NOZOMI NETWORKS

C++ micro-problems

lock-free

C++ Micro-Problems

- Herb Sutter (2013) “My Favorite C++ 10-Liner” https://channel9.msdn.com/Events/GoingNative/2013/My-Favorite-Cpp-10-Liner
- Indeed some* problems can be solved in 10 lines
- We will show you three of them

(*) Not *all* problems unfortunately, but we are working on it

SLIDEWARE

WARNING

MAY CONTAIN TRACES OF
SIMPLIFIED CONTENTS

Highlights

1. Lock-free basics: the atomic_list
2. Micro-problems
3. The law of the instrument

Highlights

1. Lock-free basics: the atomic_list
2. Micro-problems
3. The law of the instrument

Lock-free basics

- User-space based synchronization algorithms
  - No mutex involved
  - Do not rely on the OS to serialize operations
  - Waste some CPU time instead
- Pros and Cons:
  - *Insanely* low overhead in the fast path
  - Poor scalability under high contention (for hardware reasons)
  - Did I mention they are hard to optimize?

The atomic linked list

The atomic linked list

The atomic linked list

- R. Kent Treiber (1986) *Systems Programming: Coping with Parallelism*
  - A lock-free linked list with front, push-front and pop-front
- DOES IT REALLY WORK IN C++?
  - Not as written
- A SUBSET of the original code is just perfect
  - Remove pop-front to get the *swiss knife of lock free programming*

The atomic linked list

atomic_list.hpp

```
template <typename T>
class atomic_list
{
        struct node_t
        {
                T value;
                node_t* next;
        };

        std::atomic<node_t*> head_{nullptr};
```

The atomic push-front

atomic_list.hpp

```
template <typename U>
void push_front(U&& u)
{
        node_t* const n = new node_t{ std::forward<U>(u), nullptr };
        node_t* h = head_.load();
        do
        {
                n->next = h;
        }
        while (!head_.compare_exchange_weak(h, n));
}
```

The atomic linked list

Scenario #1: push-front + front

atomic_list.hpp

```
const T* front() const
{
        if (node_t* h = head_.load())
                return &(h->value);
        else
                return nullptr;
}
```

Scenario #2: push-front + pop-all

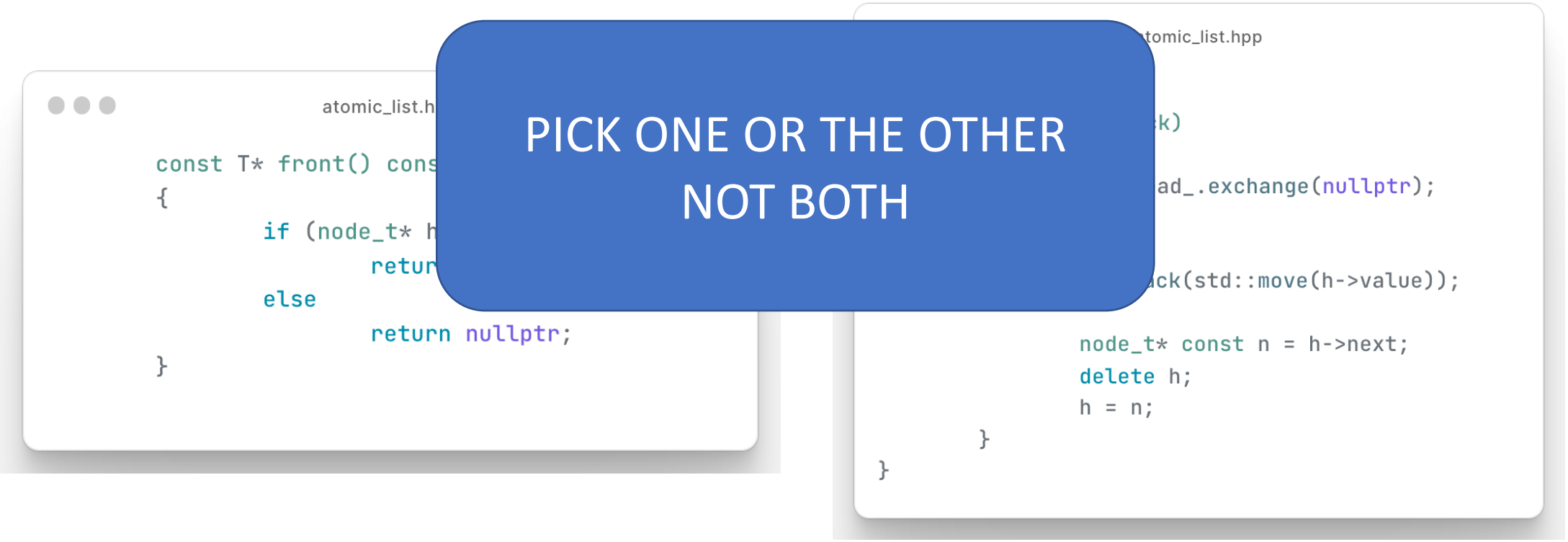
atomic_list.hpp

```
template <typename X>
void pop_all(X callback)
{
        node_t* h = head_.exchange(nullptr);
        while (h)
        {
                callback(std::move(h->value));

                node_t* const n = h->next;
                delete h;
                h = n;
        }
}
```

The atomic linked list

Scenario #1: push-front + front

Scenario #2: push-front + pop-all



The atomic linked list

- Other combinations/variations are UNSAFE
  - There is no control on the lifetime of nodes
  - We solve the problem by *never deleting*
- Note in fact there's no destructor
  - Even destruction of the list may be problematic
  - But it would be: `pop_all` with an empty callback

The atomic linked list

- Whenever `front` is safe, we can add the const-equivalent of `pop_all`

atomic_list.hpp

```
template <typename X>
const T* find_if(X predicate) const
{
  for (node_t* h = head_.load(); h; h = h->next)
  {
    if (predicate(h->value))
      return &h->value;
  }
  return nullptr;
}

template <typename X>
void visit(X callback) const
{
  for (node_t* h = head_.load(); h; h = h->next)
    callback(h->value);
}
```

The atomic linked list

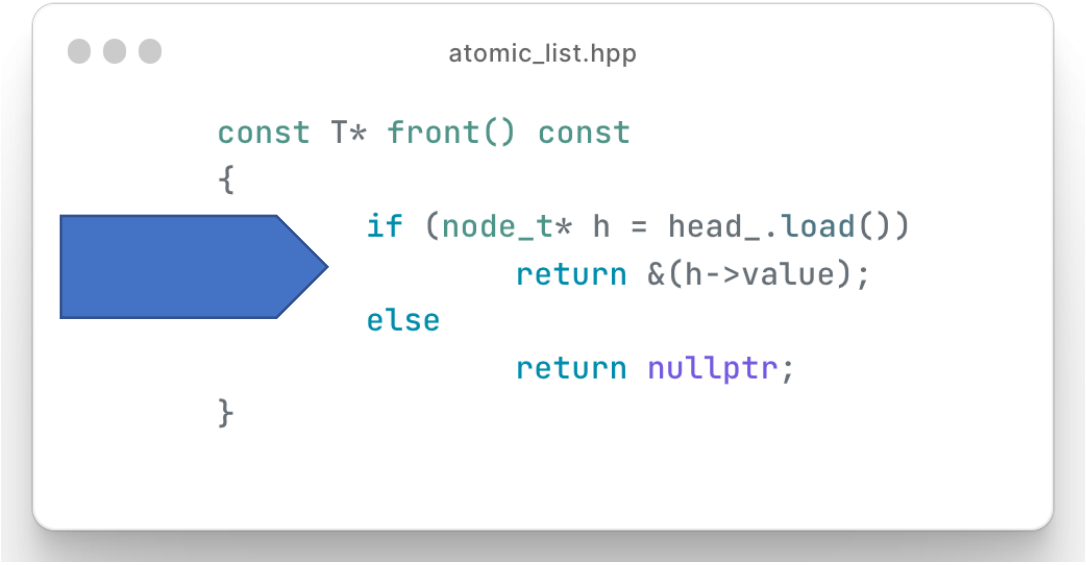
- Example 1: call `front` while a different thread runs `pop_all`

atomic_list.hpp

```
const T* front() const
{
        if (node_t* h = head_.load())
                return &(h->value);
        else
                return nullptr;
}
```

The atomic linked list

- Example 1: call `front` while a different thread runs `pop_all`

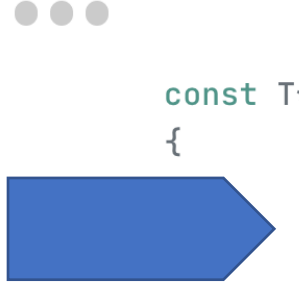


The atomic linked list

- Example 1: call `front` while a different thread runs `pop_all`

atomic_list.hpp

```
const T* front() const
{
        if (node_t* h = head_.load())
                return &(h->value);
        else
                return nullptr;
}
```



atomic_list.hpp

```
template <typename X>
void pop_all(X callback)
{
        node_t* h = head_.exchange(nullptr);
        while (h)
        {
                callback(std::move(h->value));

                node_t* const n = h->next;
                delete h;
                h = n;
        }
}
```

The atomic linked list

- Example 1: call `front` while a different thread runs `pop_all`

atomic_list.hpp

```
const T* front() const
{
        if (node_t* h = head_.load())
                return &(h->value);
        else
                return nullptr;
}
```

atomic_list.hpp

```
template <typename X>
void pop_all(X callback)
{
        node_t* h = head_.exchange(nullptr);
        while (h)
        {
                callback(std::move(h->value));

                node_t* const n = h->next;
                delete h;
                h = n;
        }
}
```

The atomic linked list

- Example 1: call `front` while a different thread runs `pop_all`

atomic_list.hpp

```
const T* front() const
{
        if (node_t* h = head_.load())
                return &(h->value);
        else
                return nullptr;
}
```

atomic_list.hpp

```
template <typename X>
void pop_all(X callback)
{
        node_t* h = head_.exchange(nullptr);
        while (h)
        {
                callback(std::move(h->value));

                node_t* const n = h->next;
                delete h;
                h = n;
        }
```

Delete object in use
(returning front by value does not fix)

The atomic linked list

- Example 2: Treiber's `pop_front`

The atomic linked list

- Example 2: Treiber's ... pop_front

atomic_list.hpp

```
template <typename U>
bool pop_front(U& result)
{
        node_t* h = head_.load();
        if (!h)
                return false;

        node_t* n = h->next;
        while (!head_.compare_exchange_weak(h, n))
                n = h->next;

        result = std::move(h->value);
        delete h;
        return true;
}
```

The atomic linked list

atomic_list.hpp

```
template <typename U>
bool pop_front(U& result)
{
                node_t* h = head_.load();
                if (!h)
                        return false;

                node_t* n = h->next;
                while (!head_.compare_exchange_weak(h, n))
                        n = h->next;

                result = std::move(h->value);
                delete h;
                return true;
}
```

The atomic linked list

Thread#1

atomic_list.hpp

```
template <typename U>
bool pop_front(U& result)
{
                node_t* h = head_.load();
                if (!h)
                        return false;

                node_t* n = h->next;
                while (!head_.compare_exchange_weak(h, n))
                        n = h->next;

                result = std::move(h->value);
                delete h;
                return true;
}
```

atomic_list.hpp

```
template <typename U>
bool pop_front(U& result)
{
                node_t* h = head_.load();
                if (!h)
                        return false;

                node_t* n = h->next;
                while (!head_.compare_exchange_weak(h, n))
                        n = h->next;

                result = std::move(h->value);
                delete h;
                return true;
}
```

The atomic linked list

Thread#1

atomic_list.hpp

```
template <typename U>
bool pop_front(U& result)
{
        node_t* h = head_.load();
        if (!h)
                return false;

        node_t* n = h->next;
        while (!head_.compare_exchange_weak(h, n))
                n = h->next;

        result = std::move(h->value);
        delete h;
        return true;
}
```

atomic_list.hpp

```
template <typename U>
bool pop_front(U& result)
{
        node_t* h = head_.load();
        if (!h)
                return false;

        node_t* n = h->next;
        while (!head_.compare_exchange_weak(h, n))
                n = h->next;

        result = std::move(h->value);
        delete h;
        return true;
}
```

The atomic linked list

Thread#1

atomic_list.hpp

```
template <typename U>
bool pop_front(U& result)
{
        node_t* h = head_.load();
        if (!h)
                return false;

        node_t* n = h->next;
        while (!head_.compare_exchange_weak(h, n))
                n = h->next;

        result = std::move(h->value);
        delete h;
        return true;
}
```

atomic_list.hpp

```
template <typename U>
bool pop_front(U& result)
{
        node_t* h = head_.load();
        if (!h)
                return false;

        node_t* n = h->next;
        while (!head_.compare_exchange_weak(h, n))
                n = h->next;

        result = std::move(h->value);
        delete h;
        return true;
```

The atomic linked list

Thread#1

atomic_list.hpp

```
template <typename U>
bool pop_front(U& result)
{
                node_t* h = head_.load();
                if (!h)
                        return false;

                node_t* n = h->next;
                while (!head_.compare_exchange_weak(h, n))
                        n = h->next;

                result = std::move(h->value);
                delete h;
                return true;
}
```

atomic_list.hpp

```
template <typename U>
bool pop_front(U& result)
{
                node_t* h = head_.load();
                if (!h)
                        return false;

                node_t* n = h->next;
                while (!head_.compare_exchange_weak(h, n))
                        n = h->next;

                result = std::move(h->value);
                delete h;
                return true;
```

The atomic linked list

Thread#1

atomic_list.hpp

```
template <typename U>
bool pop_front(U& result)
{
        node_t* h = head_.load();
        if (!h)
                return false;

        node_t* n = h->next;
        while (!head_.compare_exchange_weak(h, n))
                n = h->next;

        result = std::move(h->value);
        delete h;
        return true;
}
```

Read from deleted memory

Thread#2

atomic_list.hpp

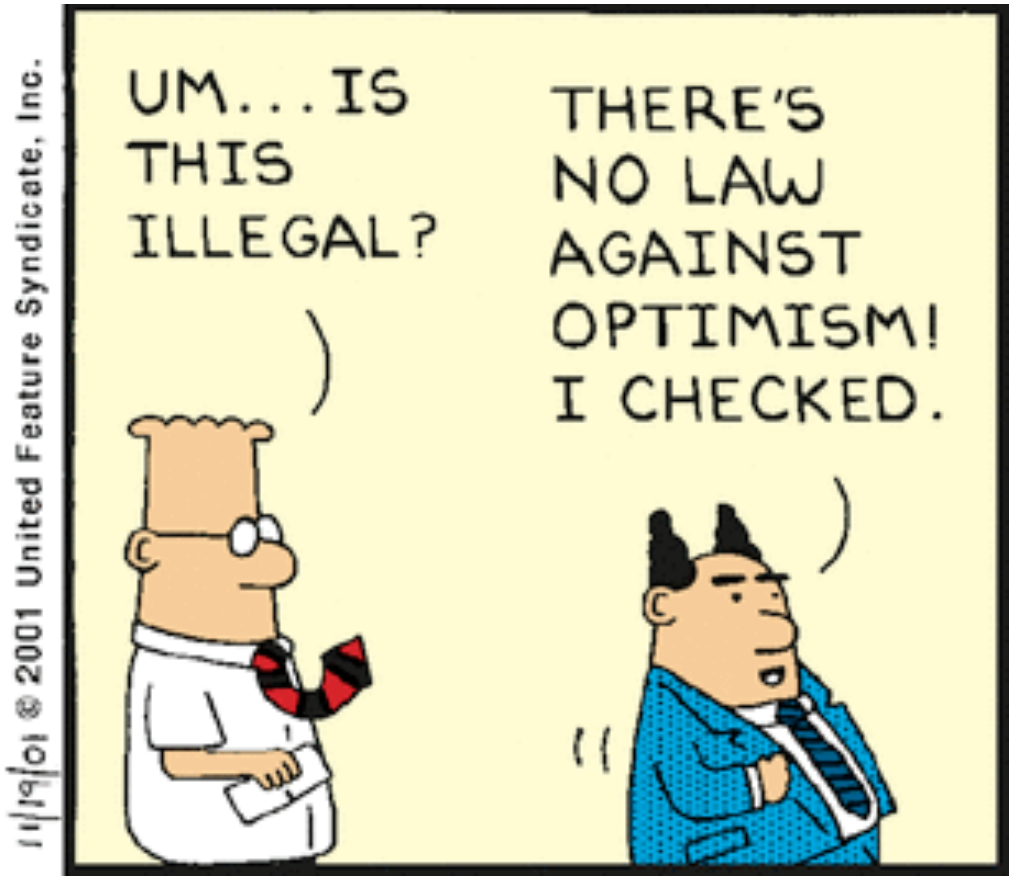
```
template <typename U>
bool pop_front(U& result)
{
        node_t* h = head_.load();
        if (!h)
                return false;

        node_t* n = h->next;
        while (!head_.compare_exchange_weak(h, n))
                n = h->next;

        result = std::move(h->value);
        delete h;
        return true;
```

How to (not) optimize

- Relax atomic memory order ❌
  - On average, low performance gain
  - Operations that are not «sequentially consistent» may not be sequenced
  - Intel processors are special
  - Testing is hard
- Pay attention to reduce sharing ✅
  - Often not usable in practice

https://www.1024cores.net/home/lock-free-algorithms/false-sharing---false

That’s enough for the pessimistic part…



Highlights

1. Lock-free basics: the atomic_list
2. Micro-problems
3. The law of the instrument

Highlights

1. Lock-free basics: the atomic_list
2. Micro-problems
3. The law of the instrument

Highlights

1. Lock-free basics: the atomic_list
2. Micro-problems
    1. Read often, write rarely
    2. MP queue
    3. Thread notifications
3. The law of the instrument

Problem #1: read often, write rarely

- We have a global variable that is
  - Not a trivial type (say T := std::string)
  - Read very often (by N threads)
  - Rarely updated (by any thread)
- The variable is accessed via two free/static functions
  - T GetGlobalValue()
  - SetGlobalValue(T newVal)
- Bonus points if you can make it
  - const T& GetGlobalValue()

Problem #1: read often, write rarely

- Naïve solution:

Problem #1: read often, write rarely

- Naïve solution:

Options.cpp

```
using T = std::string;

std::mutex myMutex; // global
T myName; // global

void SetMyName(const T& newVal)
{
  std::lock_guard g(myMutex);
  myName = newVal;
}

T GetMyName()
{
  std::lock_guard g(myMutex);
  return myName;
}
```

Problem #1: read often, write rarely

- Naïve solution:

Options.cpp

```
using T = std::string;

std::mutex myMutex; // global
T myName; // global

void SetMyName(const T& newVal)
{
  std::lock_guard g(myMutex);
  myName = newVal;
}

T GetMyName()
{
  std::lock_guard g(myMutex);
  return myName;
}
```

copy

Problem #1: read often, write rarely

- Naïve solution:

Options.cpp

```
using T = std::string;

std::mutex myMutex; // global
T myName; // global

void SetMyName(const T& newVal)
{
  std::lock_guard g(myMutex);
  myName = newVal;
}

T GetMyName()
{
  std::lock_guard g(myMutex);
  return myName;
}
```

Problem #1: read often, write rarely

- Naïve solution:

Options.cpp

```
using T = std::string;

std::mutex myMutex; // global
T myN

void

  std
  myN
}

T GetMyName()
{
  std::lock_guard g(myMutex);
  return myName;
}
```

SIDE NOTE

To see a good lock-free solution,
you may have to question something
{hypotheses, properties, …}
that *superficially* looks fundamental

Problem #1: read often, write rarely

- Naïve solution:

SIDE NOTE

To see a good lock-free solution,
you may have to question something
{hypotheses, properties, …}
that *superficially* looks fundamental

- Do I really need to keep a single value in memory?

Problem #1: read often, write rarely

- Solution:
  - *Never delete old values*
  - Use a global `atomic_list<T>` with front/push_front

- Optional:
  - Clear the list in the destructor (it may be acceptable to leak)
  - Wrap `front` to return a default instead of `nullptr`

Problem #1: read often, write rarely

Options.cpp

```cpp
#include "atomic_list.h"

struct NameList
{
   atomic_list<std::string> data;

   ~NameList()
   {
     data.pop_all([](auto&&){});
   }
} myName;

const std::string& GetMyName()
{
  static const std::string empty{};
  if (auto ptr = myName.data.front())
    return *ptr;
  else
    return empty;
}

void SetMyName(std::string name)
{
  myName.data.push_front(std::move(name));
}
```

Problem #1: read often, write rarely

reference

Options.cpp

```cpp
#include "atomic_list.h"

struct NameList
{
   atomic_list<std::string> data;

   ~NameList()
   {
     data.pop_all([](auto&&){});
   }
} myName;

const std::string& GetMyName()
{
  static const std::string empty{};
  if (auto ptr = myName.data.front())
    return *ptr;
  else
    return empty;
}

void SetMyName(std::string name)
{
  myName.data.push_front(std::move(name));
}
```

Problem #2: MP queue

- Producers and consumers (threads)
- An object of type T represents some «work» to do
- Producers create T's and *move* them in a «shared list»
  - Order is not relevant
- Consumer(s) remove T's from the «shared list»
  - Work items are «consumed»

Problem #2: MP queue

- ACTIVE LOGGING
  - Many threads produce strings that must be appended to a log file
  - One thread gathers them, adds a timestamp, etc. and writes

- Solution: use an atomic_list<T> with push_front / pop_all

Problem #2: MP queue

ActiveLog.h

```
#include <...>

using LogLine = std::tuple<time_t, const char*, std::string, long>;

atomic_list<LogLine>& LogQueue();

#define LOG_D(...)  \
    LogQueue().push_front(LogLine{now(), "DEBUG", fmt::format(__VA_ARGS__), __LINE__})
```

Problem #2: MP queue

ActiveLog.cpp

```
#include <...>

struct ActiveLog
{
  atomic_list<LogLine> logLines;
  std::atomic<bool> logExit = false;

  std::ofstream logFile;
  std::thread logWriter;

  void Flush();
  ActiveLog(const char* file);
  ~ActiveLog();
}

// ...

ActiveLog theLog("/var/log/async.log");

atomic_list<LogLine>& LogQueue()
{
  return theLog.logLines;
}
```

Problem #2: MP queue

Let's zoom here…

ActiveLog.cpp

```
#include <...>

struct ActiveLog
{
  atomic_list<LogLine> logLines;
  std::atomic<bool> logExit = false;

  std::ofstream logFile;
  std::thread logWriter;

  void Flush();
  ActiveLog(const char* file);
  ~ActiveLog();
}

// ...

ActiveLog theLog("/var/log/async.log");

atomic_list<LogLine>& LogQueue()
{
  return theLog.logLines;
}
```

Problem #2: MP queue

ActiveLog.cpp

```
#include <...>

struct ActiveLog
{
  atomic_list<LogLine> logLines;
  std::atomic<bool> logExit = false;

  std::ofstream logFile;
  std::thread logWriter;

  void Flush();
  ActiveLog(const char* file);
  ~ActiveLog();
}

// ...

ActiveLog theLog("/var/log/async.log");

atomic_list<LogLine>& LogQueue()
{
  return theLog.logLines;
}
```

Problem #2: MP queue

ActiveLog.cpp

```
ActiveLog::ActiveLog(const char* file) : logFile(file)
{
  logWriter = std::thread([this]{
    while (!logExit)
    {
      std::this_thread::sleep_for(2_s);
      Flush();
    }
  });
}

ActiveLog::~ActiveLog()
{
  logExit = true;
  logWriter.join();
  Flush();
}
```

ActiveLog.cpp

```
void ActiveLog::Flush()
{
  std::vector<std::string> tmp;
  logLines.pop_all([&tmp](auto&& t) {
    auto& [time, label, msg, line] = t;
    tmp.push_back(fmt::format(...));
  });

  std::sort(tmp.begin(), tmp.end());
  for (auto& line : tmp)
    logFile << line << '\n';

  logFile << std::flush;
}
```

Problem #2: MP queue

ActiveLog.cpp

```
ActiveLog::ActiveLog(const char* file) : logFile(file)
{
  logWriter = std::thread([this]{
    while (!logExit)
    {
      std::this_thread::sleep_for(2_s);
      Flush();
    }
  });
}

ActiveLog::~ActiveLog()
{
  logExit = true;
  logWriter.join();
  Flush();
}
```

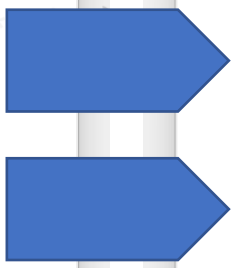
ActiveLog.cpp

```
void ActiveLog::Flush()
{
  std::vector<std::string> tmp;
  logLines.pop_all([&tmp](auto&& t) {
    auto& [time, label, msg, line] = t;
    tmp.push_back(fmt::format(...));
  });

  std::sort(tmp.begin(), tmp.end());
  for (auto& line : tmp)
    logFile << line << '\n';

  logFile << std::flush;
}
```

Problem #2: MP queue

ActiveLog.cpp

```
ActiveLog::ActiveLog(const char* file) : logFile(file)
{
  logWriter = std::thread([this]{
    while (!logExit)
    {
      std::this_thread::sleep_for(2_s);
      Flush();
    }
  });
}

ActiveLog::~ActiveLog()
{
  logExit = true;
  logWriter.join();
  Flush();
}
```



ActiveLog.cpp

```
void ActiveLog::Flush()
{
  std::vector<std::string> tmp;
  logLines.pop_all([&tmp](auto&& t) {
    auto& [time, label, msg, line] = t;
    tmp.push_back(fmt::format(...));
  });

  std::sort(tmp.begin(), tmp.end());
  for (auto& line : tmp)
    logFile << line << '\n';

  logFile << std::flush;
}
```

Problem #2: MP queue

ActiveLog.cpp

```
ActiveLog::ActiveLog(const char* file) : logFile(file)
{
  logWriter = std::thread([this]{
    while (!logExit)
    {
      std::this_thread::sleep_for(2_s);
      Flush();
    }
  });
}

ActiveLog::~ActiveLog()
{
  logExit = true;
  logWriter.join();
  Flush();
}
```

Delegate some work to the writer thread

ActiveLog.cpp

```
void ActiveLog::Flush()
{
  std::vector<std::string> tmp;
  logLines.pop_all([&tmp](auto&& t) {
    auto& [time, label, msg, line] = t;
    tmp.push_back(fmt::format(...));
  });

  std::sort(tmp.begin(), tmp.end());
  for (auto& line : tmp)
    logFile << line << '\n';

  logFile << std::flush;
}
```

Problem #3: Thread notifications

- Suppose we have a thread-safe global «map» where multiple threads execute concurrent lookups
- Assume for simplicity that the global map is read-only and is occasionally reloaded
  - Think anti-virus signatures…

Lookup.cpp

```
// this is expensive
return GlobalMap().Lookup(myKey);
```

Problem #3: Thread notifications

- As lookups are expensive, we decide to add a thread-local cache

Problem #3: Thread notifications

Lookup.cpp

```
struct LazyValue
{
        K& myKey;

        operator V() const
        { return GlobalMap().Lookup(myKey); }
};

thread_local unordered_map<K, V> myCache;
auto [iter, added] = myCache.try_emplace(myKey, LazyValue{myKey});
return iter->second;
```

Problem #3: Thread notifications

- As lookups are expensive, we decide to add a thread-local cache
- But how do we *clear* the local caches?

Lookup.cpp

```
struct LazyValue
{
        K& myKey;

        operator V() const
        { return GlobalMap().Lookup(myKey); }
};

thread_local unordered_map<K, V> myCache;
auto [iter, added] = myCache.try_emplace(myKey, LazyValue{myKey});
return iter->second;
```

Problem #3: Thread notifications

- As lookups are expensive, we decide to add a thread-local cache
- But how do we *clear* the local caches?

Problem #3: Thread notifications

- Solution: a global «hub» that can broadcast a boolean
- A thread-local message is connected to the hub on construction

a_different_thread.cpp

```
{
  // ...
  h.broadcast(true);
}
```

Lookup.cpp

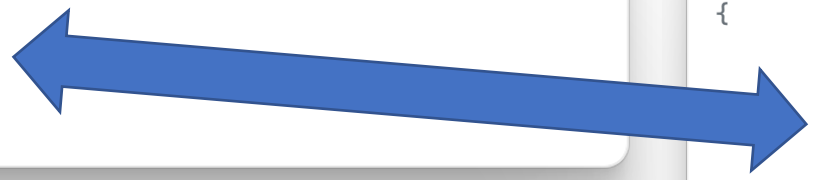
```
notification_hub<bool> h;

V SomeFunc(K myKey)
{
        thread_local unordered_map<K, V> myCache;

        thread_local notification_hub<bool>::message msg(h);
        if (msg.get_and_clear())
                myCache.clear();

        auto [iter, added] = myCache.try_emplace(myKey, LazyValue{myKey});
        // ...
}
```

Problem #3: Thread notifications

- Solution: a global «hub» that can broadcast a boolean
- A thread-local message is connected to the hub on construction

a_different_thread.cpp

```
{
  // ...
  h.broadcast(true);
}
```



Lookup.cpp

```
notification_hub<bool> h;

V SomeFunc(K myKey)
{
        thread_local unordered_map<K, V> myCache;

        thread_local notification_hub<bool>::message msg(h);
        if (msg.get_and_clear())
                myCache.clear();

        auto [iter, added] = myCache.try_emplace(myKey, LazyValue{myKey});
        // ...
}
```

Problem #3: Thread notifications

- The ”hub” is a linked list of reference-counted nodes
- A thread_local message
  - pushes a node in the hub on creation
  - …and keeps a reference to it
  - So the reference count is 2 (1 for the hub, 1 for the thread)

- `hub.broadcast` walks the list and stores `true` in every node
  - then it’s up to each thread to pick up the message

Problem #3: Thread notifications

notification_hub.h

```
template <typename T>
class notification_hub
{
    struct node_t
    {
        std::atomic<T> value{T()};
        std::atomic<int> ref_count{2};
        node_t* next{nullptr};

        static void push_front(std::atomic<node_t*>& head, node_t* const n);
        static void release(node_t* const n);
    };

    std::atomic<node_t*> head_ { nullptr };

    template <typename X> void on_nodes(X action);
    template <typename X> void on_messages(X action);

  public:

    void broadcast(T value)
    {
        on_messages([value](auto& msg) { msg.store(value); });
    }

    ~notification_hub()
    {
        on_nodes([](node_t* n) { node_t::release(n); });
    }
```

Problem #3: Thread notifications

notification_hub.h

```
template <typename T>
class notification_hub
{
    struct node_t
    {
        std::atomic<T> value{T()};
        std::atomic<int> ref_count{2};
        node_t* next{nullptr};

        static void push_front(std::atomic<node_t*>& head, node_t* const n);
        static void release(node_t* const n);
    };

    std::atomic<node_t*> head_ { nullptr };

    template <typename X> void on_nodes(X action);
    template <typename X> void on_messages(X action);

  public:

    void broadcast(T value)
    {
        on_messages([value](auto& msg) { msg.store(value); });
    }

    ~notification_hub()
    {
        on_nodes([](node_t* n) { node_t::release(n); });
    }
```

Problem #3: Thread notifications

notification_hub.h

```
template <typename T>
class notification_hub
{
    struct node_t
    {
        std::atomic<T> value{T()};
        std::atomic<int> ref_count{2};
        node_t* next{nullptr};

        static void push_front(std::atomic<node_t*>& head, node_t* const n);
        static void release(node_t* const n);
    };

    std::atomic<node_t*> head_ { nullptr };

    template <typename X> void on_nodes(X action);
    template <typename X> void on_messages(X action);

  public:

    void broadcast(T value)
    {
        on_messages([value](auto& msg) { msg.store(value); });
    }

    ~notification_hub()
    {
        on_nodes([](node_t* n) { node_t::release(n); });
    }
```

visit

Problem #3: Thread notifications

notification_hub.h

```
template <typename T>
class notification_hub
{
    struct node_t
    {
        std::atomic<T> value{T()};
        std::atomic<int> ref_count{2};
        node_t* next{nullptr};

        static void push_front(std::atomic<node_t*>& head, node_t* const n);
        static void release(node_t* const n);
    };

    std::atomic<node_t*> head_ { nullptr };

    template <typename X> void on_nodes(X action);
    template <typename X> void on_messages(X action);

  public:

    void broadcast(T value)
    {
        on_messages([value](auto& msg) { msg.store(value); });
    }

    ~notification_hub()
    {
        on_nodes([](node_t* n) { node_t::release(n); });
    }
```

Problem #3: Thread notifications

notification_hub.h

```
template <typename T>
class notification_hub
{
    struct node_t
    {
        std::atomic<T> value{T()};
        std::atomic<int> ref_count{2};
        node_t* next{nullptr};

        static void push_front(std::atomic<node_t*>& head, node_t* const n);
        static void release(node_t* const n);
    };

    std::atomic<node_t*> head_ { nullptr };

    template <typename X> void on_nodes(X action);
    template <typename X> void on_messages(X action);

  public:

    void broadcast(T value)
    {
        on_messages([value](auto& msg) { msg.store(value); });
    }

    ~notification_hub()
    {
        on_nodes([](node_t* n) { node_t::release(n); });
    }
```

notification_hub.h

```
    class message
    {
        node_t* const self_;

      public:
        message(notification_hub& h)
        : self_{new node_t}
        {
            node_t::push_front(h.head_, self_);
        }

        ~message() {
            node_t::release(self_);
        }

        T get_and_clear() {
            return self_->value.exchange(T{});
        }

        bool connected() const {
            return self_->ref_count == 2;
        }
    };
```

Problem #3: Thread notifications

notification_hub.h

```
template <typename T>
class notification_hub
{
    struct node_t
    {
        std::atomic<T> value{T()};
        std::atomic<int> ref_count{2};
        node_t* next{nullptr};

        static void push_front(std::atomic<node_t*>& head, node_t* const n);
        static void release(node_t* const n);
    };

    std::atomic<node_t*> head_ { nullptr };

    template <typename X> void on_nodes(X action);
    template <typename X> void on_messages(X action);

  public:

    void broadcast(T value)
    {
        on_messages([value](auto& msg) { msg.store(value); });
    }

    ~notification_hub()
    {
        on_nodes([](node_t* n) { node_t::release(n); });
    }
```

notification_hub.h

```
class message
{
    node_t* const self_;

  public:
    message(notification_hub& h)
    : self_{new node_t}
    {
        node_t::push_front(h.head_, self_);
    }

    ~message() {
        node_t::release(self_);
    }

    T get_and_clear() {
        return self_->value.exchange(T{});
    }

    bool connected() const {
        return self_->ref_count == 2;
    }
};
```

Problem #3: Thread notifications

notification_hub.h

```
void node_t::push_front(std::atomic<node_t*>& head, node_t* const n)
{
    node_t* h = head.load();
    do
    {
        n->next = h;
    }
    while (!head.compare_exchange_weak(h, n));
}

void node_t::release(node_t* const n)
{
    if (!(--n->ref_count))
        delete n;
}
```

notification_hub.h

```
template <typename X>
void on_nodes(X action)
{
    node_t* n = head_.load();
    while (n) {
        node_t* const next = n->next;
        action(n);
        n = next;
    }
}

template <typename X>
void on_messages(X action)
{
    on_nodes([&](node_t* n) {
        if (n->ref_count == 2) action(n->value);
    });
}
```

Problem #3: Thread notifications

notification_hub.h

```
void node_t::push_front(std::atomic<node_t*>& head, node_t* const n)
{
    node_t* h = head.load();
    do
    {
        n->next = h;
    }
    while (!head.compare_exchange_weak(h, n));
}

void node_t::release(node_t* const n)
{
    if (!(--n->ref_count))
        delete n;
}
```

notification_hub.h

```
template <typename X>
void on_nodes(X action)
{
    node_t* n = head_.load();
    while (n) {
        node_t* const next = n->next;
        action(n);
        n = next;
    }
}

template <typename X>
void on_messages(X action)
{
    on_nodes([&](node_t* n) {
        if (n->ref_count == 2) action(n->value);
    });
}
```

Problem #3: Thread notifications

notification_hub.h

```
void node_t::push_front(std::atomic<node_t*>& head, node_t* const n)
{
    node_t* h = head.load();
    do
    {
        n->next = h;
    }
    while (!head.compare_exchange_weak(h, n));
}

void node_t::release(node_t* const n)
{
    if (!(--n->ref_count))
        delete n;
}
```

visit

notification_hub.h

```
template <typename X>
void on_nodes(X action)
{
    node_t* n = head_.load();
    while (n) {
        node_t* const next = n->next;
        action(n);
        n = next;
    }
}

template <typename X>
void on_messages(X action)
{
    on_nodes([&](node_t* n) {
        if (n->ref_count == 2) action(n->value);
    });
}
```

Highlights

1. Lock-free basics: the atomic_list
2. Micro-problems
3. The law of the instrument

Highlights

1. Lock-free basics: the atomic_list
2. Micro-problems
3. The law of the instrument

The law of the instrument

- To a man with a hammer, everything looks like a nail
- To a man with an atomic_list, everything looks like a push_front

- When you have an apparently unrelated problem, consider solving it with an atomic_list

The law of the instrument

NOZOMI NETWORKS

Thank You!

Nozomi Networks accelerates digital transformation by protecting the world's critical infrastructure, industrial and government organizations from cyber threats. Our solution delivers exceptional network and asset visibility, threat detection, and insights for OT and IoT environments. Customers rely on us to minimize risk and complexity while maximizing operational resilience.

nozominetworks.com

Bonus slide: an atomic hashtable

- Assume atomic_list is written as:

atomic_list.hpp

```
template <typename U>
void push_front(U&& u)
{
  push_front(new node_t { std::forward<U>(u), nullptr });
}

void push_front(node_t* n)
{
  // as before...
```

Bonus slide: an atomic hashtable

- Using T= std::tuple<std::atomic<size_t>, const K, V>
- Create an array of atomic_list<T>
  - Reserve is mandatory
- Find scans the appropriate bucket
- Insert is a push_front in the appropriate bucket
  - Yes, we have duplicated elements…
  - …But find will stop on the first (i.e. the last insert)
  - Erase just flips the bits of the hash

Bonus slide: an atomic hashtable

Bonus slide: an atomic hashtable

atomic_hashtable.hpp

```
template <typename K, typename V>
class atomic_hashtable
{
  enum { HASH, KEY, VALUE };
  using tuple_t = std::tuple<std::atomic<size_t>, const K, V>;
  using list_t = atomic<list<tuple_t>>;

  std::unique_ptr<list_t []> bucket_;
  const unsigned size_;

 public:
    atomic_hashtable(unsigned n) // reserve is mandatory
    : bucket_(new list_t[n]), size_(n)
    {
    }

    const V* find(const K& key) const;
    void insert(const K& key, const V& value);
    void erase(const K& key);
};
```

Bonus slide: an atomic hashtable

atomic_hashtable.hpp

```
template <typename K, typename V>
class atomic_hashtable
{
  enum { HASH, KEY, VALUE };
  using tuple_t = std::tuple<std::atomic<size_t>, const K, V>;
  using list_t = atomic<list<tuple_t>>;

  std::unique_ptr<list_t []> bucket_;
  const unsigned size_;

 public:
    atomic_hashtable(unsigned n) // reserve is mandatory
    : bucket_(new list_t[n]), size_(n)
    {
    }

    const V* find(const K& key) const;
    void insert(const K& key, const V& value);
    void erase(const K& key);
};
```

atomic_hashtable.hpp

```
const V* find(const K& key) const
{
  const auto hash = std::hash<K>{}(key);
  auto elem = bucket_[hash % size_].find_if([&](const tuple_t& t)
  {
    return std::get<HASH>(t) == hash && std::get<KEY>(t) == key;
  });

  return elem ? &std::get<VALUE>(*elem) : nullptr;
}

void insert(const K& key, const V& value)
{
  const auto hash = std::hash<K>{}(key);
  bucket_[hash % size_].push_front(new tuple_t { hash, key, value });
}

void erase(const K& key)
{
  const auto hash = std::hash<K>{}(key);
  bucket_[hash % size_].visit([&](tuple_t& t)
  {
    if (std::get<HASH>(t) == hash && std::get<KEY>(t) == key)
      std::get<HASH>(t) ^= ~0ull;
  });
}
```