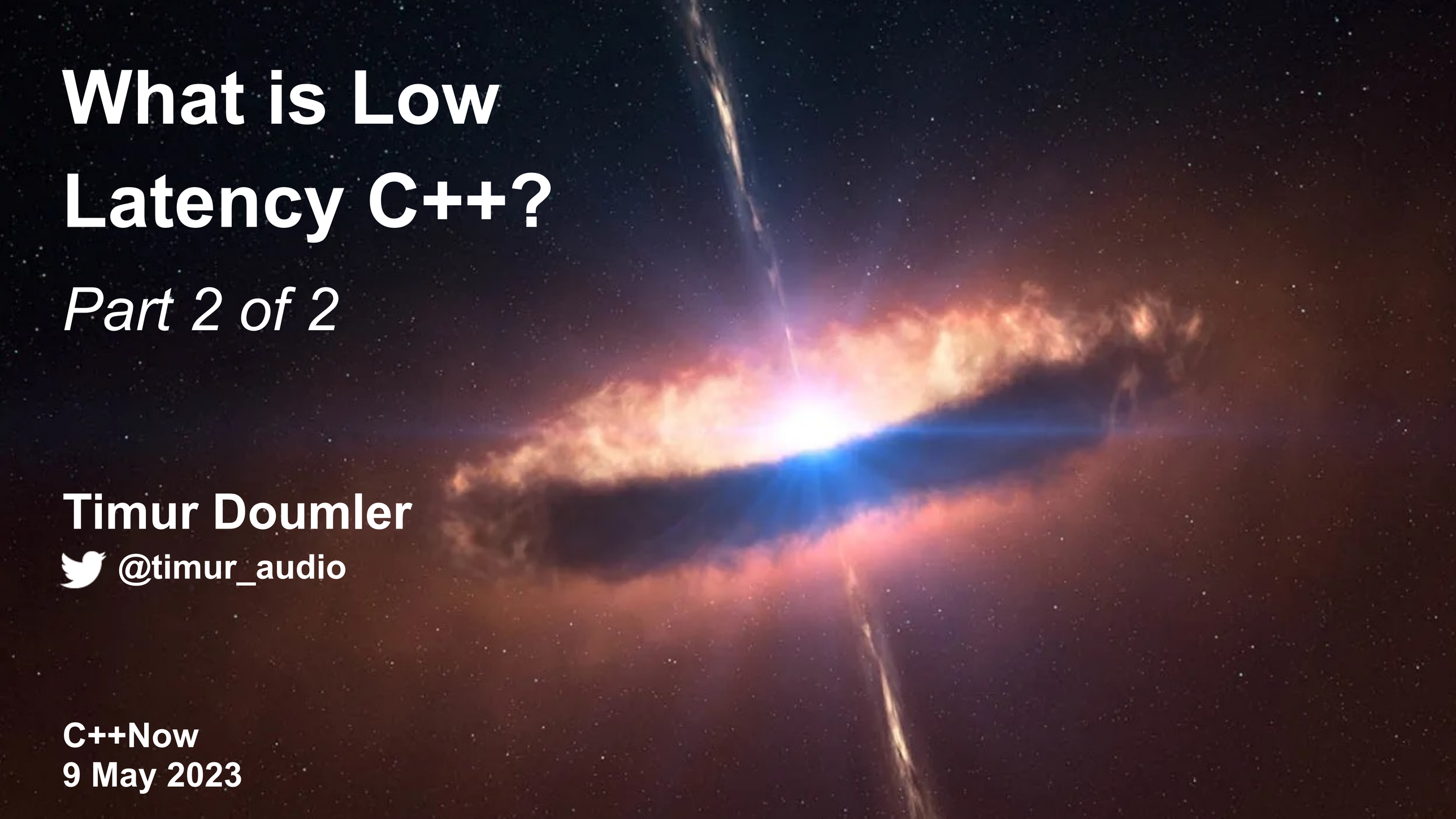
What is Low Latency C++?

Part 2 of 2



Timur Doumler

 @timur_audio

C++Now
9 May 2023

C++ techniques for low-latency programming

- Two categories:
  - Efficient programming
  - **Programming for deterministic execution time**

- Most crucial thing: measuring!

What not to do in the hot path:

- Dynamic memory allocations/deallocations
- blocking the thread
- I/O
- exceptions
- context switches / mode switches (user/kernel space)
- syscalls
- calling into unknown code
- loops without definite bounds
- algorithms > $O(1)$, or with no statically known upper bound on N

What not to do in the hot path:

- **Dynamic memory allocations/deallocations**
- blocking the thread
- I/O
- exceptions
- context switches / mode switches (user/kernel space)
- syscalls
- calling into unknown code
- loops without definite bounds
- algorithms > *O*(1), or with no statically known upper bound on *N*

Avoiding allocations

- Do not use data types that allocate
- Do not use algorithms that allocate
- Do not use data structures that allocate

Avoiding allocations

- **Do not use data types that allocate**
- Do not use algorithms that allocate
- Do not use data structures that allocate

Avoiding allocations

- Do not use data types that allocate
- **Do not use algorithms that allocate**
- Do not use data structures that allocate

#include <algorithm>

#include <algorithm>

- Which of them allocate memory?
- The standard doesn't say
- But most don't

#include <algorithm>

Let `comp` be `less{}` and `proj` be `identity{}` for the overloads with no parameters by those names.

Preconditions: For the overloads in namespace `std`, `RandomAccessIterator` meets the *Cpp17ValueSwappable* requirements ([swappable.requirements]) and the type of `*first` meets the *Cpp17MoveConstructible* (Table 28) and *Cpp17MoveAssignable* (Table 30) requirements.

Effects: Sorts the elements in the range [`first, last`) with respect to `comp` and `proj`.

Returns: `last` for the overloads in namespace `ranges`.

Complexity: Let N be `last - first`. If enough extra memory is available, $N \log(N)$ comparisons. Otherwise, at most $N \log^2(N)$ comparisons. In either case, twice as many projections as the number of comparisons.

Remarks: Stable ([algorithm.stable]).

```
// allocating standard algorithms

std::stable_sort
std::stable_partition
std::inplace_merge
```

Avoiding allocations

- Do not use data types that allocate
- Do not use algorithms that allocate
- **Do not use data structures that allocate**

```
// do use:

std::array
std::pair
std::tuple
std::optional
std::variant
```

```
// do use:

std::array
std::pair
std::tuple
std::optional
std::variant
```

```
// do not use:

std::any
std::function
std::vector, std::deque, std::list...
```

Custom allocators

google / tcmalloc Public

Watch 85 | Fork 327 | Star 3.1k

Code | Issues 28 | Pull requests 7 | Zenhub | Actions | Projects | Security | Insights

master | 3 branches | 0 tags | Go to file | Add file | Code

v-gogte and Copybara-Service Moves Print functions to sharde... ... 52ba74a 10 hours ago 1,126 commits

ci	Allow unlimited stdout bytes for gcc tests on CI.	15 days ago
docs	Release the lifetime (deallocation) profiler.	7 days ago
tcmalloc	Moves Print functions to sharded transfer cache.	10 hours ago
.bazelrc	Wire up fuzz tests to a driver.	2 years ago
.gitignore	Added a gitignore	3 years ago
CONTRIBUTING.md	Initial commit	3 years ago
LICENSE	Initial commit	3 years ago
README.md	Internal changes	6 months ago
WORKSPACE	Add fuzzer for SizeMap to test size class validation logic.	21 hours ago

README.md

TCMalloc

This repository contains the TCMalloc C++ code.

TCMalloc is Google's customized implementation of C's `malloc()` and C++'s `operator new` used for memory allocation within our C and C++ code. TCMalloc is a fast, multi-threaded malloc implementation.

About

No description, website, or topics provided.

Readme
Apache-2.0 license
Code of conduct
3.1k stars
85 watching
327 forks

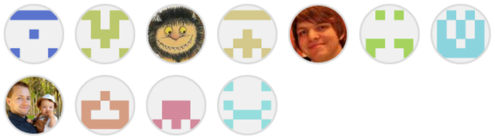
Releases

No releases published

Packages

No packages published

Contributors 43



+ 32 contributors

General-purpose allocators

- tcmalloc, rpmalloc…

  are not good for ultra low latency and real-time:

  - minimising average cost, not worst case
  - not constant time
  - multithreaded (locks)
  - eventually go to OS to request dynamic memory

Custom allocators

- Preallocate everything

Custom allocators

- Preallocate everything
  - Monotonic allocators
    - `std::pmr::monotonic_buffer_resource`
  - Pool allocators
    - `std::pmr::unsynchronised_pool_resource`
  - Frame allocators
  - Arena allocators
  - Double-ended allocators for big/small buffers

Custom allocators

- Preallocate everything
  - Monotonic allocators
    - `std::pmr::monotonic_buffer_resource`
  - Pool allocators
    - `std::pmr::unsynchronised_pool_resource`
  - Frame allocators
  - Arena allocators
  - Double-ended allocators for big/small buffers
- Lock-free allocators (require helper thread)

```
std::array<float, 1024> stack_memory;

void process(buffer& b)
{
    std::pmr::monotonic_buffer_resource monotonic_buffer(
        stack_memory.data(),
        stack_memory.size(),
        std::pmr::null_memory_resource());

    using allocator_t = std::pmr::polymorphic_allocator<float>;
    allocator_t allocator(&monotonic_buffer);

    std::pmr::vector<float> my_vector(b.size(), 0.0f, allocator);
}
```

```
std::array<float, 1024> stack_memory;

void process(buffer& b)
{
    std::pmr::monotonic_buffer_resource monotonic_buffer(
        stack_memory.data(),
        stack_memory.size(),
        std::pmr::null_memory_resource());

    using allocator_t = std::pmr::polymorphic_allocator<float>;
    allocator_t allocator(&monotonic_buffer);

    std::pmr::vector<float> my_vector(b.size(), 0.0f, allocator);
}
```

```
std::array<float, 1024> stack_memory;

void process(buffer& b)
{
    std::pmr::monotonic_buffer_resource monotonic_buffer(
        stack_memory.data(),
        stack_memory.size(),
        std::pmr::null_memory_resource());

    using allocator_t = std::pmr::polymorphic_allocator<float>;
    allocator_t allocator(&monotonic_buffer);

    std::pmr::vector<float> my_vector(b.size(), 0.0f, allocator);
}
```

```
// Alternatives: fixed-capacity versions
// All elements are stored inside the object itself (on the stack)

static_vector<T, Capacity>
```

```
// Alternatives: fixed-capacity versions
// All elements are stored inside the object itself (on the stack)

static_vector<T, Capacity>
```

- Gonzalo Brito: P0843 (static_vector proposal)
- David Stone: *"Faster, Easier, Simpler Vectors" (CppCon 2021)*
- David Stone: *"Implementing static_vector: How Hard Could It Be? (CppCon 2021)*

```
// Alternatives: fixed-capacity versions
// All elements are stored inside the object itself (on the stack)

static_vector<T, Capacity>
inplace_function<R(Args...), Size, Alignment>
inplace_any<Size, Alignment>
```

Non-allocating utilities: design tradeoffs

Non-allocating utilities: design tradeoffs

- std::variant is always non-allocating
- but boost::variant is not

```
struct ThrowsOnConstruction
{
    ThrowsOnConstruction() { throw std::exception(); }
};

void variantTest()
{
    std::variant<int, ThrowsOnConstruction> var = 42; // var now holds an int

    try
    {
        var.emplace<ThrowsOnConstruction>(); // ctor fails!
    }
    catch (std::exception&)
    {
        // what does var hold now?
    }
}
```

```
struct ThrowsOnConstruction
{
    ThrowsOnConstruction() { throw std::exception(); }
};

void variantTest()
{
    std::variant<int, ThrowsOnConstruction> var = 42; // var now holds an int

    try
    {
        var.emplace<ThrowsOnConstruction>(); // ctor fails!
    }
    catch (std::exception&)
    {
        // what does var hold now?
    }
}
```

```cpp
struct ThrowsOnConstruction
{
    ThrowsOnConstruction() { throw std::exception(); }
};

void variantTest()
{
    std::variant<int, ThrowsOnConstruction> var = 42; // var now holds an int

    try
    {
        var.emplace<ThrowsOnConstruction>(); // ctor fails!
    }
    catch (std::exception&)
    {
        // what does var hold now?
    }
}
```

```
struct ThrowsOnConstruction
{
    ThrowsOnConstruction() { throw std::exception(); }
};

void variantTest()
{
    std::variant<int, ThrowsOnConstruction> var = 42; // var now holds an int

    try
    {
        var.emplace<ThrowsOnConstruction>();
    }
    catch (std::exception&)
    {
        // Boost.Variant: still holds an int.
    }
}
```

```
struct ThrowsOnConstruction
{
    ThrowsOnConstruction() { throw std::exception(); }
};

void variantTest()
{
    std::variant<int, ThrowsOnConstruction> var = 42; // var now holds an int

    try
    {                                                 // Boost.Variant:
        var.emplace<ThrowsOnConstruction>(); // temporary heap backup :(
    }
    catch (std::exception&)
    {
        // Boost.Variant: still holds an int.
    }
}
```

```
struct ThrowsOnConstruction
{
    ThrowsOnConstruction() { throw std::exception(); }
};

void variantTest()
{
    std::variant<int, ThrowsOnConstruction> var = 42; // var now holds an int

    try
    {                                                 // std::variant:
        var.emplace<ThrowsOnConstruction>(); // no heap allocation :)
    }
    catch (std::exception&)
    {
        // var.valueless_by_exception() == true
    }
}
```

Lambdas

Lambdas

```
void process(buffer& b)
{
    std::array a = {1, 1, 2, 3, 5, 8, 13};

    // this will never allocate:
    auto f = [=] {
        return std::accumulate (a.begin(), a.end(), 0);
    };
}
```

Lambdas

```
void process(buffer& b)
{
    std::array a = {1, 1, 2, 3, 5, 8, 13};

    // this will never allocate:
    auto f = [=] {
        return std::accumulate (a.begin(), a.end(), 0);
    };

    do_something(b, f);
}
```

Coroutines

Coroutines

```
// generates the sequence 0, 1, 2, ...
generator<int> f()
{
    int i = 0;
    while (true)
        co_yield i++;
}
```

Coroutines

```
// generates the sequence 0, 1, 2, ...
generator<int> f()
{
    int i = 0;
    while (true)
        co_yield i++;
}

void process(buffer& b)
{
    auto gen = f();
    do_something(b, gen);
}
```

Coroutines

```
// generates the sequence 0, 1, 2, ...
generator<int> f()
{
    int i = 0;
    while (true)
        co_yield I++;
}

void process(buffer& b)
{
    auto gen = f();   // may perform dynamic allocation :(
    do_something(b, gen);
}
```

Options

- rely on the optimiser?

  → *Eyal Zedaka: “Using Coroutines to Implement C++ Exceptions for Freestanding Environments” (CppCon 2021)*
- create and suspend coroutine frame upfront
- write your own promise type, defining its own custom `operator new` and `operator delete`
- Don’t use coroutines in a low-latency/real-time scenario

What not to do in the hot path:

- Dynamic memory allocations/deallocations
- **blocking the thread**
- I/O
- exceptions
- context switches / mode switches (user/kernel space)
- syscalls
- calling into unknown code
- loops without definite bounds
- algorithms > *O*(1), or with no statically known upper bound on *N*

Concurrency

Terminology

- atomic == indivisible, race-free
- lock-free == at least one thread is guaranteed to make progress
- wait-free == all threads are guaranteed to make progress

Terminology

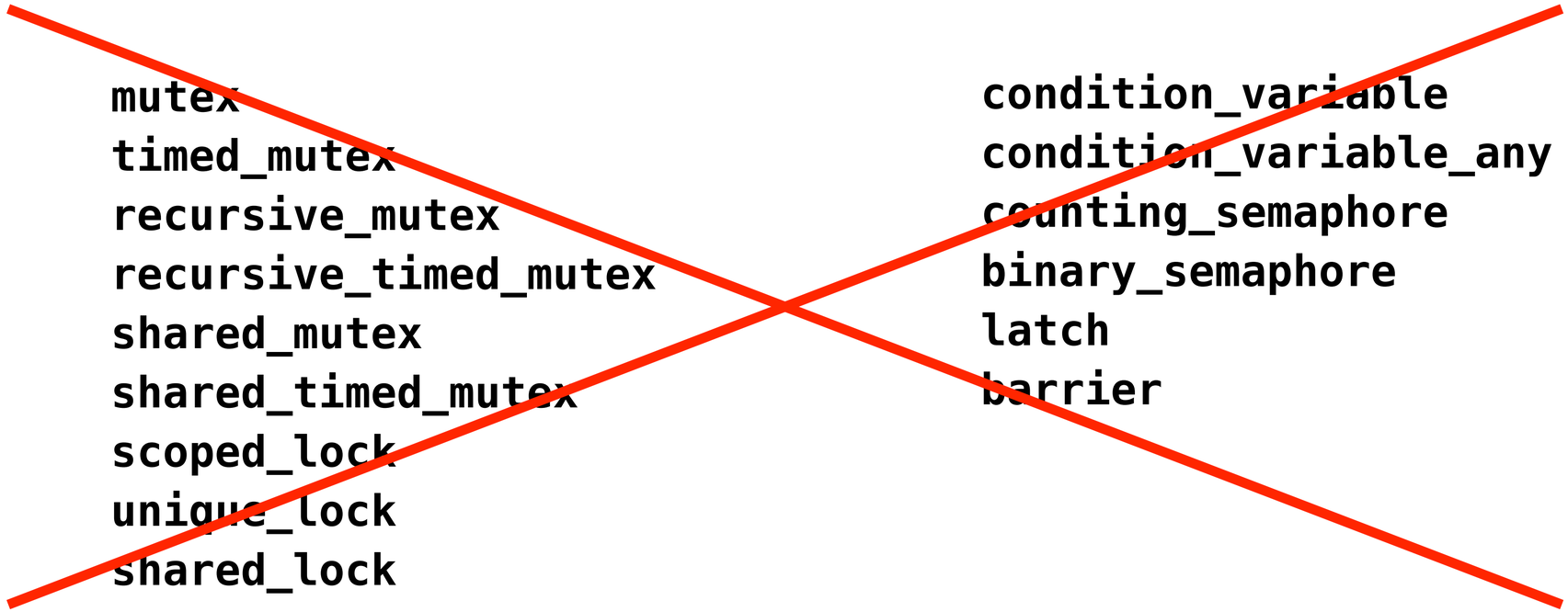
- atomic == indivisible, race-free
- lock-free == at least one thread is guaranteed to make progress
- **wait-free == all threads are guaranteed to make progress**

Concurrency

```
mutex
timed_mutex
recursive_mutex
recursive_timed_mutex
shared_mutex
shared_timed_mutex
scoped_lock
unique_lock
shared_lock
```

```
condition_variable
condition_variable_any
counting_semaphore
binary_semaphore
latch
barrier
```

Concurrency



Wait-free concurrency

- Can't block hot path (no mutexes, no spinning)
- Can't do any syscalls
- Can't do anything with unbounded/non-deterministic runtime

Wait-free concurrency

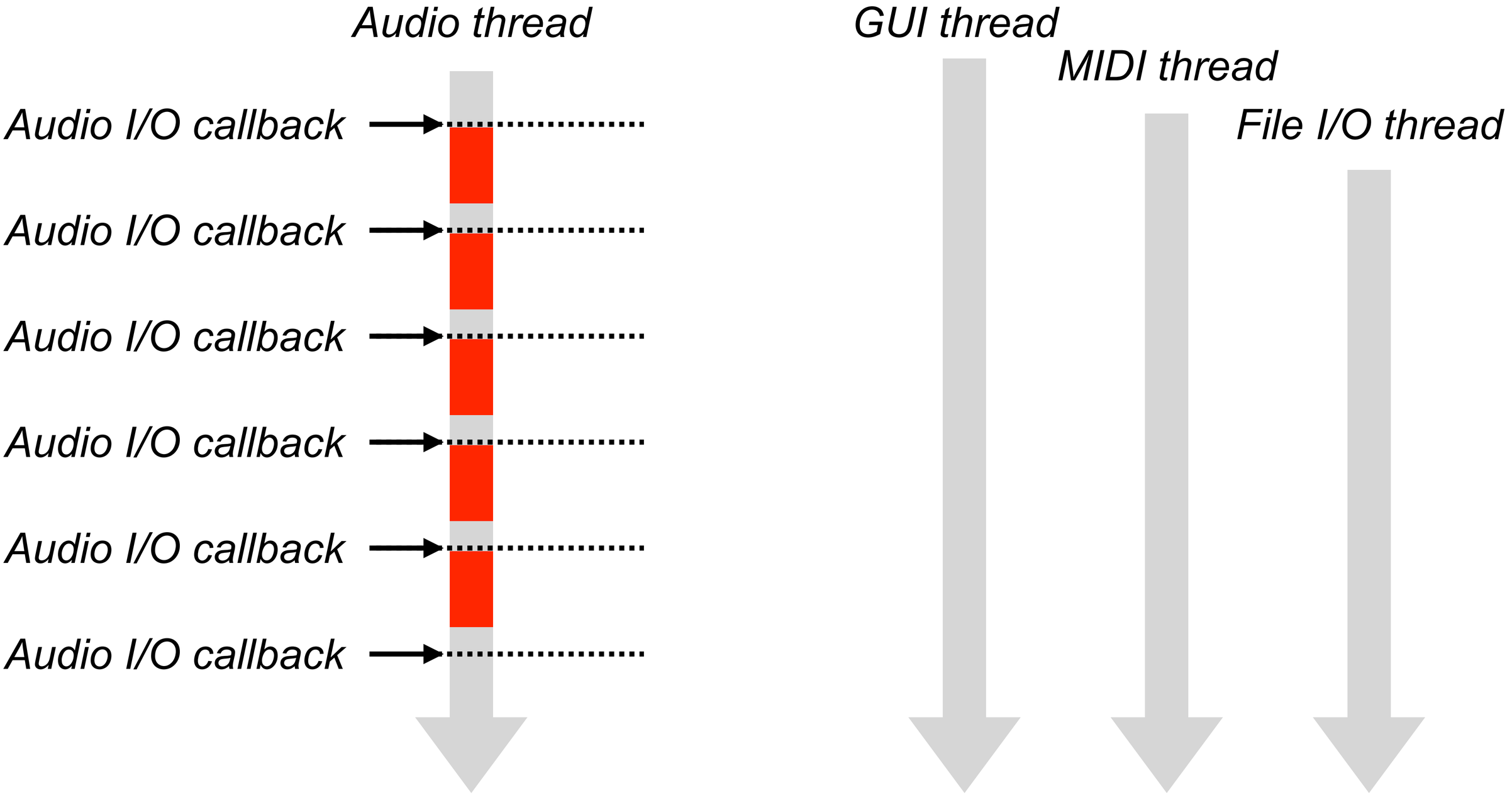
- Can't block hot path (no mutexes, no spinning)
- Can't do any syscalls
- Can't do anything with unbounded/non-deterministic runtime
- Only available synchronisation mechanism:
  - `std::atomic`
  - `std::atomic_ref` (since C++20)
  - never on hot path:
    - spin on `std::atomic`, `compare_exchange` in a loop, etc.
    - use `std::atomic<T>::wait/notify_one/notify_all`
  - always:
    - `static_assert(std::atomic<T>::is_always_lock_free);`

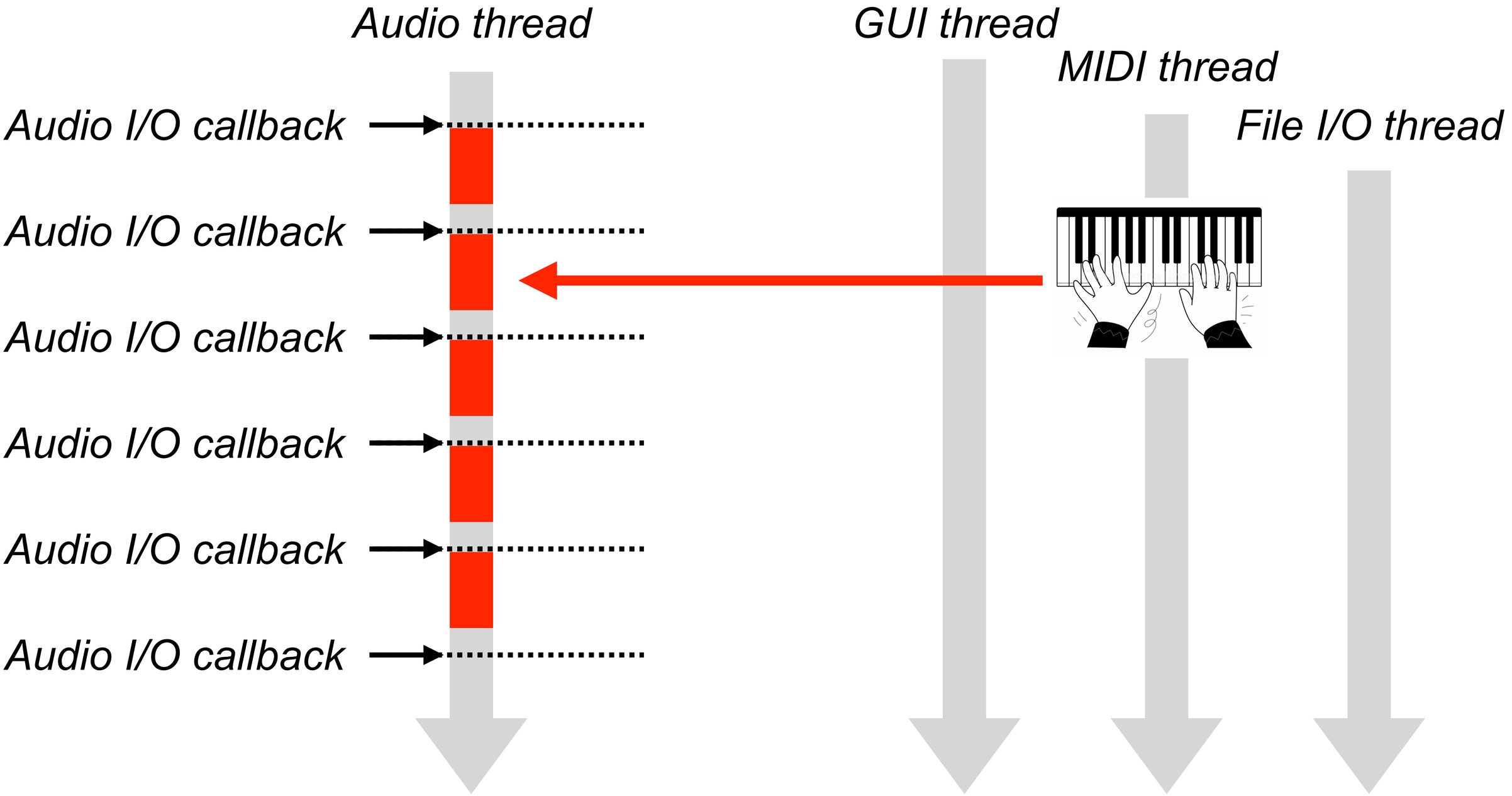
Wait-free concurrency

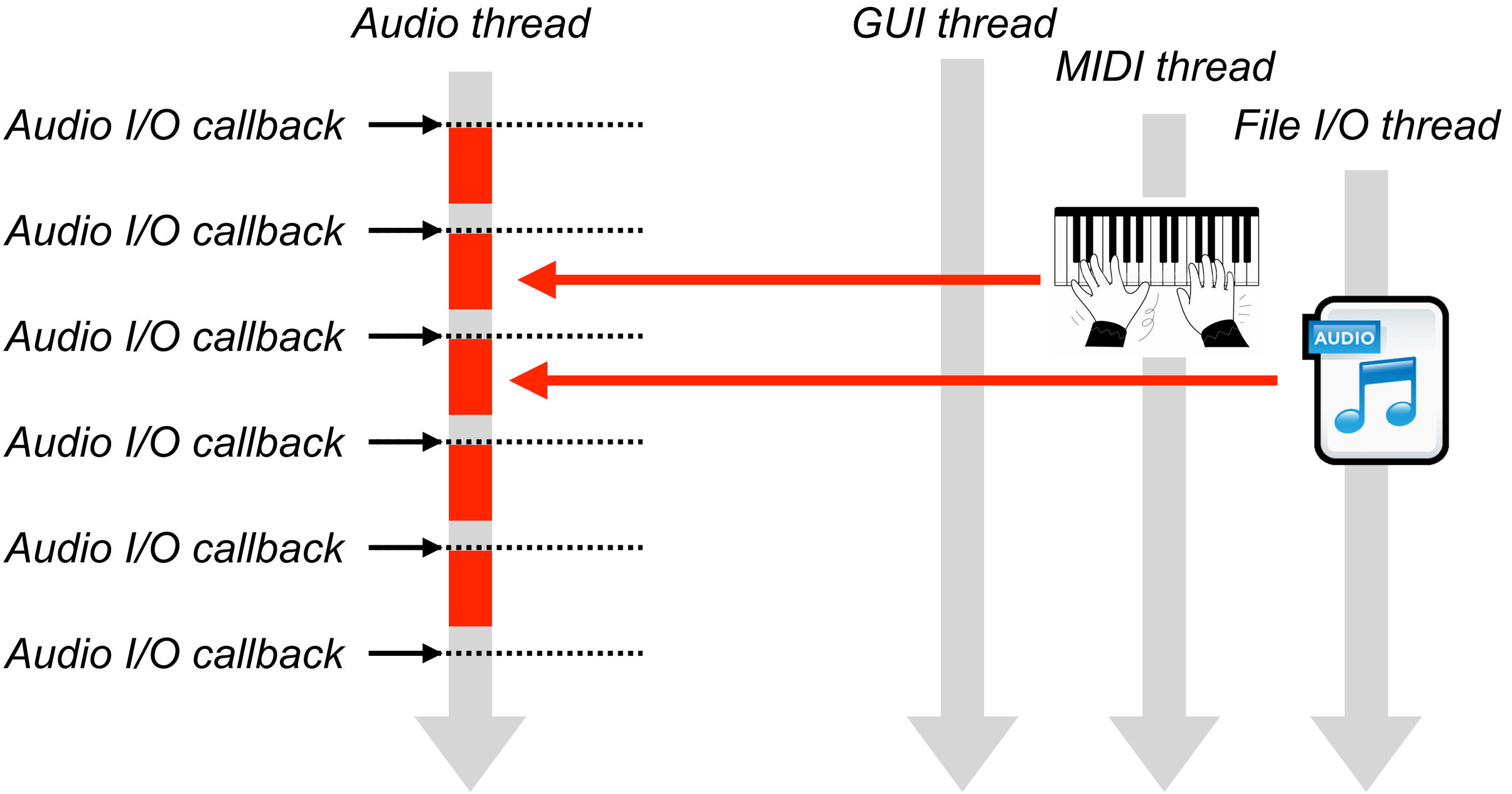
- Passing data to/from "hot path" thread
  - wait-free queue
- Sharing data with "hot path" thread
  - Hot path reads
    - try_lock
    - "spin-on-write"
    - RCU
  - Hot path writes
    - Double-buffering
    - SeqLock

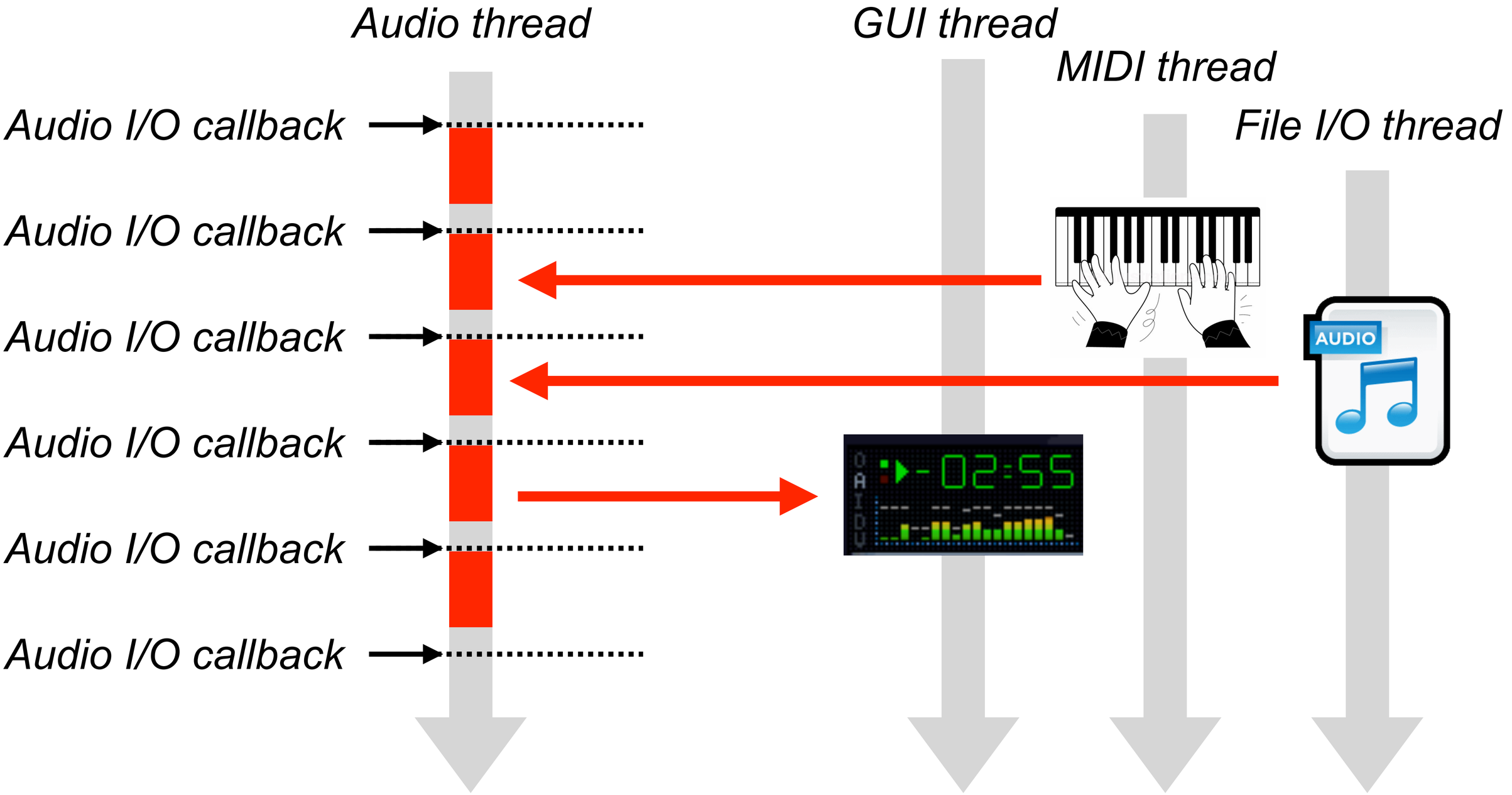
Wait-free concurrency

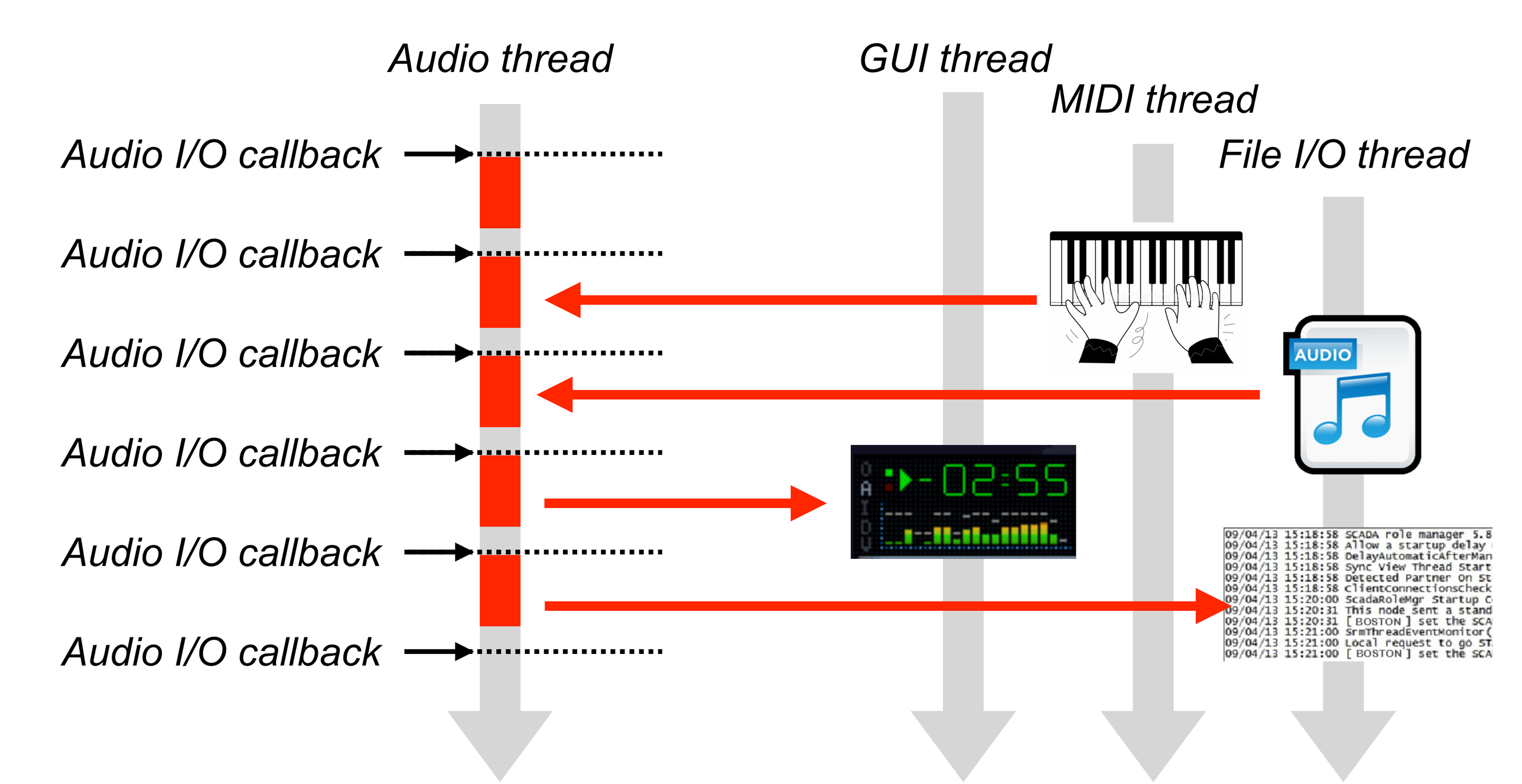
- **Passing data to/from "hot path" thread**
  - wait-free queue
- Sharing data with "hot path" thread
  - Hot path reads
    - Spinlock try_lock
    - "spin-on-write"
    - RCU
  - Hot path writes
    - Double-buffering
    - SeqLock


Audio thread
GUI thread
MIDI thread
File I/O thread
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio I/O callback


Audio thread
GUI thread
MIDI thread
File I/O thread
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio I/O callback

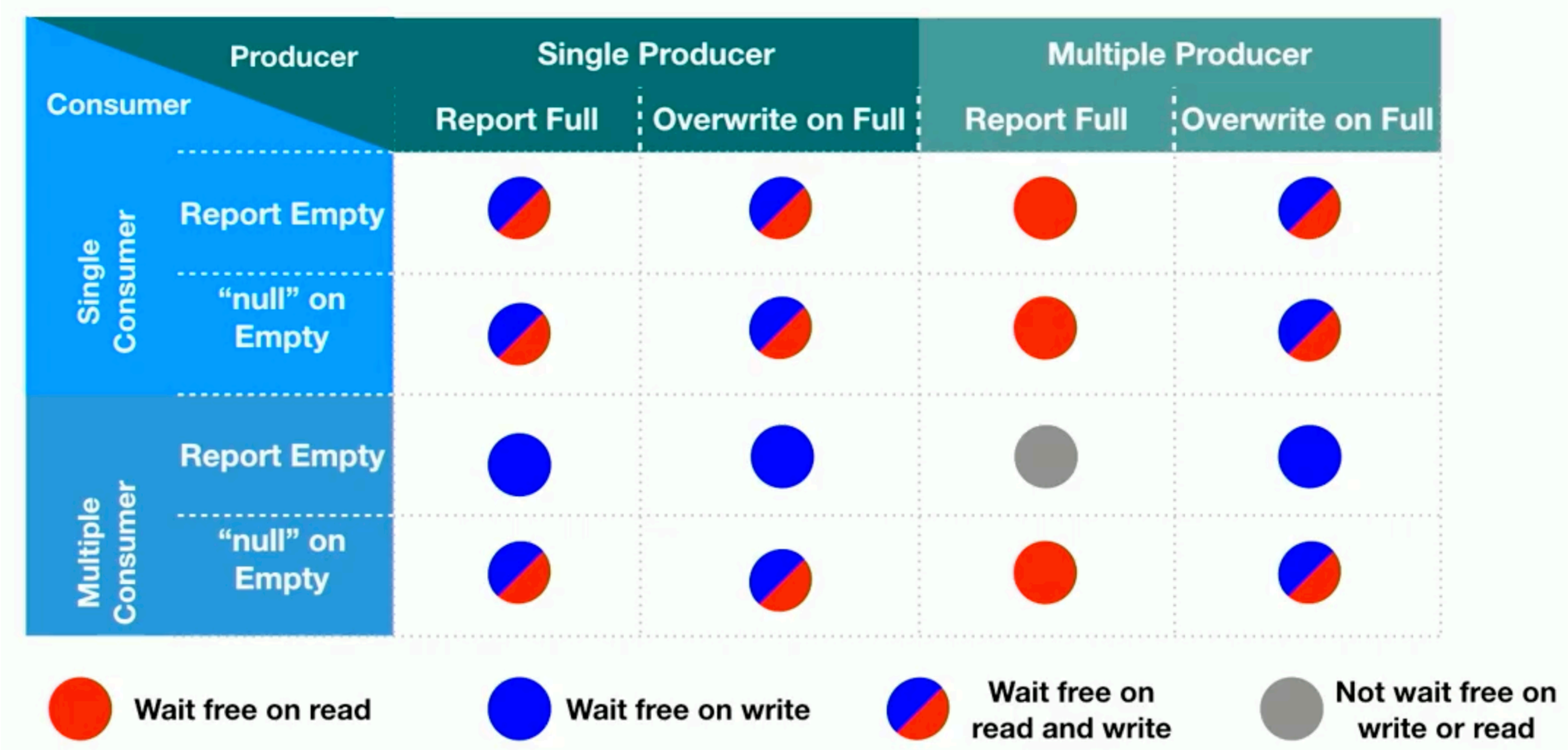

Audio thread
GUI thread
MIDI thread
File I/O thread
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio I/O callback
AUDIO


Audio thread
GUI thread
MIDI thread
File I/O thread
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio I/O callback
AUDIO


Audio thread
GUI thread
MIDI thread
File I/O thread
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio I/O callback
AUDIO
-02:55
09/04/13 15:18:58 SCADA role manager 5.8
09/04/13 15:18:58 Allow a startup delay
09/04/13 15:18:58 DelayAutomaticAfterMan
09/04/13 15:18:58 Sync View Thread Start
09/04/13 15:18:58 Detected Partner On St
09/04/13 15:18:58 ClientConnectionsCheck
09/04/13 15:20:00 ScadaRoleMgr Startup C
09/04/13 15:20:31 This node sent a stand
09/04/13 15:20:31 [BOSTON] set the SCA
09/04/13 15:21:00 SrmThreadEventMonitor(
09/04/13 15:21:00 Local request to go ST
09/04/13 15:21:00 [BOSTON] set the SCA

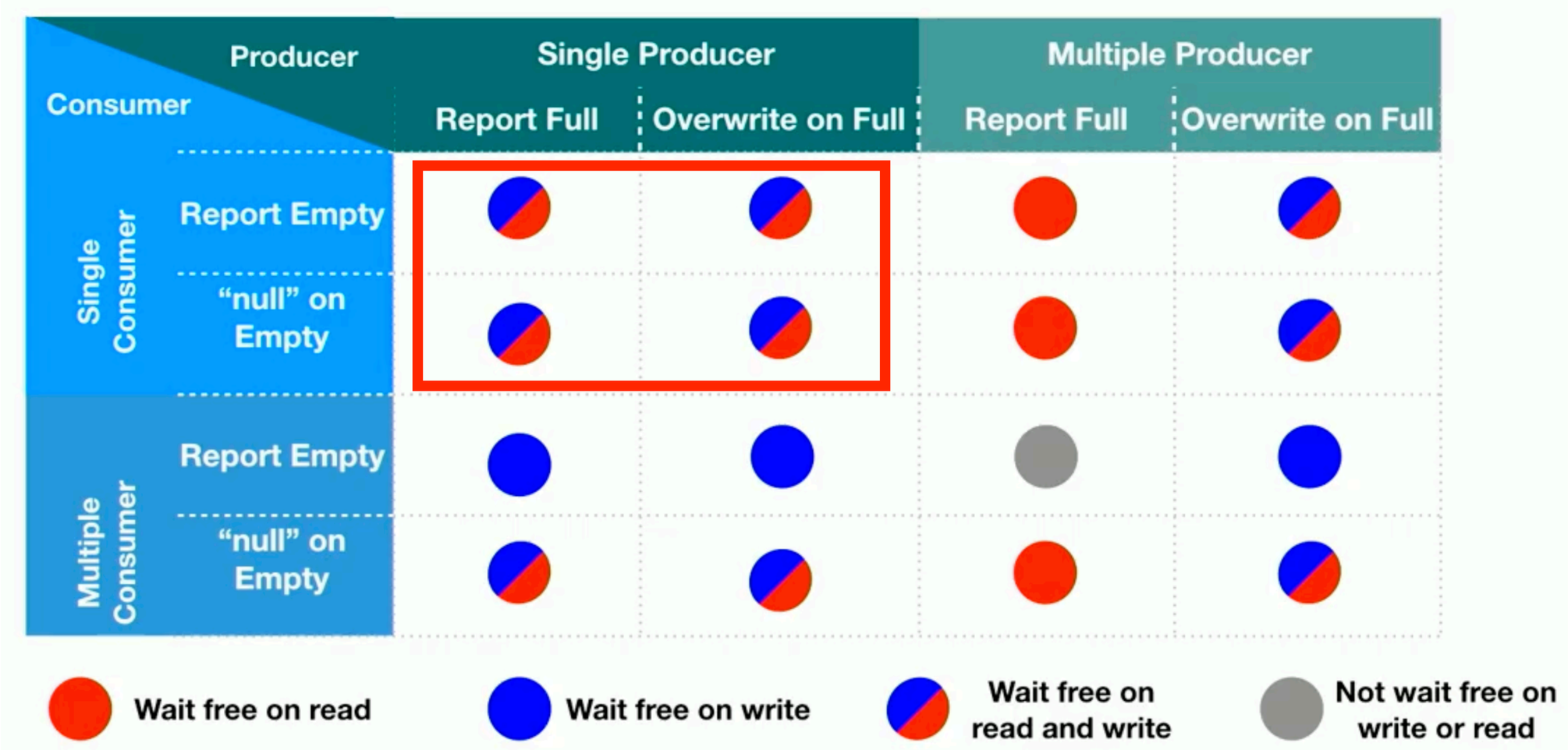
The cost of various lock-free queues

(from "Real-time 101" by Fabian Renn-Giles & Dave Rowland)

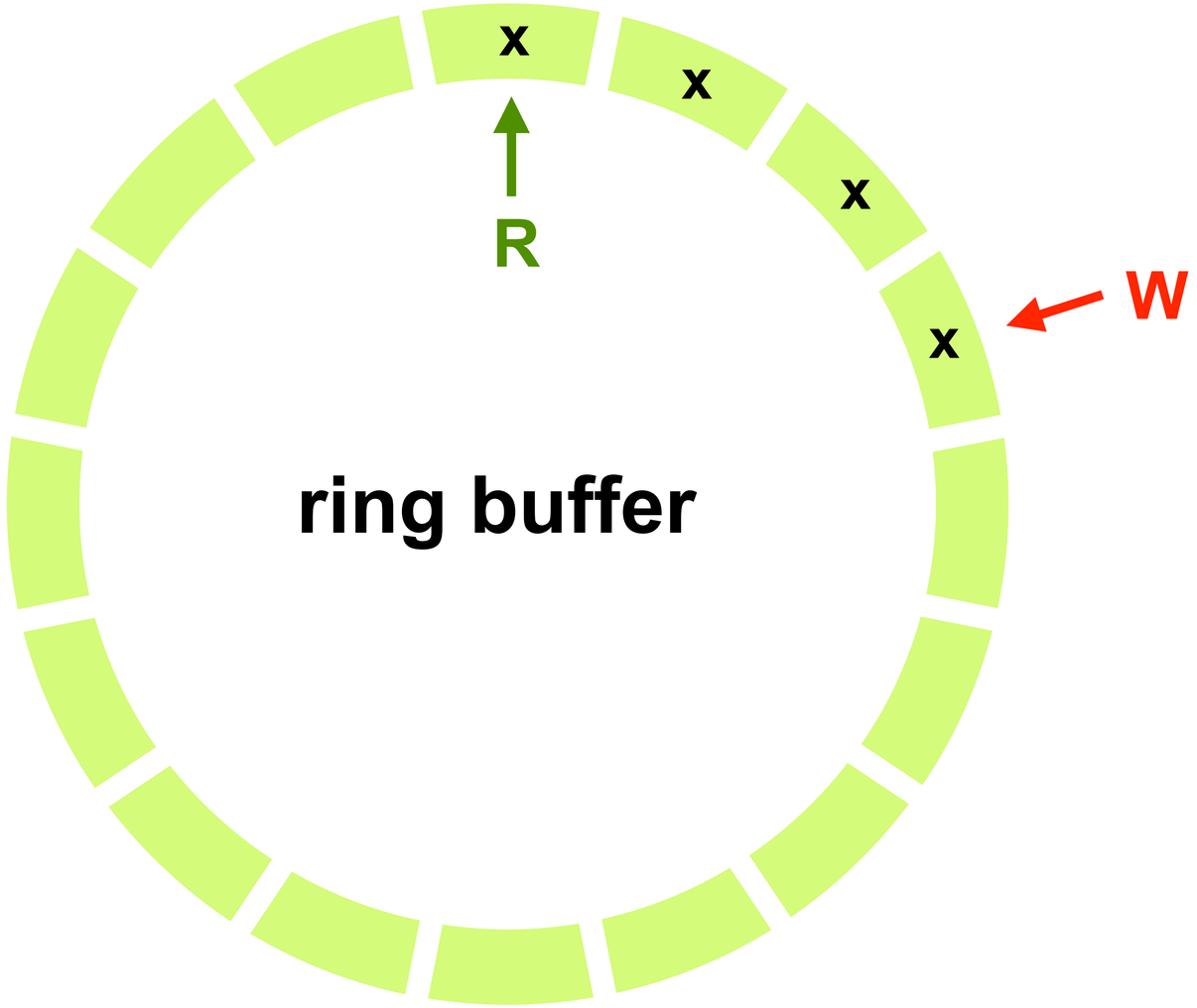


The cost of various lock-free queues

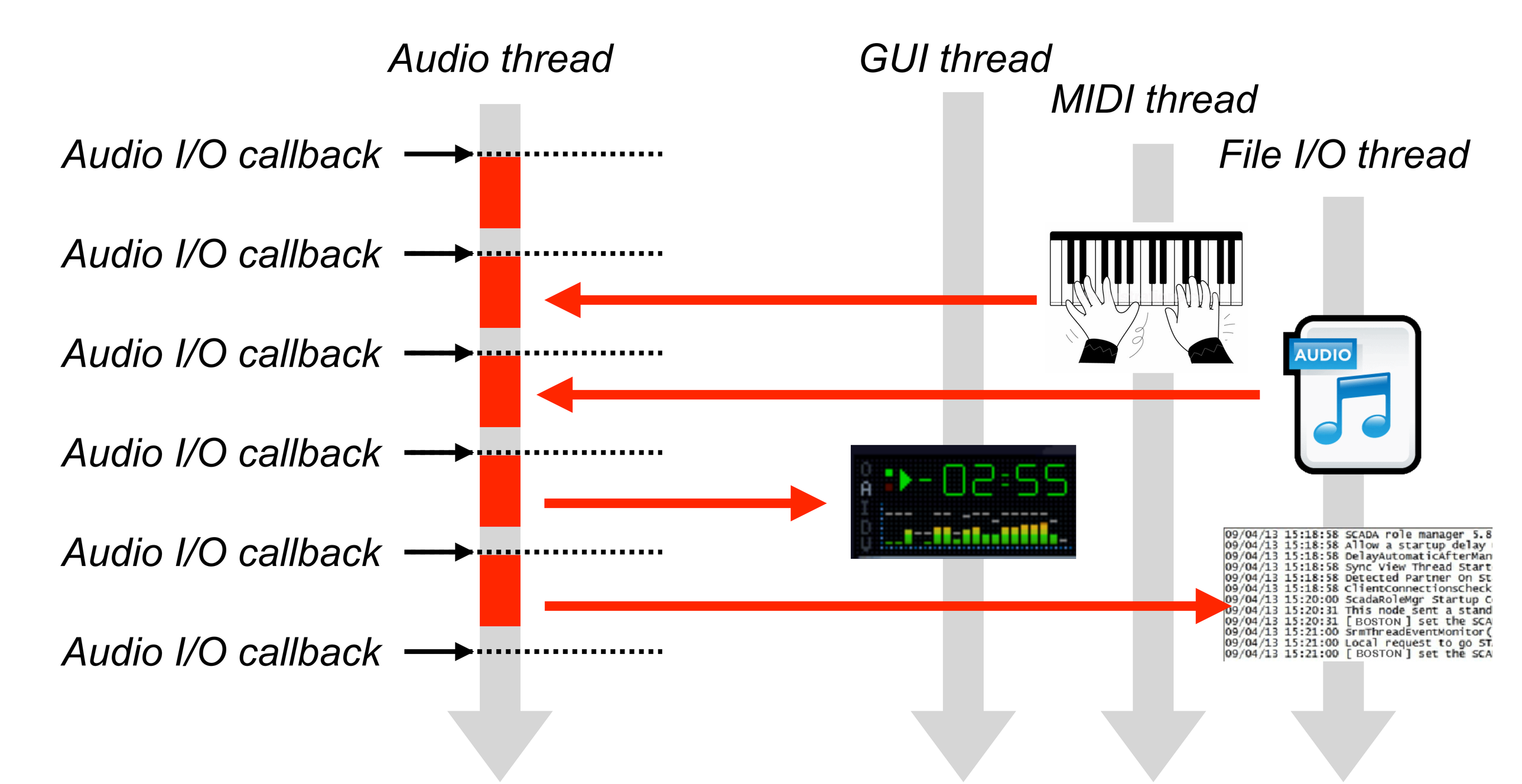
(from "Real-time 101" by Fabian Renn-Giles & Dave Rowland)



Single-producer single-consumer queue

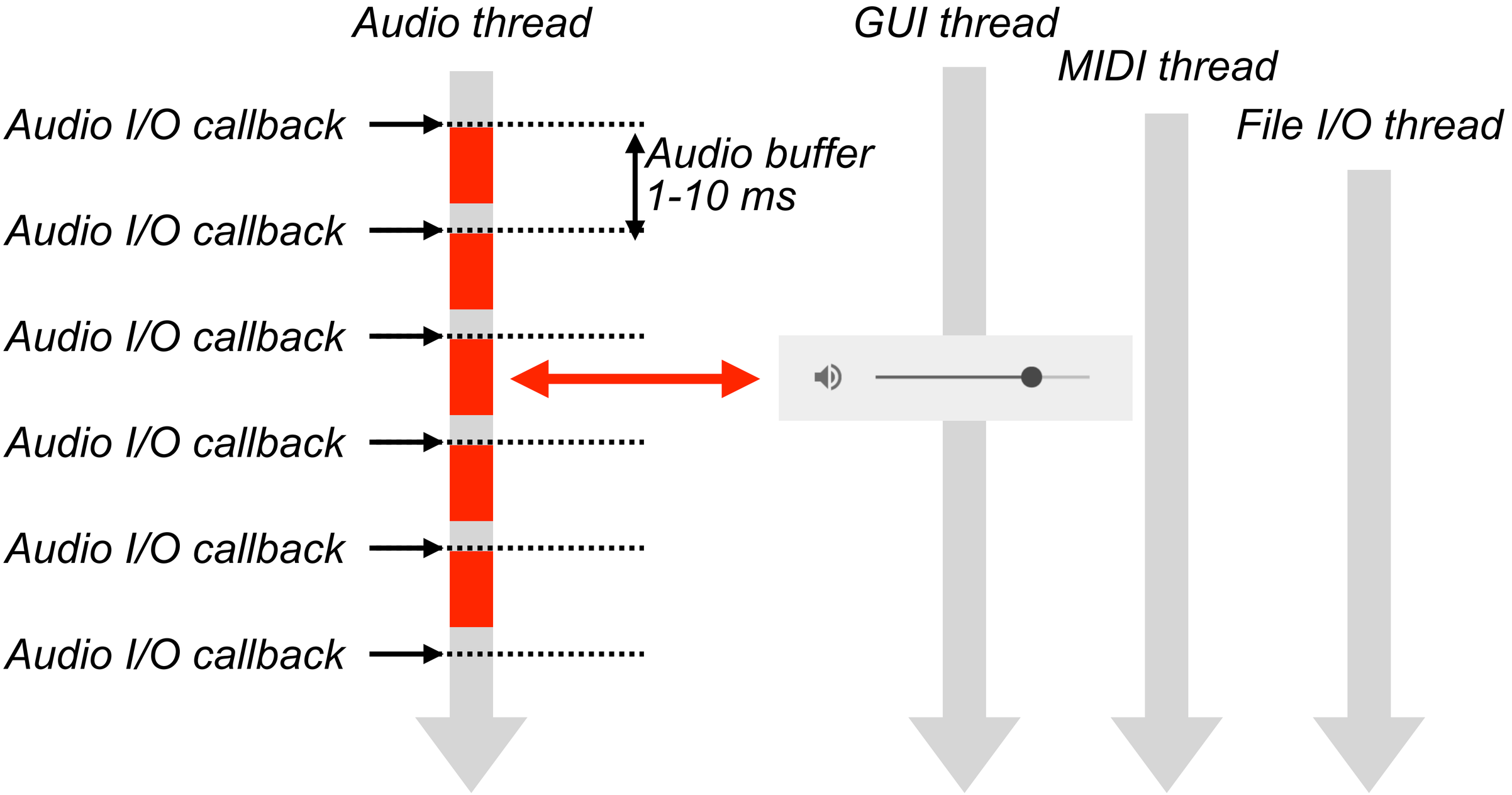


```
template <typename T, size_t size>
struct LockFreeQueue {
    bool push(const T& newElement) {
        auto oldWritePos = writePos.load();
        auto newWritePos = getPositionAfter(oldWritePos);

        if (newWritePos == readPos.load())
            return false;

        ringBuffer[oldWritePos] = newElement;

        writePos.store(newWritePos);
        return true;
    }

    bool pop(T& returnedElement) {
        auto oldWritePos = writePos.load();
        auto oldReadPos = readPos.load();

        if (oldWritePos == oldReadPos)
            return false;

        returnedElement = std::move(ringBuffer[oldReadPos]);

        readPos.store(getPositionAfter(oldReadPos));
        return true;
    }

private:
    static constexpr size_t getPositionAfter(size_t pos) noexcept {
        return ++pos == ringBufferSize ? 0 : pos;
    }

    static constexpr size_t ringBufferSize = size + 1;
    std::array<T, ringBufferSize> ringBuffer;
    std::atomic<size_t> readPos = 0, writePos = 0;
};
```


Audio thread
GUI thread
MIDI thread
File I/O thread
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio I/O callback
AUDIO
-02:55
09/04/13 15:18:58 SCADA role manager 5.8
09/04/13 15:18:58 Allow a startup delay
09/04/13 15:18:58 DelayAutomaticAfterMan
09/04/13 15:18:58 Sync View Thread Start
09/04/13 15:18:58 Detected Partner On St
09/04/13 15:18:58 ClientConnectionsCheck
09/04/13 15:20:00 ScadaRoleMgr Startup C
09/04/13 15:20:31 This node sent a stand
09/04/13 15:20:31 [BOSTON] set the SCA
09/04/13 15:21:00 SrmThreadEventMonitor(
09/04/13 15:21:00 Local request to go ST
09/04/13 15:21:00 [BOSTON] set the SCA

Wait-free concurrency

- Passing data to/from "hot path" thread
  - wait-free queue
- **Sharing data with "hot path" thread**
  - Hot path reads
    - try_lock
    - "spin-on-write"
    - RCU
  - Hot path writes
    - Double-buffering
    - SeqLock


Audio thread
GUI thread
MIDI thread
File I/O thread
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio buffer
1-10 ms

```
float gain;

void process(audio_buffer& buffer) {
    for (auto& sample : buffer)
        sample *= gain;
}

void set_gain(float new_gain) {
    gain = new_gain;
}
```

```
float gain;



void process(audio_buffer& buffer) {
    for (auto& sample : buffer)
        sample *= gain;  // data race
}

void set_gain(float new_gain) {
    gain = new_gain;  // data race
}
```

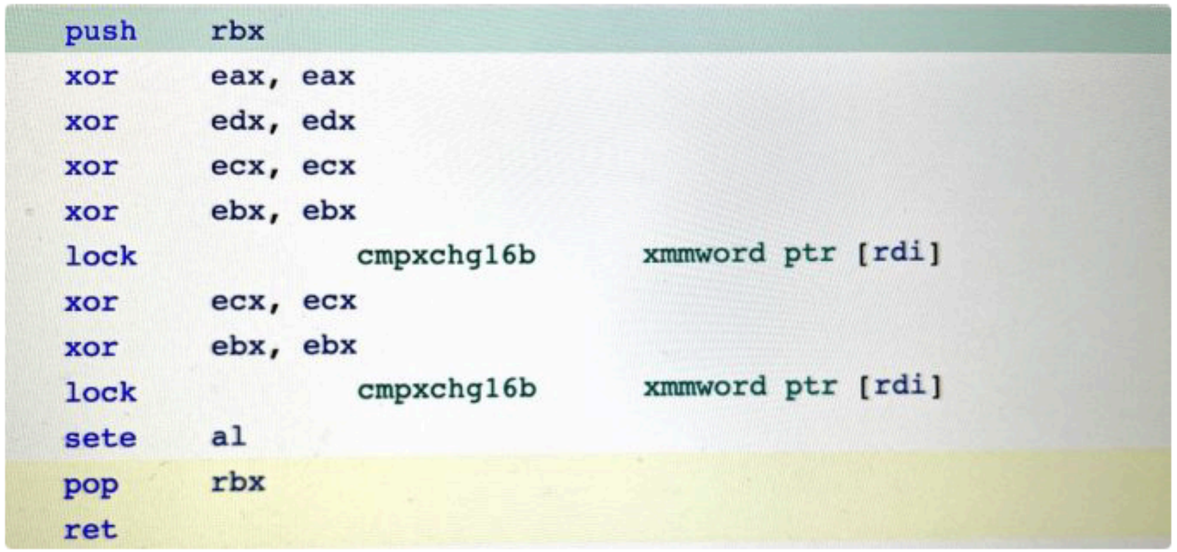
```
std::atomic<float> gain;


void process(audio_buffer& buffer) {
    for (auto& sample : buffer)
        sample *= gain.load();
}

void set_gain(float new_gain) {
    gain.store(new_gain);
}
```

```
std::atomic<float> gain;



void process(audio_buffer& buffer) {
    for (auto& sample : buffer)
        sample *= gain.load();
}

void set_gain(float new_gain) {
    gain.store(new_gain);
}
```

```
std::atomic<float> gain;


void process(audio_buffer& buffer) {
    float current_gain *= gain.load();
    for (auto& sample : buffer)
        sample *= current_gain;
}

void set_gain(float new_gain) {
    gain.store(new_gain);
}
```

```
std::atomic<float> gain;
static_assert(std::atomic<float>::is_always_lock_free);

void process(audio_buffer& buffer) {
    float current_gain *= gain.load();
    for (auto& sample : buffer)
        sample *= current_gain;
}

void set_gain(float new_gain) {
    gain.store(new_gain);
}
```

```
static_assert(std::atomic<std::complex<double>>::is_always_lock_free);
```



```
push    rbx
xor     eax, eax
xor     edx, edx
xor     ecx, ecx
xor     ebx, ebx
lock            cmpxchg16b      xmmword ptr [rdi]
xor     ecx, ecx
xor     ebx, ebx
lock            cmpxchg16b      xmmword ptr [rdi]
sete    al
pop     rbx
ret
```

DWCAS in C++

31 MARCH 2022 / TIMUR DOUMLER / 2 COMMENTS

Implementing your own lock-free data structures using standard C++ isn't something you should attempt unless you really, really know what you are doing (this article summarises why). But you can never become an expert in something if you don't try, so I went in and did it anyway. Among the stumbling blocks I found there was one that I found particularly surprising, so I decided to write a blog post about it: what happens if your lock-free data structure relies on DWCAS.

DWCAS (Double-width compare-and-swap) is a CPU instruction performing an atomic compare-and-swap operation on a memory location that's double the native word size. In other words, it gives you 128-bit atomic lock-free variables on a 64-bit CPU. It is available on every x86-64 chip (`test cmpxchg16b` in x86 assembly) apart from some early AMD and Intel Core 2 chips (we're talking 2008

Search ... SEARCH

RECENT POSTS

DWCAS in C++

Trip report: C++ Siberia 2020

Using locks in real-time audio processing, safely

Trip report: February 2020 ISO C++ committee meeting, Prague

How to make a container from a C++20 range

RECENT COMMENTS

Timur Doumler on DWCAS in C++

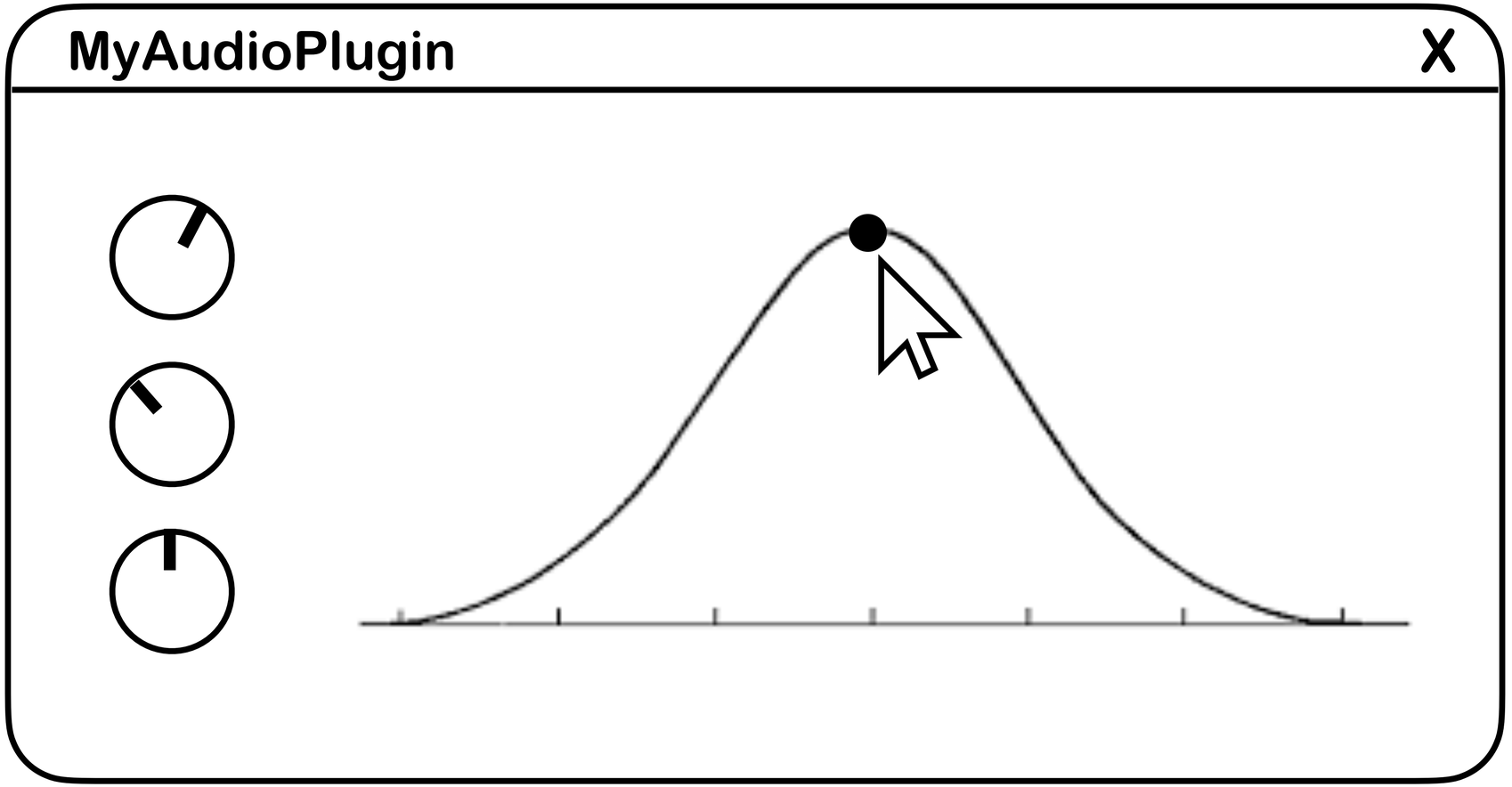
Jim Cownie on DWCAS in C++

How to iterate over a range-v3 action? – Windows Questions on How to make a container from a C++20 range

How to iterate over a range-v3 action? –

	Apple Clang v13	Clang 13	GCC 11	MSVC 17 2022
x86	N/A	✅	✅	✅
x64	✅	⚠️ needs -mcx16 beware ODR violations!	❌	❌
armv7	N/A	❌	✅	N/A
arm64	✅	✅	❌	❌

Example



```
struct biquad_coefficients {
    float b0, b1, b2, a1, a2;
};
```

```
struct biquad_coefficients {
    float b0, b1, b2, a1, a2;
};

std::atomic<biquad_coefficients> coeffs;

void process(audio_buffer& buffer) {
    process_biquad(buffer, coeffs.load());
}

void update_coeffs(biquad_coefficients new_coeffs) {
    coeffs.store(new_coeffs);
}
```

```
struct biquad_coefficients {
    float b0, b1, b2, a1, a2;
};

std::atomic<biquad_coefficients> coeffs;

void process(audio_buffer& buffer) {
    process_biquad(buffer, coeffs.load());  // locks a mutex
}

void update_coeffs(biquad_coefficients new_coeffs) {
    coeffs.store(new_coeffs);  // locks a mutex
}
```

```
struct biquad_coefficients {
    float b0, b1, b2, a1, a2;
};

std::atomic<biquad_coefficients> coeffs;
static_assert(std::atomic<biquad_coefficients>::is_always_lock_free); // false

void process(audio_buffer& buffer) {
    process_biquad(buffer, coeffs.load());
}

void update_coeffs(biquad_coefficients new_coeffs) {
    coeffs.store(new_coeffs);
}
```

Wait-free concurrency

- Passing data to/from "hot path" thread
  - wait-free queue
- **Sharing data with "hot path" thread**
  - Hot path reads
    - try_lock
    - "spin-on-write"
    - RCU
  - Hot path writes
    - Double-buffering
    - SeqLock

Wait-free concurrency

- Passing data to/from "hot path" thread
  - wait-free queue
- Sharing data with "hot path" thread
  - Hot path reads
    - **try_lock – if reading the data can fail**
    - "spin-on-write"
    - RCU
  - Hot path writes
    - Double-buffering
    - SeqLock

```
biquad_coefficients coeffs;
std::mutex mtx;

void process(audio_buffer& buffer) {
    if (std::unique_lock lock(mtx, std::try_to_lock);
        lock.owns_lock())
        process_biquad(buffer, coeffs);
    else
        /* fallback strategy */
}

void update_coeffs(biquad_coefficients new_coeffs) {
    std::lock_guard lock(mtx);
    coeffs = new_coeffs;
}
```

```
biquad_coefficients coeffs;
std::mutex mtx;

void process(audio_buffer& buffer) {
    if (std::unique_lock lock(mtx, std::try_to_lock);
        lock.owns_lock())
        process_biquad(buffer, coeffs);
    else
        /* fallback strategy */
}

void update_coeffs(biquad_coefficients new_coeffs) {
    std::lock_guard lock(mtx);
    coeffs = new_coeffs;
}
```

```
biquad_coefficients coeffs;
std::mutex mtx;

void process(audio_buffer& buffer) {
    if (std::unique_lock lock(mtx, std::try_to_lock);
        lock.owns_lock())
        process_biquad(buffer, coeffs);
    else
        /* fallback strategy */
}

void update_coeffs(biquad_coefficients new_coeffs) {
    std::lock_guard lock(mtx);
    coeffs = new_coeffs;
}
```

```
biquad_coefficients coeffs;
std::mutex mtx;

void process(audio_buffer& buffer) {
    if (std::unique_lock lock(mtx, std::try_to_lock);
        lock.owns_lock())
        process_biquad(buffer, coeffs);
    else
        /* fallback strategy */
}

void update_coeffs(biquad_coefficients new_coeffs) {
    std::lock_guard lock(mtx);
    coeffs = new_coeffs;
}
```

```
biquad_coefficients coeffs;
std::mutex mtx;  // don't use std::mutex for this!

void process(audio_buffer& buffer) {
    if (std::unique_lock lock(mtx, std::try_to_lock);
        lock.owns_lock())
        process_biquad(buffer, coeffs);
    else
        /* fallback strategy */
}  // <-- mtx.unlock() called here - not deterministic with std::mutex

void update_coeffs(biquad_coefficients new_coeffs) {
    std::lock_guard lock(mtx);
    coeffs = new_coeffs;
}
```


ADC20
AUDIO
DEVELOPER
CONFERENCE
Using Locks in Real-Time
Audio Processing, Safely
Timur Doumler
0:10 / 52:54

```
biquad_coefficients coeffs;
crill::spin_mutex mtx;  // Use spinlock with progressive fallback

void process(audio_buffer& buffer) {
    if (std::unique_lock lock(mtx, std::try_to_lock);
        lock.owns_lock())
        process_biquad(buffer, coeffs);
    else
        /* fallback strategy */
}

void update_coeffs(biquad_coefficients new_coeffs) {
    std::lock_guard lock(mtx);
    coeffs = new_coeffs;
}
```

github.com/crill-dev/crill

crill-dev / crill Public

Edit Pins | Watch 12 | Fork 3 | Starred 89

Code | Issues 7 | Pull requests 1 | Zenhub | Discussions | Actions | Projects | Wiki | Security | Insights | ...

main | 4 branches | 0 tags | Go to file | Add file | Code

timuraudio seqlock: fixed documentation which was erroneously stating that ... ... bedcf27 on Feb 8 28 commits

.github/workflows	Added GitHub Actions workflow to build and test	9 months ago
include	seqlock: fixed documentation which was erroneously stating that r...	3 months ago
tests	seqlock_object: fixed default constructor which was constructing ...	3 months ago
.gitignore	Added .gitignore	10 months ago
CMakeLists.txt	Added seqlock_object	3 months ago
LICENSE.txt	Added license	10 months ago
README.md	Fixed wrong github actions badge URL	9 months ago

README.md

Crill

Build and test passing

Crill stands for **C**ross-platform **R**eal-time, **I**/O, and **L**ow-latency **L**ibrary. It is a C++ header-only library offering generic utilities for low-latency and real-time code.

Crill is currently a work in progress.

About

The crill library

Readme
BSL-1.0 license
89 stars
12 watching
3 forks
Report repository

Releases

No releases published
Create a new release

Packages

No packages published
Publish your first package

Languages

C++ 99.6% Other 0.4%

try_lock

- Use with spinlock, not std::mutex!
- Use progressive backoff to avoid wasting energy
- Tradeoffs:
  - Reader always wait-free
  - Single reader
  - Only works if reading is allowed to fail
  - Reader blocks writer(s) who needs to spin
  - one writer (or multiple writers who block each other)

```
biquad_coefficients coeffs;
crill::spin_mutex mtx;

void process(audio_buffer& buffer) {
    if (std::unique_lock lock(mtx, std::try_to_lock);
        lock.owns_lock())
        process_biquad(buffer, coeffs);
    else
        /* fallback strategy - what if there is none? */
}

void update_coeffs(biquad_coefficients new_coeffs) {
    std::lock_guard lock(mtx);
    coeffs = new_coeffs;
}
```

```
std::atomic<biquad_coefficients*> coeffs;
```

```
std::atomic<biquad_coefficients*> coeffs;

void process(audio_buffer& buffer) {
    auto* current_coeffs = coeffs.load();
    process_biquad(buffer, *current_coeffs);
    // what to do with current_coeffs?
}

void update_coeffs(biquad_coefficients new_coeffs) {
    auto* new_coeffs = new biquad_coefficients(new_coeffs);
    auto* old_coeffs = coeffs.exchange(new_coeffs);
    // what to do with old_coeffs?
}
```

Wait-free concurrency

- Passing data to/from "hot path" thread
  - wait-free queue
- Sharing data with "hot path" thread
  - Hot path reads
    - try_lock
    - **"spin-on-write"**
    - RCU
  - Hot path writes
    - Double-buffering
    - SeqLock


Meeting C++ 2019
David Rowland
Fabian Renn-Giles
Real-time 101
Real-time 101
David Rowland & Fabian Renn-Giles
@drowaudio @hogliux
0:01 / 47:55 • Intro

```
std::unique_ptr<biquad_coefficients> storage;
std::atomic<biquad_coefficients*> coeffs;

void process(audio_buffer& buffer) {
    auto* current_coeffs = coeffs.exchange(nullptr);
    process_biquad(buffer, *current_coeffs);
    coeffs.store(current_coeffs);
}

void update_coeffs(biquad_coefficients new_coeffs) {
    auto new_coeffs = std::make_unique<biquad_coefficients>(new_coeffs);
    for (auto* expected = storage.get();
         !coeffs.compare_exchange_weak(expected, new_coeffs.get());
         expected = storage.get() /* spin */;
    storage = std::move(new_coeffs);
}
```

```
std::unique_ptr<biquad_coefficients> storage;
std::atomic<biquad_coefficients*> coeffs;

void process(audio_buffer& buffer) {
    auto* current_coeffs = coeffs.exchange(nullptr);
    process_biquad(buffer, *current_coeffs);
    coeffs.store(current_coeffs);
}

void update_coeffs(biquad_coefficients new_coeffs) {
    auto new_coeffs = std::make_unique<biquad_coefficients>(new_coeffs);
    for (auto* expected = storage.get();
         !coeffs.compare_exchange_weak(expected, new_coeffs.get());
         expected = storage.get() /* spin */;
    storage = std::move(new_coeffs);
}
```

```
std::unique_ptr<biquad_coefficients> storage;
std::atomic<biquad_coefficients*> coeffs;

void process(audio_buffer& buffer) {
    auto* current_coeffs = coeffs.exchange(nullptr);
    process_biquad(buffer, *current_coeffs);
    coeffs.store(current_coeffs);
}

void update_coeffs(biquad_coefficients new_coeffs) {
    auto new_coeffs = std::make_unique<biquad_coefficients>(new_coeffs);
    for (auto* expected = storage.get();
         !coeffs.compare_exchange_weak(expected, new_coeffs.get());
         expected = storage.get() /* spin */;
    storage = std::move(new_coeffs);
}
```

```
std::unique_ptr<biquad_coefficients> storage;
std::atomic<biquad_coefficients*> coeffs;

void process(audio_buffer& buffer) {
    auto* current_coeffs = coeffs.exchange(nullptr);
    process_biquad(buffer, *current_coeffs);
    coeffs.store(current_coeffs);
}

void update_coeffs(biquad_coefficients new_coeffs) {
    auto new_coeffs = std::make_unique<biquad_coefficients>(new_coeffs);
    for (auto* expected = storage.get();
         !coeffs.compare_exchange_weak(expected, new_coeffs.get());
         expected = storage.get() /* spin */;
    storage = std::move(new_coeffs);
}
```

```
std::unique_ptr<biquad_coefficients> storage;
std::atomic<biquad_coefficients*> coeffs;

void process(audio_buffer& buffer) {
    auto* current_coeffs = coeffs.exchange(nullptr);
    process_biquad(buffer, *current_coeffs);
    coeffs.store(current_coeffs);
}

void update_coeffs(biquad_coefficients new_coeffs) {
    auto new_coeffs = std::make_unique<biquad_coefficients>(new_coeffs);
    for (auto* expected = storage.get();
         !coeffs.compare_exchange_weak(expected, new_coeffs.get());
         expected = storage.get() /* spin */;
    storage = std::move(new_coeffs);
}
```

```
std::unique_ptr<biquad_coefficients> storage;
std::atomic<biquad_coefficients*> coeffs;

void process(audio_buffer& buffer) {
    auto* current_coeffs = coeffs.exchange(nullptr);
    process_biquad(buffer, *current_coeffs);
    coeffs.store(current_coeffs);
}

void update_coeffs(biquad_coefficients new_coeffs) {
    auto new_coeffs = std::make_unique<biquad_coefficients>(new_coeffs);
    for (auto* expected = storage.get();
         !coeffs.compare_exchange_weak(expected, new_coeffs.get());
         expected = storage.get() /* spin */;
    storage = std::move(new_coeffs);
}
```

```
crill::spin_on_write_object<biquad_coefficients> coeffs;

void process(audio_buffer& buffer) {
    auto read_ptr = coeffs.lock_read();
    process_biquad(buffer, *read_ptr);
}

void update_coeffs(biquad_coefficients new_coeffs) {
    coeffs.update(new_coeffs);
}
```

```
crill::spin_on_write_object<biquad_coefficients> coeffs;

void process(audio_buffer& buffer) {
    auto read_ptr = coeffs.lock_read();
    process_biquad(buffer, *read_ptr);
}

void update_coeffs(biquad_coefficients new_coeffs) {
    coeffs.update(new_coeffs);
}
```

```
crill::spin_on_write_object<biquad_coefficients> coeffs;

void process(audio_buffer& buffer) {
    auto read_ptr = coeffs.lock_read();
    process_biquad(buffer, *read_ptr);
}

void update_coeffs(biquad_coefficients new_coeffs) {
    coeffs.update(new_coeffs);
}
```

```
crill::spin_on_write_object<biquad_coefficients> coeffs;

void process(audio_buffer& buffer) {
    auto read_ptr = coeffs.lock_read();
    process_biquad(buffer, *read_ptr);
}

void update_coeffs(biquad_coefficients new_coeffs) {
    coeffs.update(new_coeffs);
}

void modify_coeffs() {
    auto write_ptr = coeffs.lock_write();
    write_ptr->a0 = 0.0f;
}
```

spin_on_write_object

- Easy to use
- Reader always wait-free
- Tradeoffs:
  - Single reader
  - read is 2x std::atomic::exchange
  - reader blocks writer(s) who need to spin
  - one writer (or multiple writers who block each other)
  - writer needs to heap-allocate + copy

Wait-free concurrency

- Passing data to/from "hot path" thread
  - wait-free queue
- Sharing data with "hot path" thread
  - Hot path reads
    - try_lock
    - "spin-on-write"
    - **RCU (Read, Copy, Update)**
  - Hot path writes
    - Double-buffering
    - SeqLock

```
std::unique_ptr<biquad_coefficients> storage;
std::atomic<biquad_coefficients*> coeffs;

void process(audio_buffer& buffer) {
    auto coeffs_ptr = coeffs.load();
    process_biquad(buffer, *coeffs_ptr);
}

void update_coeffs(biquad_coefficients new_coeffs) {
    auto new_coeffs = std::make_unique<biquad_coefficients>(new_coeffs);
    coeffs.store(new_coeffs_ptr);
    storage.reset(new_coeffs_ptr);
}
```

```
std::unique_ptr<biquad_coefficients> storage;
std::atomic<biquad_coefficients*> coeffs;

void process(audio_buffer& buffer) {
    auto coeffs_ptr = coeffs.load();
    process_biquad(buffer, *coeffs_ptr);
}

void update_coeffs(biquad_coefficients new_coeffs) {
    auto new_coeffs = std::make_unique<biquad_coefficients>(new_coeffs);
    coeffs.store(new_coeffs_ptr);
    storage.reset(new_coeffs_ptr);
}
```

```
std::unique_ptr<biquad_coefficients> storage;
std::atomic<biquad_coefficients*> coeffs;

void process(audio_buffer& buffer) {
    auto coeffs_ptr = coeffs.load();
    process_biquad(buffer, *coeffs_ptr);
}

void update_coeffs(biquad_coefficients new_coeffs) {
    auto new_coeffs = std::make_unique<biquad_coefficients>(new_coeffs);
    coeffs.store(new_coeffs_ptr);
    storage.reset(new_coeffs_ptr);
}
```

```
std::unique_ptr<biquad_coefficients> storage;
std::atomic<biquad_coefficients*> coeffs;

void process(audio_buffer& buffer) {
    auto coeffs_ptr = coeffs.load();
    process_biquad(buffer, *coeffs_ptr);
}

void update_coeffs(biquad_coefficients new_coeffs) {
    auto new_coeffs = std::make_unique<biquad_coefficients>(new_coeffs);
    coeffs.store(new_coeffs_ptr);
    storage.reset(new_coeffs_ptr);
}
```

```
std::unique_ptr<biquad_coefficients> storage;
std::atomic<biquad_coefficients*> coeffs;

void process(audio_buffer& buffer) {
    auto coeffs_ptr = coeffs.load();
    process_biquad(buffer, *coeffs_ptr);
}

void update_coeffs(biquad_coefficients new_coeffs) {
    auto new_coeffs = std::make_unique<biquad_coefficients>(new_coeffs);
    coeffs.store(new_coeffs_ptr);
    storage.reset(new_coeffs_ptr);
}
```

```
std::unique_ptr<biquad_coefficients> storage;
std::atomic<biquad_coefficients*> coeffs;

void process(audio_buffer& buffer) {
    auto coeffs_ptr = coeffs.load();
    process_biquad(buffer, *coeffs_ptr);
}

void update_coeffs(biquad_coefficients new_coeffs) {
    auto new_coeffs = std::make_unique<biquad_coefficients>(new_coeffs);
    coeffs.store(new_coeffs_ptr);
    storage.reset(new_coeffs_ptr);
}
```

```
std::unique_ptr<biquad_coefficients> storage;
std::atomic<biquad_coefficients*> coeffs;

void process(audio_buffer& buffer) {
    auto coeffs_ptr = coeffs.load();
    process_biquad(buffer, *coeffs_ptr);
}

void update_coeffs(biquad_coefficients new_coeffs) {
    auto new_coeffs = std::make_unique<biquad_coefficients>(new_coeffs);
    coeffs.store(new_coeffs_ptr);
    storage.reset(new_coeffs_ptr);   // Can't do that! might still be read!
}
```

```
std::unique_ptr<biquad_coefficients> storage;
std::atomic<biquad_coefficients*> coeffs;

void process(audio_buffer& buffer) {
    auto coeffs_ptr = coeffs.load();
    process_biquad(buffer, *coeffs_ptr);
}

void update_coeffs(biquad_coefficients new_coeffs) {
    auto new_coeffs = std::make_unique<biquad_coefficients>(new_coeffs);
    coeffs.store(new_coeffs_ptr);
    auto* old_ptr = storage.release();   // What to do with old_ptr?
    storage.reset(new_coeffs);           // Nothing -> memory leak!
}
```

```
std::unique_ptr<biquad_coefficients> storage;
std::atomic<biquad_coefficients*> coeffs;

void process(audio_buffer& buffer) {
    auto coeffs_ptr = coeffs.load();
    process_biquad(buffer, *coeffs_ptr);
}

void update_coeffs(biquad_coefficients new_coeffs) {
    auto new_coeffs = std::make_unique<biquad_coefficients>(new_coeffs);
    coeffs.store(new_coeffs_ptr);
    auto* old_ptr = storage.release();   // What to do with old_ptr?
    storage.reset(new_coeffs);           // Delete "later, when it's safe"
}                                        // -> "Deferred reclamation"
```

RCU

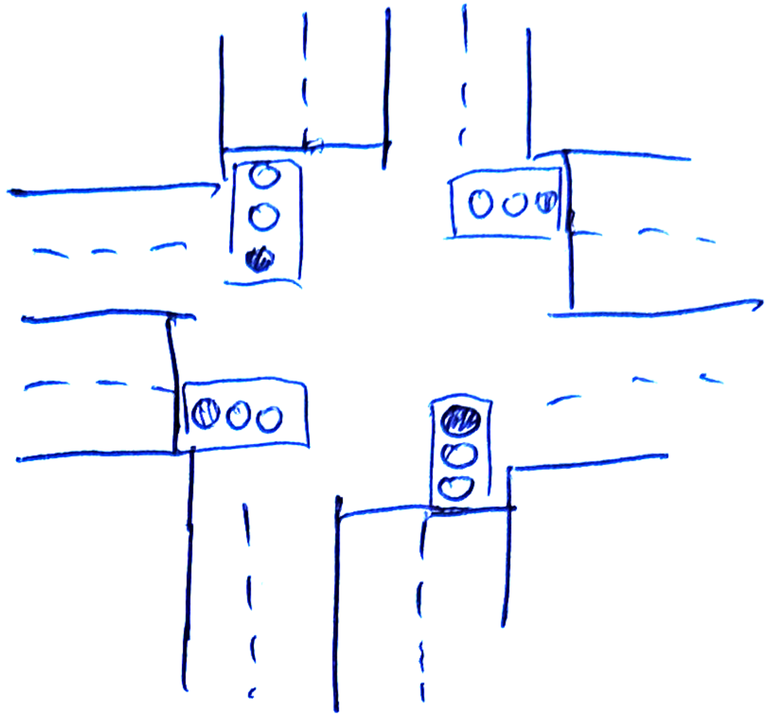
- read is now a single atomic load (= no overhead on modern platforms)
- Multiple concurrent readers & writers
- Readers don't block writers
- But we need to solve deferred reclamation problem!
  - hard...

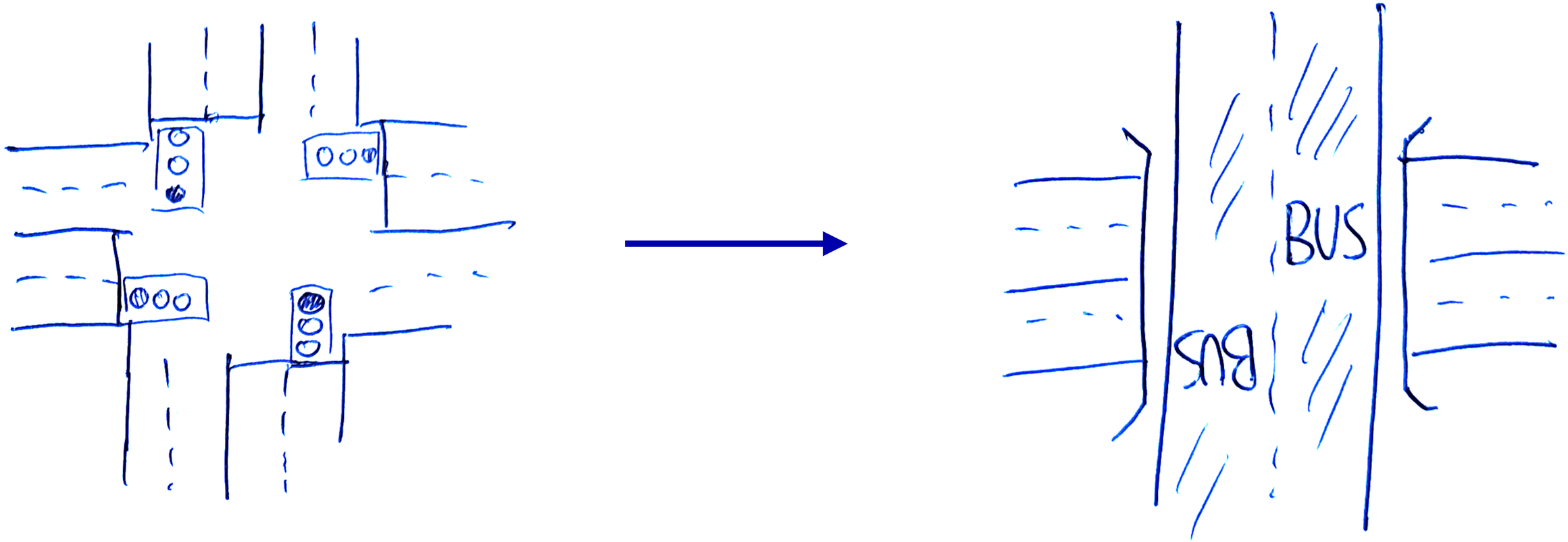
RCU

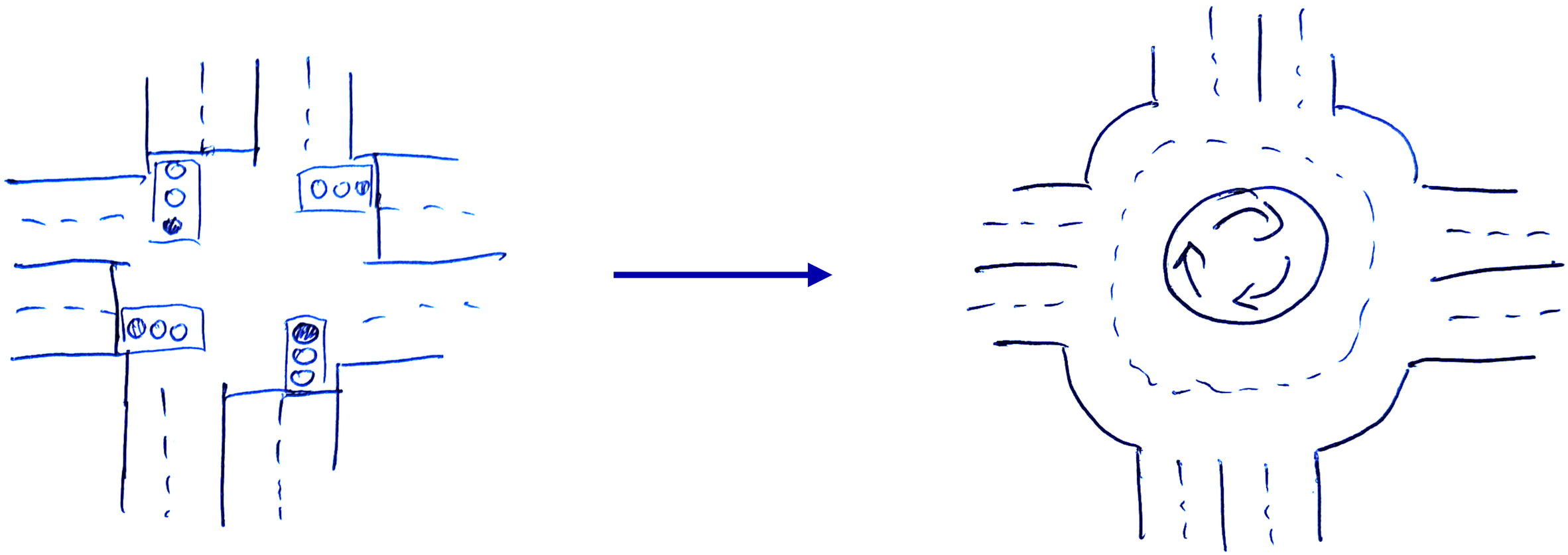
- read is now a single atomic load (= no overhead on modern platforms)
- Multiple concurrent readers & writers
- Readers don't block writers
- But we need to solve deferred reclamation problem!
  - hard...
  - same problem needs to be solved for **lock-free data structures**

Lock-free data structures

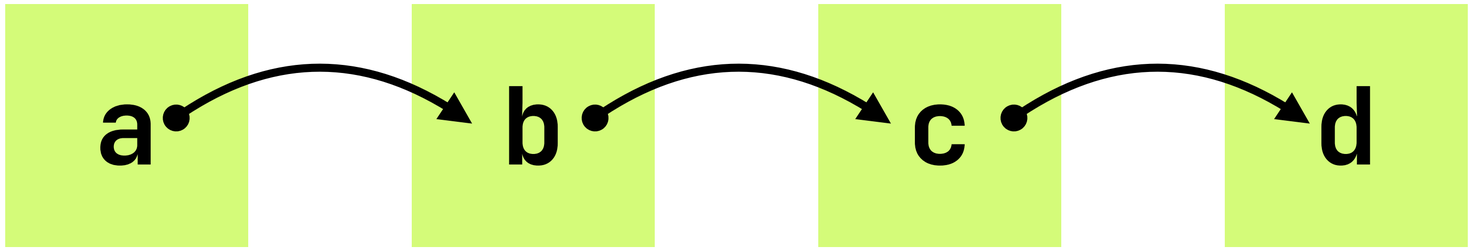
- Queue
- Stack
- Linked list



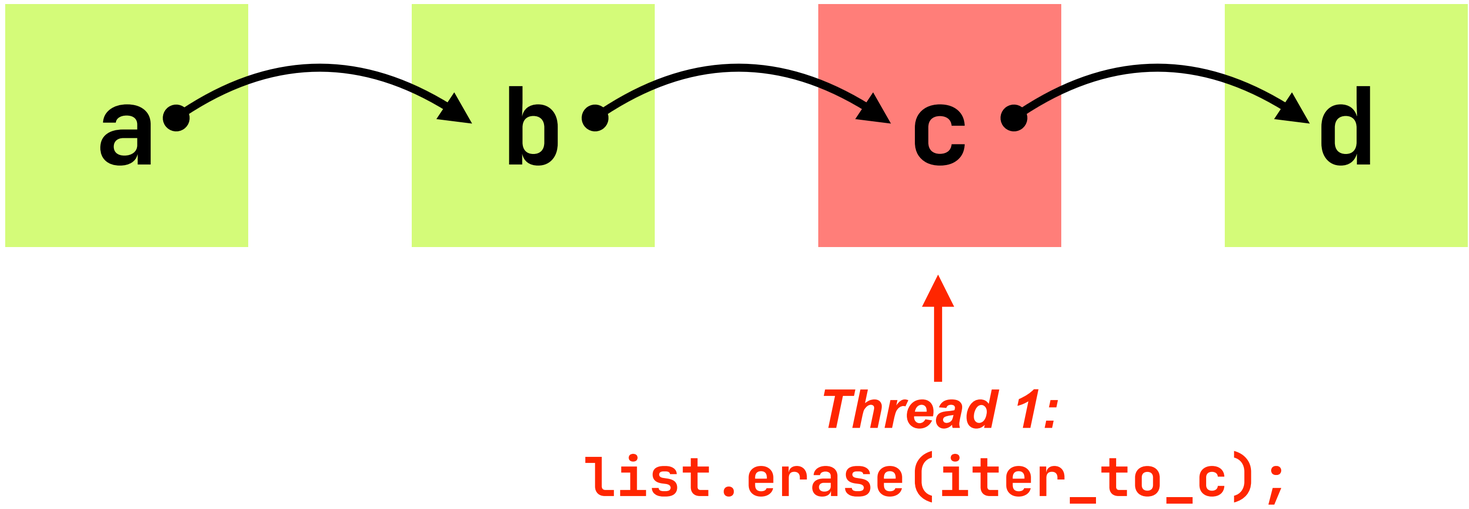

BUS
BUS



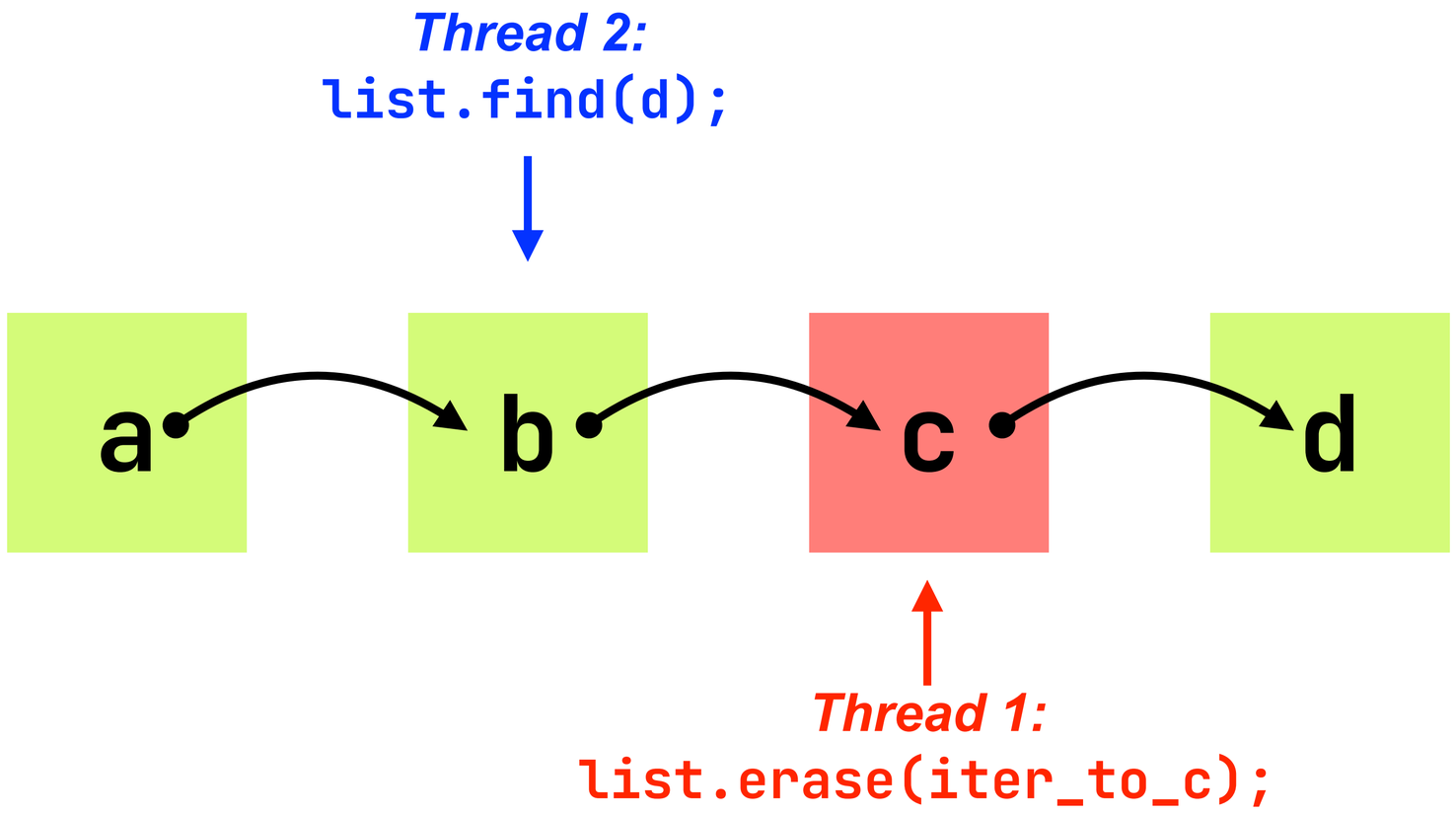
Example: lock-free linked list



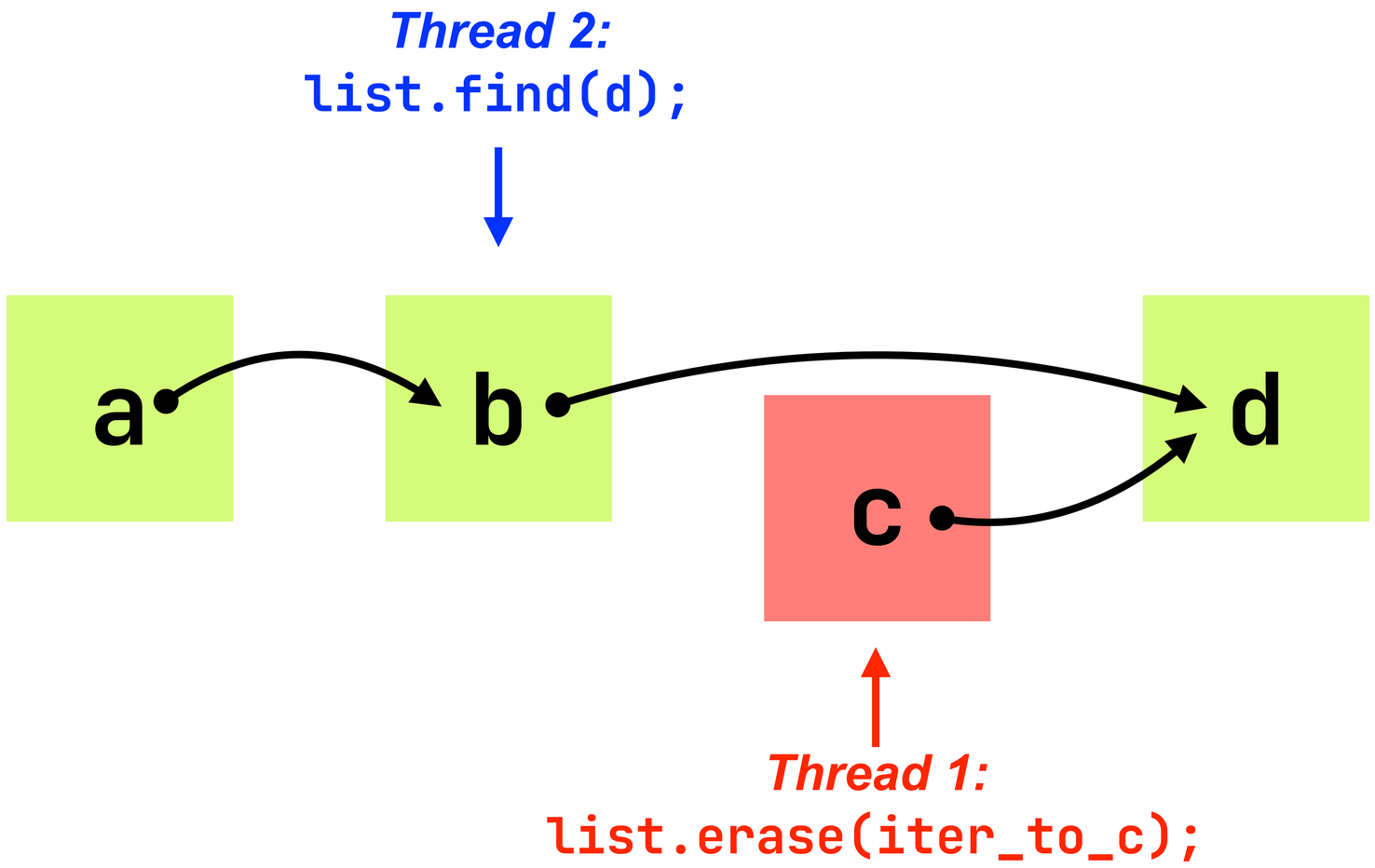
Example: lock-free linked list



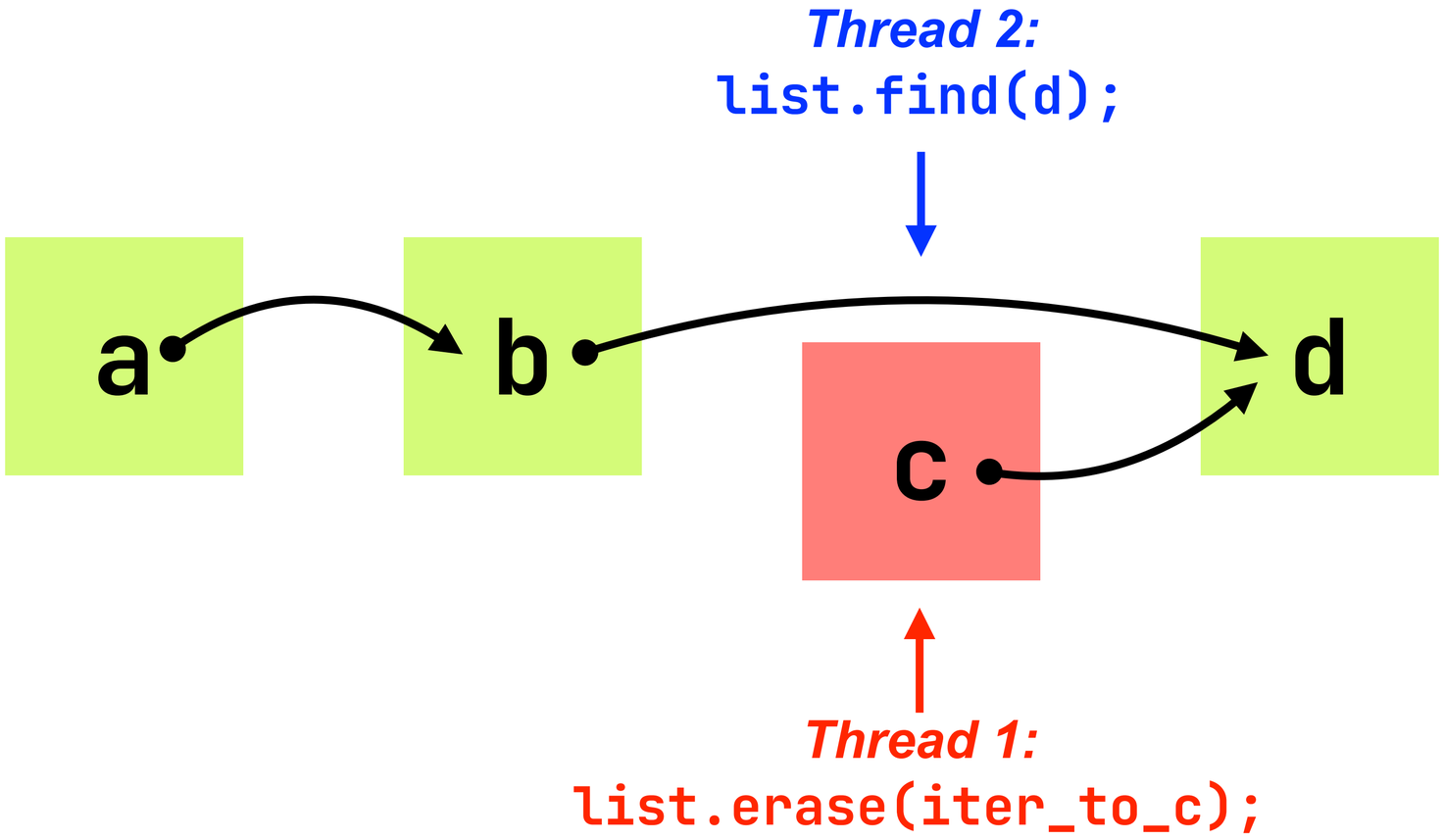
Example: lock-free linked list

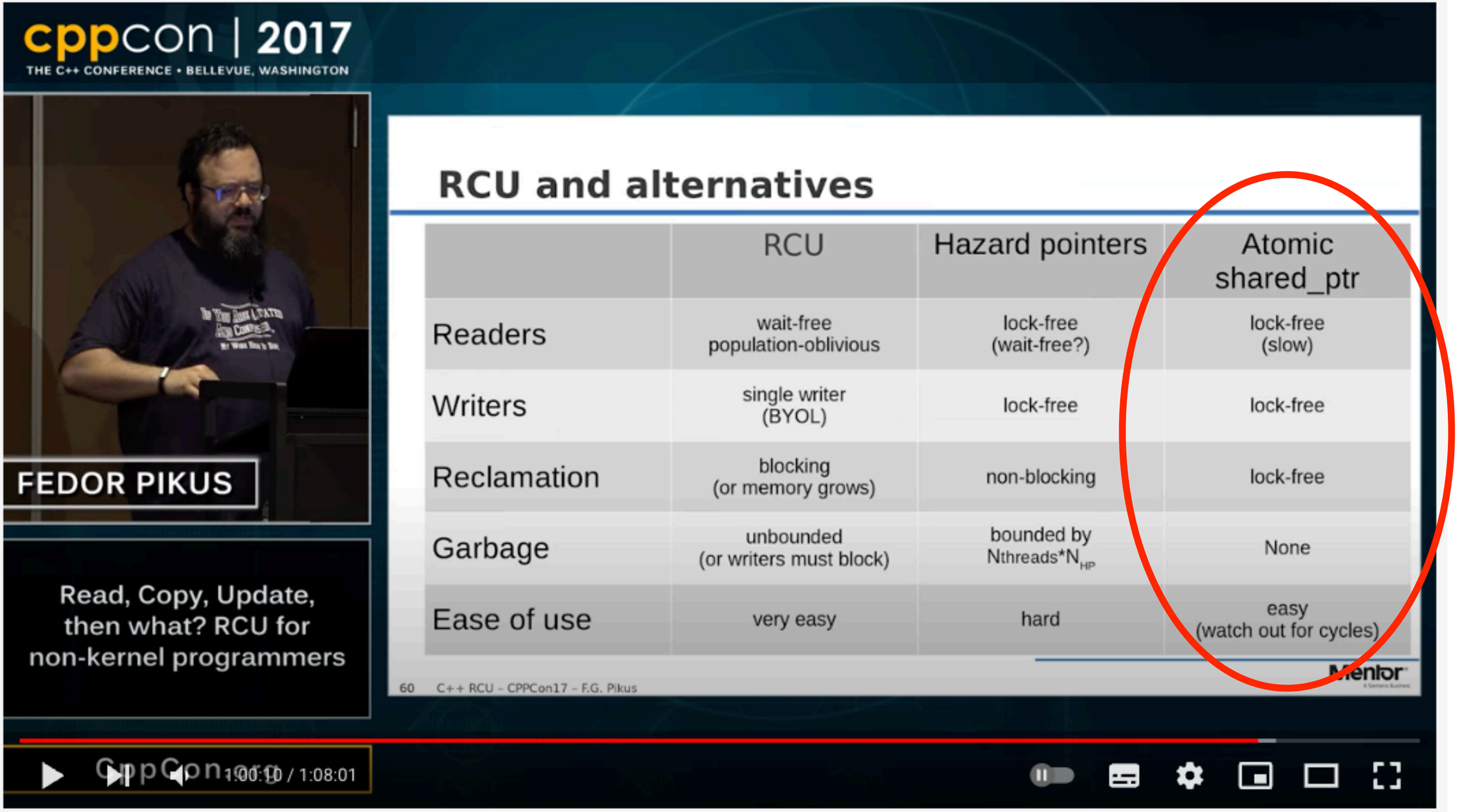


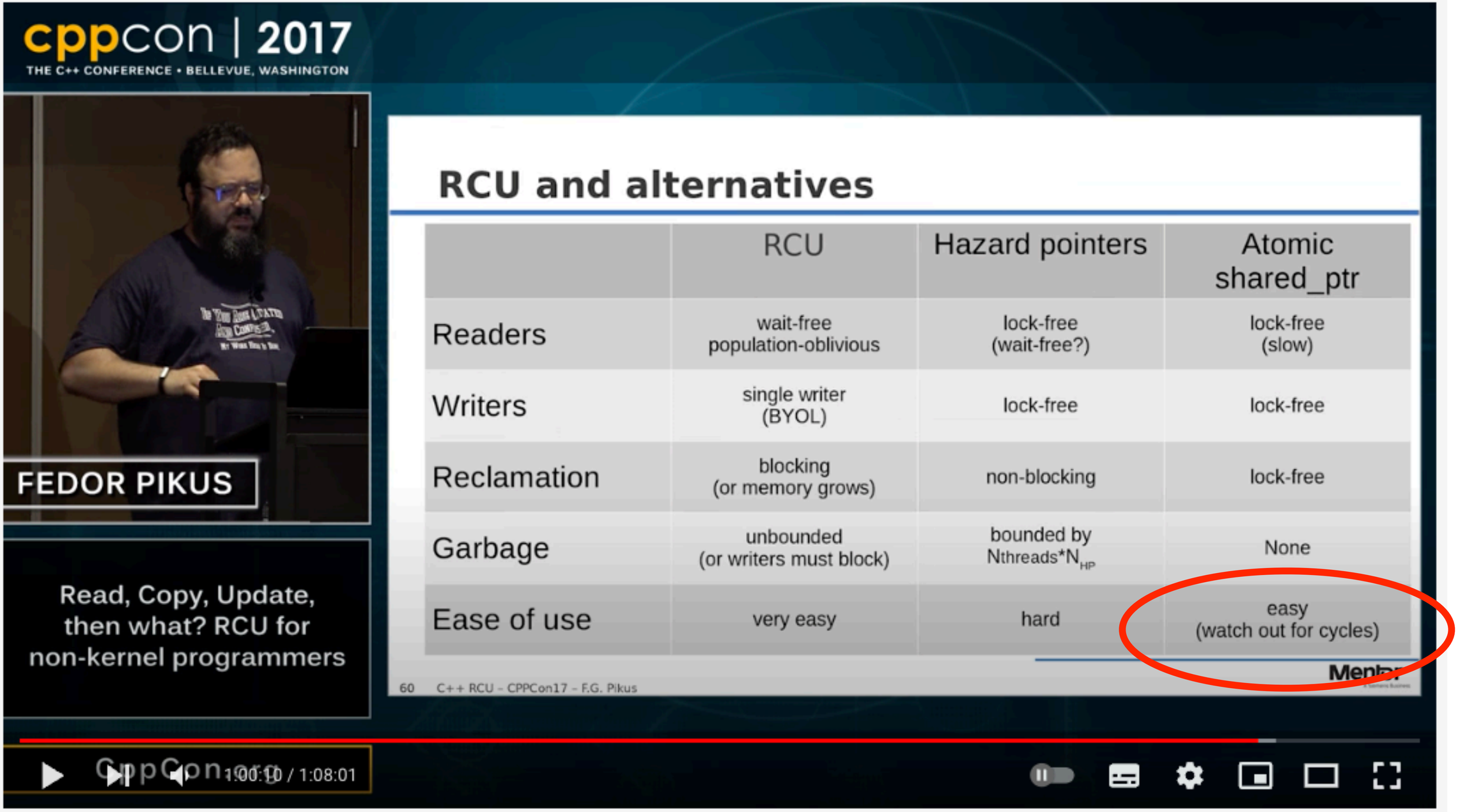
Example: lock-free linked list



Example: lock-free linked list



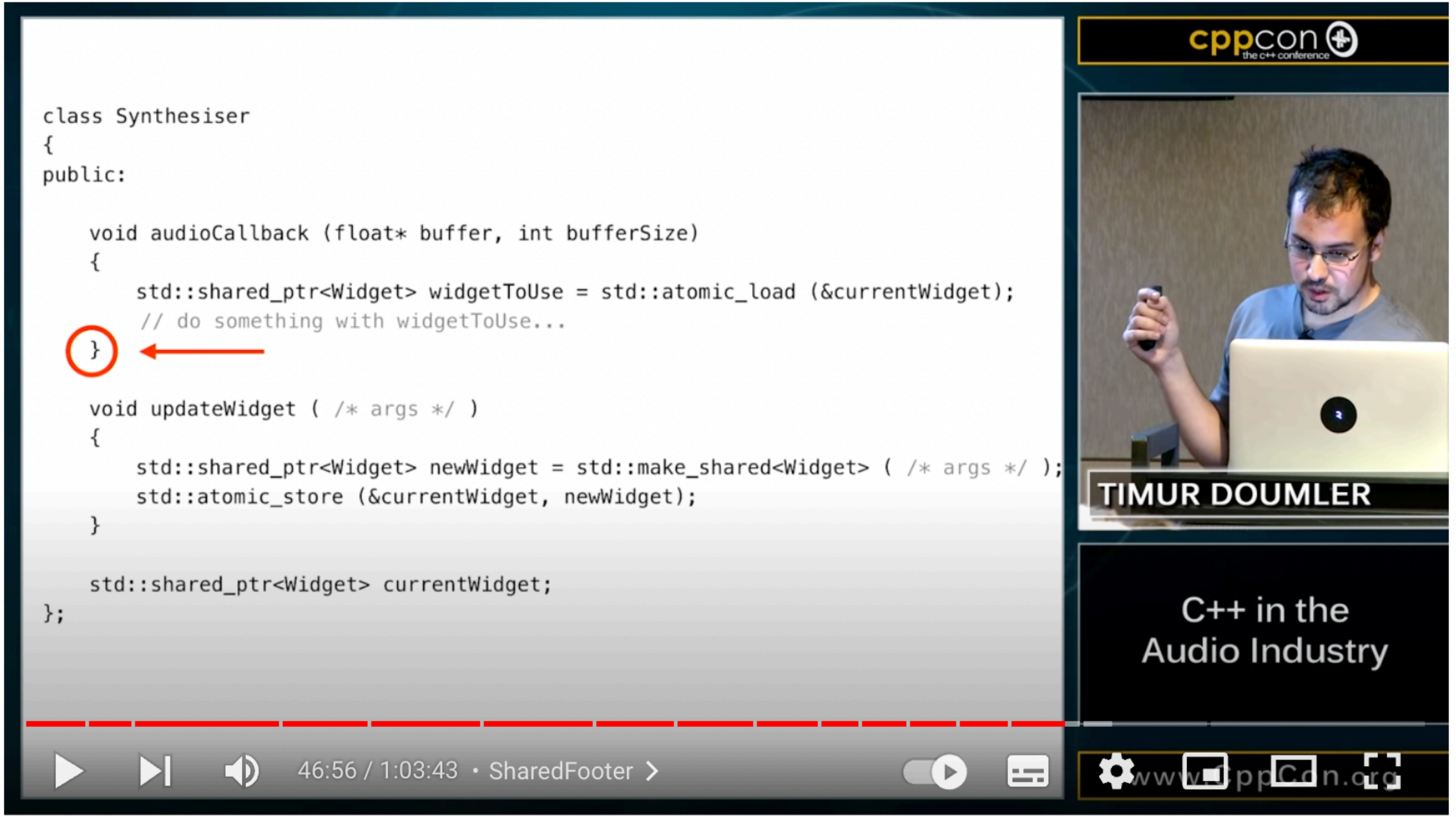

cppcon | 2017
THE C++ CONFERENCE • BELLEVUE, WASHINGTON
FEDOR PIKUS
Read, Copy, Update, then what? RCU for non-kernel programmers
RCU and alternatives
RCU
Hazard pointers
Atomic shared_ptr
Readers
wait-free population-oblivious
lock-free (wait-free?)
lock-free (slow)
Writers
single writer (BYOL)
lock-free
lock-free
Reclamation
blocking (or memory grows)
non-blocking
lock-free
Garbage
unbounded (or writers must block)
bounded by Nthreads*N_HP
None
Ease of use
very easy
hard
easy (watch out for cycles)
60
C++ RCU – CPPCon17 – F.G. Pikus
Mentor
CppCon.org
1:00:10 / 1:08:01

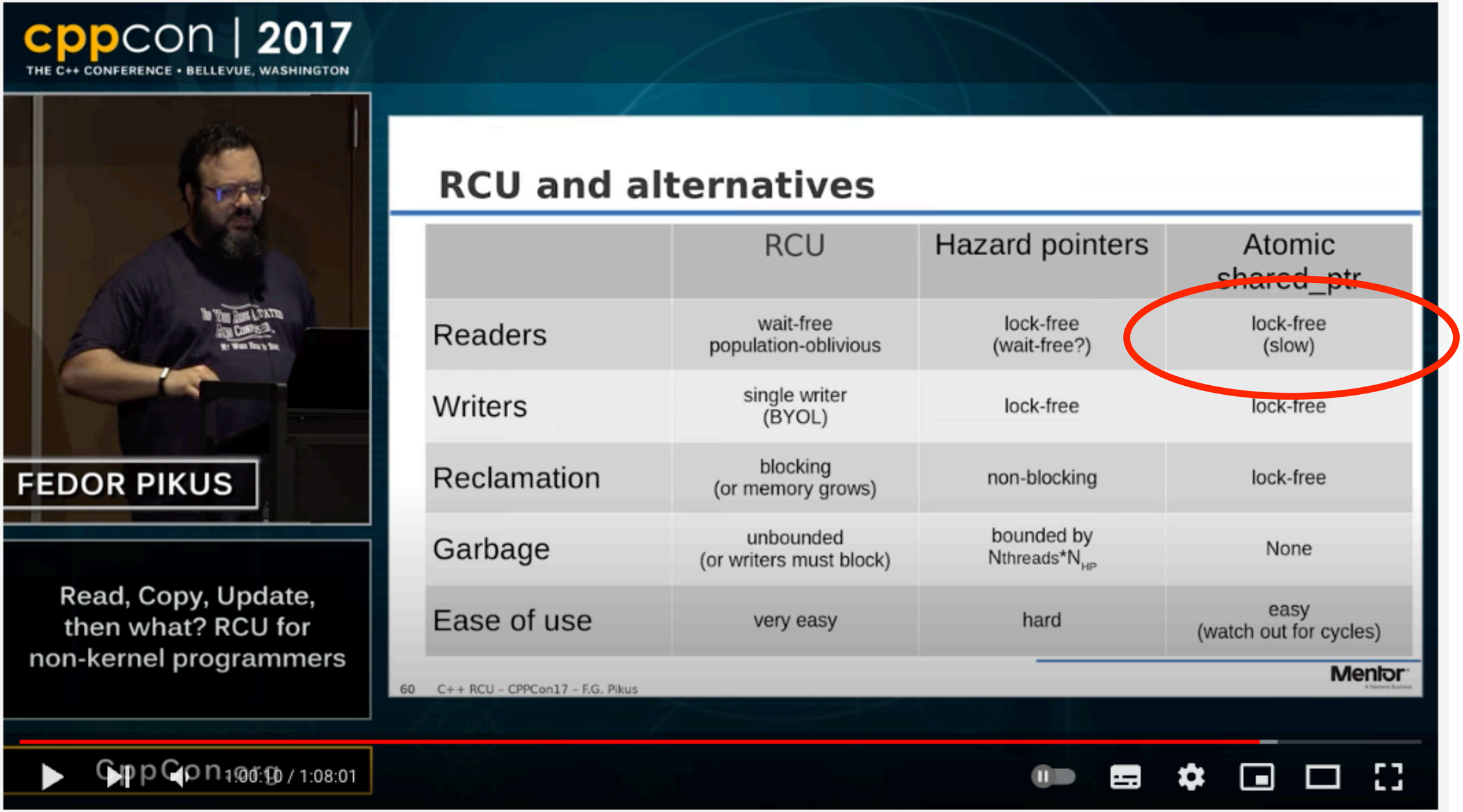

cppcon | 2017
THE C++ CONFERENCE • BELLEVUE, WASHINGTON
FEDOR PIKUS
Read, Copy, Update, then what? RCU for non-kernel programmers
RCU and alternatives
RCU
Hazard pointers
Atomic shared_ptr
Readers
wait-free population-oblivious
lock-free (wait-free?)
lock-free (slow)
Writers
single writer (BYOL)
lock-free
lock-free
Reclamation
blocking (or memory grows)
non-blocking
lock-free
Garbage
unbounded (or writers must block)
bounded by Nthreads*N_HP
None
Ease of use
very easy
hard
easy (watch out for cycles)
60
C++ RCU - CPPCon17 - F.G. Pikus
1:00:10 / 1:08:01

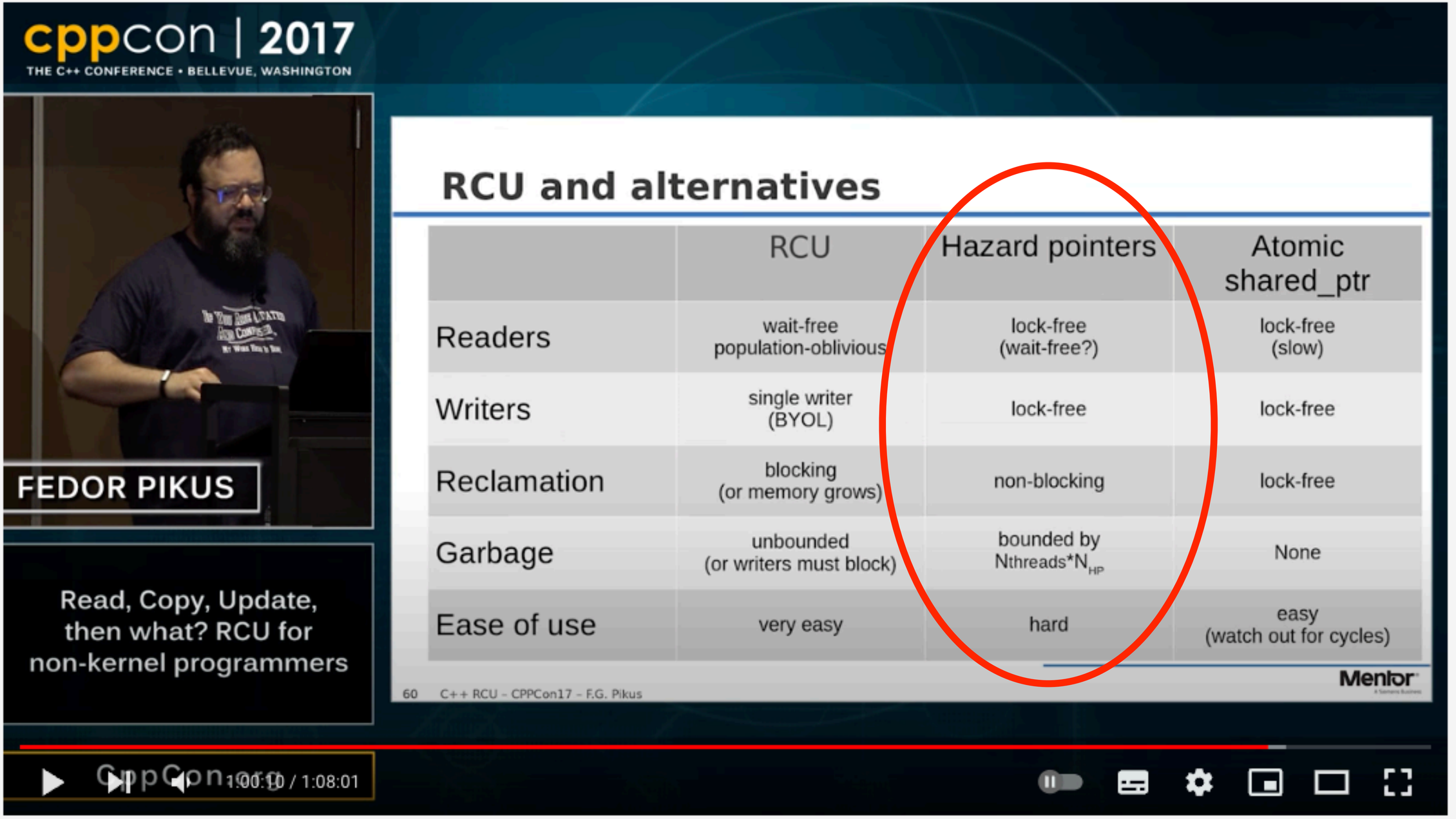
```
// Author: Herb Sutter

template<typename T>
class concurrent_stack {
    struct Node {
        T t;
        shared_ptr<Node> next;
    };
    atomic<shared_ptr<Node>> head;
    concurrent_stack(concurrent_stack&) = delete;
    void operator=(concurrent_stack&) = delete;

public:
    concurrent_stack() = default;
    ~concurrent_stack() = default;

    class reference {
        shared_ptr<Node> p;
    public:
        reference(shared_ptr<Node> p_) : p{p_} {}
        T &operator*() { return p->t; }
        T *operator->() { return &p->t; }
    };

    auto find(T t) const {
        auto p = head.load();
        p = p->next;
        return reference(move(p));
    }

    auto front() const { return reference(head); }

    void push_front(T t) {
        auto p = make_shared<Node>();
        p->t = t;
        p->next = head;
        while(!head.compare_exchange_weak(p->next, p))
            ;
    }

    void pop_front() {
        auto p = head.load();
        while (p && !head.compare_exchange_weak(p, p->next))
            ;
    }
};
```

```
// Author: Herb Sutter

template<typename T>
class concurrent_stack {
    struct Node {
        T t;
        shared_ptr<Node> next;
    };
    atomic<shared_ptr<Node>> head;
    concurrent_stack(concurrent_stack&) = delete;
    void operator=(concurrent_stack&) = delete;

public:
    concurrent_stack() = default;
    ~concurrent_stack() = default;

    class reference {
        shared_ptr<Node> p;
    public:
        reference(shared_ptr<Node> p_) : p{p_} {}
        T &operator*() { return p->t; }
        T *operator->() { return &p->t; }
    };

    auto find(T t) const {
        auto p = head.load();
        p = p->next;
        return reference(move(p));
    }

    auto front() const { return reference(head); }

    void push_front(T t) {
        auto p = make_shared<Node>();
        p->t = t;
        p->next = head;
        while(!head.compare_exchange_weak(p->next, p))
            ;
    }

    void pop_front() {
        auto p = head.load();
        while (p && !head.compare_exchange_weak(p, p->next))
            ;
    }
};
```


2022
A Lock-free Atomic shared_ptr
Timur Doumler
C++ now
0:18 / 1:32:01


class Synthesiser
{
public:
void audioCallback (float* buffer, int bufferSize)
{
std::shared_ptr<Widget> widgetToUse = std::atomic_load (&currentWidget);
// do something with widgetToUse...
}
void updateWidget (/* args */)
{
std::shared_ptr<Widget> newWidget = std::make_shared<Widget> (/* args */);
std::atomic_store (&currentWidget, newWidget);
}
std::shared_ptr<Widget> currentWidget;
};
cppcon
the c++ conference
TIMUR DOUMLER
C++ in the
Audio Industry
46:56 / 1:03:43 • SharedFooter
www.CppCon.org


cppcon | 2017
THE C++ CONFERENCE • BELLEVUE, WASHINGTON
FEDOR PIKUS
Read, Copy, Update, then what? RCU for non-kernel programmers
RCU and alternatives
RCU
Hazard pointers
Atomic shared_ptr
Readers
wait-free population-oblivious
lock-free (wait-free?)
lock-free (slow)
Writers
single writer (BYOL)
lock-free
lock-free
Reclamation
blocking (or memory grows)
non-blocking
lock-free
Garbage
unbounded (or writers must block)
bounded by Nthreads*N_HP
None
Ease of use
very easy
hard
easy (watch out for cycles)
60 C++ RCU – CPPCon17 – F.G. Pikus
Mentor
1:00:10 / 1:08:01

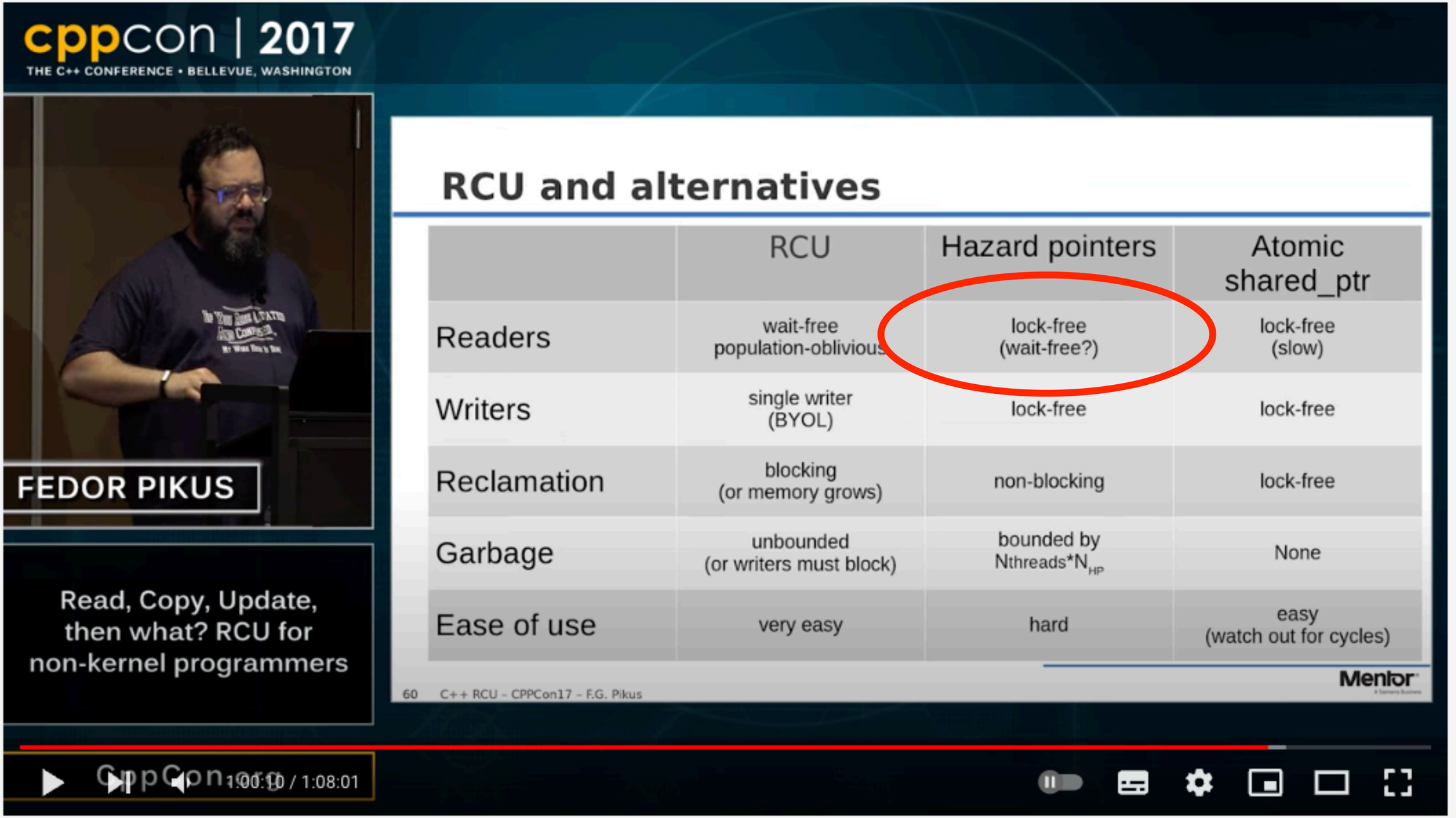

cppcon | 2017
THE C++ CONFERENCE • BELLEVUE, WASHINGTON
FEDOR PIKUS
Read, Copy, Update, then what? RCU for non-kernel programmers
RCU and alternatives
RCU
Hazard pointers
Atomic shared_ptr
Readers
wait-free population-oblivious
lock-free (wait-free?)
lock-free (slow)
Writers
single writer (BYOL)
lock-free
lock-free
Reclamation
blocking (or memory grows)
non-blocking
lock-free
Garbage
unbounded (or writers must block)
bounded by Nthreads*N_HP
None
Ease of use
very easy
hard
easy (watch out for cycles)
60
C++ RCU – CPPCon17 – F.G. Pikus
Mentor
1:00:10 / 1:08:01

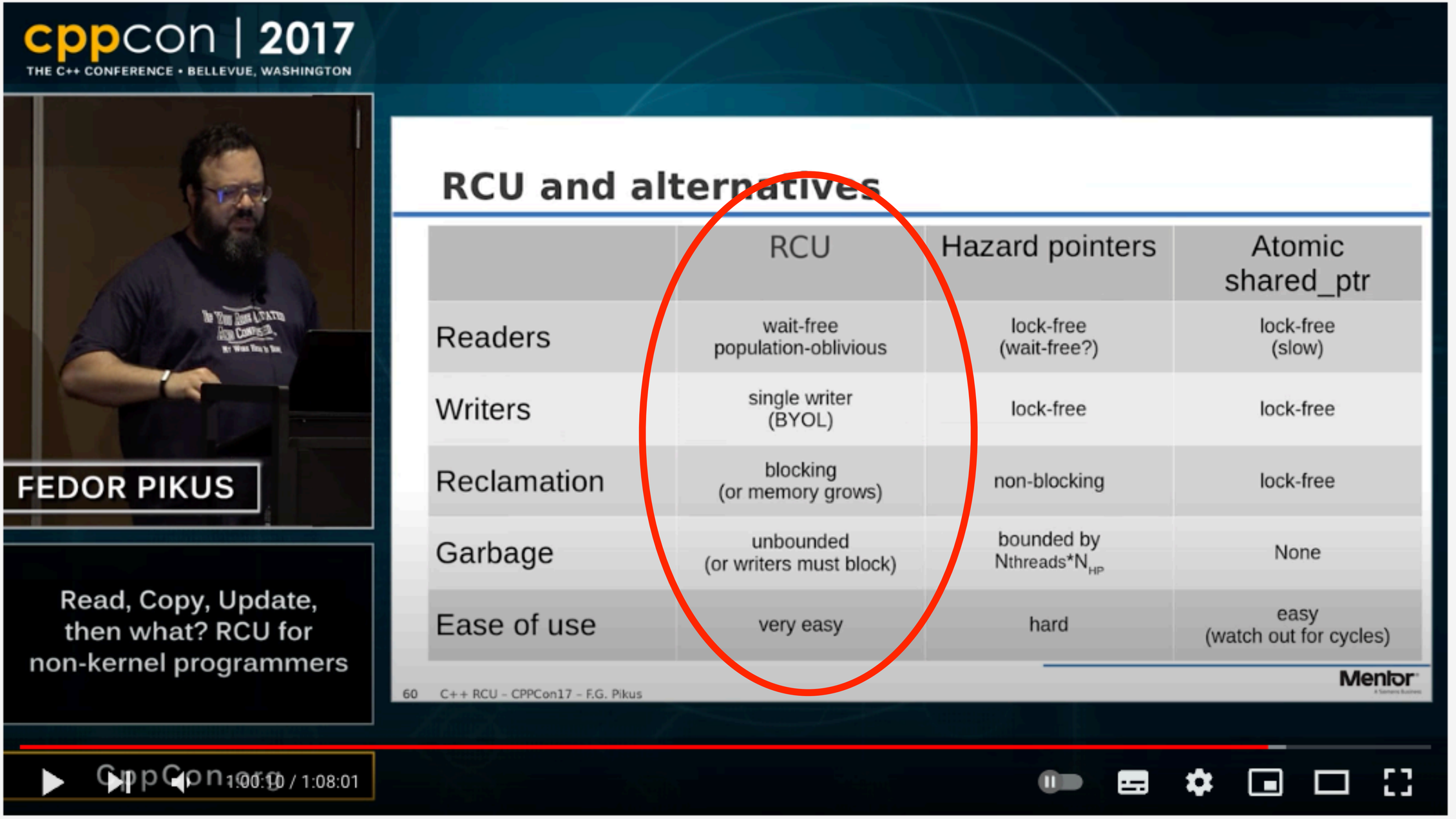
Document Number: P2530R3
Date: 2023-03-02
Project WG21, LWG
Reply to: Maged M. Michael
maged.michael@gmail.com

Hazard Pointers for C++26

Authors:
Maged M. Michael, Michael Wong, Paul McKenney, Andrew Hunter, Daisy S. Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher, Mathias Stearn

Email:
maged.michael@gmail.com, michael@codeplay.com, paulmck@kernel.org, andrewhhunter@gmail.com, dhollman@google.com, cxx@jfbastien.com, hboehm@google.com, davidtgoldblatt@gmail.com, frank.birbacher@gmail.com, redbeard0531+isocpp@gmail.com


cppcon | 2017
THE C++ CONFERENCE • BELLEVUE, WASHINGTON
FEDOR PIKUS
Read, Copy, Update, then what? RCU for non-kernel programmers
RCU and alternatives
RCU
Hazard pointers
Atomic shared_ptr
Readers
wait-free population-oblivious
lock-free (wait-free?)
lock-free (slow)
Writers
single writer (BYOL)
lock-free
lock-free
Reclamation
blocking (or memory grows)
non-blocking
lock-free
Garbage
unbounded (or writers must block)
bounded by Nthreads*N_HP
None
Ease of use
very easy
hard
easy (watch out for cycles)
60
C++ RCU – CPPCon17 – F.G. Pikus
Mentor
1:00:10 / 1:08:01

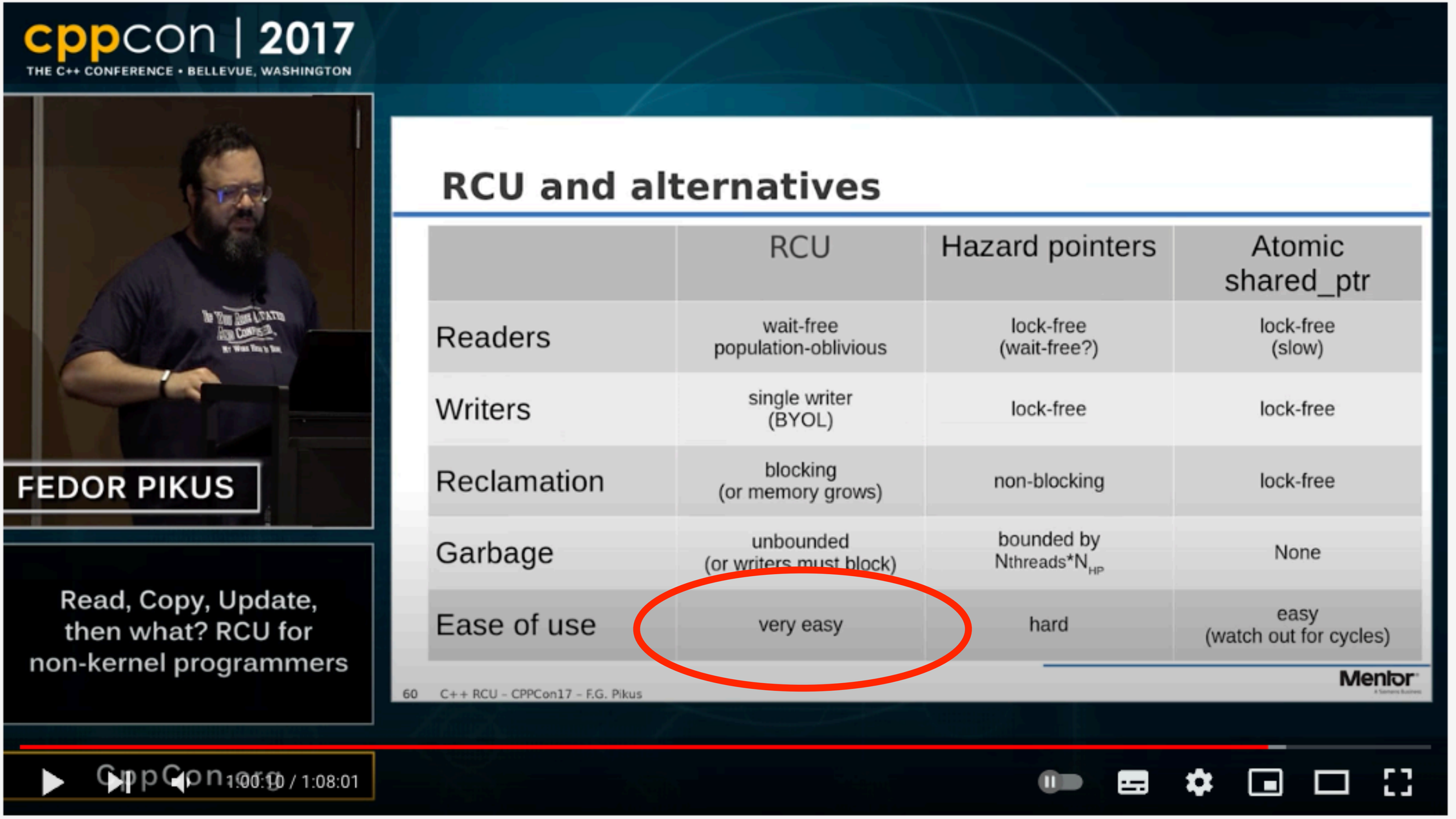

cppcon | 2017
THE C++ CONFERENCE • BELLEVUE, WASHINGTON
FEDOR PIKUS
Read, Copy, Update, then what? RCU for non-kernel programmers
RCU and alternatives
RCU
Hazard pointers
Atomic shared_ptr
Readers
wait-free population-oblivious
lock-free (wait-free?)
lock-free (slow)
Writers
single writer (BYOL)
lock-free
lock-free
Reclamation
blocking (or memory grows)
non-blocking
lock-free
Garbage
unbounded (or writers must block)
bounded by Nthreads*N_HP
None
Ease of use
very easy
hard
easy (watch out for cycles)
60
C++ RCU – CPPCon17 – F.G. Pikus
Mentor
1:00:10 / 1:08:01

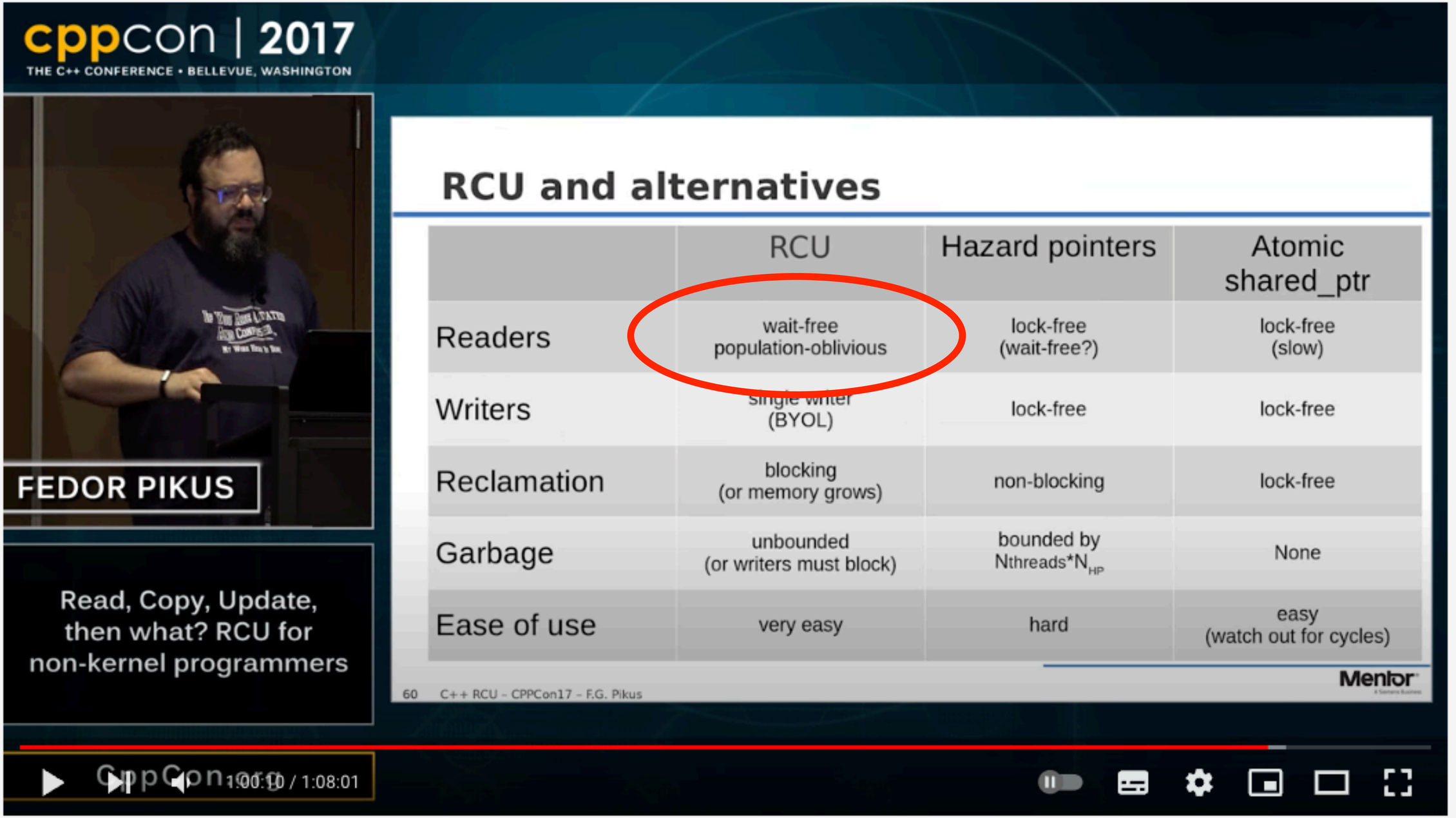
Document Number: P2545R4
Date: 2023-03-08
Revises: None
Reply to: Paul E. McKenney
Meta
paulmckrcu@gmail.com

Read-Copy Update (RCU)

Authors:
Paul McKenney, Michael Wong, Maged M. Michael, Andrew Hunter, Daisy Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher, Erik Rigtorp, Tomasz Kamiński, Olivier Giroux, David Vernet, Timur Doumler

email:
paulmckrcu@gmail.com, michael@codeplay.com, maged.michael@acm.org, andrewhhunter@gmail.com, dhollman@google.com, cxx@jfbastien.com, hboehm@google.com, davidtgoldblatt@gmail.com, frank.birbacher@gmail.com, erik@rigtorp.se, tomaszkam@gmail.com, ogiroux@apple.com, dvernet@meta.com, papers@timur.audio


cppcon | 2017
THE C++ CONFERENCE • BELLEVUE, WASHINGTON
FEDOR PIKUS
Read, Copy, Update, then what? RCU for non-kernel programmers
RCU and alternatives
RCU
Hazard pointers
Atomic shared_ptr
Readers
wait-free population-oblivious
lock-free (wait-free?)
lock-free (slow)
Writers
single writer (BYOL)
lock-free
lock-free
Reclamation
blocking (or memory grows)
non-blocking
lock-free
Garbage
unbounded (or writers must block)
bounded by Nthreads*N_HP
None
Ease of use
very easy
hard
easy (watch out for cycles)
60
C++ RCU – CPPCon17 – F.G. Pikus
Mentor
1:00:10 / 1:08:01


cppcon | 2017
THE C++ CONFERENCE • BELLEVUE, WASHINGTON
FEDOR PIKUS
Read, Copy, Update, then what? RCU for non-kernel programmers
RCU and alternatives
RCU
Hazard pointers
Atomic shared_ptr
Readers
wait-free population-oblivious
lock-free (wait-free?)
lock-free (slow)
Writers
single writer (BYOL)
lock-free
lock-free
Reclamation
blocking (or memory grows)
non-blocking
lock-free
Garbage
unbounded (or writers must block)
bounded by Nthreads*N_HP
None
Ease of use
very easy
hard
easy (watch out for cycles)
60 C++ RCU – CPPCon17 – F.G. Pikus
Mentor
CppCon.org
1:00:10 / 1:08:01

Document number: P0561R6
Date: 2022-01-27
Reply to: Geoff Romer <gromer@google.com>, Andrew Hunter <andrewhhunter@gmail.com>
Audience: Concurrency Study Group, Library Working Group

An RAII Interface for Deferred Reclamation

Background

For purposes of this paper, *deferred reclamation* refers to a pattern of concurrent data sharing with two components: *readers* access the data while holding reader locks, which guarantee that the data will remain live while the lock is held. Meanwhile one or more *updaters* update the data by replacing it with a newly-allocated value. All subsequent readers will see the new value, but the old value is not destroyed until all readers accessing it have released their locks. Readers never block the updater or other readers, and the updater never blocks readers. Updates are inherently costly, because they require allocating and constructing new data values, so they are expected to be rare compared to reads.

```
crill::defer_reclaim_object<biquad_coefficients> coeffs;

void process(audio_buffer& buffer) {
    auto read_ptr = coeffs.lock_read();
    process_biquad(buffer, *read_ptr);
}

void update_coeffs(biquad_coefficients new_coeffs) {
    coeffs.update(new_coeffs);
}
```

```
crill::defer_reclaim_object<biquad_coefficients> coeffs;

void process(audio_buffer& buffer) {
    auto read_ptr = coeffs.lock_read();
    process_biquad(buffer, *read_ptr);
}

void update_coeffs(biquad_coefficients new_coeffs) {
    coeffs.update(new_coeffs);
}
```

```
crill::defer_reclaim_object<biquad_coefficients> coeffs;

void process(audio_buffer& buffer) {
    auto read_ptr = coeffs.lock_read();
    process_biquad(buffer, *read_ptr);
}

void update_coeffs(biquad_coefficients new_coeffs) {
    coeffs.update(new_coeffs);
}
```

```
crill::defer_reclaim_object<biquad_coefficients> coeffs;

void process(audio_buffer& buffer) {
    auto read_ptr = coeffs.lock_read();
    process_biquad(buffer, *read_ptr);
}

void update_coeffs(biquad_coefficients new_coeffs) {
    coeffs.update(new_coeffs);
}
```

```
crill::defer_reclaim_object<biquad_coefficients> coeffs;

void process(audio_buffer& buffer) {
    auto read_ptr = coeffs.lock_read();
    process_biquad(buffer, *read_ptr);
}

void update_coeffs(biquad_coefficients new_coeffs) {
    coeffs.update(new_coeffs);
}

void timer_callback() {
    coeffs.reclaim();
}
```


ADC22
THREAD SYNCHRONISATION IN REAL-TIME AUDIO PROCESSING WITH RCU
(READ-COPY-UPDATE)
TIMUR DOUMLER
0:09 / 55:11
HD

defer_reclaim_object

- Reading always wait-free
- Multiple readers
  - algorithm simplifies greatly if single reader
- read is single atomic load (= no overhead on modern platforms)
- Readers do not block writers
- Writer needs to do heap allocation
- User needs to manage reclamation

Variation: reclaim_on_write_object

- Reading always wait-free
- Multiple readers
  - algorithm simplifies greatly if single reader
- read is single atomic load (= no overhead on modern platforms)
- Writer waits for active reader(s) to finish
- Writing is fast and does not require heap allocation
- No need to manage reclamation

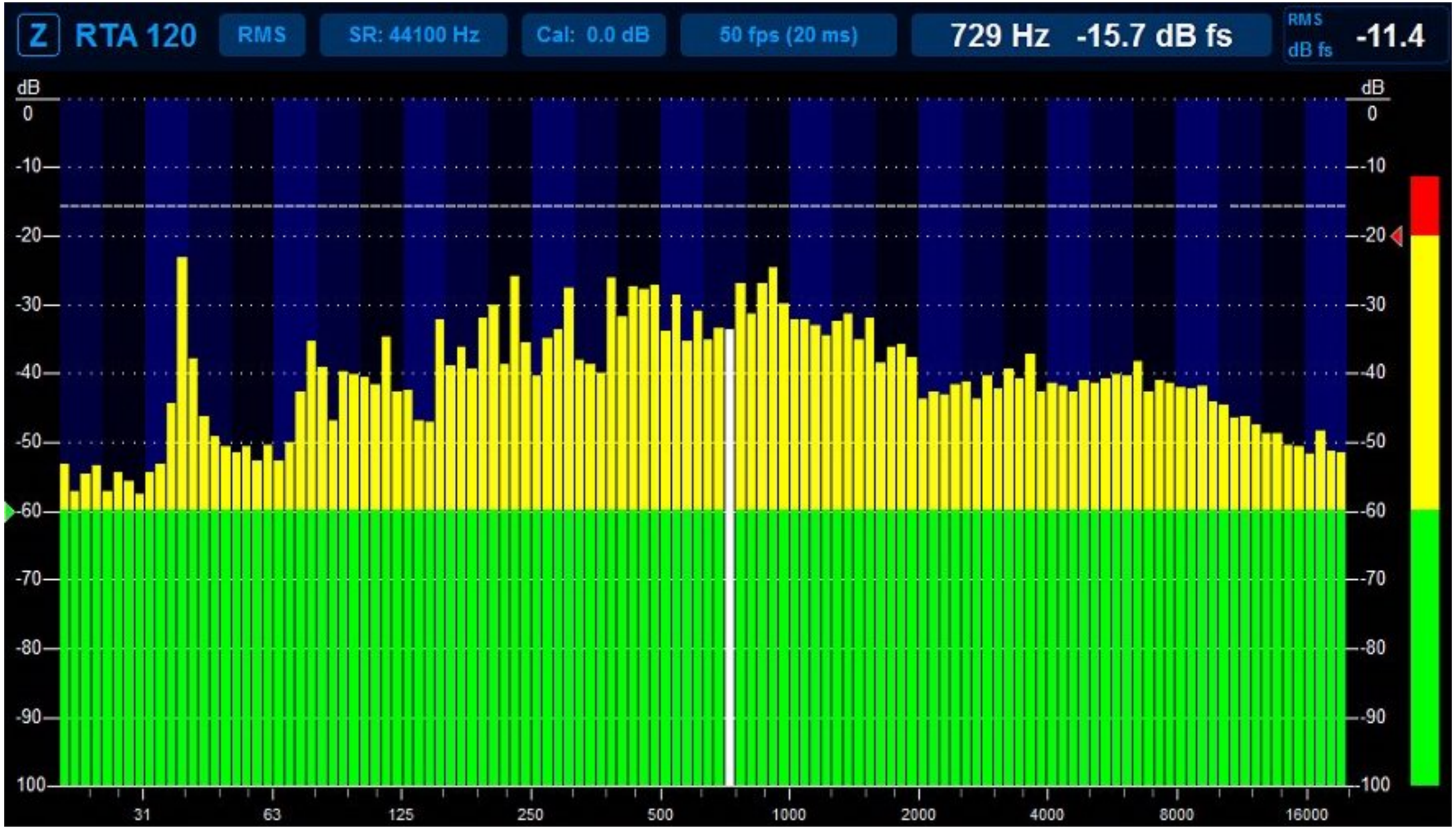
Wait-free concurrency

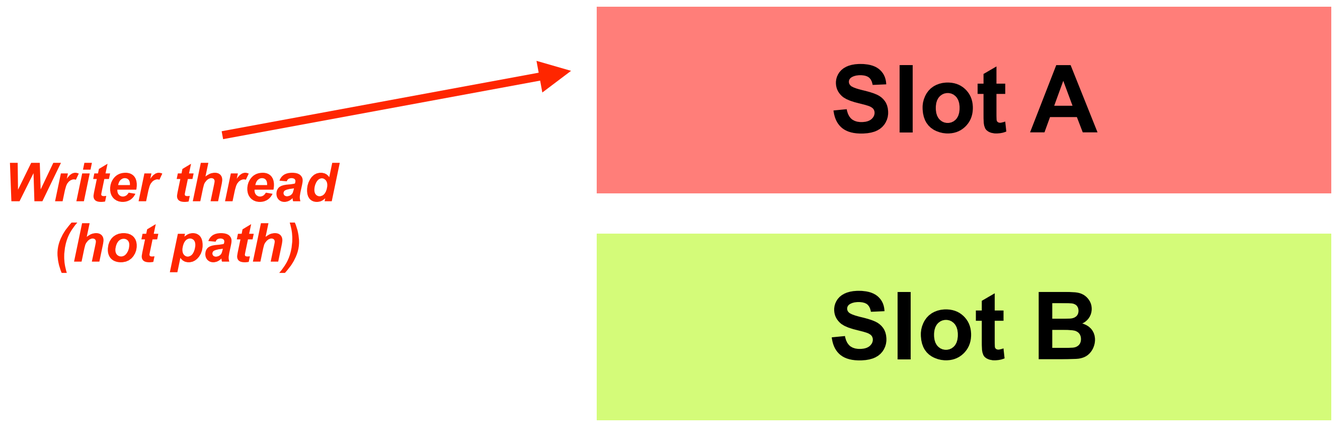
- Passing data to/from "hot path" thread
  - wait-free queue
- Sharing data with "hot path" thread
  - Hot path reads
    - try_lock
    - "spin-on-write"
    - RCU
  - Hot path writes
    - Double-buffering
    - SeqLock

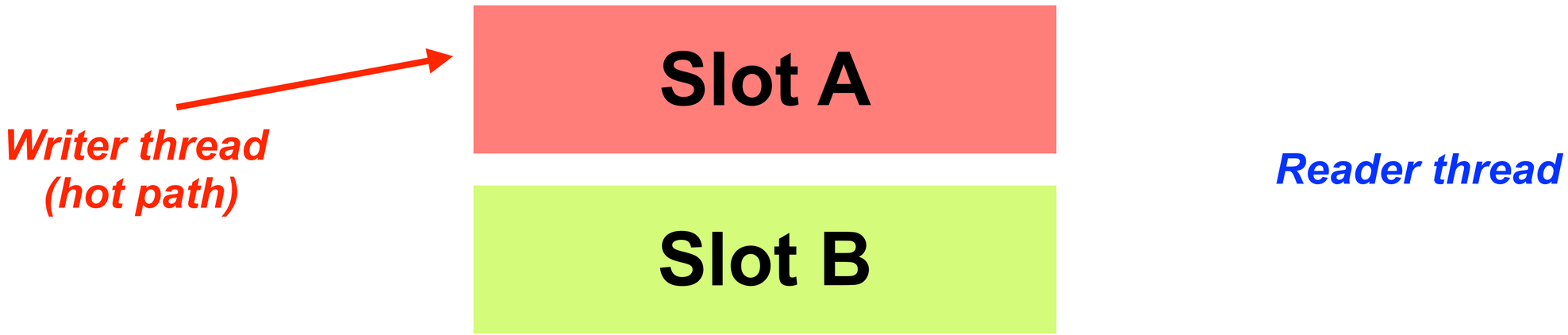
Wait-free concurrency

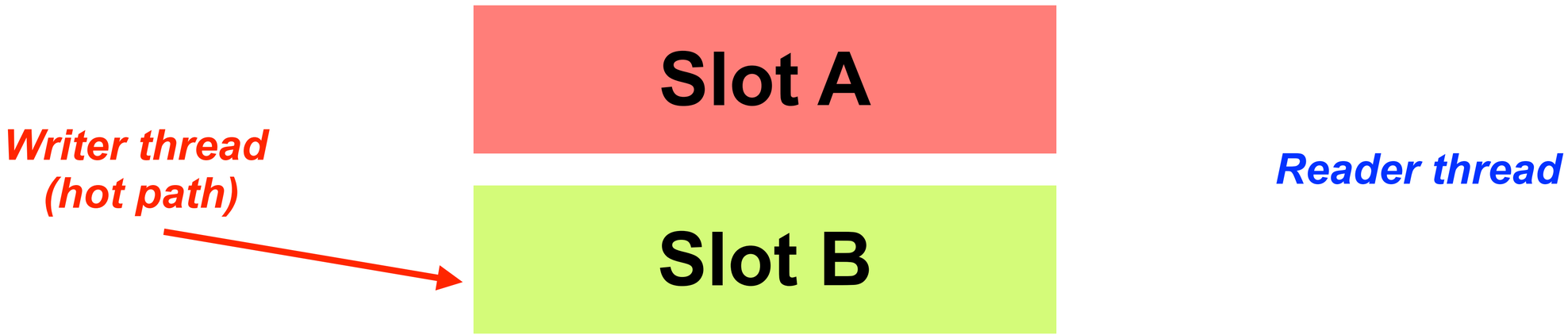
- Passing data to/from "hot path" thread
  - wait-free queue
- Sharing data with "hot path" thread
  - Hot path reads
    - try_lock
    - "spin-on-write"
    - RCU
  - Hot path writes
    - **Double-buffering**
    - SeqLock

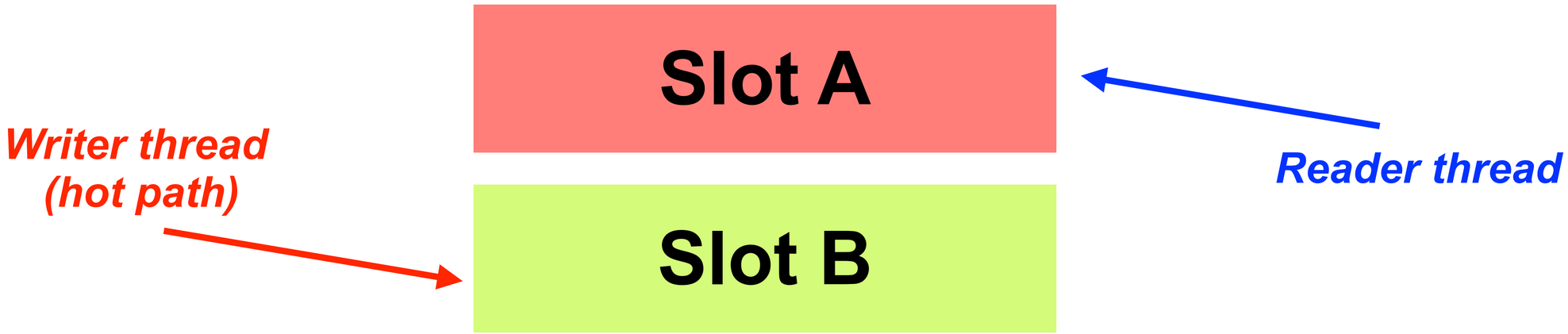

Meeting C++ 2019
David Rowland
Fabian Renn-Giles
Real-time 101
Real-time 101
David Rowland & Fabian Renn-Giles
@drowaudio @hogliux
0:01 / 47:55 • Intro


RTA 120
RMS
SR: 44100 Hz
Cal: 0.0 dB
50 fps (20 ms)
729 Hz -15.7 dB fs
RMS
dB fs
-11.4
dB
0
-10
-20
-30
-40
-50
-60
-70
-80
-90
100
31
63
125
250
500
1000
2000
4000
8000
16000


Writer thread
(hot path)
Slot A
Slot B


Writer thread
(hot path)
Slot A
Slot B
Reader thread


Slot A
Slot B
Writer thread
(hot path)
Reader thread


Slot A
Slot B
Writer thread
(hot path)
Reader thread

```
std::array<frequency_spectrum, 2> slots;
std::atomic<int> idx = {0};

void process(audio_buffer& audio_in) {
    auto spectrum = calculate_spectrum(audio_in);
    int write_idx = idx.load();
    slots[write_idx] = spectrum;
}

void update_spectrum() {
    int read_idx = idx.fetch_xor(1);
    draw_spectrum(slots[read_idx]);
}
```

```
std::array<frequency_spectrum, 2> slots;
std::atomic<int> idx = {0};

void process(audio_buffer& audio_in) {
    auto spectrum = calculate_spectrum(audio_in);
    int write_idx = idx.load();
    slots[write_idx] = spectrum;
}

void update_spectrum() {
    int read_idx = idx.fetch_xor(1);
    draw_spectrum(slots[read_idx]);
}
```

```
std::array<frequency_spectrum, 2> slots;
std::atomic<int> idx = {0};

void process(audio_buffer& audio_in) {
    auto spectrum = calculate_spectrum(audio_in);
    int write_idx = idx.load();
    slots[write_idx] = spectrum;
}

void update_spectrum() {
    int read_idx = idx.fetch_xor(1);
    draw_spectrum(slots[read_idx]);
}
```

```
std::array<frequency_spectrum, 2> slots;
std::atomic<int> idx = {0};

void process(audio_buffer& audio_in) {
    auto spectrum = calculate_spectrum(audio_in);
    int write_idx = idx.load();
    slots[write_idx] = spectrum;
}

void update_spectrum() {
    int read_idx = idx.fetch_xor(1);
    draw_spectrum(slots[read_idx]);
}
```

```
std::array<frequency_spectrum, 2> slots;
std::atomic<int> idx = {0};

enum {
    BIT_IDX = (1 << 0),
    BIT_NEWDATA = (1 << 1),
    BIT_BUSY = (1 << 2)
};

void process(audio_buffer& audio_in) {
    auto spectrum = calculate_spectrum(audio_in);
    int write_idx = idx.fetch_or(BIT_BUSY) & BIT_IDX;
    slots[write_idx] = spectrum;
    idx.store ((write_idx & BIT_IDX) | BIT_NEWDATA);
}

void update_spectrum() {
    int read_idx = idx.fetch_xor(1);
    if ((read_idx & BIT_NEWDATA) != 0) {
        int new_idx;
        do {
            read_idx &= ~BIT_BUSY;
```

```
std::array<frequency_spectrum, 2> slots;
std::atomic<int> idx = {0};

enum {
    BIT_IDX = (1 << 0),
    BIT_NEWDATA = (1 << 1),
    BIT_BUSY = (1 << 2)
};

void process(audio_buffer& audio_in) {
    auto spectrum = calculate_spectrum(audio_in);
    int write_idx = idx.fetch_or(BIT_BUSY) & BIT_IDX;
    slots[write_idx] = spectrum;
    idx.store ((write_idx & BIT_IDX) | BIT_NEWDATA);
}

void update_spectrum() {
    int read_idx = idx.fetch_xor(1);
    if ((read_idx & BIT_NEWDATA) != 0) {
        int new_idx;
        do {
            read_idx &= ~BIT_BUSY;
```

```
std::array<frequency_spectrum, 2> slots;
std::atomic<int> idx = {0};

enum {
    BIT_IDX = (1 << 0),
    BIT_NEWDATA = (1 << 1),
    BIT_BUSY = (1 << 2)
};

void process(audio_buffer& audio_in) {
    auto spectrum = calculate_spectrum(audio_in);
    int write_idx = idx.fetch_or(BIT_BUSY) & BIT_IDX;
    slots[write_idx] = spectrum;
    idx.store ((write_idx & BIT_IDX) | BIT_NEWDATA);
}

void update_spectrum() {
    int read_idx = idx.fetch_xor(1);
    if ((read_idx & BIT_NEWDATA) != 0) {
        int new_idx;
        do {
            read_idx &= ~BIT_BUSY;
```

```
};

void process(audio_buffer& audio_in) {
    auto spectrum = calculate_spectrum(audio_in);
    int write_idx = idx.fetch_or(BIT_BUSY) & BIT_IDX;
    slots[write_idx] = spectrum;
    idx.store ((write_idx & BIT_IDX) | BIT_NEWDATA);
}

void update_spectrum() {
    int read_idx = idx.fetch_xor(1);
    if ((read_idx & BIT_NEWDATA) != 0) {
        int new_idx;
        do {
            read_idx &= ~BIT_BUSY;
            new_idx = (read_idx ^ BIT_IDX) & BIT_IDX;
        } while (! idx.compare_exchange_weak(read_idx, new_idx));
        read_idx = new_idx;
    }
    draw_spectrum(slots[(read_idx & BIT_IDX) ^ 1]);
}
```

```
};

void process(audio_buffer& audio_in) {
    auto spectrum = calculate_spectrum(audio_in);
    int write_idx = idx.fetch_or(BIT_BUSY) & BIT_IDX;
    slots[write_idx] = spectrum;
    idx.store ((write_idx & BIT_IDX) | BIT_NEWDATA);
}

void update_spectrum() {
    int read_idx = idx.fetch_xor(1);
    if ((read_idx & BIT_NEWDATA) != 0) {
        int new_idx;
        do {
            read_idx &= ~BIT_BUSY;
            new_idx = (read_idx ^ BIT_IDX) & BIT_IDX;
        } while (! idx.compare_exchange_weak(read_idx, new_idx));
        read_idx = new_idx;
    }
    draw_spectrum(slots[(read_idx & BIT_IDX) ^ 1]);
}
```

```
};

void process(audio_buffer& audio_in) {
    auto spectrum = calculate_spectrum(audio_in);
    int write_idx = idx.fetch_or(BIT_BUSY) & BIT_IDX;
    slots[write_idx] = spectrum;
    idx.store ((write_idx & BIT_IDX) | BIT_NEWDATA);
}

void update_spectrum() {
    int read_idx = idx.fetch_xor(1);
    if ((read_idx & BIT_NEWDATA) != 0) {
        int new_idx;
        do {
            read_idx &= ~BIT_BUSY;
            new_idx = (read_idx ^ BIT_IDX) & BIT_IDX;
        } while (! idx.compare_exchange_weak(read_idx, new_idx));
        read_idx = new_idx;
    }
    draw_spectrum(slots[(read_idx & BIT_IDX) ^ 1]);
}
```

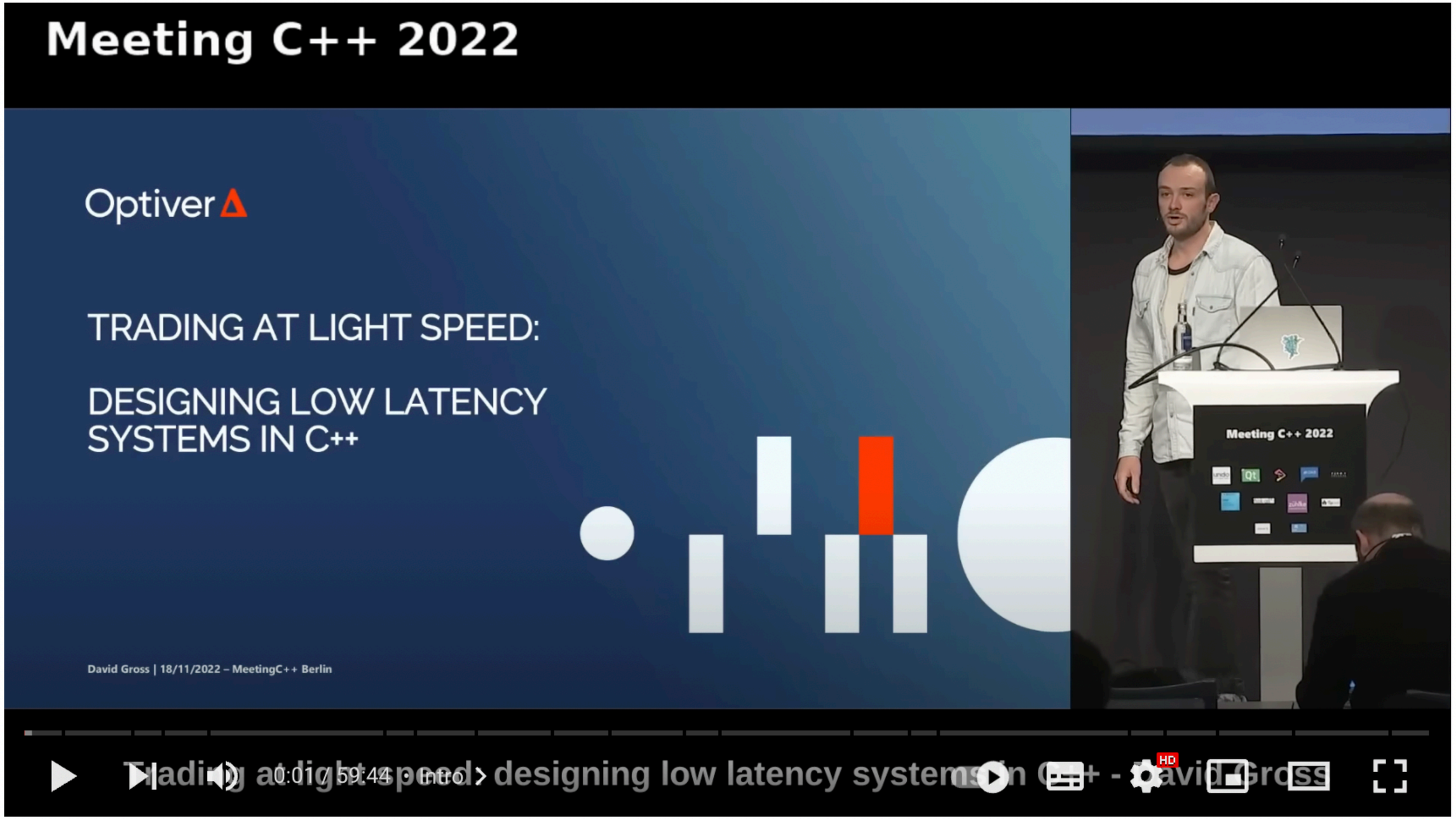
```
};

void process(audio_buffer& audio_in) {
    auto spectrum = calculate_spectrum(audio_in);
    int write_idx = idx.fetch_or(BIT_BUSY) & BIT_IDX;
    slots[write_idx] = spectrum;
    idx.store ((write_idx & BIT_IDX) | BIT_NEWDATA);
}

void update_spectrum() {
    int read_idx = idx.fetch_xor(1);
    if ((read_idx & BIT_NEWDATA) != 0) {
        int new_idx;
        do {
            read_idx &= ~BIT_BUSY;
            new_idx = (read_idx ^ BIT_IDX) & BIT_IDX;
        } while (! idx.compare_exchange_weak(read_idx, new_idx));
        read_idx = new_idx;
    }
    draw_spectrum(slots[(read_idx & BIT_IDX) ^ 1]);
}
```

Double-buffering

- Writing always wait-free
- Good solution if writing happens continuously, lots of data
- Tradeoffs:
  - Single reader, single writer
  - Writer blocks reader (has to spin on a CAS loop)
  - Overhead of copying data on both reader & writer

Wait-free concurrency

- Passing data to/from "hot path" thread
  - wait-free queue
- Sharing data with "hot path" thread
  - Hot path reads
    - try_lock
    - "spin-on-write"
    - RCU
  - Hot path writes
    - Double-buffering
    - **SeqLock**


Meeting C++ 2022
Optiver
TRADING AT LIGHT SPEED:
DESIGNING LOW LATENCY SYSTEMS IN C++
David Gross | 18/11/2022 – MeetingC++ Berlin
Meeting C++ 2022
0:01 / 59:44 • Intro
Trading at light speed: designing low latency systems in C++ - David Gross

```
std::atomic<std::size_t> seq = 0;

void store(T t) noexcept {
    std::size_t old_seq = seq.fetch_add(1);

    // write data...

    seq.store(old_seq + 2);
}
```

```
std::atomic<std::size_t> seq = 0;

void store(T t) noexcept {
    std::size_t old_seq = seq.fetch_add(1);

    // write data...

    seq.store(old_seq + 2);
}

bool try_load(T& t) const noexcept {
    std::size_t seq1 = seq.load();
    if (seq1 % 2 != 0)
        return false;

    // read data...

    std::size_t seq2 = seq.load(std::memory_order_relaxed);
    return seq1 == seq2;
}
```

```
std::atomic<std::size_t> seq = 0;

void store(T t) noexcept {
    std::size_t old_seq = seq.load();
    seq.store(old_seq + 1);

    // write data...

    seq.store(old_seq + 2);
}

bool try_load(T& t) const noexcept {
    std::size_t seq1 = seq.load();
    if (seq1 % 2 != 0)
        return false;

    // read data...

    std::size_t seq2 = seq.load(std::memory_order_relaxed);
    return seq1 == seq2;
}
```

```
std::atomic<std::size_t> seq = 0;

void store(T t) noexcept {
    std::size_t old_seq = seq.load(std::memory_order_relaxed);
    seq.store(old_seq + 1, std::memory_order_relaxed);

    std::atomic_thread_fence(std::memory_order_release);
    // write data...

    seq.store(old_seq + 2, std::memory_order_release);
}

bool try_load(T& t) const noexcept {
    std::size_t seq1 = seq.load(std::memory_order_acquire);
    if (seq1 % 2 != 0)
        return false;

    // read data...
    std::atomic_thread_fence(std::memory_order_acquire);

    std::size_t seq2 = seq.load(std::memory_order_relaxed);
    return seq1 == seq2;
}
```

```
std::atomic<std::size_t> seq = 0;

void store(T t) noexcept {
    std::size_t old_seq = seq.load(std::memory_order_relaxed);
    seq.store(old_seq + 1, std::memory_order_relaxed);

    std::atomic_thread_fence(std::memory_order_release);
    // write data...

    seq.store(old_seq + 2, std::memory_order_release);
}

bool try_load(T& t) const noexcept {
    std::size_t seq1 = seq.load(std::memory_order_acquire);
    if (seq1 % 2 != 0)
        return false;

    // read data...
    std::atomic_thread_fence(std::memory_order_acquire);

    std::size_t seq2 = seq.load(std::memory_order_relaxed);
    return seq1 == seq2;
}
```

Doc. No.: WG21/P1478R8
Date: 2022-11-9
Reply-to: Hans-J. Boehm
Email: hboehm@google.com
Authors: Hans-J. Boehm
Audience: LWG
Target: Concurrency TS 2

P1478R8: Byte-wise atomic memcpy

Several prior papers have suggested mechanisms that allow for nonatomic accesses that behave like atomics in some way. There are several possible use cases. Here we focus on seqlocks which, in our experience, seem to generate the strongest demand for such a feature.

This proposal is intended to be as simple as possible. It is in theory, but only in theory, a pure library facility that can be implemented without compiler support. We expect that practical implementations will implement the new facilities as aliases for existing `memcpy` implementations. This cannot be done by the user in portable code, since it requires additional assumptions about the memcpy implementation. Hence there is a strong argument for including it in the standard library.

There have been a number of prior proposals in this space. Most recently P0690 suggested "tearable atomics". Other solutions were proposed in N3710, which suggested more complex handling for speculative nonatomics loads. This proposal is closest in title to P0603. In a sense this returns to the original intent of that proposal, and is arguably the simplest and narrowest proposal.

Can Seqlocks Get Along with Programming Language Memory Models?

Hans-J. Boehm
HP Laboratories
Hans.Boehm@hp.com

Abstract

Seqlocks are an important synchronization mechanism and represent a significant improvement over conventional reader-writer locks in some contexts. They avoid the need to update a synchronization variable during a reader critical section, and hence improve performance by avoiding cache coherence misses on the lock object itself. Unfortunately, they rely on speculative racing loads inside the critical section. This makes them an interesting problem case for programming-language-level memory models that emphasize data-race-free programming. We analyze a variety of implementation alternatives within the C++11 memory model, and briefly address the corresponding issue in Java. In the process, we observe that there may be a use for "read-dont-modify-write" operations, i.e. read-modify-write operations that atomically write back the original value, without modifying it, solely for the memory model consequences, and that it may be useful for compilers to optimize such operations.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming; D.3.2 [*Programming Languages*]: Language Classifications—C++; D.4.1 [*Operating Systems*]: Process Management—Mutual exclusion

rently by multiple threads, there are several alternatives to protecting them:

mutexes or traditional reader-writer locks These do not take advantage of infrequent writes. They further have the crucial disadvantage that even read accesses update the state of the lock. Thus if many threads try to read the data at the same time, they will still generate cache contention, as each thread in turn tries to obtain exclusive access to the cache line holding the lock. This is true even in the case of a conventional reader-writer lock, where the actual data read accesses can normally proceed in parallel on a shared cache line.

RCU An alternative mechanism is to encapsulate all the shared data in a separately allocated object, which is referenced by a globally accessible pointer p. Read accesses dereference p and access the shared data. Write accesses construct a copy of the data and atomically replace p. Reads may read slightly stale data, since they may continue to follow a previously read version of p. A non-trivial protocol [16] or a garbage collector is required to eventually reclaim the separately allocated objects.

seqlocks The data can be "protected" by a sequence number. The sequence number starts at zero, and is incremented before and after writing the object. Each reader checks

SeqLock

- Writing always wait-free
- Good solution if writing happens more rarely, data not too large
- Tradeoffs:
  - Single writer, multiple readers
  - Readers lock-free but not wait-free
    (might have to retry unbounded nr of times)
  - Data must be trivially copyable
  - Overhead of copying data *atomically* on both reader & writer

Wait-free concurrency

- Passing data to/from "hot path" thread
  - wait-free queue
- Sharing data with "hot path" thread
  - Hot path reads
    - try_lock
    - "spin-on-write"
    - RCU
  - Hot path writes
    - Double-buffering
    - SeqLock

What not to do in the hot path:

- Dynamic memory allocations/deallocations
- blocking the thread
- **I/O**
- exceptions
- context switches / mode switches (user/kernel space)
- syscalls
- calling into unknown code
- loops without definite bounds
- algorithms > *O*(1), or with no statically known upper bound on *N*

I/O from the hot path

- With other threads: Push message into wait-free spsc queue
- With other processes: Shared memory
- With hardware: Direct Memory Access (DMA)

Reading data from disk on the hot path


Multi Rack
01. Concert Strings
01-16
17-32
33-48
49-64
KSP
aux
12. Full String Ensemble Sm
Output: st. 1
Voices: 0 Max: 200
Purge
MIDI Ch: [A] 1
Memory: 161.75 MB
Tune
0.00
AMADEUS
SYMPHONIC ORCHESTRA
STRING ENSEMBLE
Attack
Vel → Att
Release
ARTICULATIONS
Sustains C-1
Staccato C#-1
Marcato D-1
Sforzando D#-1
Spiccato E-1
Pizzicato F-1
Bartok Pizzicato F#-1
Half Trill G-1
WholeTrill G#-1
Tremolo A-1
LEGATO
SET KEY SW
KS
KS
PERFORMANCE
Volume MIDI CC
1 Mod Wheel
CC Sensitivity
Vel Sensitivity
Vibrato Control
VIB
NV-VIB
X-Fade NV-VB
EQUALIZER
LOW
MID
HIGH
AMBIENCE
Hall
ON/OFF
AMOUNT
SIZE
FILTER
LO PASS
HI PASS
+1

Reading data from disk on the hot path

- Pre-load into RAM and lock address range (to prevent swap-out)
  - `mlock` (POSIX)
  - `VirtualLock` (Windows)
- Disk streaming
  - Pre-load into RAM first ~ 100 ms of every possible sound
  - Once it starts playing, start filling in the rest from disk
    (on a background thread)

What not to do in the hot path:

- Dynamic memory allocations/deallocations
- blocking the thread
- I/O
- **exceptions**
- context switches / mode switches (user/kernel space)
- syscalls
- calling into unknown code
- loops without definite bounds
- algorithms > *O*(1), or with no statically known upper bound on *N*

Zero-overhead deterministic exceptions: Throwing values

Document Number: **P0709 R4** Date: 2019-08-04
Reply-to: Herb Sutter (hsutter@microsoft.com)
Audience: EWG, LEWG

R4: All sections, but esp. the design in §4.3 (allocation failure), are updated with LEWG+EWG Cologne feedback.

Abstract

Divergent error handling has fractured the C++ community into incompatible dialects, because of long-standing unresolved problems in C++ exception handling. This paper enumerates four interrelated problems in C++ error handling. Although these could be four papers, I believe it is important to consider them together.

§4.1: "C++" projects commonly ban exceptions, because today's dynamic exception types violate the zero-overhead principle, and do not have statically boundable space and time costs. In particular, `throw` requires dynamic allocation and `catch` of a type requires RTTI. — We must at minimum enable all C++ projects to enable exception handling and to use the standard language and library. This paper proposes extending C++'s exception handling to let functions declare that they throw a *statically known type by value*, so that the implementation can opt into an efficient implementation (a compatible ABI extension). Code that uses only this efficient exception handling has zero space and time overhead compared to returning error codes.

Zero-overhead deterministic exceptions: Throwing values

Document Number: **P0709 R4** Date: 2019-08-04
Reply-to: Herb Sutter (hsutter@microsoft.com)
Audience: EWG, LEWG

R4: All sections, but esp. the design in §4.3 (allocation failure), are updated with LEWG+EWG Cologne feedback.

Abstract

Divergent error handling has fractured the C++ community into incompatible dialects, because of long-standing unresolved problems in C++ exception handling. This paper enumerates four interrelated problems in C++ error handling. Although these could be four papers, I believe it is important to consider them together.

§4.1: "C++" projects commonly ban exceptions, because today's dynamic exception types violate the zero-overhead principle, and do not have statically boundable space and time costs. In particular, `throw` requires dynamic allocation and `catch` of a type requires RTTI. — We must at minimum enable all C++ projects to enable exception handling and to use the standard language and library. This paper proposes extending C++'s exception handling to let functions declare that they throw a *statically known type by value*, so that the implementation can opt into an efficient implementation (a compatible ABI extension). Code that uses only this efficient exception handling has zero space and time overhead compared to returning error codes.

Error handling

- No exceptions!
  - either compile with no exceptions
  - or ban exceptions from "hot path"
- Error codes (ugly)
- std::optional (no info about error)
- std::expected (since C++23)

Error handling

- No exceptions!
  - either compile with no exceptions
  - or ban exceptions from "hot path"
- Error codes (ugly)
- std::optional (no info about error)
- std::expected (since C++23)

Timur Doumler: "How C++23 changes the way we write code" (MeetingC++ 2022)

What not to do in the hot path:

- Dynamic memory allocations/deallocations
- blocking the thread
- I/O
- exceptions
- **context switches / mode switches (user/kernel space)**
- syscalls
- calling into unknown code
- loops without definite bounds
- algorithms > *O*(1), or with no statically known upper bound on *N*

Avoiding context switches/mode switches

- Mainstream operating systems: Thread priority

Avoiding context switches/mode switches

- Mainstream operating systems: Thread priority
- Real-time operating systems: deterministic thread scheduler

Avoiding context switches/mode switches

- Mainstream operating systems: Thread priority
- Real-time operating systems: deterministic thread scheduler
- If you control the hardware:
  - Kernel bypass

Avoiding context switches/mode switches

- Mainstream operating systems: Thread priority
- Real-time operating systems: deterministic thread scheduler
- If you control the hardware:
  - Kernel bypass
- If your hot path is in a single thread,
  and you don't care about efficiency of other threads:
  - Turn off hyperthreading
  - Pin hot path thread to one CPU core

What not to do in the hot path:

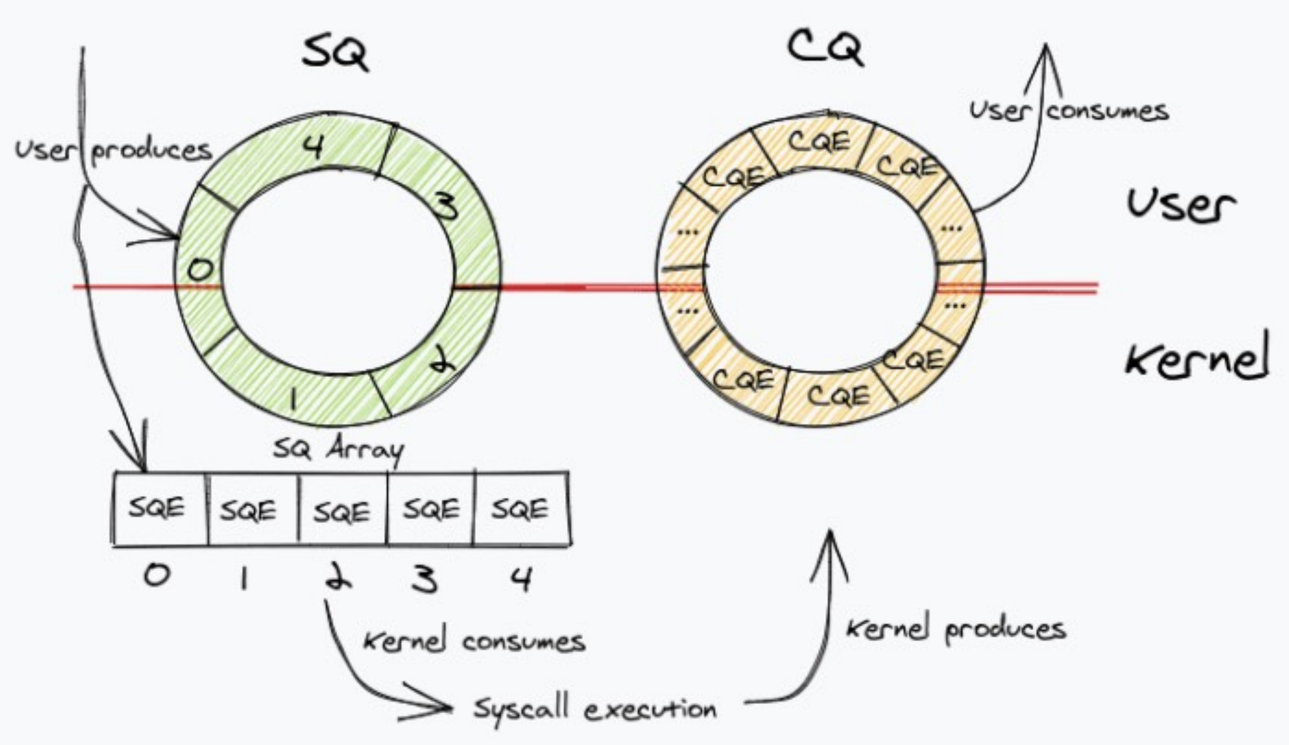
- Dynamic memory allocations/deallocations
- blocking the thread
- I/O
- exceptions
- context switches / mode switches (user/kernel space)
- **syscalls**
- calling into unknown code
- loops without definite bounds
- algorithms > *O*(1), or with no statically known upper bound on *N*

Avoiding syscalls

- Userspace libraries

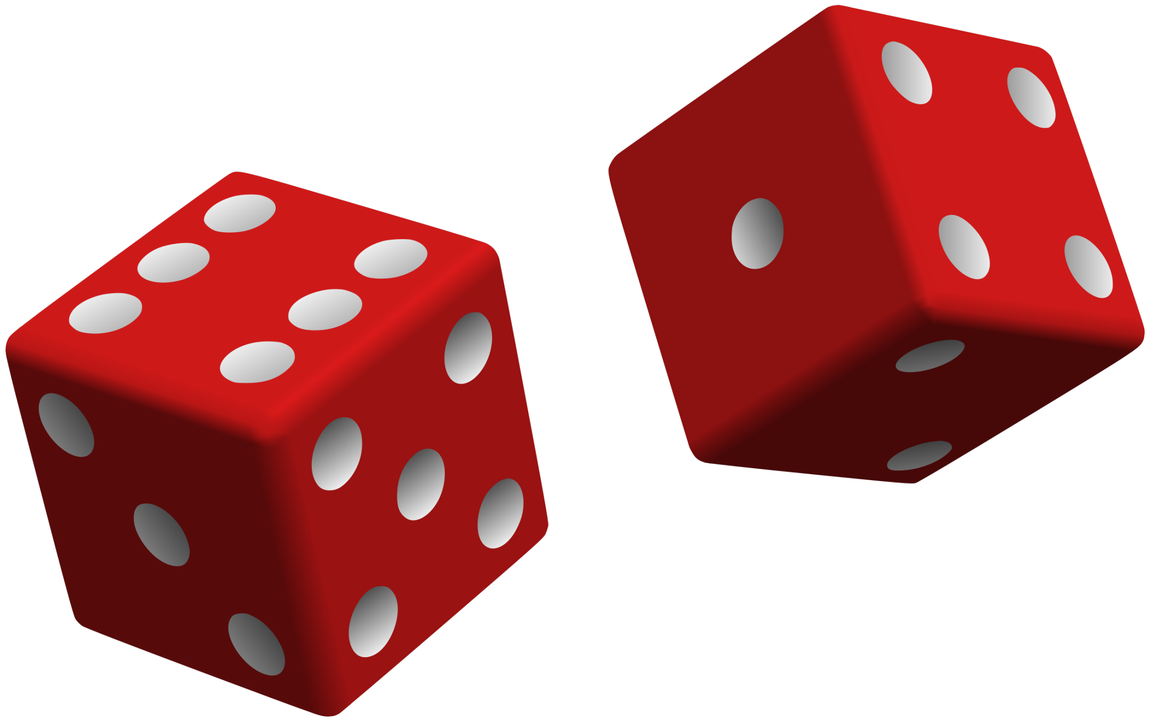
Avoiding syscalls

- Userspace libraries
- Linux: io_uring?



What not to do in the hot path:

- Dynamic memory allocations/deallocations
- blocking the thread
- I/O
- exceptions
- context switches / mode switches (user/kernel space)
- syscalls
- **calling into unknown code**
- **loops without definite bounds**
- **algorithms > *O*(1), or with no statically known upper bound on *N***



```
// returns a random float in the interval [0, 1)
float get_random_sample()
{
    return float(std::rand()) / float(INT_MAX);
}

void process(buffer& b)
{
   // fill buffer with random white noise:
   std::ranges::fill(b, get_random_sample);
}
```

```
// returns a random float in the interval [0, 1)
float get_random_sample()
{
    return float(std::rand()) / float(INT_MAX);
}

void process(buffer& b)
{
   // fill buffer with random white noise:
   std::ranges::fill(b, get_random_sample);
}
```

26 Numerics library [numerics]

26.6 Random number generation [rand]

26.6.10 Low-quality random number generation [c.math.rand]

1 [*Note 1*: The header `<cstdlib>` declares the functions described in this subclause. — *end note*]

```
int rand();
void srand(unsigned int seed);
```

2 *Effects*: The `rand` and `srand` functions have the semantics specified in the C standard library.

3 *Remarks*: The implementation may specify that particular library functions may call `rand`. It is implementation-defined whether the `rand` function may introduce data races ([res.on.data.races]).

[*Note 2*: The other random number generation facilities in this document ([rand]) are often preferable to `rand`, because `rand`'s underlying algorithm is unspecified. Use of `rand` therefore continues to be non-portable, with unpredictable and oft-questionable quality and performance. — *end note*]

SEE ALSO: ISO C 7.22.2

26 Numerics library [numerics]

26.6 Random number generation [rand]

26.6.10 Low-quality random number generation [c.math.rand]

1 [*Note 1*: The header `<cstdlib>` declares the functions described in this subclause. — *end note*]

```
int rand();
void srand(unsigned int seed);
```

2 *Effects*: The `rand` and `srand` functions have the semantics specified in the C standard library.

3 *Remarks*: The implementation may specify that particular library functions may call `rand`. It is implementation-defined whether the `rand` function may introduce data races ([res.on.data.races]).

[*Note 2*: The other random number generation facilities in this document ([rand]) are often preferable to `rand`, because `rand`'s underlying algorithm is unspecified. Use of `rand` therefore continues to be non-portable, with unpredictable and oft-questionable quality and performance. — *end note*]

SEE ALSO: ISO C 7.22.2

26 Numerics library [numerics]

26.6 Random number generation [rand]

26.6.10 Low-quality random number generation [c.math.rand]

1 [*Note 1*: The header `<cstdlib>` declares the functions described in this subclause. — *end note*]

```
int rand();
void srand(unsigned int seed);
```

2 *Effects*: The `rand` and `srand` functions have the semantics specified in the C standard library.

3 *Remarks*: The implementation may specify that particular library functions may call `rand`. It is implementation-defined whether the `rand` function may introduce data races ([res.on.data.races]).

[*Note 2*: The other random number generation facilities in this document ([rand]) are often preferable to `rand`, because `rand`'s underlying algorithm is unspecified. Use of `rand` therefore continues to be non-portable, with unpredictable and oft-questionable quality and performance. — *end note*]

SEE ALSO: ISO C 7.22.2

```
// C++ random number generators:

mersenne_twister_engine
linear_congruential_engine
subtract_with_carry_engine
```

1 A *uniform random bit generator* `g` of type `G` is a function object returning unsigned integer values such that each value in the range of possible results has (ideally) equal probability of being returned.

[*Note 1*: The degree to which `g`'s results approximate the ideal is often determined statistically. — *end note*]

```
template<class G>
  concept uniform_random_bit_generator =
    invocable<G&> && unsigned_integral<invoke_result_t<G&>> &&
    requires {
      { G::min() } -> same_as<invoke_result_t<G&>>;
      { G::max() } -> same_as<invoke_result_t<G&>>;
      requires bool_constant<(G::min() < G::max())>::value;
    };
```

2 Let `g` be an object of type `G`. `G` models `uniform_random_bit_generator` only if

(2.1) — `G::min() <= g()`,

(2.2) — `g() <= G::max()`, and

(2.3) — `g()` has amortized constant complexity.

3 A class `G` meets the *uniform random bit generator* requirements if `G` models `uniform_random_bit_generator`, `invoke_result_t<G&>` is an unsigned integer type ([basic.fundamental]), and `G` provides a nested *typedef-name* `result_type` that denotes the same type as `invoke_result_t<G&>`.

Timur Doumler 🇫🇮 @timur_audio · Jan 7, 2020

Question: In C++, how do I generate random numbers in a context that requires the code to finish in a deterministic amount of time (realtime audio callback)?

All the random number engines in std:: are only *amortised* constant time according to the standard, so they're out 😕

19 2 22

Corentin @Cor3ntin · Jan 7, 2020

I am not entirely sure but I think it's still deterministic if not constant. mersen twister has an internal state of fixed known size N and every N numbers that state is recomputed, so it's O(N) then O(1) N-1 times, then O(1) again. I believe N is rather small (32?).

1

Peter Bindels 💉⏳ @dascandy42 · Jan 7, 2020

N is 624.

1 2

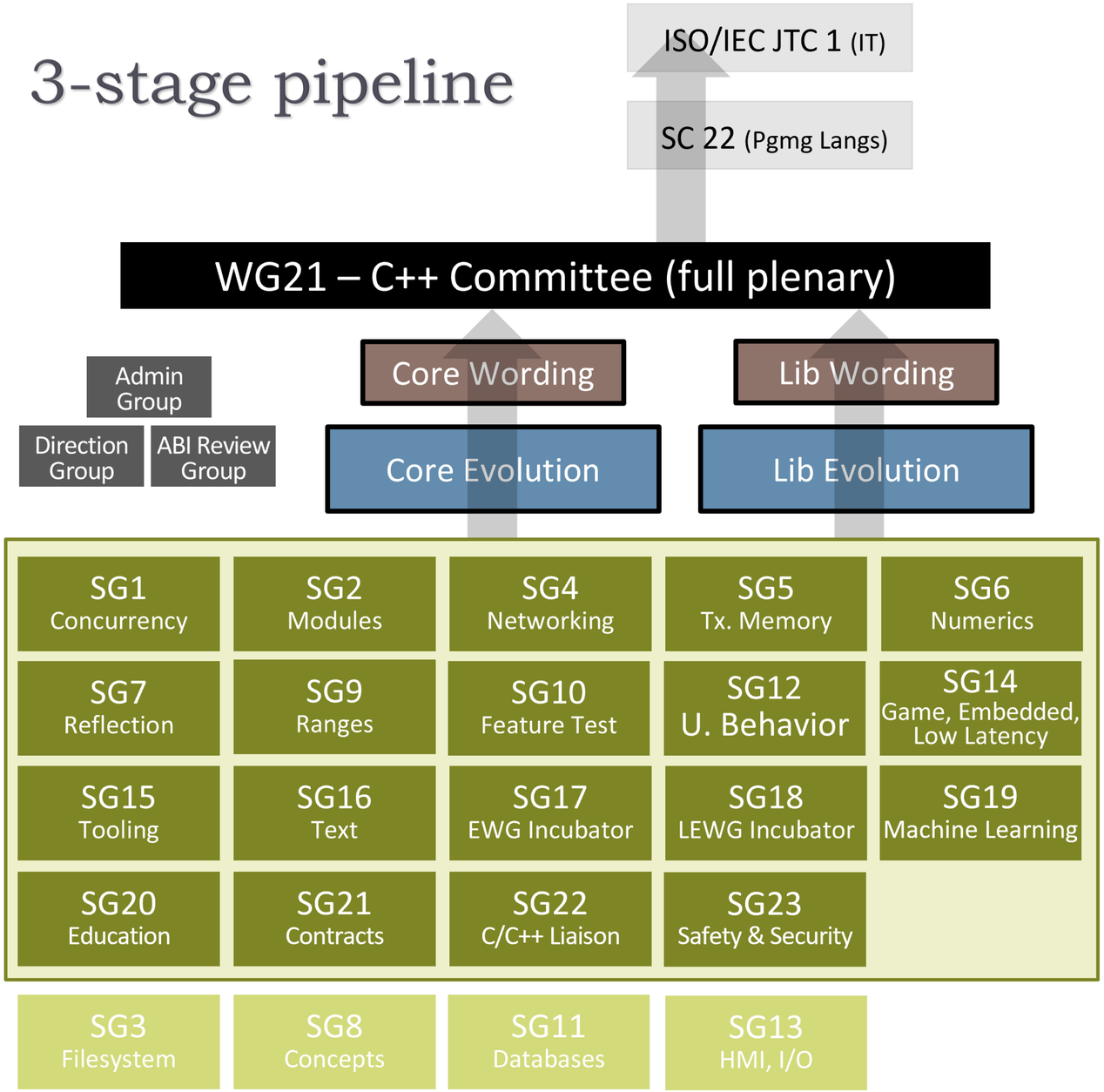
Corentin @Cor3ntin · Jan 7, 2020

I can see that being a problem on a toaster.

2


Cppcon 2022
The C++ Conference
September 12th-16th
Fast, High-Quality
Pseudo-Random Numbers
for Non-Cryptographers
ROTH MICHAELS
Cppcon
The C++ Conference
2022
September 12th-16th
Play (k)
0:10 / 55:42
Roth Michaels
Fast, High-Quality
Pseudo-Random Numbers for
Non-Cryptographers

3-stage pipeline



(F)DIS Approval

CD & PDTS Approval

Internal Approval

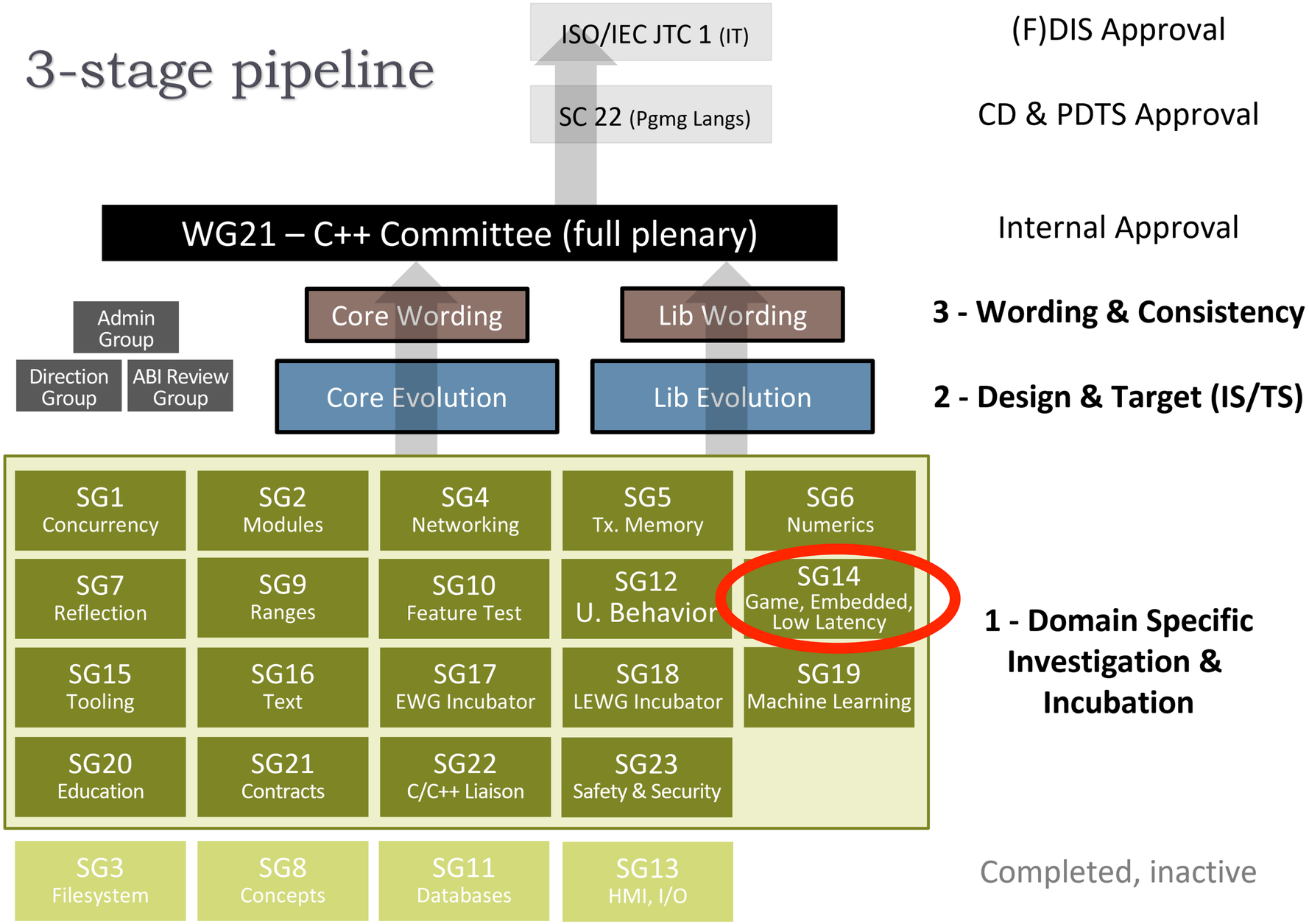
3 - Wording & Consistency

2 - Design & Target (IS/TS)

1 - Domain Specific Investigation & Incubation

Completed, inactive

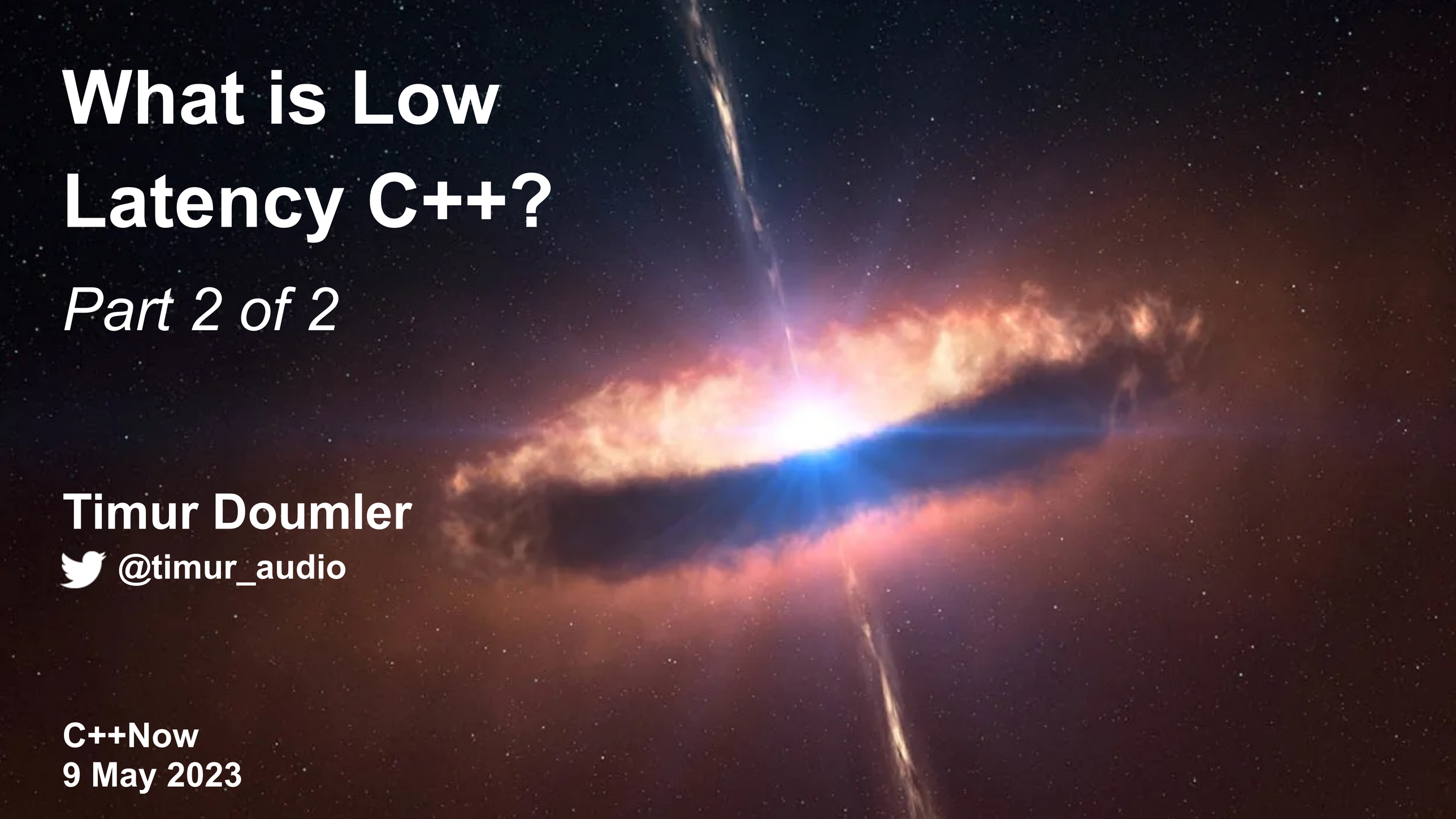
3-stage pipeline



github.com/crill-dev/crill

What is Low Latency C++?

Part 2 of 2



Timur Doumler

 @timur_audio

C++Now
9 May 2023


The international
C++ conference
in the UK, by the sea
28th - 30th June 2023
Workshops: 27th June

C++ and Safety

Timur Doumler

60 minute session | beginner | intermediate | advanced

13:45-14:45, Thursday, 29th June 2023

Organisations such as the National Security Angency (NSA) and the National Institute of Standards and Techology (NIST) are currently urging developers to move away from programming languages that are not memory safe. C++ is arguably not a "safe" programming language in its current form. Why is that? And should we do anything about it? If yes, what, and how? Have we arrived at a crossroads for the future evolution of C++? What does "safety" even mean, and how is it different from "security" and "correctness"?

In this talk, we attempt to give useful definitions for these terms. For safety in particular, we can distinguish between functional safety and language safety, and identify different aspects of language safety (of which memory safety is one). We discuss how and why C++ is considered "unsafe" and what consequences follow from that for different domains and use cases. We look at how other programming languages, such as Java, Rust, and Val avoid such safety issues, what tradeoffs are involved in these strategies, and why we can't easily adopt any of them for C++. We consider the tooling available today to mitigate safety issues in C++, such as sanitisers and static analysers, and their limitations. Finally, we look at the future evolution of C++ and discuss the current work on C++ Contracts and other recent proposals targeted at making C++ more safe.

The C++ Undefined Behaviour Survey

- 3 simple questions (one of which is optional)
- anonymous

The C++ Undefined Behaviour Survey

- 3 simple questions (one of which is optional)
- anonymous
- https://timur.audio/survey