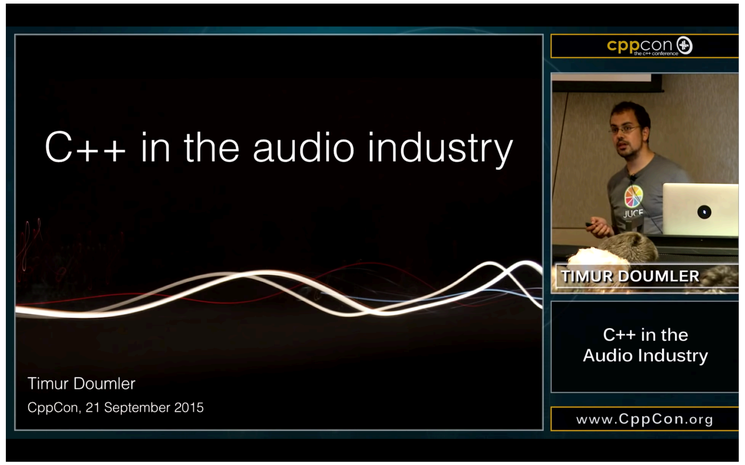
Real-time 101

David Rowland & Fabian Renn-Giles

@drowaudio @hogliux

Prior Art

- C++ in the Audio Industry
  - Timur Doumler
  - CppCon 2015



- Time Waits for Nothing
  - Ross Bencina
  - http://www.rossbencina.com/code/real-time-audio-programming-101-time-waits-for-nothing

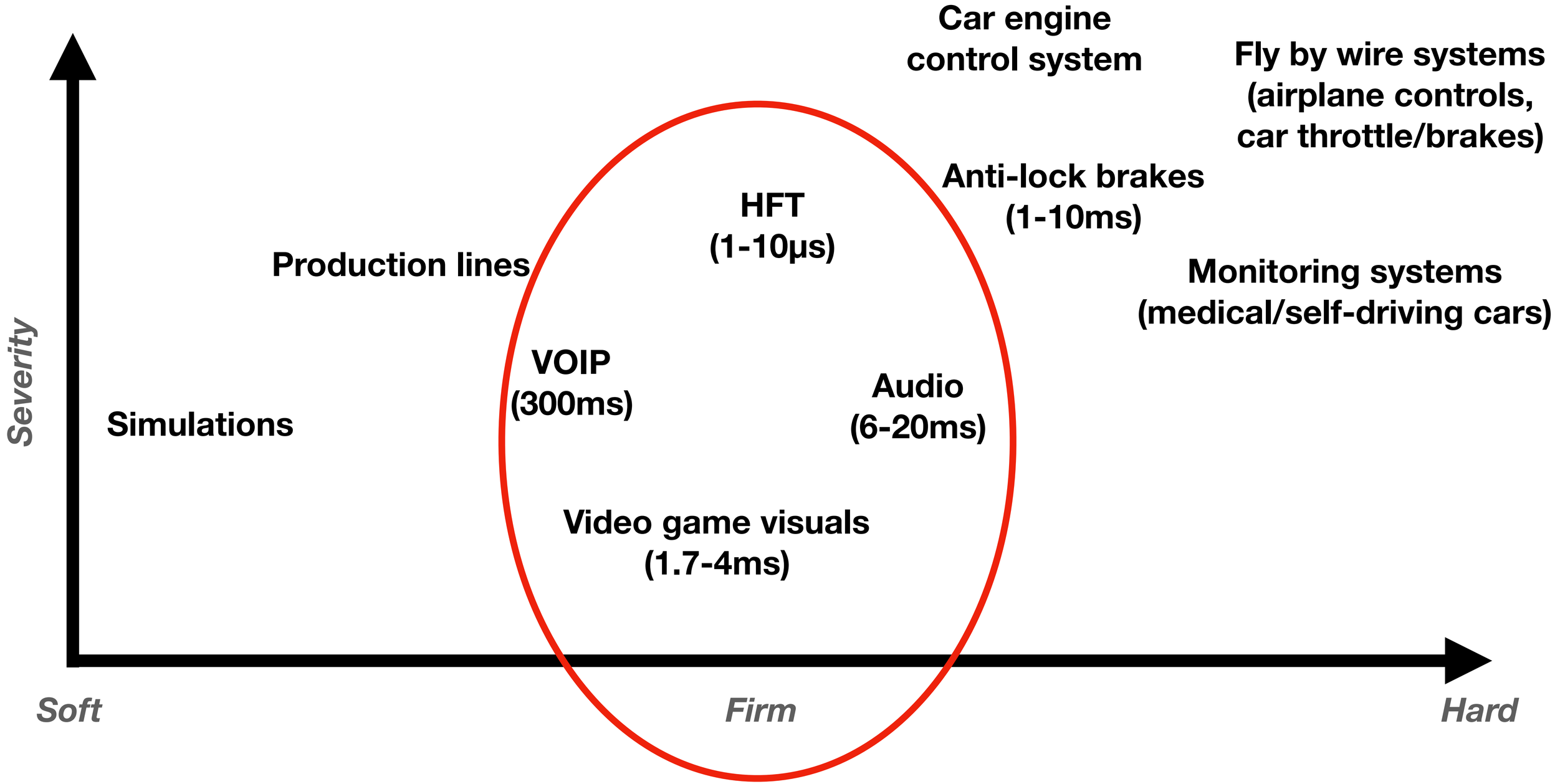
1. What Does Real-time Mean?

“A system is said to be real-time if the total correctness of an operation depends not only upon its logical correctness, but also upon the time in which it is performed.”

–Shin, K.G.; Ramanathan, P. (Jan 1994)

How Hard is Hard?

- "*Real-time systems, as well as their deadlines, are classified by the consequence of missing a deadline*" - Wikipedia
  - **Hard** – missing a deadline is a total system failure
  - **Firm** – infrequent deadline misses are tolerable, but may degrade the system's quality of service. The usefulness of a result is zero after its deadline
  - **Soft** – the usefulness of a result degrades after its deadline, thereby degrading the system's quality of service


Car engine
control system
Fly by wire systems
(airplane controls,
car throttle/brakes)
Anti-lock brakes
(1-10ms)
HFT
(1-10µs)
Production lines
Monitoring systems
(medical/self-driving cars)
VOIP
(300ms)
Audio
(6-20ms)
Simulations
Video game visuals
(1.7-4ms)
Severity
Soft
Firm
Hard

Types of Limiting Factors

- **Latency**
  - Time to generate output from an input

- **Bandwidth**
  - How much processing can you do in your acceptable latency window

- **Jitter**
  - The difference in latency

What Could Possibly Go Wrong?

- **Game/Visuals** - dropped/stuck frames
- **Audio** - clicks/glitches in output
- **HFT** - lose $10m per minute
  (Core C++ 2019: Nimrod Sapir - High Frequency Trading and Ultra Low Latency development techniques)
- **Life-support/monitoring** - deaths

Real-time as a Contract

- In software, if you state your API is “real-time”, you are making a contract that you will complete in a deterministic amount of time*
  - *Depending on resource contention within the system
- There is no way in C++ to state this programatically

2. Real-time and Resources

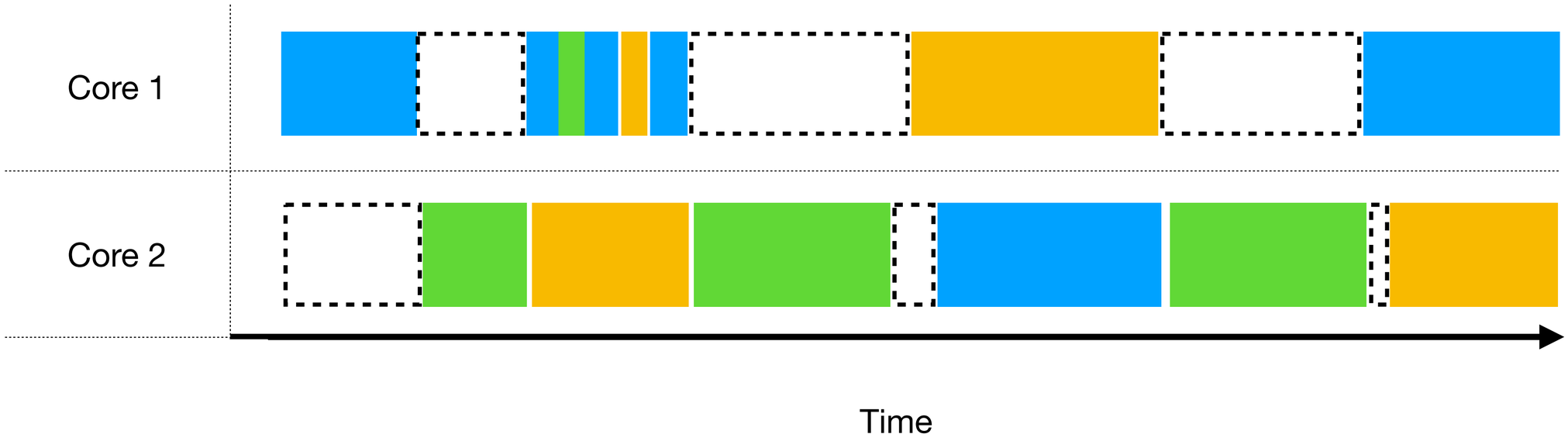
Thread Scheduling

- Threads are scheduled by the OS (ignoring process-scope threads)
- How frequently and how long they get run for will depend on thread priority and the scheduling algorithm used

Thread Sleep/Wake

- Largely OS dependant
- Usually implemented as a queue of threads that are “running”, “sleeping” or “ready”
  - Automatically via indirect system calls such as IO
  - Programatically via system calls such as:
    - std::this_thread::sleep_for()
    - std::condition_variable::notify_one/all()
    - std::condition_variable::wait()
- Thread scheduler will pick a thread from the list to run
- Varying algorithms based on:
  - Priority
  - Last run time
- Can involve a context switch

Normal Thread Schedule



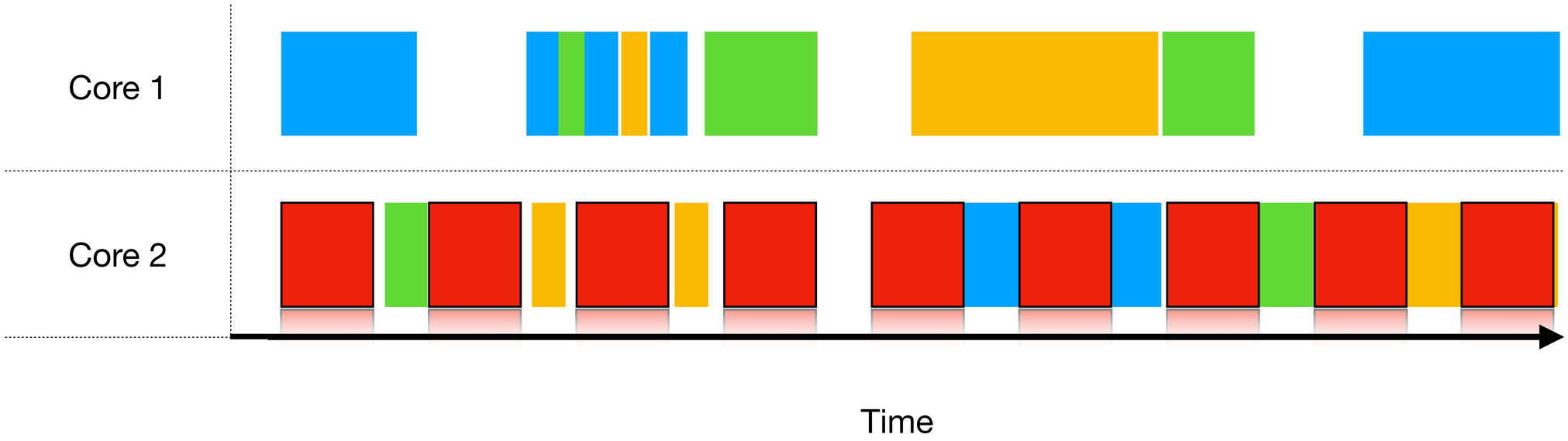
Real-time Threads

- Real-time threads can give you an uninterrupted period of time to run
- If you exceed this you may be put to sleep (and hence you can miss your deadlines)
- You may be declassified as as a “real-time” thread and hence have less time next callback
- Don’t miss your deadlines

https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/scheduler/scheduler.html

Don’t miss your deadlines!

Real-time Thread Schedule



Problems to Real-time

	Real-time	Non-real-time
CPU work	✔	✔
Context switches	✔ (avoid)	✔
Memory access	✔ (non-paged)	✔
System calls	✘	✔
Allocations	✘	✔
Deallocations	✘	✔
Exceptions	✘	✔
Priority Inversion	✘	✔

Context Switching

- Saves the state of the currently executing thread
- Loads the state of a new thread to execute
  - Registers (including machine-state-flags)
  - Stack pointer
  - Program counter
  - Address space (when switching between processes)
- Might be required to move from user-mode to kernel-mode e.g. waiting for an IO resource
- Cache contention

Memory Access

- Where is the memory located?
  - Cache - L1, L2 etc.
  - RAM
  - Disk
- Page faults
  - Memory could be paged to disk if it is large and the system runs out of memory between RT callbacks
  - Constantly “poke” memory with a dedicated low-priority thread
  - mlock()/munlock() (POSIX)
  - VirtualLock()/VirtualUnlock() (Windows)

Memory Access

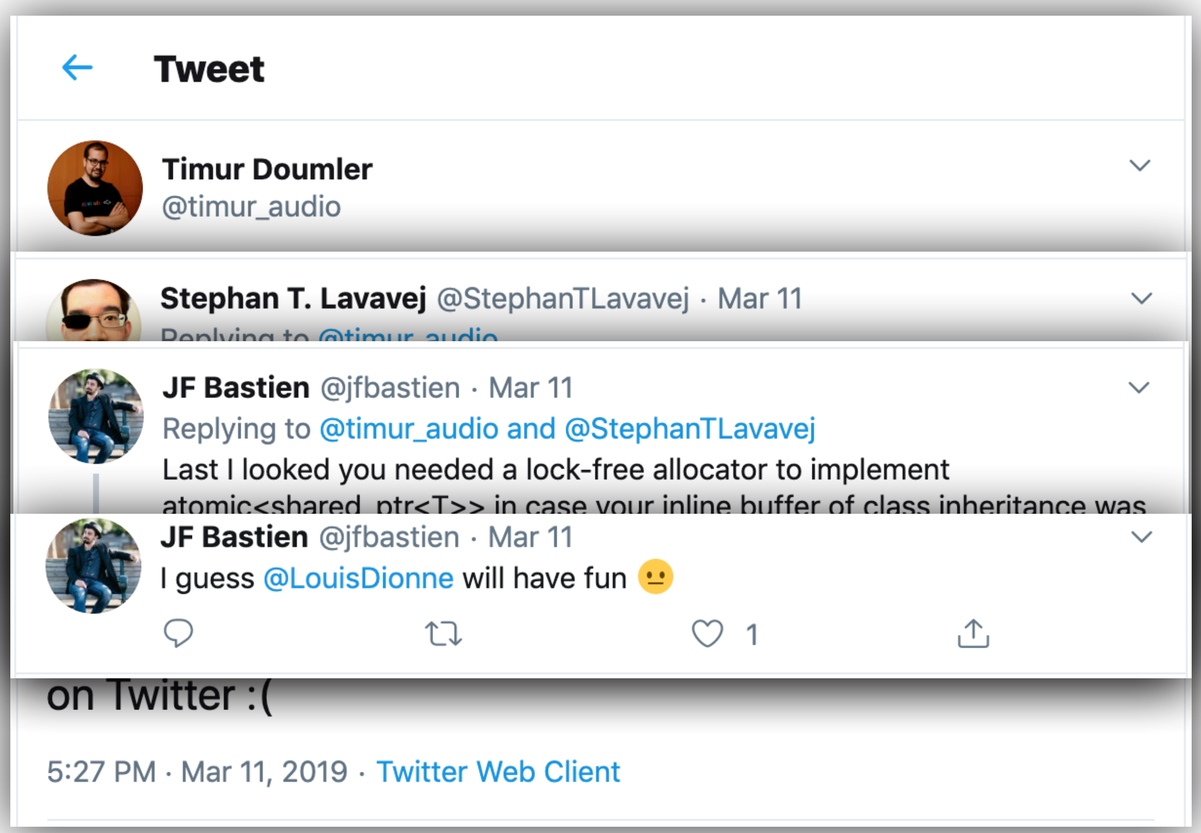
Priority Inversion

```
// Shared state and mutex to synchronise access to it
std::vector<float> vec;
std::mutex mutex;


// Thread 1 - High priority
{
    std::scoped_lock<std::mutex> lock (mutex);      // Lock access to vec
    std::for_each (vec.begin(), vec.end(),          // Process vec
                   [] (auto& f) { f *= 0.1f; });
}

// Thread 2 - Low priority
{
    std::scoped_lock<std::mutex> lock (mutex);      // Lock access to vec
    vec.resize (500'000);                           // Perform expensive operation
}                                                   // Could be de-scheduled
```

```
template <class _Tp>
_LIBCPP_AVAILABILITY_ATOMIC_SHARED_PTR
void
atomic_store(shared_ptr<_Tp>* __p, shared_ptr<_Tp> __r)
{
    __sp_mut& __m = __get_sp_mut(__p);
    __m.lock();
    __p->swap(__r);
    __m.unlock();
}
```

atomic<shared_ptr<T>>

- `std::atomic<std::shared_ptr<T>>` probably won't be lock free :(



Making Copies

- Copies of objects can involve system calls and synchronisation
- Even if you've avoided a data race

```
void updateData (const std::vector<float>& newData)
{
   data = newData;
}
```

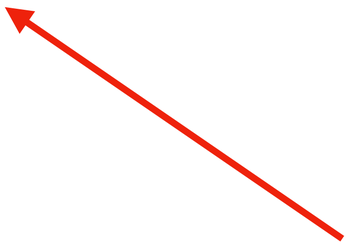
data could resize and allocate

Hidden Copies

- Even moves can result in hidden costs
- Especially when dealing with legacy code which may not have proper move constructors/assignment operators

```
class LegacyObject
{
public:
    LegacyObject();
    LegacyObject (const LegacyObject&);
    LegacyObject& operator=(const LegacyObject&);
    ...
};

struct Parent
{
    std::vector<int> vec;
    LegacyObject obj;
};
```

```
int main()
{
    Parent a, b;
    a = std::move (b);

    return 0;
}
```



`LegacyObject` has no move operator so the copy assignment operator will be called

Blocking vs. Non-wait-free vs. Wait Free

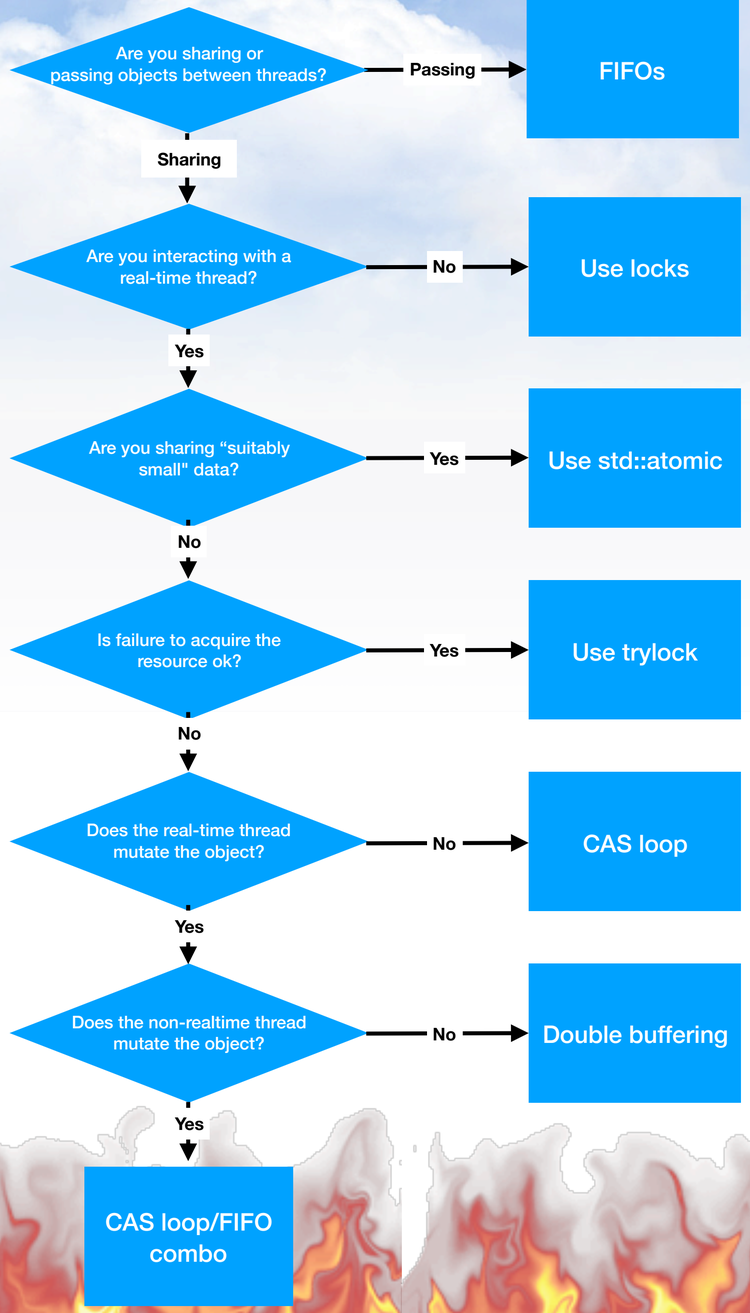
Blocking	Non-wait-free	Wait-free
May context switch for example due to a lock, system call etc.	Execution time is unbounded	Execution time is bounded*
Caches likely to be invalidated	Must contain a loop (which is unbounded)	No unbounded loops
Memory may be swapped	Blocking operations are never wait-free (but not vice versa)	

In Summary

- Don’t block!
  - **System calls**
  - **Waiting to acquire a lock** of any kind

3. Winning in Real-time

What are you sharing between threads?

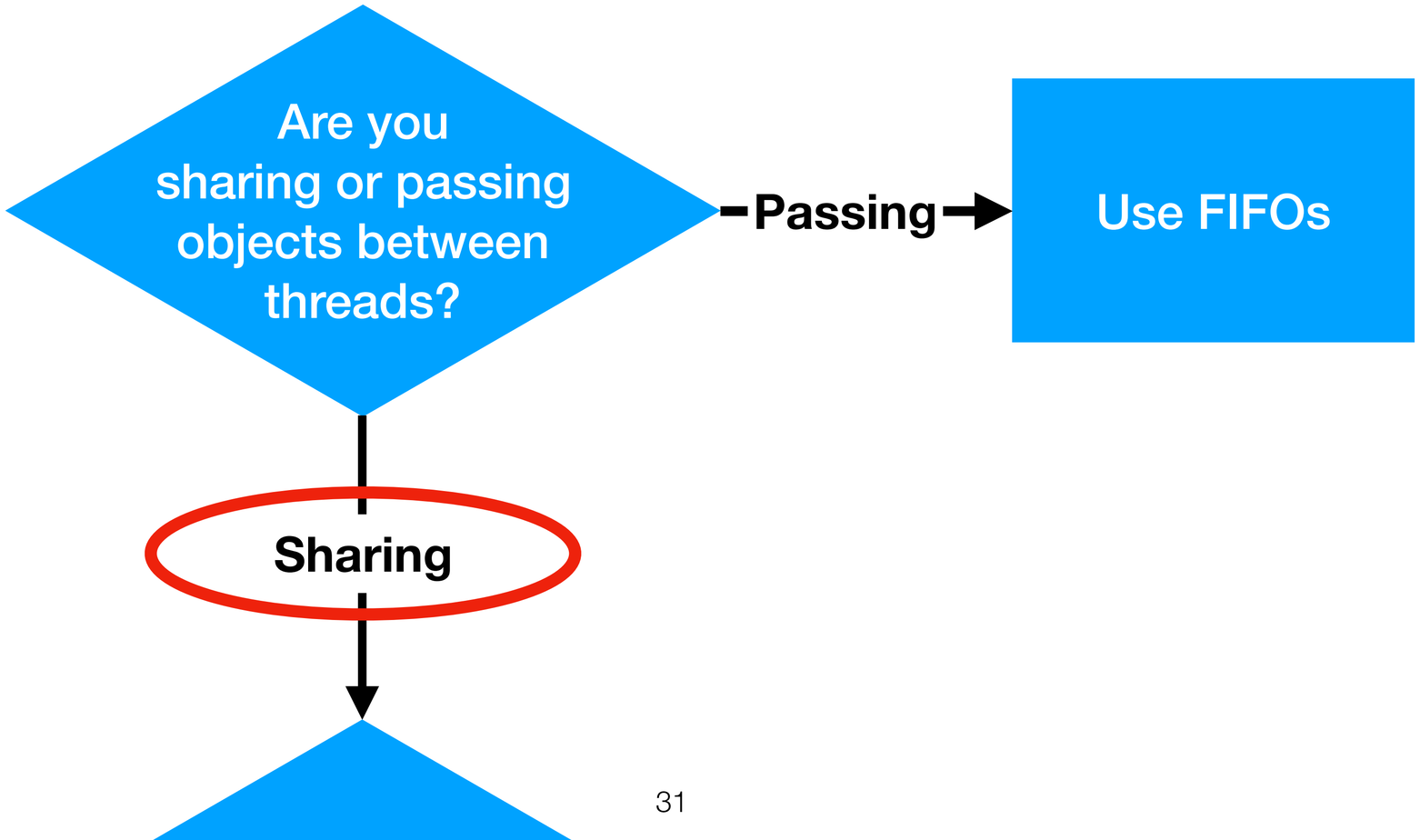


The `farbot` Library

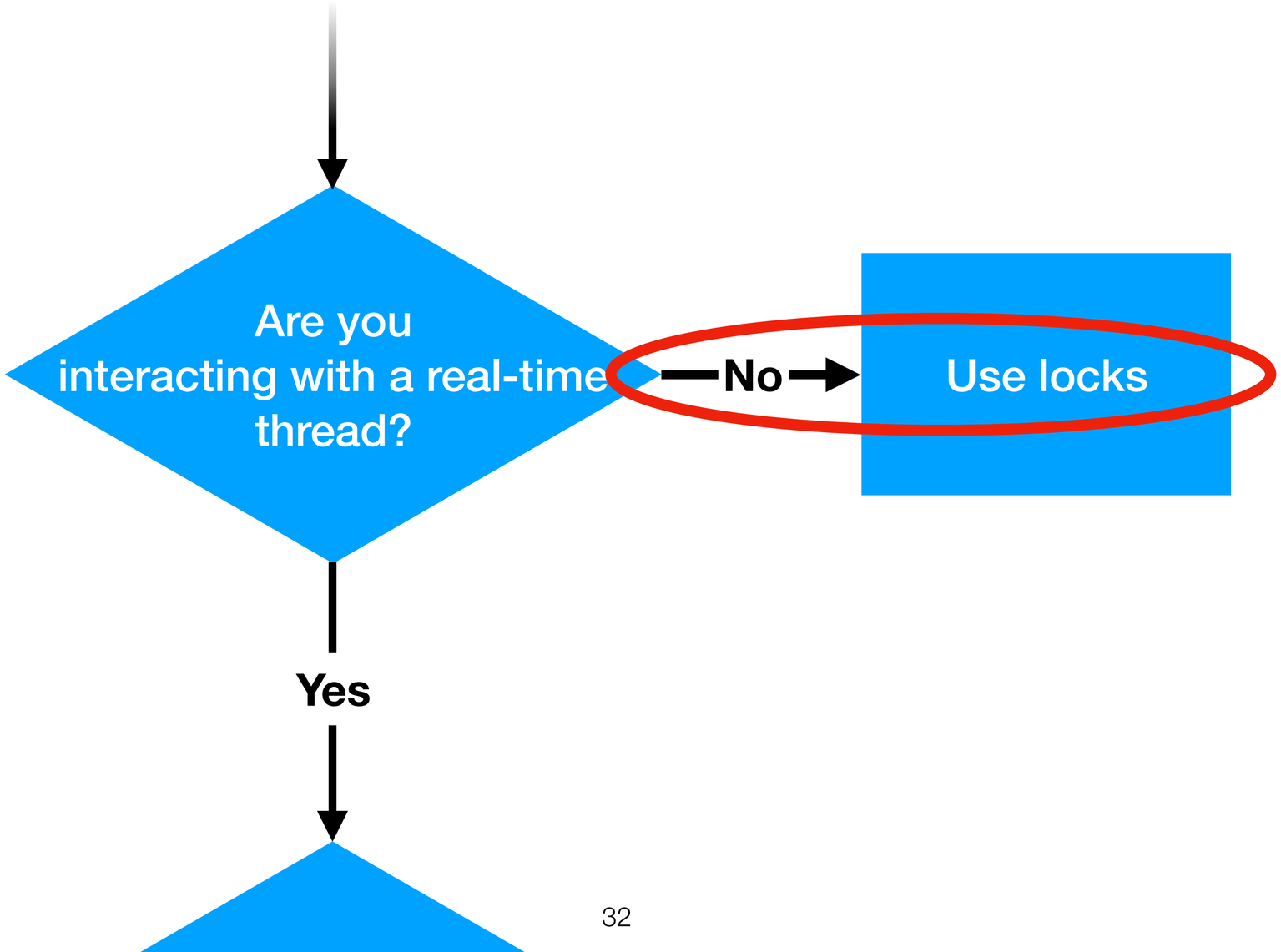
- A collection of realtime design patterns and debugging tools
- Collected over years of realtime programming
- Uses `std::memory_order_*` to optimise performance
- Most design patterns have been used in production code
- Still early alpha development!
- Please contribute:

https://github.com/hogliux/farbot

Sharing or Passing?

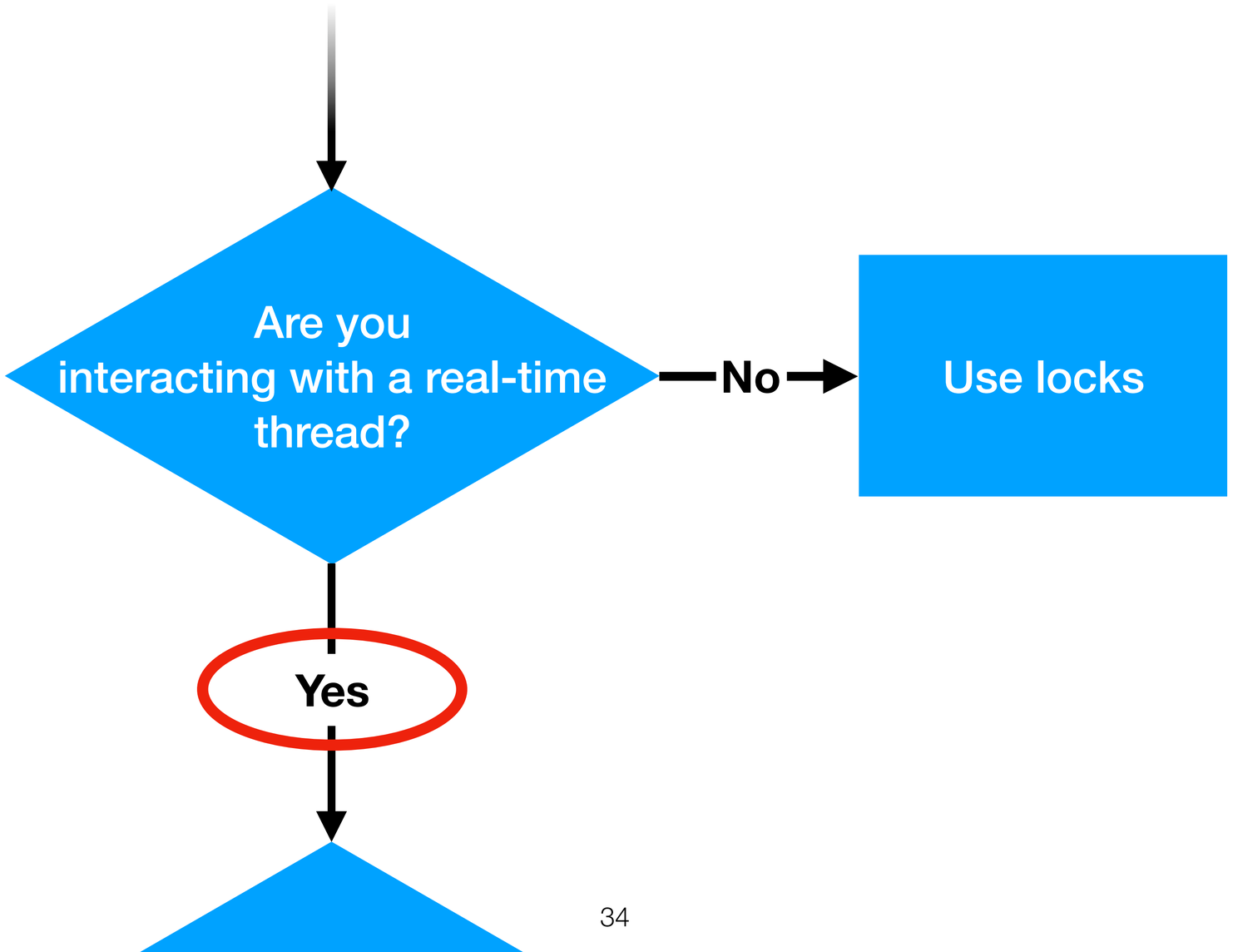


Can you use a lock?

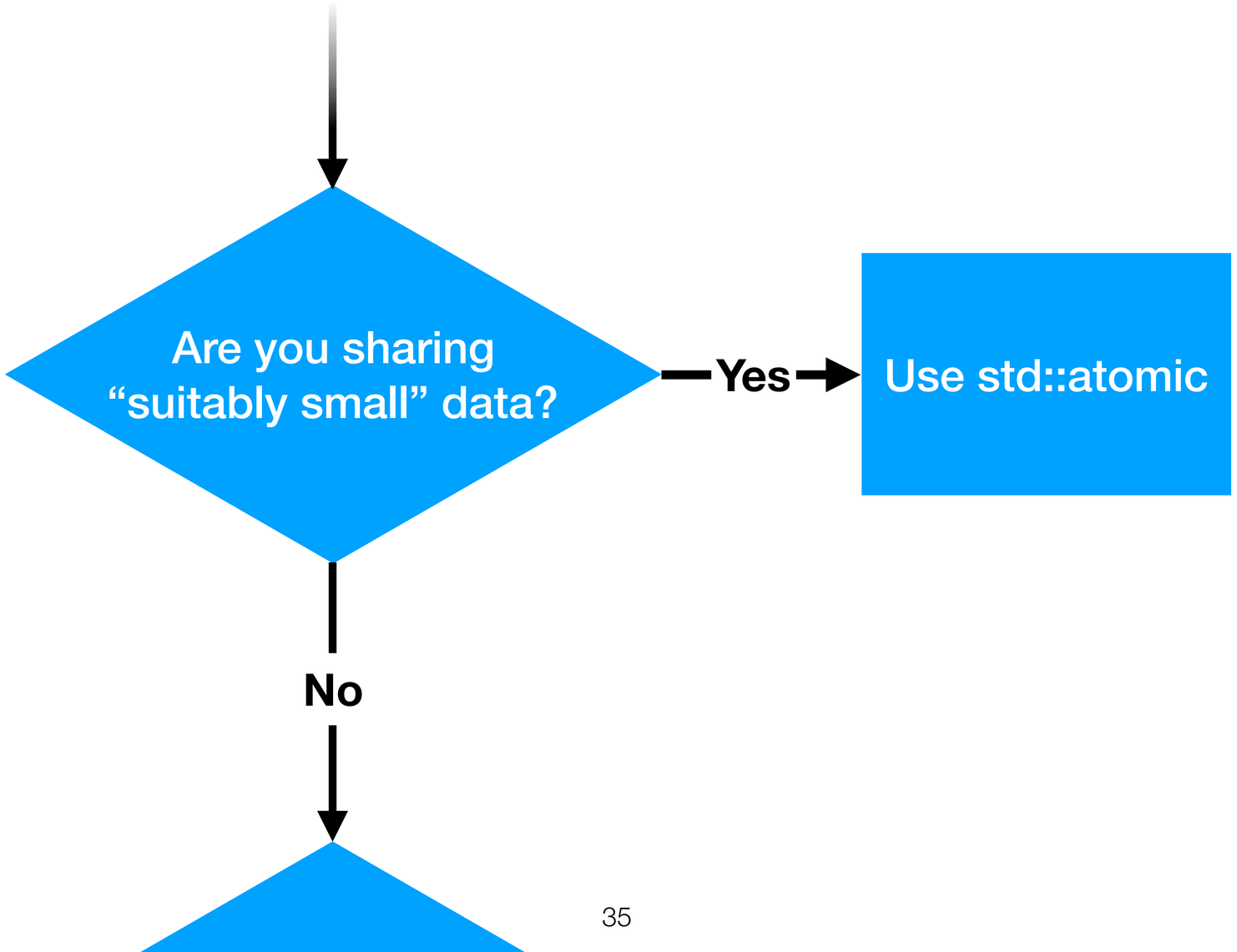


std::mutex

Can you use a lock?



How big is your data?



```
auto gain = 1.0f;
```

```
void processSensorData (float* sensorInOut, int n)
{
    // do some dsp
    ...


    for (int i = 0; i < n; ++i)
        sensorInOut[i] *= gain;
}

// called on another thread
void setSensorGain (float newGain)
{
    gain = newGain;
}
```

- Is this ok?
- How about on a single core machine?
- Sharing data between threads where one of them is a write is a data-race
- A data-race is undefined behaviour

```
auto gain = 1.0f;
```

```
void processSensorData (float* sensorInOut, int n)
{
    // do some dsp
    ...

    register auto gain_copy = gain;
    for (int i = 0; i < n; ++i)
        sensorInOut[i] *= gain_copy;
}

// called on another thread
void setSensorGain (float newGain)
{
    gain = newGain;
}
```

```
auto gain = 1.0f;

void realtimeThreadEntry()
{
    while (rocketFlying)
    {
        ...
        processSensorData (sensorData, 512);
    }
}

void processSensorData (float* sensorInOut, int n)
{
    // do some dsp ...

    for (int i = 0; i < n; ++i)
        sensorInOut[i] *= gain;
}

// called on another thread
void setSensorGain (float newGain)
{
    gain = newGain;
}
```

```
auto gain = 1.0f;

void realtimeThreadEntry()
{

    while (rocketFlying)
    {
        // do some dsp ...

        for (int i = 0; i < n; ++i)
           sensorInOut[i] *= gain;
    }
}

// called on another thread
void setSensorGain (float newGain)
{
    gain = newGain;
}
```

```
auto gain = 1.0f;

void realtimeThreadEntry()
{
    register auto gain_copy = gain;
    while (rocketFlying)
    {
        // do some dsp ...

        for (int i = 0; i < n; ++i)
           sensorInOut[i] *= gain_copy;
    }
}

// called on another thread
void setSensorGain (float newGain)
{
    gain = newGain;
}
```

Cached

No effect

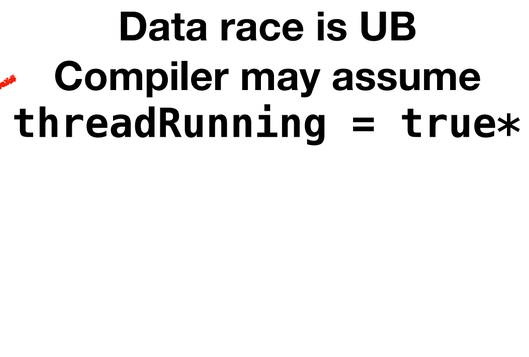
Data race

↓

Undefined behaviour

↓

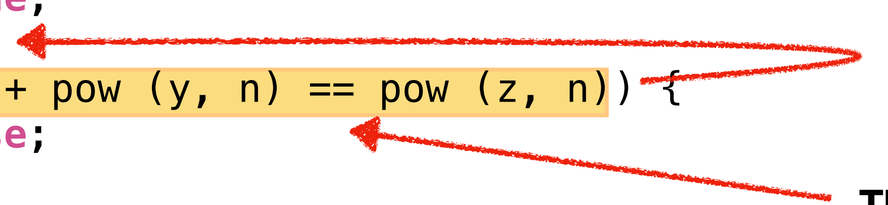
Anything can happen!
(Including exploding rockets)




```
bool threadRunning;

bool proveFermatsLastTheorem() // Thread 1 {
    threadRunning = true;
    for (int n = 3; threadRunning; ++n) {
        if (pow (x, n) + pow (y, n) == pow (z, n)) {
            return false;
        }
    }

    return true;
}

void testTheorem () {
    bool result;
    startThread ([] () (result = proveFermatsLastTheorem));
    Sleep (2000);
    threadRunning = false;
    std::cout << result << std::endl;
}
```

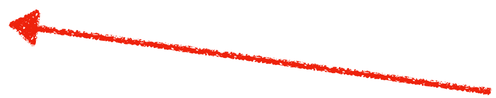
*** A valid C++ compiler is allowed to assume `threadRunning` is always `true` (due to data-race being UB). Most compilers we tested will nevertheless do the right thing here (i.e. check `threadRunning` every iteration) - but they are not required to. Your code may break when you update compiler versions for example.**

```
bool threadRunning;

bool proveFermatsLastTheorem() // Thread 1 {
    threadRunning = true;
    while (true) {
        if (pow (x, n) + pow (y, n) == pow (z, n)) {
            return false;
        }
        ++n
    }

    return true;
}

void testTheorem () {
    bool result;
    startThread ([] () (result = proveFermatsLastTheorem));
    Sleep (2000);
    threadRunning = false;
    std::cout << result << std::endl;
}
```



This is real loop condition

```
bool threadRunning;

bool proveFermatsLastTheorem() // Thread 1 {
    threadRunning = true;
    while (pow (x, n) + pow (y, n) != pow (z, n)) ++n;

    return false;
    return true;
}

void testTheorem () {
    bool result;
    startThread ([] () (result = proveFermatsLastTheorem));
    Sleep (2000);
    threadRunning = false;
    std::cout << result << std::endl;
}
```



Dead-code elimination

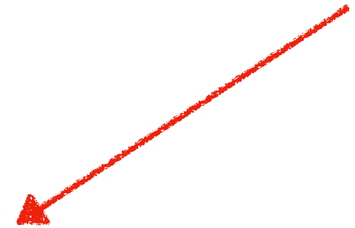
```
bool threadRunning;

bool proveFermatsLastTheorem() // Thread 1 {
    threadRunning = true;
    while (pow (x, n) + pow (y, n) != pow (z, n)) ++n;

    return false;

}

void testTheorem () {
    bool result;
    startThread ([] () (result = proveFermatsLastTheorem));
    Sleep (2000);
    threadRunning = false;
    std::cout << result << std::endl;
}
```

1. **A function that never returns is UB: C++ may assume that functions will <u>always</u> return***
2. **The only way this function can return is be returning false**
3. **The function does not have any side-effects**

*** if it doesn't call any IO or `[[no_return]]` is not specified**

```
bool threadRunning;

bool proveFermatsLastTheorem() // Thread 1 {


    return false;

}

void testTheorem () {
    bool result;
    startThread ([] () (result = proveFermatsLastTheorem));
    Sleep (2000);
    threadRunning = false;
    std::cout << result << std::endl;
}
```

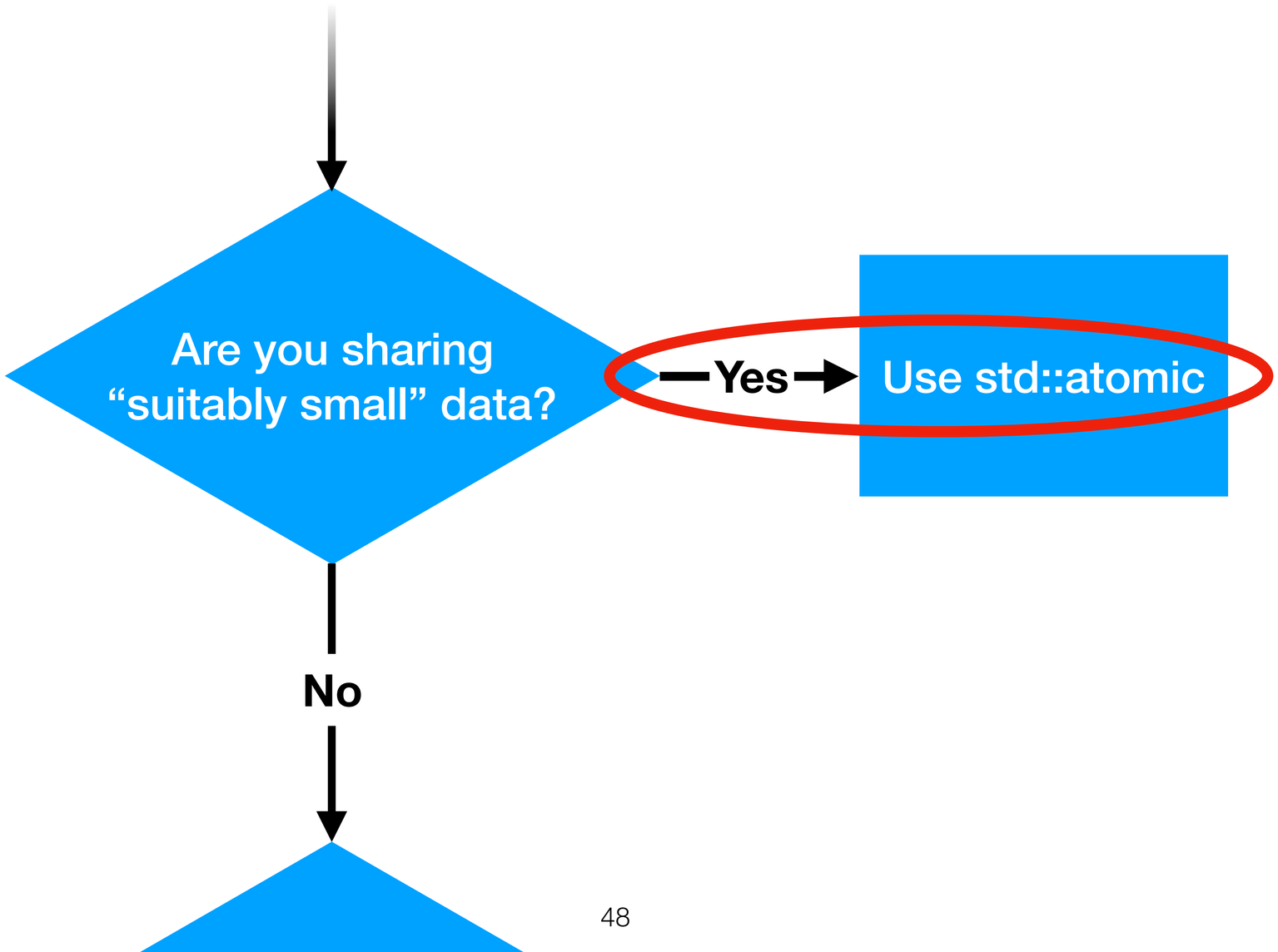
With data-races branches of your code can be executed which seem impossible to reach!



```
bool threadRunning;

bool proveFermatsLastTheorem() // Thread 1 {
    threadRunning = true;
    for (int n = 3; threadRunning; ++n) {
        if (pow (x, n) + pow (y, n) == pow (z, n)) {
            return false;
        }
    }

    return true;
}

void testTheorem () {
    bool result;
    startThread ([] () (result = proveFermatsLastTheorem));
    Sleep (2000);
    threadRunning = false;
    std::cout << result << std::endl;
}
```

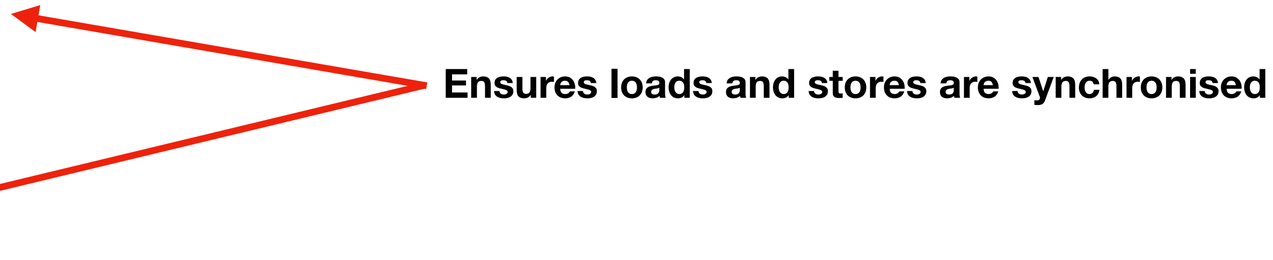
Don’t write data-races!

How big is your data?



```cpp
std::atomic<float> gain (1.0f);
```

```cpp
void processSensorData (float* sensorInOut, int n)
{
    // do some dsp
    ...

    for (int i = 0; i < n; ++i)
        sensorInOut[i] *= gain.load();
}

// called on another thread
void setSensorGain (float newGain)
{
    gain.store (newGain);
}
```



```cpp
std::atomic<float> gain (1.0f);
static_assert (std::atomic<float>::is_always_lock_free);

void processSensorData (float* sensorInOut, int n)
{
    // do some dsp
    ...

    for (int i = 0; i < n; ++i)
        sensorInOut[i] *= gain.load();
}

// called on another thread
void setSensorGain (float newGain)
{
    gain.store (newGain);
}
```

Ensure manipulating a float is lock-free on your machine

Ensures loads and stores are synchronised

std::atomic<>

- Ensure “tear free” & synchronised manipulation of shared data
- May use traditional locks if data-type cannot be manipulated atomically in hardware. Always check **`std::atomic<>::is_always_lock_free`**!
- Only a subset of manipulations are supported:
  - Store
  - Load
  - Atomic addition/subtraction
  - exchange/compare-exchange
- More info here: https://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/

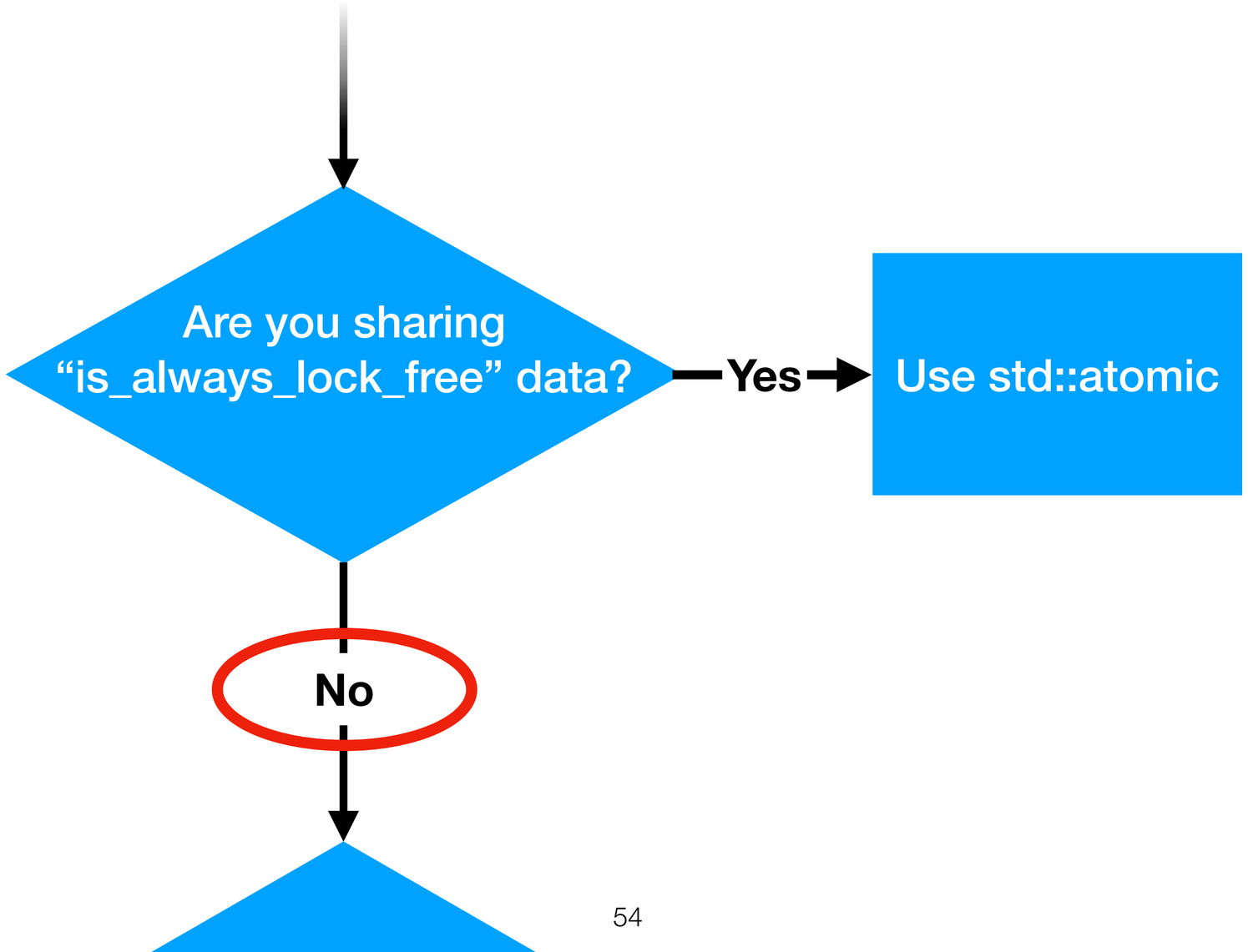
std::memory_order

- Can be specified to relax the memory ordering of atomic operations
- If you don’t care about the ordering of operations, you might get better performance using **`memory_order_relaxed`**
- More info here: https://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/

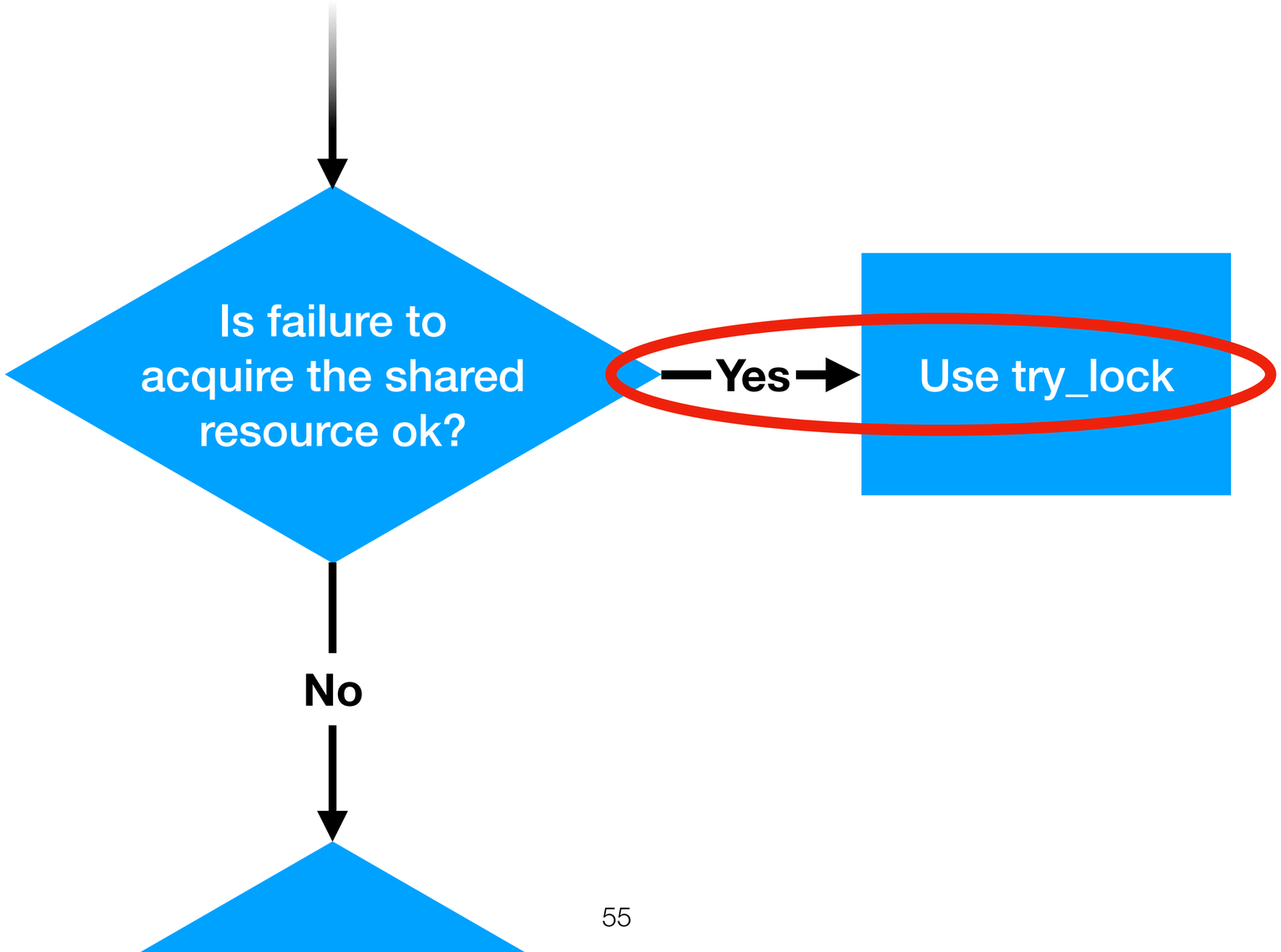
atomic Summary

- Scenario:
  - Multiple threads may need to mutate the data
- Trade-off:
  - Data is small: `std::atomic<>::is_always_lock_free == true`
  - Only certain operations are allowed
- Examples:
  - Sharing small data between threads
  - Gain values, level meters, automation values, parameters etc.

How big is your data?



Seriously, can you use a lock?



```
class WavetableSynthesizer
{
public:
    void audioCallback()
    {
        if (std::unique_lock<mutex> tryLock (mutex, std::try_to_lock); tryLock.owns_lock())
        {
            // Do something with wavetable
        }
        else
        {
            // Do something else as wavetable is not available
        }
    }

    void updateWavetable (/* args */)
    {
        // Create new Wavetable
        auto newWavetable = std::make_unique<Wavetable> (/* args */);

        {
            std::lock_guard<std::mutex> lock (mutex);
            std::swap (wavetable, newWavetable);
        }

        // Delete old wavetable here to lock for least time possible
    }

private:
    mutex mutex;
    std::unique_ptr<Wavetable> wavetable;
};
```

What happens here?

What is mutex?

std::mutex<>

- **`std::mutex::try_lock()`** is wait-free
  - Can fail spuriously
- **`std::mutex::unlock()`** can block

```
class WavetableSynthesizer
{
public:
    void audioCallback()
    {
        if (std::unique_lock<spin_lock> tryLock (mutex, std::try_to_lock); tryLock.owns_lock())
        {
            // Do something with wavetable
        }
        else
        {
            // Do something else as wavetable is not available
        }
    }

    void updateWavetable (/* args */)
    {
        // Create new Wavetable
        auto newWavetable = std::make_unique<Wavetable> (/* args */);

        {
            std::lock_guard<spin_lock> lock (mutex);
            std::swap (wavetable, newWavetable);
        }

        // Delete old wavetable here to lock for least time possible
    }

private:
    spin_lock mutex;
    std::unique_ptr<Wavetable> wavetable;
};
```

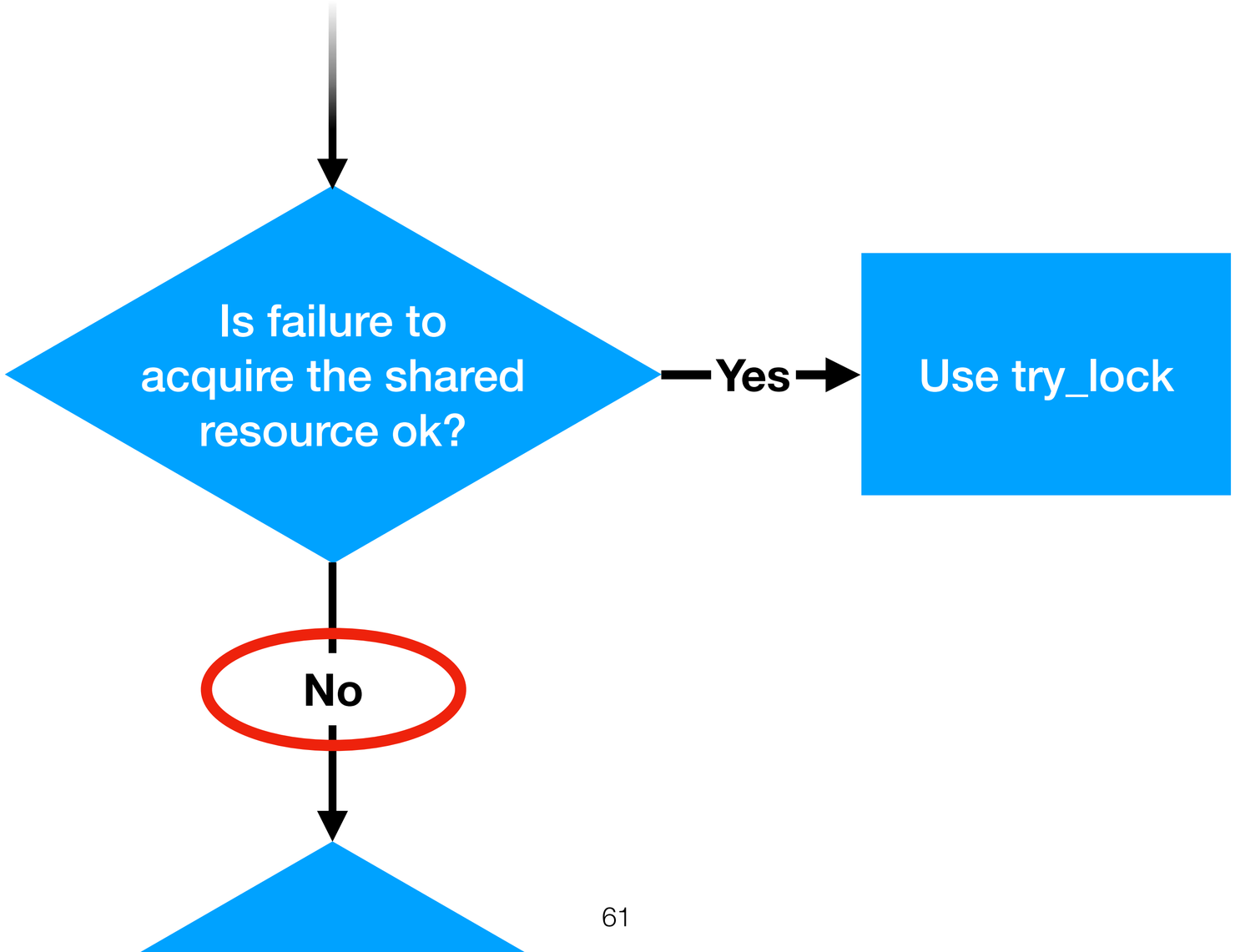
Basic spin_lock

```
class spin_lock
{
public:
    void lock() noexcept        { while (flag.test_and_set()); }
    void unlock() noexcept      { flag.clear(); }
    bool try_lock() noexcept    { return ! flag.test_and_set(); }

private:
    std::atomic_flag flag = ATOMIC_FLAG_INIT;
};
```

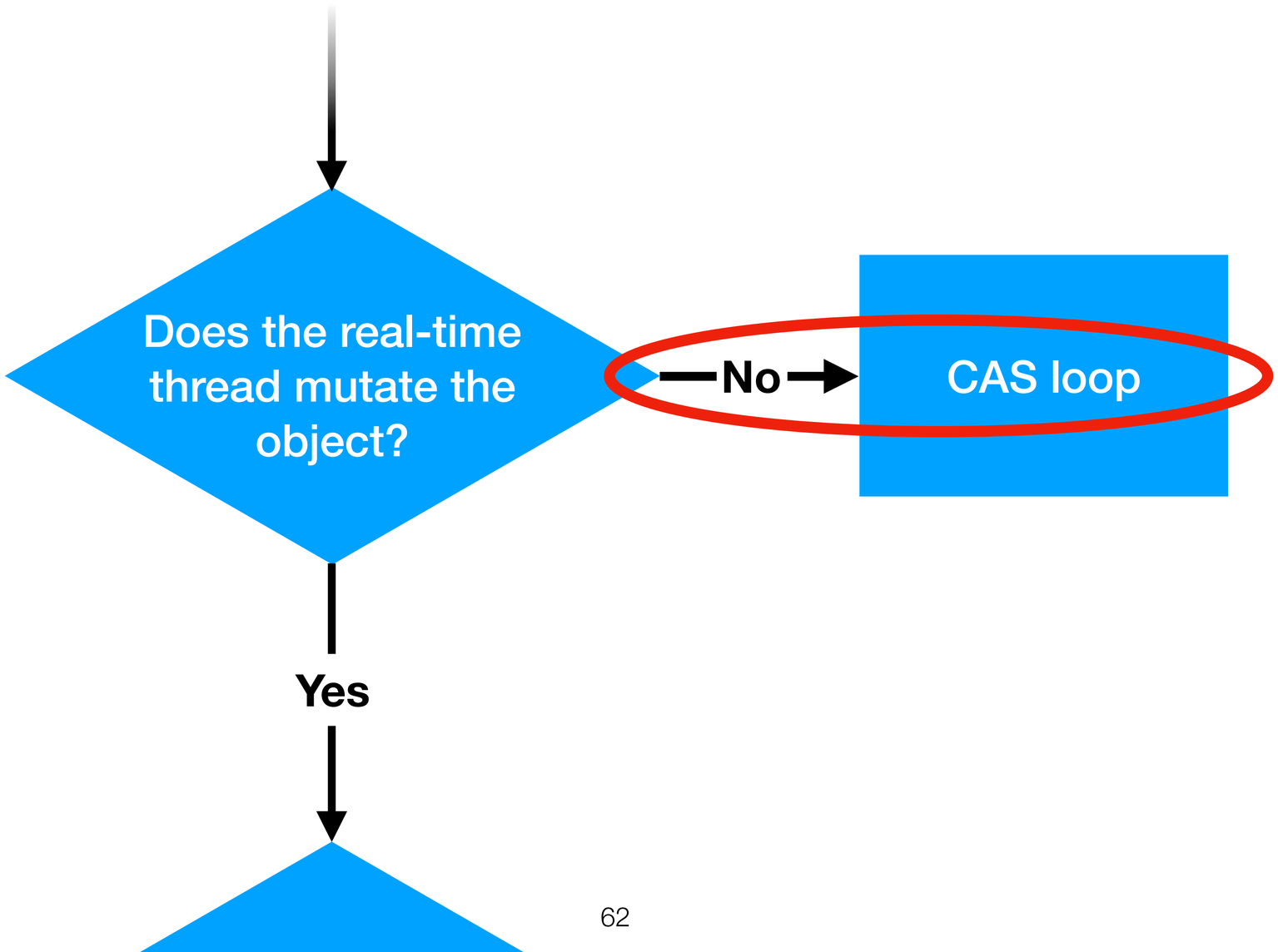
try_lock Summary

- Scenario:
  - Data is big: `std::atomic<>::is_always_lock_free == false`
  - Failure to acquire the resource is ok
- Trade-off:
  - Non-real-time thread waits on real-time thread for access to the resource
  - Real-time thread will have to fail gracefully
- Examples:
  - Passing large data to the real-time thread for exclusive use
  - Audio samples, wavetables, filter coefficients etc.

Seriously, can you use a lock?



Real-time Mutating?



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

BiquadCoeffecients calculateLowPassCoeffecients (float freq);

void audioThread (const float* src, float* dst, size_t n)
{
    static float lv1, lv2;

    for (size_t i = 0; i < n; ++i)
    {
        auto input = src[i];
        auto output = (input * coeffs.b0) + lv1;
        dst[i] = output;

        lv1 = (input * coeffs.b1) - (output* coeffs.a1) + lv2;
        lv2 = (input * coeffs.b2) - (output* coeffs.a2);
    }
}

void updateFrequencyParameter (float newValue)
{
    coeffs = calculateLowPassCoeffecients (newValue);
}
```

```
struct BiquadCoeffecients  {  float b0, b1, b2, a1, a2; };
std::atomic<BiquadCoeffecients> coeffs;


BiquadCoeffecients calculateLowPassCoeffecients (float freq);

void audioThread (const float* src, float* dst, size_t n)
{
    static float lv1, lv2;
    auto local_coeffs = coeffs.load();

    for (size_t i = 0; i < n; ++i)
    {
        auto input = src[i];
        auto output = (input * local_coeffs.b0) + lv1;
        dst[i] = output;

        lv1 = (input * local_coeffs.b1) - (output* local_coeffs.a1) + lv2;
        lv2 = (input * local_coeffs.b2) - (output* local_coeffs.a2);
    }
}

void updateFrequencyParameter (float newValue)
{
    coeffs = calculateLowPassCoeffecients (newValue);
}
```

```
struct BiquadCoeffecients  {  float b0, b1, b2, a1, a2; };
std::atomic<BiquadCoeffecients> coeffs;
static_assert (std::atomic<BiquadCoefficients>::is_always_lock_free);

BiquadCoeffecients calculateLowPassCoeffecients (float freq);

void audioThread (const float* src, float* dst, size_t n)
{
    static float lv1, lv2;
    auto local_coeffs = coeffs.load();

    for (size_t i = 0; i < n; ++i)
    {
        auto input = src[i];
        auto output = (input * local_coeffs.b0) + lv1;
        dst[i] = output;

        lv1 = (input * local_coeffs.b1) - (output* local_coeffs.a1) + lv2;
        lv2 = (input * local_coeffs.b2) - (output* local_coeffs.a2);
    }
}

void updateFrequencyParameter (float newValue)
{
    coeffs = calculateLowPassCoeffecients (newValue);
}
```

Fails!

The CAS Exchange Loop

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

BiquadCoeffecients calculateLowPassCoeffecients (float freq);

void audioThread (const float* src, float* dst, size_t n)
{

    processBiquad (src, dst, n, coeffs);
}

void updateFrequencyParameter (float newValue)
{
    coeffs = calculateLowPassCoeffecients (newValue);
}
```

The CAS Exchange Loop

```
struct BiquadCoeffecients  {  float b0, b1, b2, a1, a2; };
std::atomic<BiquadCoeffecients*> coeffs;

BiquadCoeffecients calculateLowPassCoeffecients (float freq);

void audioThread (const float* src, float* dst, size_t n)
{
    auto* coeffsCopy = coeffs.load();
    processBiquad (src, dst, n, coeffsCopy);
}

void updateFrequencyParameter (float newValue)
{
    coeffs = new BiquadCoeffecients (calculateLowPassCoeffecients (newValue));
}
```

Works but memory leak!

The CAS Exchange Loop

```
struct BiquadCoeffecients  {  float b0, b1, b2, a1, a2; };
BiquadCoeffecients* coeffs;
std::atomic<bool> isInAudioThread { false };

BiquadCoeffecients calculateLowPassCoeffecients (float freq);

void audioThread (const float* src, float* dst, size_t n)
{
    isInAudioThread = true;
    auto* coeffsCopy = coeffs;
    processBiquad (src, dst, n, coeffsCopy);
    isInAudioThread = false;
}

void updateFrequencyParameter (float newValue)
{
    auto* ptr = new BiquadCoeffecients (calculateLowPassCoeffecients (newValue));

    while (isInAudioThread.load())
        ;

    std::swap (ptr, coeffs);
    delete ptr;
}
```

Spin whilst in audio thread

isInAudioThread could be changed here

✗ **ABA problem!**

The CAS Exchange Loop

```
struct BiquadCoeffecients  {  float b0, b1, b2, a1, a2; };
std::unique_ptr<BiquadCoeffecients> storage { std::make_unique<BiquadCoeffecients>() };
std::atomic<BiquadCoeffecients*> biquadCoeffs;

void processAudio (float* buffer)
{
    auto* coeffs = biquadCoeffs.exchange (nullptr); // set biquadCoeffs to nullptr while in processing audio

    processBiquad (*coeffs, buffer);

    biquadCoeffs = coeffs;
}

void changeBiquadParameters (BiquadCoeffecients newCoeffs)
{
    auto newBiquad = std::make_unique<BiquadCoeffecients> (newCoeffs);

    for (auto* expected = storage.get(); // spin while the realtime thread is processing
         ! biquadCoeffs.compare_exchange_strong (expected, newBiquad.get());
         expected = storage.get());

    storage = std::move (newBiquad);
}
```

Changes on real-time thread will be lost

Old storage now deleted

✔ **Works!**

farbot's NonRealtimeMutatable

```
struct BiquadCoeffecients  {  float b0, b1, b2, a1, a2; };
NonRealtimeMutatable<BiquadCoeffecients> biquadCoeffs;

void processAudio (float* buffer)
{
    auto& coeffs = biquadCoeffs.realtimeAcquire();

    processBiquad (coeffs, buffer);

    biquadCoeffs.realtimeRelease();
}

void changeBiquadParameters (BiquadCoeffecients newCoeffs)
{
    auto& coeffs = biquadCoeffs.nonRealtimeAcquire();

    coeffs = newCoeffs;

    biquadCoeffs.nonRealtimeRelease();
}
```

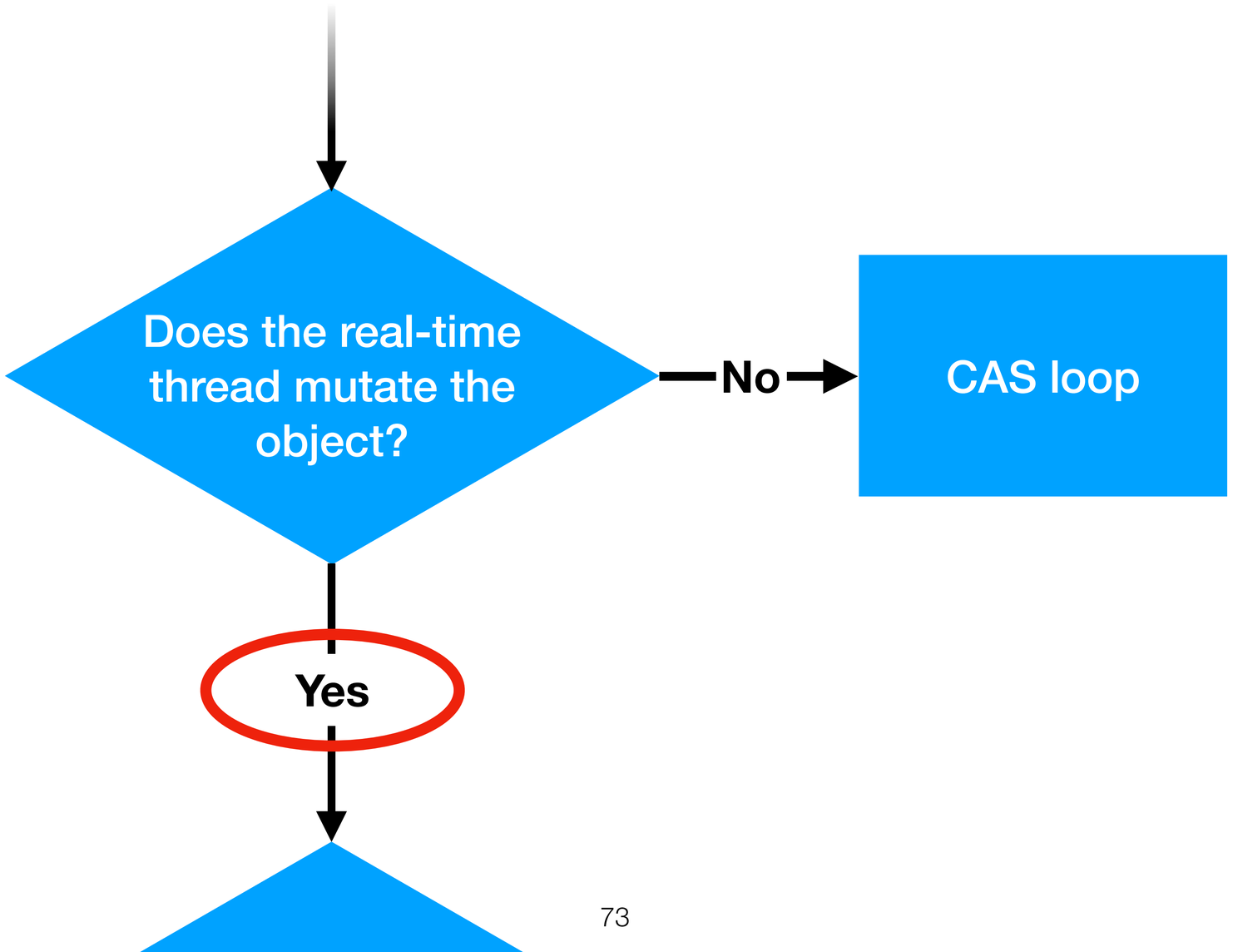
farbot's NonRealtimeMutatable

```
struct BiquadCoeffecients  {  float b0, b1, b2, a1, a2; };
NonRealtimeMutatable<BiquadCoeffecients> biquadCoeffs;

void processAudio (float* buffer)
{
    NonRealtimeMutatable<BiquadCoeffecients>::ScopedAccess<true> coeffs(biquadCoeffs);

    processBiquad (*coeffs, buffer);


}

void changeBiquadParameters (BiquadCoeffecients newCoeffs)
{
    NonRealtimeMutatable<BiquadCoeffecients>::ScopedAccess<false> coeffs(biquadCoeffs);

    *coeffs = newCoeffs;


}
```

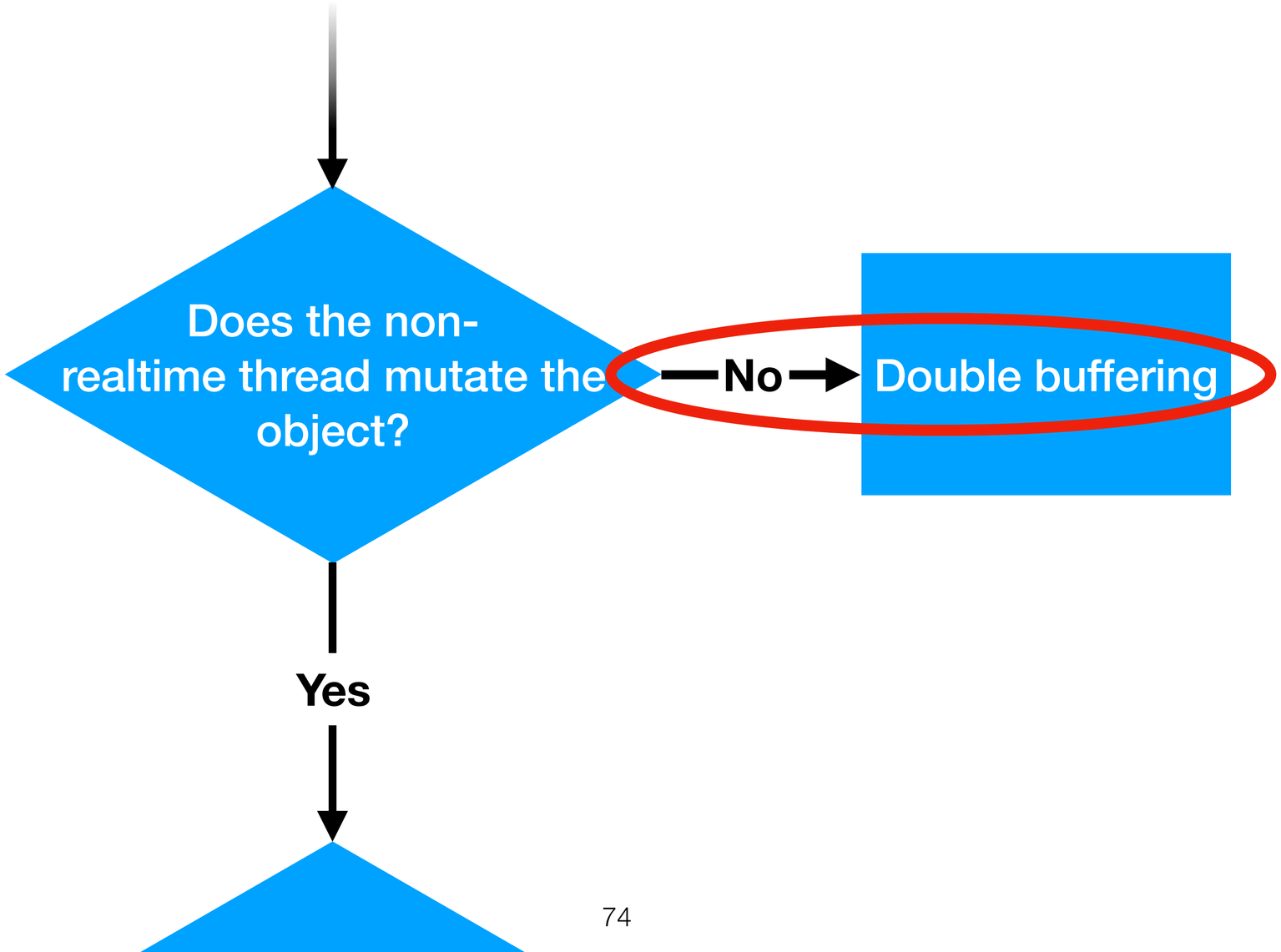
Non-real-time Mutate Summary

- Scenario:
  - Data is big: `std::atomic<>::is_always_lock_free == false`
  - The non-real-time thread ***can*** mutate the object
  - Real-time thread will not fail to acquire the resource
- Trade-off:
  - The real-time thread ***can not*** mutate the object
  - Non-real-time thread will wait on the real-time thread
  - Overhead of copying on the non-real-time thread
- Examples:
  - Sharing large data from the non-real-time thread to the real-time thread
  - Audio samples, wavetables, filter coefficients etc.

Real-time Mutating?

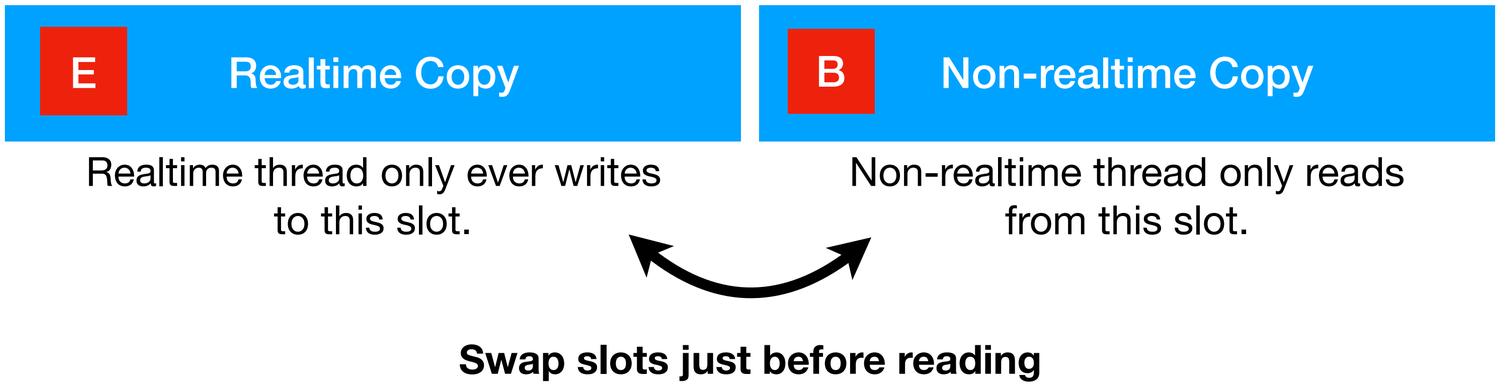


Non-real-time Mutating?



Double Buffering

Use two buffers: one for the realtime thread, one for the non-realtime thread



1. Both slots are pre-initialised with valid data
2. Realtime thread can write to realtime slot without interference
3. When non-realtime thread wants to read data, the slots are swapped
4. Realtime thread can continue to write to realtime thread while non-realtime thread reads

Double Buffering

```
using FrequencySpectrum = std::array<float, 512>;

std::array<FrequencySpectrum,2> mostRecentSpectrum;
std::atomic<int> idx = {0};

void processAudio (const float* buffer, size_t n)
{
    auto freqSpec = calculateSpectrum (buffer, n);

    mostRecentSpectrum[idx.load()] = freqSpec;
}

void updateSpectrumUIButtonClicked()
{
    auto i = idx.fetch_xor (1);
    displaySpectrum (mostRecentSpectrum[i]);
}
```

`idx` denotes current slot of realtime thread (`idx XOR 1` denotes slot of non-realtime thread)

Realtime thread writes to it's slot (let's say slot `0`)

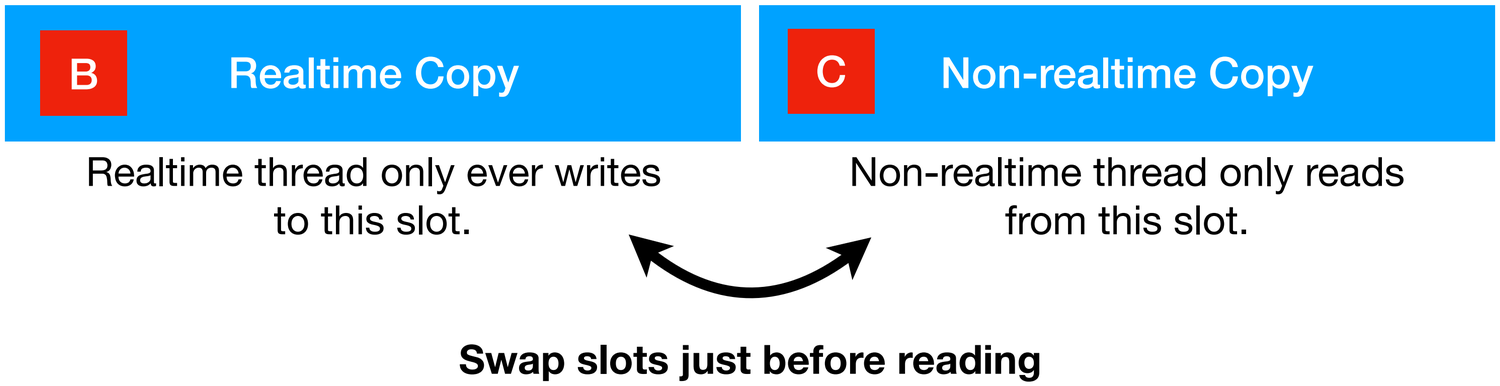
`idx` is swapped. Any new writes now go to slot 1. Old value (`0`) is assigned to `i`.

slot `0` is displayed. As writes now go to slot 1, realtime thread can't overwrite us while displaying.

➔ Unfortunately, calling `updateSpectrumUIButtonClicked` twice in a row will show old data!

Double Buffering

Problem: reading twice in a row:



Old data when non-realtime thread reads twice in a row

➔ Use `NEW_DATA` flag to indicate new data is available

Double Buffering

```
using FrequencySpectrum = std::array<float, 512>;

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

std::array<FrequencySpectrum,2> mostRecentSpectrum;
std::atomic<int> idx = {0};

void processAudio (const float* buffer, size_t n)
{
    auto freqSpec = calculateSpectrum (buffer, n);

    auto i = idx.load() & BIT_IDX;
    mostRecentSpectrum[i] = freqSpec;
    idx.store ((i & BIT_IDX) | BIT_NEWDATA);
}

void updateSpectrumUIButtonClicked()
{
    auto current = idx.load();

    if ((current & BIT_NEWDATA) != 0)
    {
        current = (current & BIT_IDX) ^ 1;
        idx.store (current);
    }

    displaySpectrum (mostRecentSpectrum[(current & BIT_IDX) ^ 1]);
}
```

Add a new bit "BIT_NEWDATA" to the index variable

However, introduced race because we now unatomically load store the idx variable

Let non-realtime thread know that new data is available

Only swap indices if new data is available

Atomically clear new data bit and increment index

Double Buffering

```
using FrequencySpectrum = std::array<float, 512>;

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

std::array<FrequencySpectrum,2> mostRecentSpectrum;
std::atomic<int> idx = {0};

void processAudio (const float* buffer, size_t n) {
    auto freqSpec = calculateSpectrum (buffer, n);

    auto i = idx.fetch_or(BIT_BUSY) & BIT_IDX;
    mostRecentSpectrum[i] = freqSpec;
    idx.store ((i & BIT_IDX) | BIT_NEWDATA);
}

void updateSpectrumUIButtonClicked() {
    auto current = idx.load();

    if ((current & BIT_NEWDATA) != 0) {
        int newValue;
        do {
            current &= ~BIT_BUSY;
            newValue = (current ^ BIT_IDX) & BIT_IDX;
        } while (! idx.compare_exchange_weak (current, newValue));

        current = newValue;
    }

    displaySpectrum(mostRecentSpectrum[(current & BIT_IDX) ^ 1]);
}
```

Add a new bit "BIT_BUSY"

BIT_BUSY is set when realtime thread is in the middle of load/ store of `idx` variable

CAS loop to ensure that idx is only incremented when BIT_BUSY is not set

farbot's RealtimeMutatable

```
using FrequencySpectrum = std::array<float, 512>;

RealtimeMutatable<FrequencySpectrum> mostRecentSpectrum;

void processAudio (const float* buffer, size_t n) {
    auto& freqSpec = mostRecentSpectrum.realtimeAcquire();

    freqSpec = calculateSpectrum (buffer, n);

    mostRecentSpectrum.realtimeRelease();
}

void updateSpectrumUIButtonClicked() {
    auto& recentSpectrum = mostRecentSpectrum.nonRealtimeAcquire();

    displaySpectrum(recentSpectrum);

    mostRecentSpectrum.nonRealtimeRelease();
}
```

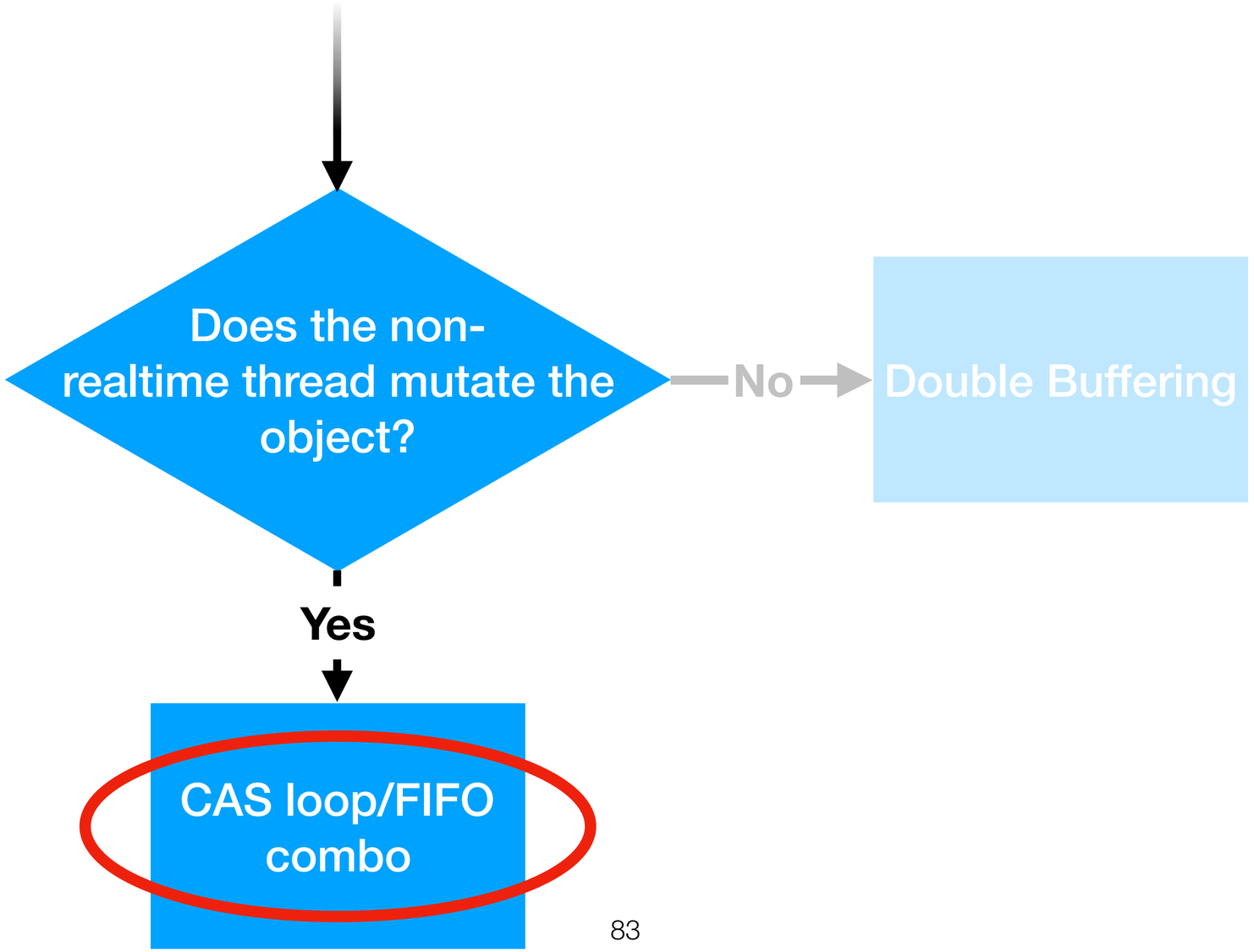
farbot's RealtimeMutatable

```
using FrequencySpectrum = std::array<float, 512>;

RealtimeMutatable<FrequencySpectrum> mostRecentSpectrum;

void processAudio (const float* buffer, size_t n) {
    RealtimeMutatable<FrequencySpectrum>::ScopedAccess<true> freqSpec(mostRecentSpectrum);

    *freqSpec = calculateSpectrum (buffer, n);
}

void updateSpectrumUIButtonClicked() {
    RealtimeMutatable<FrequencySpectrum>::ScopedAccess<false> recentSpectrum(mostRecentSpectrum);

    displaySpectrum(*recentSpectrum);
}
```

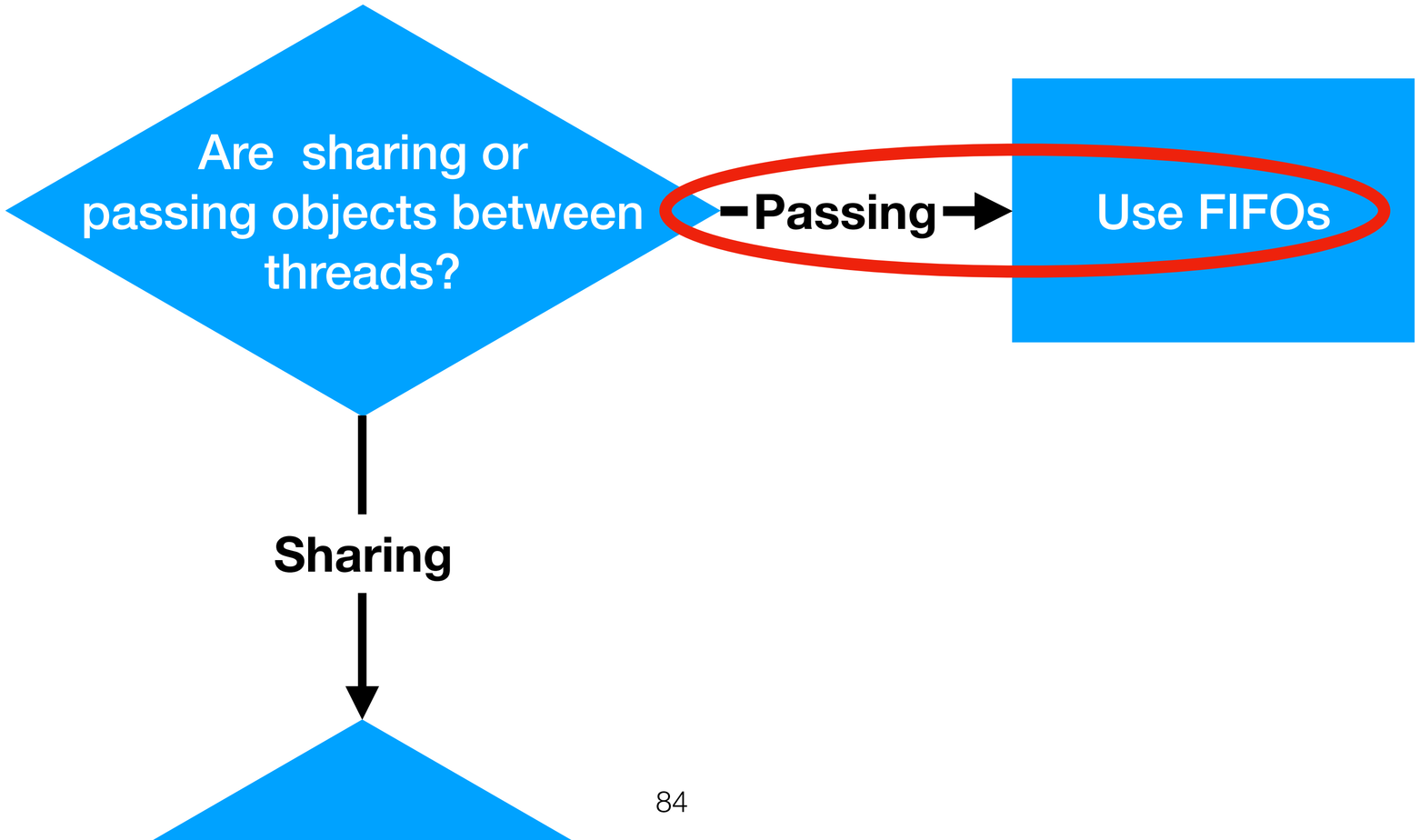
Real-time Mutate Summary

- Scenario:
  - Data is big: `std::atomic<>::is_always_lock_free == false`
  - The real-time thread ***can*** mutate the object
  - Real-time thread will not fail to acquire the resource
- Trade-off:
  - The non-real-time thread ***can not*** mutate the object
  - Non-real-time thread will wait on the real-time thread
  - Overhead of copying on the non-real-time & real-time thread
- Examples:
  - Sharing large data from the real-time thread to the non-real-time thread
  - GUI visualisations, frequency spectrums, oscilloscopes etc.

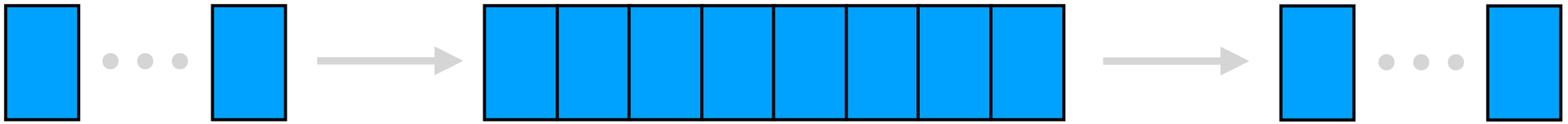
Both Mutating



Sharing or Passing?

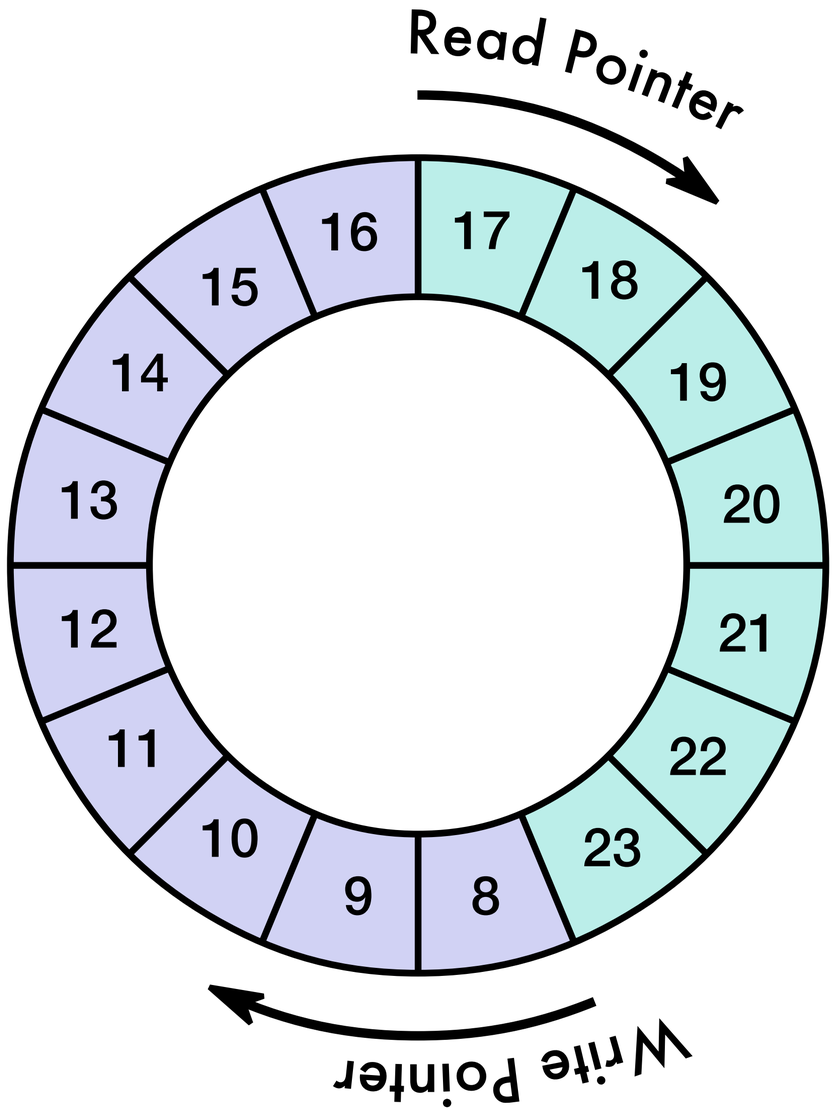


The humble FIFO



- First-In, First-Out data structures
  - Perfect for passing data/objects from one thread to the other
  - Useful if losing data/objects is not an option
    - `(Non)RealtimeMutatable` loses data if written to twice

The humble FIFO



- Realtime code use ring buffers to implement a FIFO
- Fixed capacity: no allocations (i.e. realtime safe)
- Various flavours

The humble FIFO

Single Consumer, Single Producer

```
template <typename T> class fifo {
public:
    bool push (T && arg) {
        auto pos  = writepos.load();
        auto next = (pos + 1) % slots.size();

        if (next == readpos.load())
            return false;

        slots[pos] = std::move (arg);
        writepos.store (next);
        return true;
    }

    bool pop(T& result) {
        auto pos = readpos.load();

        if (pos == writepos.load())
            return false;

        result = std::move (slots[pos]);
        readpos.store ((pos + 1) % slots.size());
        return true;
    }
private:
    std::vector<T> slots = {}; std::atomic<int> readpos = {0}, writepos = {0};
};
```

Get the current write position

Is there room in the fifo?

Write object to current write position

Update write position

Get the current read position

Is there something in the fifo?

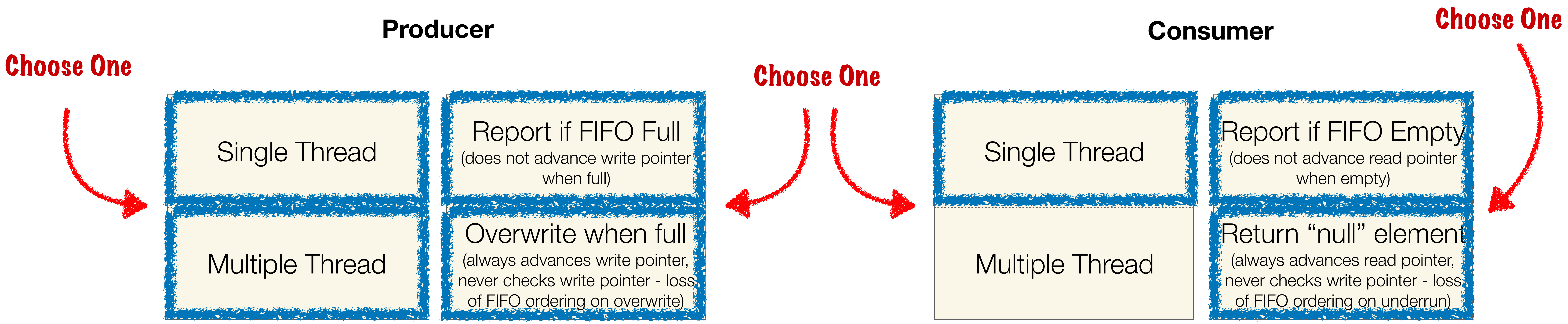
Read object from current read position

Update read position

Which FIFO is right for you?

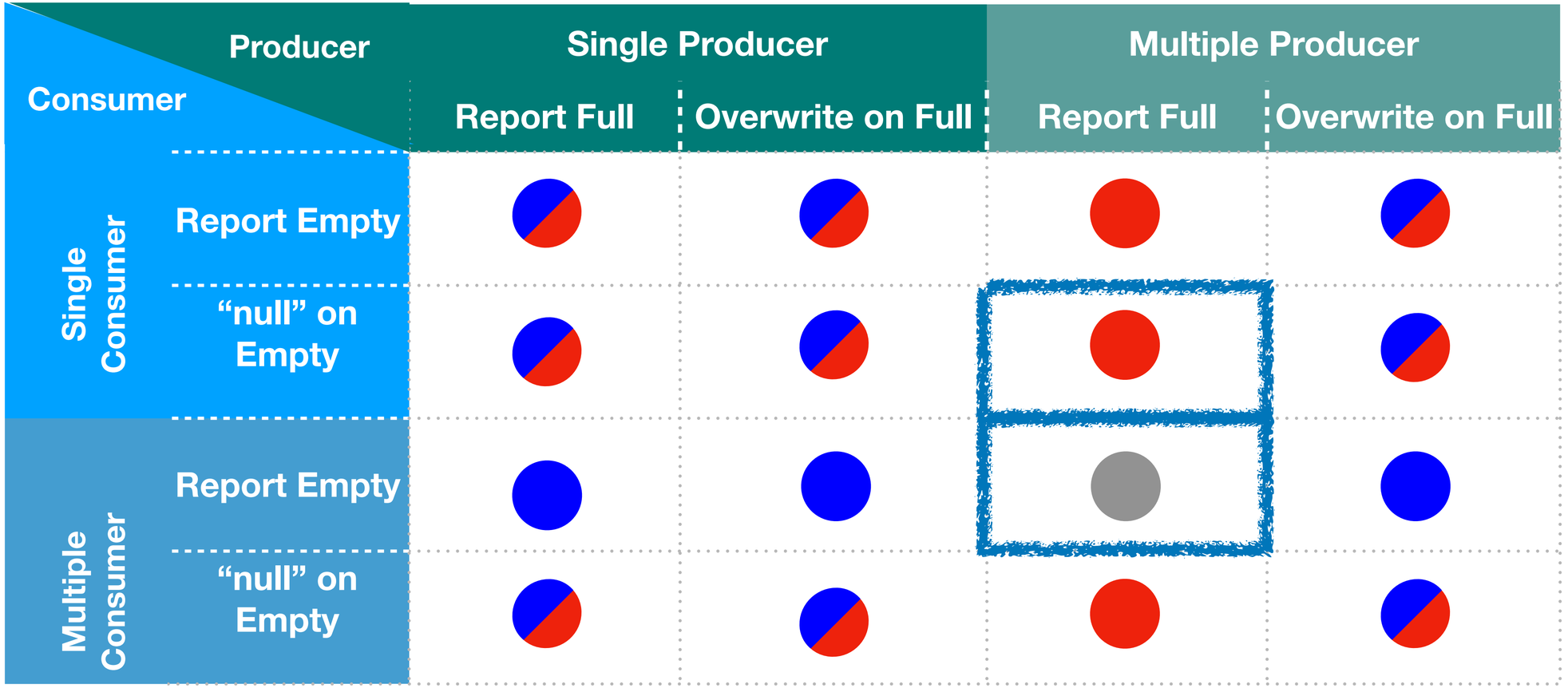
Ask yourself two questions?

1. Will more than one thread concurrently read/write into the FIFO?
2. What should happen if the FIFO is full when writing/empty when reading?



Example 1: Audio thread "sending" "downsampled" audio to UI thread to display as waveform
Example 2: Peak detection from multiple sensors

Costs of various FIFOs



Wait free on read

Wait free on write

Wait free on read and write

Not wait free on write or read

farbot's FIFO

- Supports all 16 variants
- Most general variant is 60% slower than boost's fifo
  - 5x faster than naïve solution (i.e. single producer, single consumer with spin locks)
- Other variants are comparable in speed with boost's fifo
- All variants are TSAN compatible (no false positives)

Alexander Krizhanovsky: http://natsys-lab.blogspot.com/2013/05/lock-free-multi-producer-multi-consumer.html

callAsync

- Defer non-realtime safe processing to non-realtime thread

```
callAsync([] () { std::cout << "Hello World!" << std::endl; });
```

- Lambda will be executed on non-realtime thread
- callAsync is realtime safe (even if lambda isn’t)
- Moving your lambda must be realtime safe!

Farbot's AsyncCaller

```
class AsyncCaller {
public:
    void callAsync(std::function<void()> && lambda) {
        auto success = queue.push(std::move(lambda));
        assert (success);
    }

    void process() {
        std::function<void()> lambda;
        while (queue.pop (lambda))
            lambda();
    }
private:
    fifo<std::function<void()>> queue;
};

AsyncCaller messageThreadExecutor;

void timerCallback() {
    messageThreadExecutor.process();
}

messageThreadExecutor.callAsync([] () { std::cout << "Hello World!" << std::endl; });
```

User needs to ensure that lambda is real-time movable

Called on realtime threads

Must not wake-up non-realtime thread as signalling another thread is not lock-free

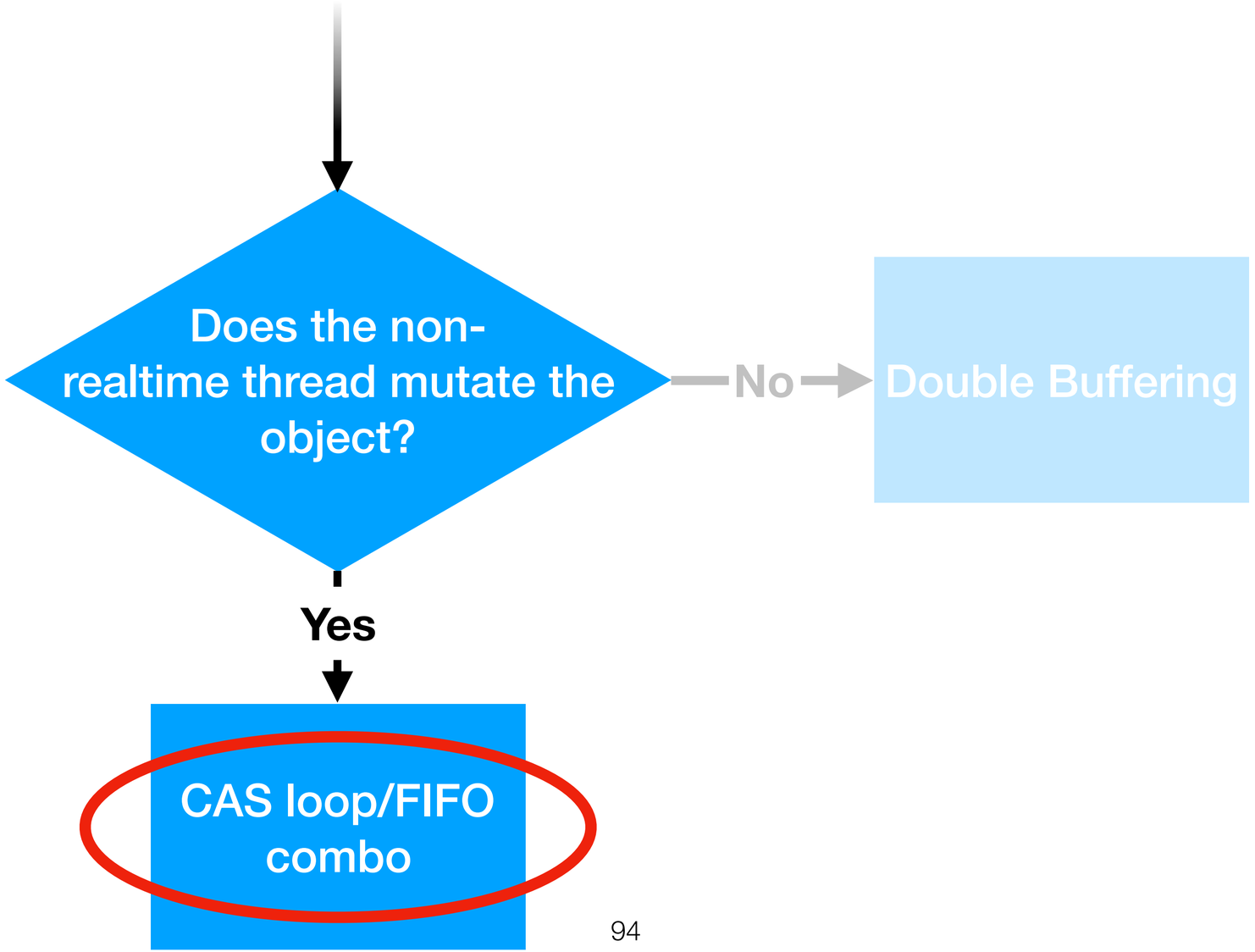
Pops lambdas from the queue

Lambdas are executed on a timer on the non-realtime thread

FIFO Summary

- Scenario:
  - Data is big: `std::atomic<>::is_always_lock_free == false`
  - Transferring objects between real-time and non-real-time threads
- Trade-off:
  - Static FIFO size
  - Behaviour when FIFO full (block/drop/overwrite)
  - Potential overhead of copying when writing and reading from the FIFO
- Examples:
  - Logging, writing input to disk (recording), reading from disk, dispatching

Both Mutating



Mutating on realtime and non-realtime

- It’s impossible to have multiple threads mutate an object without locking all the involved threads

- Choose a single thread to be in charge of mutating (can be a realtime thread) and other threads pass messages to this thread which describe the changes they want to make

Mutating on realtime and non-realtime

```
struct SourceList {
    std::array<const float*, MAX_SOURCES> buffers = {};
    int numSources = 0;
};

RealtimeMutatable<SourceList> sharedSourceList;
AsyncCaller realtimeThreadCaller;
```

- Realtime audio thread which mixes audio from multiple sources
- User can add/remove sources via GUI (i.e. non realtime thread)
- Sources can also be added/removed from realtime event streams (i.e. realtime thread)
- Realtime events should be processed instantaneously
  - We choose the realtime thread to be the mutating thread

Mutating on realtime and non-realtime

```
void addSource (const float* src) {
    if (! isRealtimeThread()) {
        realtimeThreadCaller.callAsync([src] () { addSource (src); });
        return;
    }
    RealtimeMutatable<SourceList>::ScopedAccess<true> sourceList (sharedSourceList);
    assert (sourceList->numSources < MAX_SOURCES);
    sourceList->buffers[sourceList->numSources++] = src;
}
```

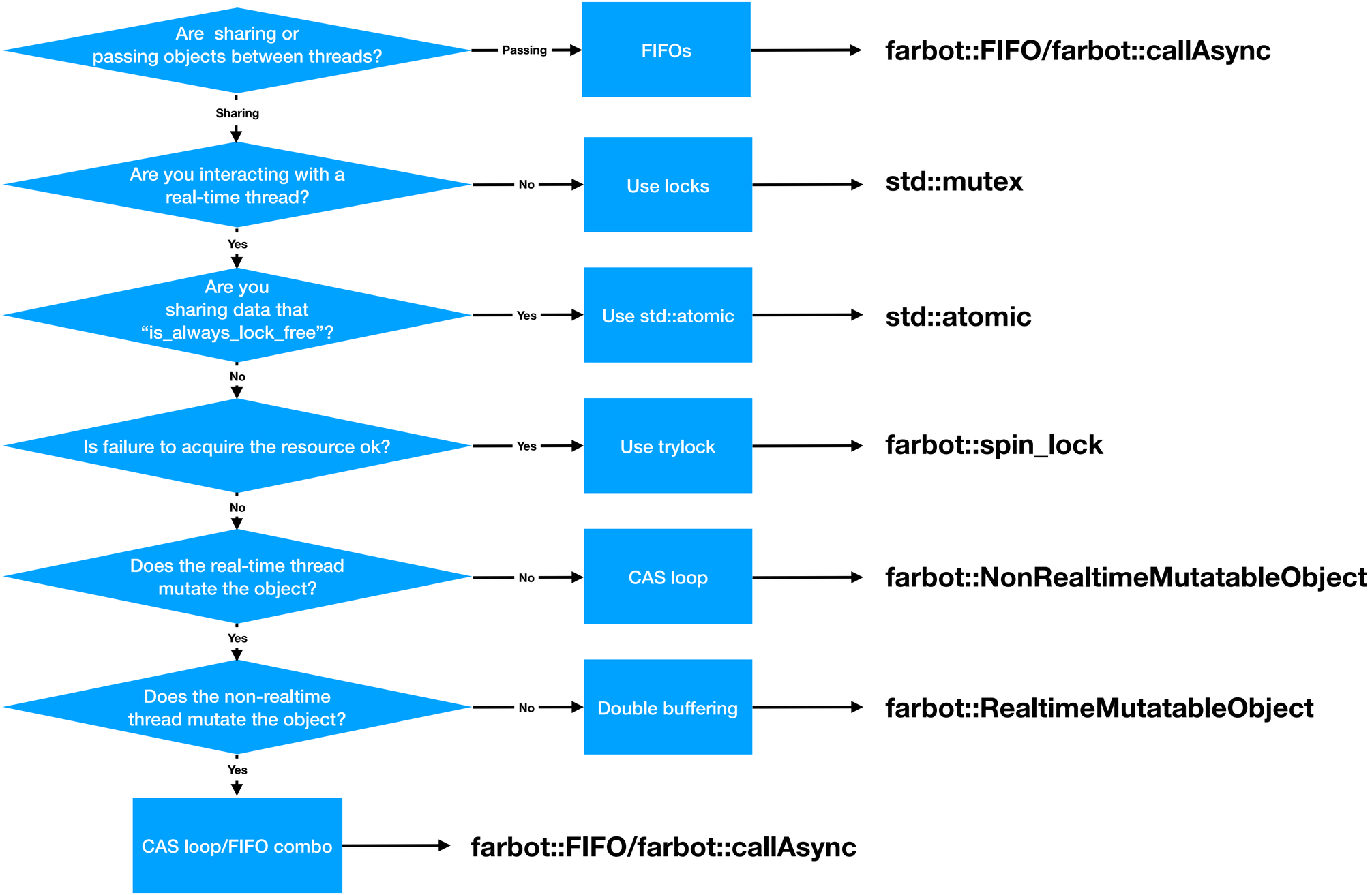
Mutating on realtime and non-realtime

```
void mixAllSources (float* output, char* realtimeEventMessages, int n) {
    processRealtimeEvents(realtimeEventMessages); // may add and remove sources
    realtimeThreadCaller.process(); // process all the lambdas

    RealtimeMutatable<SourceList>::ScopedAccess<true> sourceList (sharedSourceList);
    for (int i = 0; i < sourceList->numSources; ++i)
        mixSource (output, sourceList->buffers[i]);
}
```

```
void printSources() {
    RealtimeMutatable<SourceList>::ScopedAccess<false> sourceList (sharedSourceList);
    for (int i = 0; i < sourceList->numSources; ++i)
        std::cout << (void*)sourceList->buffers[i] << std::endl;
}
```

Real-time & Non-real-time Summary

- Scenario:
  - Data is big: `std::atomic<>::is_always_lock_free == false`
  - Sharing data between real-time and non-real-time threads
  - Both threads can mutate data
- Trade-off:
  - One thread needs to own the data
  - Same trade-offs as FIFOs & `(Non)RealTimeMutatableObjects`
  - Complexity
- Examples:
  - Managing lists and dynamic streams where losing packets is not acceptable


Are sharing or passing objects between threads?
Passing
FIFOs
farbot::FIFO/farbot::callAsync
Sharing
Are you interacting with a real-time thread?
No
Use locks
std::mutex
Yes
Are you sharing data that "is_always_lock_free"?
Yes
Use std::atomic
std::atomic
No
Is failure to acquire the resource ok?
Yes
Use trylock
farbot::spin_lock
No
Does the real-time thread mutate the object?
No
CAS loop
farbot::NonRealtimeMutatableObject
Yes
Does the non-realtime thread mutate the object?
No
Double buffering
farbot::RealtimeMutatableObject
Yes
CAS loop/FIFO combo
farbot::FIFO/farbot::callAsync


...and relax, you escaped hell

How to Debug

- Use tools such as Tsan
  - Make sure you test your production code
  - Changes such as durations can cause side effects like sleeping
  - Logging etc. can cause synchronisation events which will change your program structure
- Use `strace` to catch system calls on particular threads
- Use `LD_PRELOAD` to catch allocations and locking

Is there any way to do this statically?

How to Debug

- Example fifo:

```
bool push (T && arg) {
    …
    slots[pos] = std::move (arg);
    …
    return true;
}
```

Is push this lock and wait free?

Only if this is lock/wait-free movable

- We need: `static_assert (std::is_realtime_move_assignable<T>::value);`

Is this possible?

How to Debug

- `farbot::is_realtime_move_assignable` etc.:
  - Does the right thing for trivial and most common STL types
  - farbot statically asserts if this is not true in many places (for example in `farbot::fifo`)
  - You need to specialise `farbot::is_realtime_move_assignable` etc. for other types to tell farbot that it is safe to move/copy
- Perfect version needs to be recursive and cannot be implemented with current versions of C++

We need language support!

Summary

- **Don’t miss your deadlines!**
- Beware of hidden costs
- Follow the flow chart
- Use instrumentation to check your code

Real-time 101

David Rowland
@drowaudio

Fabian Renn-Giles
@hogliux

https://github.com/hogliux/farbot