Single Producer Single Consumer Lock-free FIFO From the Ground Up

CHARLES FRASCH

Cppcon
The C++ Conference

2023
October 01 - 06

Charlie Frasch
charles.frasch@gmail.com

Senior Core Developer - IEX Group
https://www.iex.io/

Code at: https://github.com/CharlesFrasch/cppcon2023

“The views expressed are my own and may not reflect the views of IEX Group”

Why another SPSC Fifo when you can get one from reliable sources such as Boost.Lockfree?

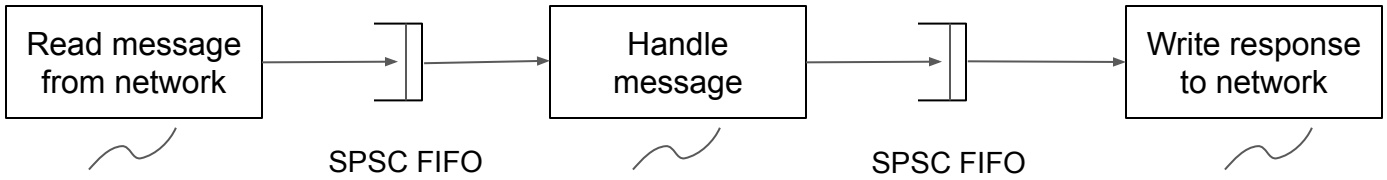
- Writing such a fifo is a fairly gentle introduction to lock free programming.
- There are some interesting performance optimizations that can be made.
- You may have some specific requirements that are not met in out-of-the box implementations.

https://www.boost.org/doc/libs/1_82_0/boost/lockfree/spsc_queue.hpp
https://github.com/rigtorp/SPSCQueue
https://www.1024cores.net/home/lock-free-algorithms/queues/unbounded-spsc-queue
https://www.dpdk.org/

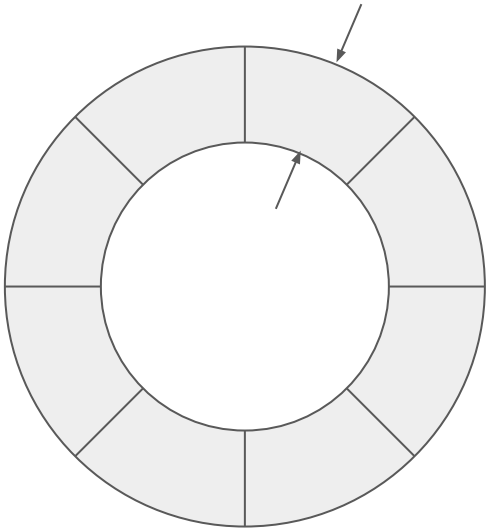
Single Producer Single Consumer Lock-Free Wait-Free Fifo

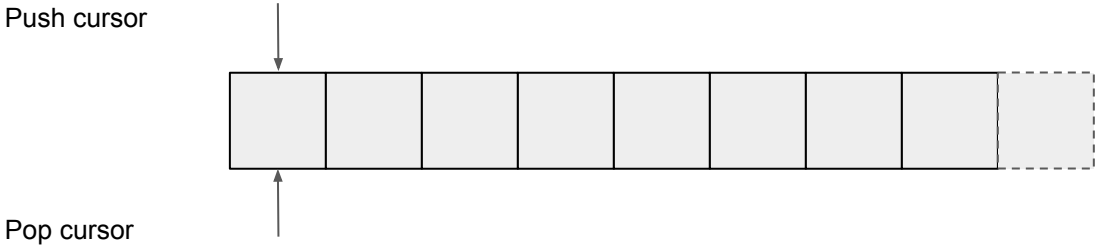
- Single producer: one producer (aka writer) thread
- Single consumer: one consumer (aka reader) thread
- Lock-free: it doesn’t use mutex locks. At any point of time, some thread will make progress.
- Wait-free: each thread moves forward regardless of other threads.
- [Circular] Fifo [or Queue]: a single, fixed-size buffer as if it were connected end-to-end. The oldest entry is processed first.

Using SPSC FIFOs to communicate between threads in a pipeline

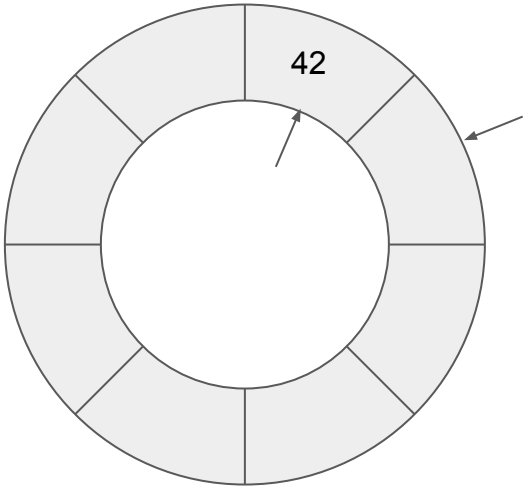


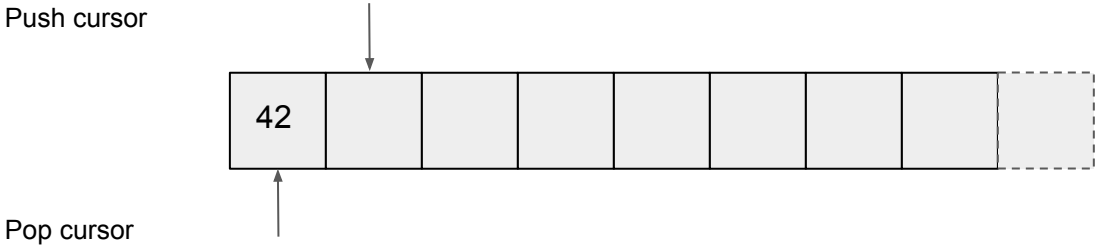
Circular Fifo - Push



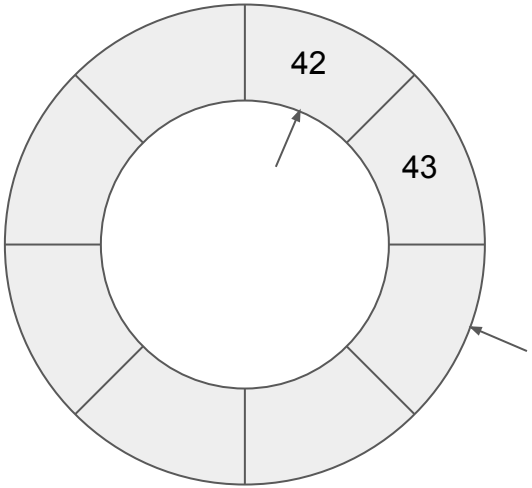


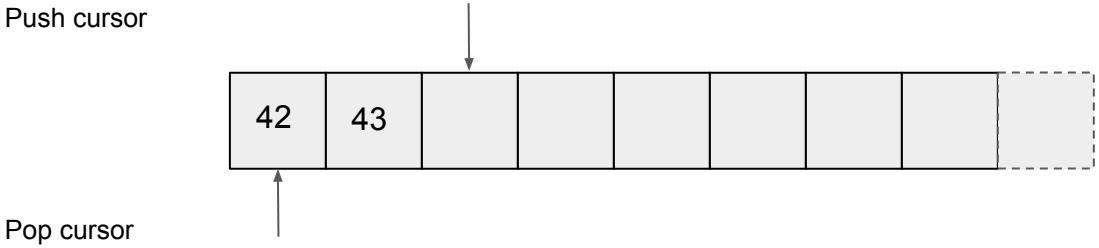
Circular Fifo - Push



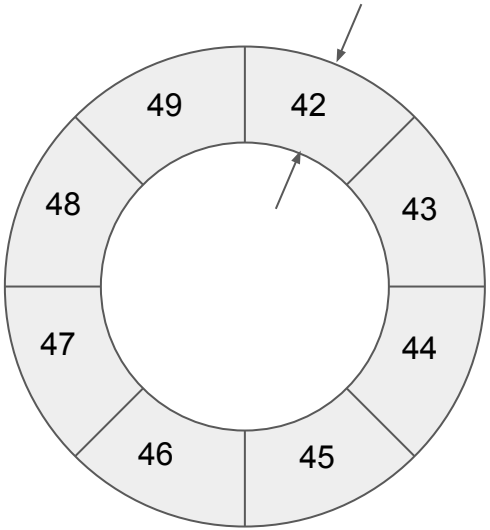


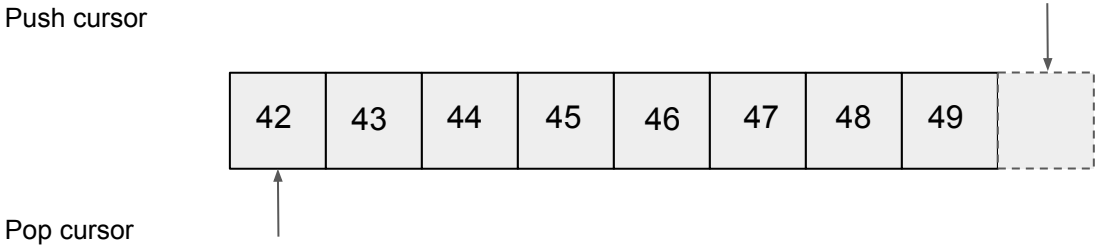
Circular Fifo - Push



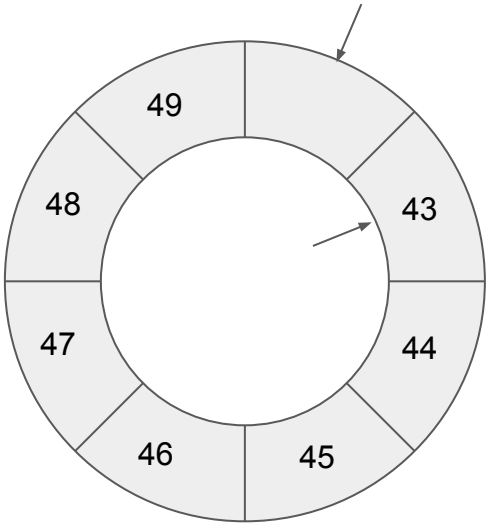


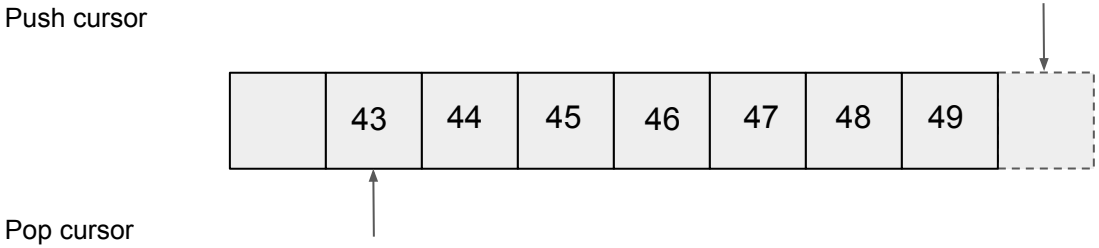
Circular Fifo - Push



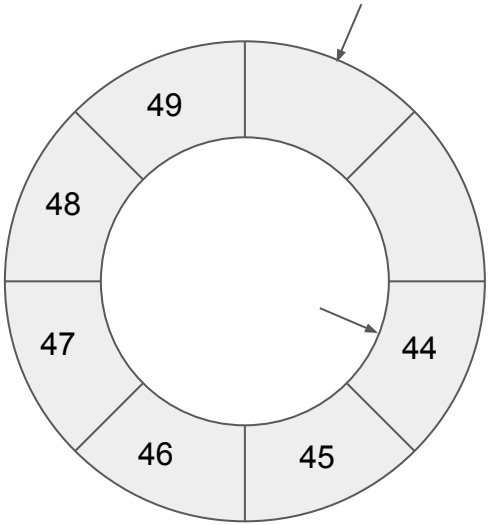


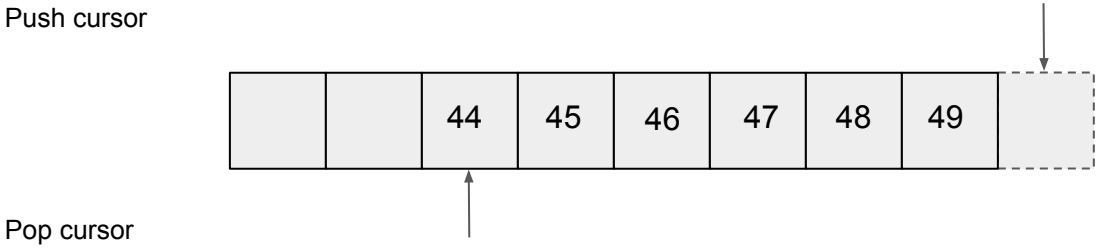
Circular Fifo - Pop



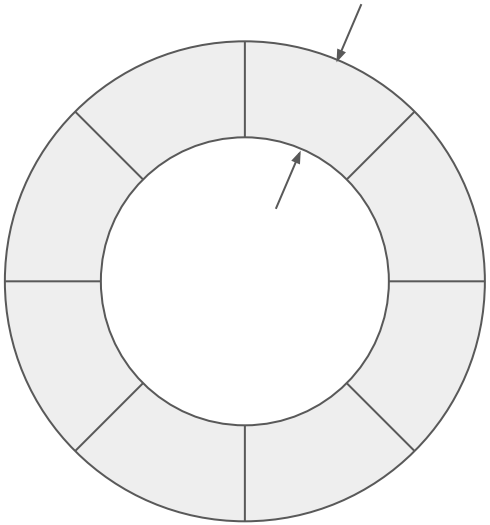


Circular Fifo - Pop

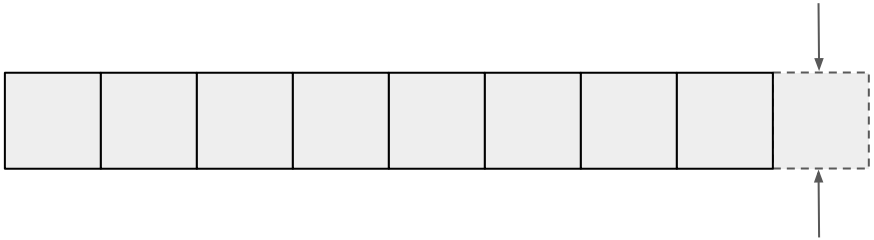




Circular Fifo - Pop



Push cursor



Pop cursor

```
template<typename T, typename Alloc = std::allocator<T>>
class Fifo1 : private Alloc
{
  std::size_t capacity_;
  T* ring_;
  std::size_t pushCursor_{};
  std::size_t popCursor_{};

public:
  explicit Fifo1(std::size_t capacity, Alloc const& alloc = Alloc{})
    : Alloc{alloc}, capacity_{capacity}, ring_{std::allocator_traits::allocate(*this, capacity)}
  {}
  ~Fifo1() {
    while(not empty()) {
      ring_[popCursor_ % capacity_].~T();
      ++popCursor_;
    }
    std::allocator_traits::deallocate(*this, ring_, capacity_);
  }

  auto capacity() const { return capacity_; }
  auto size() const { return pushCursor_ - popCursor_; }
  auto empty() const { return size() == 0; }
  auto full() const { return size() == capacity(); }

  auto push(T const& value);
  auto pop(T* value);
};
```

```
auto Fifo1::push(T const& value) {
  if (full()) {
      return false;
  }
  new (&ring_[pushCursor_ % capacity_]) T(value);
  ++pushCursor_;
  return true;
}

auto Fifo1::pop(T& value) {
  if (empty()) {
    return false;
  }
  value = ring_[popCursor_ % capacity_];
  ring_[popCursor_ % capacity_].~T();
  ++popCursor_;
  return true;
}
```

```
cfrasch@Charles-PC:~/cppcon2023/build/debug$ ./unitTests --gtest_filter=FifoTest/0.*
Running main() from ./googletest/src/gtest_main.cc
Note: Google Test filter = FifoTest/0.*
[==========] Running 6 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 6 tests from FifoTest/0, where TypeParam = Fifo1<int>
[ RUN      ] FifoTest/0.properties
[       OK ] FifoTest/0.properties (0 ms)
[ RUN      ] FifoTest/0.initialConditions
[       OK ] FifoTest/0.initialConditions (0 ms)
[ RUN      ] FifoTest/0.push
[       OK ] FifoTest/0.push (0 ms)
[ RUN      ] FifoTest/0.pop
[       OK ] FifoTest/0.pop (0 ms)
[ RUN      ] FifoTest/0.popFullFifo
[       OK ] FifoTest/0.popFullFifo (0 ms)
[ RUN      ] FifoTest/0.popEmpty
[       OK ] FifoTest/0.popEmpty (0 ms)
[----------] 6 tests from FifoTest/0 (0 ms total)
[----------] Global test environment tear-down
[==========] 6 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 6 tests.
```

```
cfrasch@Charles-PC:~/cppcon2023/build/debug$ ./unitTests --gtest_filter=FifoTest/0.*
Running main() from ./googletest/src/gtest_main.cc
Note: Google Test filter = FifoTest/0.*
[==========] Running 6 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 6 tests from FifoTest/0, where TypeParam = Fifo1<int>
[ RUN      ] FifoTest/0.properties
[       OK ] FifoTest/0.properties (0 ms)
[ RUN      ] FifoTest/0.initialConditions
[       OK ] FifoTest/0.initialConditions (0 ms)
[ RUN      ] FifoTest/0.push
[       OK ] FifoTest/0.push (0 ms)
[ RUN      ] FifoTest/0.pop
[       OK ] FifoTest/0.pop (0 ms)
[ RUN      ] FifoTest/0.popFullFifo
[       OK ] FifoTest/0.popFullFifo (0 ms)
[ RUN      ] FifoTest/0.popEmpty
[       OK ] FifoTest/0.popEmpty (0 ms)
[----------] 6 tests from FifoTest/0 (0 ms total)
[----------] Global test environment tear-down
[==========] 6 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 6 tests.
cfrasch@Charles-PC:~/cppcon2023/build/debug$ ./fifo1
Fifo1:  7,553,693 ops/s
```

```
cfrasch@Charles-PC:~/cppcon2023/build/debug$ ./unitTests --gtest_filter=FifoTest/0.*
Running main() from ./googletest/src/gtest_main.cc
Note: Google Test filter = FifoTest/0.*
[==========] Running 6 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 6 tests from FifoTest/0, where TypeParam = Fifo1<int>
[ RUN      ] FifoTest/0.properties
[       OK ] FifoTest/0.properties (0 ms)
[ RUN      ] FifoTest/0.initialConditions
[       OK ] FifoTest/0.initialConditions (0 ms)
[ RUN      ] FifoTest/0.push
[       OK ] FifoTest/0.push (0 ms)
[ RUN      ] FifoTest/0.pop
[       OK ] FifoTest/0.pop (0 ms)
[ RUN      ] FifoTest/0.popFullFifo
[       OK ] FifoTest/0.popFullFifo (0 ms)
[ RUN      ] FifoTest/0.popEmpty
[       OK ] FifoTest/0.popEmpty (0 ms)
[----------] 6 tests from FifoTest/0 (0 ms total)
[----------] Global test environment tear-down
[==========] 6 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 6 tests.
cfrasch@Charles-PC:~/cppcon2023/build/debug$ ./fifo1
Fifo1:  7,553,693 ops/s
cfrasch@Charles-PC:~/cppcon2023/build/release$ ./fifo1
```

```
cfrasch@Charles-PC:~/cppcon2023/build/debug$ ./unitTests --gtest_filter=FifoTest/0.*
Running main() from ./googletest/src/gtest_main.cc
Note: Google Test filter = FifoTest/0.*
[==========] Running 6 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 6 tests from FifoTest/0, where TypeParam = Fifo1<int>
[ RUN      ] FifoTest/0.properties
[       OK ] FifoTest/0.properties (0 ms)
[ RUN      ] FifoTest/0.initialConditions
[       OK ] FifoTest/0.initialConditions (0 ms)
[ RUN      ] FifoTest/0.push
[       OK ] FifoTest/0.push (0 ms)
[ RUN      ] FifoTest/0.pop
[       OK ] FifoTest/0.pop (0 ms)
[ RUN      ] FifoTest/0.popFullFifo
[       OK ] FifoTest/0.popFullFifo (0 ms)
[ RUN      ] FifoTest/0.popEmpty
[       OK ] FifoTest/0.popEmpty (0 ms)
[----------] 6 tests from FifoTest/0 (0 ms total)
[----------] Global test environment tear-down
[==========] 6 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 6 tests.
cfrasch@Charles-PC:~/cppcon2023/build/debug$ ./fifo1
Fifo1:  7,553,693 ops/s
cfrasch@Charles-PC:~/cppcon2023/build/release$ ./fifo1
terminate called after throwing an instance of 'std::runtime_error'
  what():  invalid value
Aborted
```

What happened?

```
cfrasch@Charles-PC:~/cppcon2023/build/debug$ ./fifo1.tsan
==================
WARNING: ThreadSanitizer: data race (pid=53590)
  Write of size 8 at 0x7ffc348b1998 by main thread:
    #0 Fifo1<std::array<long, 25ul> >::push(std::array<long, 25ul> const&) /home/cfrasch/cppcon2023/Fifo1.hpp:38
(fifo1.tsan+0x3608)
    #1 void bench<Fifo1<std::array<long, 25ul> > >(char const*, int, int) /home/cfrasch/cppcon2023/bench.hpp:54
(fifo1.tsan+0x376b)
    #2 void bench<Fifo1>(char const*, int, char**) /home/cfrasch/cppcon2023/bench.hpp:78 (fifo1.tsan+0x2d87)
    #3 main /home/cfrasch/cppcon2023/fifo1.cpp:5 (fifo1.tsan+0x2769)
…
```

A data race happened

```
cfrasch@Charles-PC:~/cppcon2023/build/debug$ ./fifo1.tsan
==================
WARNING: ThreadSanitizer: data race (pid=53590)
  Write of size 8 at 0x7ffc348b1998 by main thread:
    #0 Fifo1<std::array<long, 25ul> >::push(std::array<long, 25ul> const&) /home/cfrasch/cppcon2023/Fifo1.hpp:38
(fifo1.tsan+0x3608)
    #1 void bench<Fifo1<std::array<long, 25ul> > >(char const*, int, int) /home/cfrasch/cppcon2023/bench.hpp:54
(fifo1.tsan+0x376b)
    #2 void bench<Fifo1>(char const*, int, char**) /home/cfrasch/cppcon2023/bench.hpp:78 (fifo1.tsan+0x2d87)
    #3 main /home/cfrasch/cppcon2023/fifo1.cpp:5 (fifo1.tsan+0x2769)
…
```

A data race happened

Memory model

…

Threads and data races

…

When an evaluation of an expression writes to a memory location and another evaluation reads or modifies the same memory location, the expressions are said to *conflict*. A program that has two conflicting evaluations has a *data race* unless

- both evaluations execute on the same thread or in the same signal handler, or
- both conflicting evaluations are atomic operations (see `std::atomic`), or
- one of the conflicting evaluations *happens-before* another (see `std::memory_order`).

If a data race occurs, the behavior of the program is undefined.

Push thread

```
if (pushCursor_ - popCursor_ != size_)
new (&ring_[pushCursor_ % size_]) T(value)
++pushCursor_
```

Pop thread

```
if (popCursor_ < pushCursor_)
value = ring_[popCursor_ % size_]
++popCursor_
```

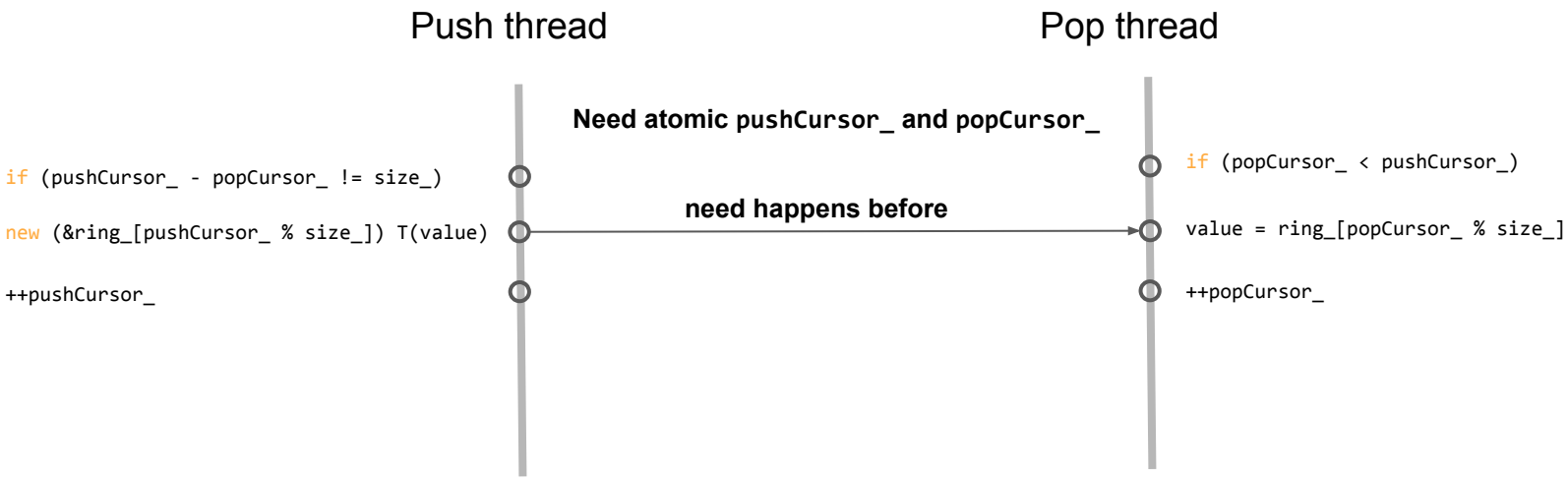
Push thread

```
if (pushCursor_ - popCursor_ != size_)
new (&ring_[pushCursor_ % size_]) T(value)
++pushCursor_
```

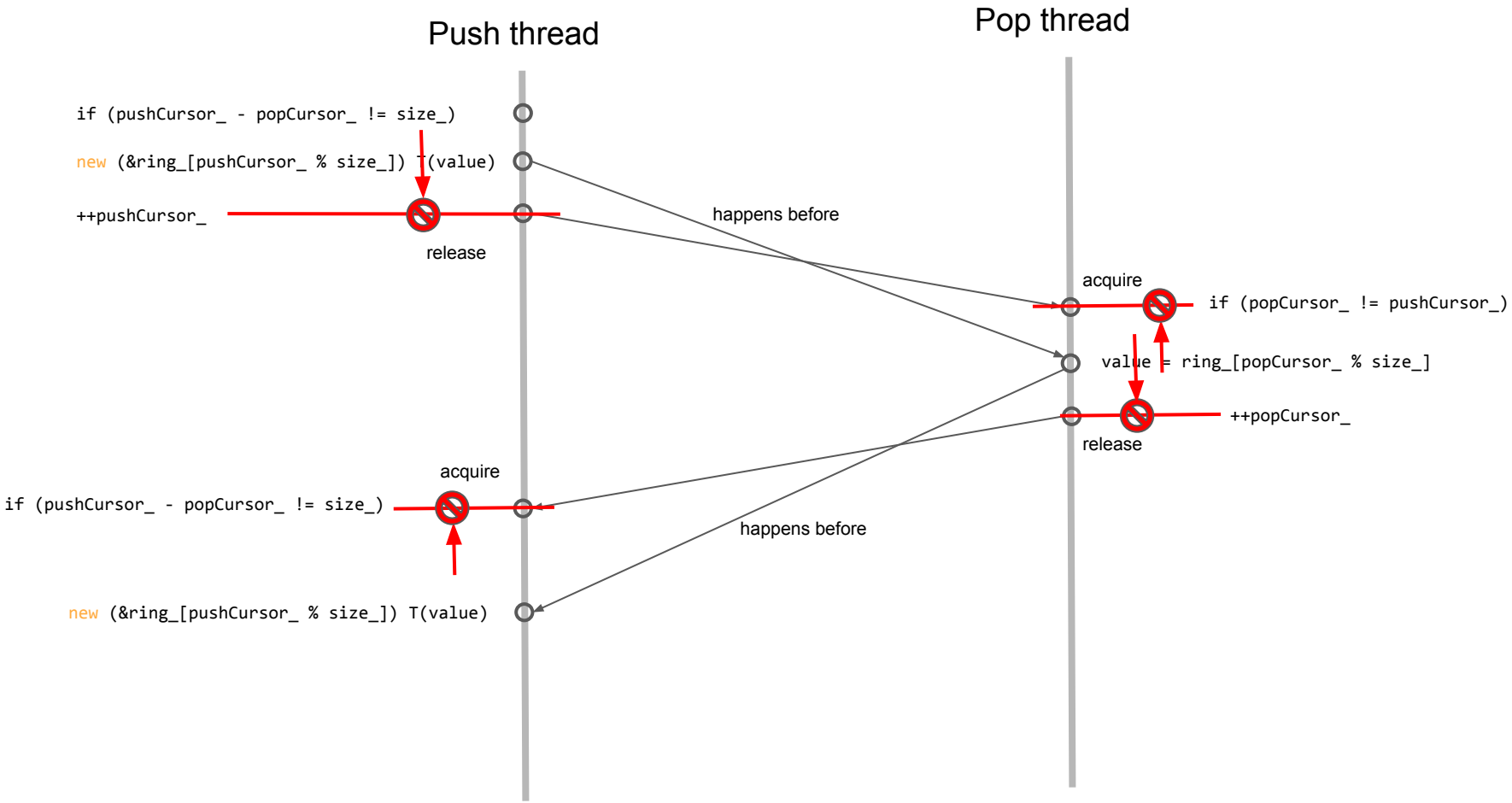
Need atomic `pushCursor_` and `popCursor_`

Pop thread

```
if (popCursor_ < pushCursor_)
value = ring_[popCursor_ % size_]
++popCursor_
```


Push thread
Pop thread
Need atomic pushCursor_ and popCursor_
if (pushCursor_ - popCursor_ != size_)
new (&ring_[pushCursor_ % size_]) T(value)
++pushCursor_
need happens before
if (popCursor_ < pushCursor_)
value = ring_[popCursor_ % size_]
++popCursor_

```
template<typename T, typename Alloc = std::allocator<T>>
class Fifo2 : private Alloc
{
  std::size_t capacity_;
  T* ring_;

  /// Loaded and stored by the push thread; loaded by the pop thread
  std::atomic<std::size_t> pushCursor_{};

  /// Loaded and stored by the pop thread; loaded by the push thread
  std::atomic<std::size_t> popCursor_{};

  static_assert<std::atomic<std::size_t>::is_always_lock_free);

public:
  explicit Fifo2(std::size_t size, Alloc const& alloc = Alloc{}) ...
  ~Fifo2() ...

  auto capacity() const ...
  auto size() const ...
  auto empty() const ...
  auto full() const ...

  auto push(T const& value) ...
  auto pop(T* value) ...
};
```


Push thread
Pop thread
if (pushCursor_ - popCursor_ != size_)
new (&ring_[pushCursor_ % size_]) T(value)
++pushCursor_
release
happens before
acquire
if (popCursor_ != pushCursor_)
value = ring_[popCursor_ % size_]
++popCursor_
release
acquire
if (pushCursor_ - popCursor_ != size_)
happens before
new (&ring_[pushCursor_ % size_]) T(value)

```
cfrasch@Charles-PC:~/cppcon2023/build/debug$ ./unitTests --gtest_filter=FifoTest/1.*
Running main() from ./googletest/src/gtest_main.cc
Note: Google Test filter = FifoTest/1.*
[==========] Running 6 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 6 tests from FifoTest/1, where TypeParam = Fifo2<int>
…
[----------] 6 tests from FifoTest/1 (0 ms total)
[----------] Global test environment tear-down
[==========] 6 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 6 tests.
cfrasch@Charles-PC:~/cppcon2023/build/debug$ ./fifo2.tsan
Fifo2:   940,241 ops/s
cfrasch@Charles-PC:~/cppcon2023/build/release$ ./fifo2
Fifo2:  12,817,132 ops/s
```

```
template<typename T, typename Alloc = std::allocator<T>>
class Fifo3 : private Alloc
{
  std::size_t capacity_;
  T* ring_;
  static_assert<std::atomic<std::size_t>::is_always_lock_free);

  alignas(std::hardware_destructive_interference_size) std::atomic<std::size_t> pushCursor_;
  alignas(std::hardware_destructive_interference_size) std::atomic<std::size_t> popCursor_;

  /// Padding to avoid false sharing with adjacent objects
  char padding_[std::hardware_destructive_interference_size - sizeof(std::size_t)];

  auto full(std::size_t pushCursor, std::size_t popCursor) const { return (pushCursor - popCursor) == capacity_; }
  static auto empty(std::size_t pushCursor, std::size_t popCursor) { return pushCursor == popCursor; }
  auto element(std::size_t cursor) { return &ring_[cursor % capacity_]; }

public:
  explicit Fifo3(std::size_t size, Alloc const& alloc = Alloc{}) ...
  ~Fifo3() ...

  auto capacity() const ...
  auto size() const ...
  auto empty() const ...
  auto full() const ...

  auto push(T const& value);
  auto pop(T* value);
};
```

```
auto Fifo3::push(T const& value) {
  auto pushCursor = pushCursor_.load(std::memory_order_relaxed);
  auto popCursor = popCursor_.load(std::memory_order_acquire);
  if (full(pushCursor, popCursor)) {
    return false;
  }
  new (element(pushCursor)) T(value);
  pushCursor_.store(pushCursor + 1, std::memory_order_release);
  return true;
}


auto Fifo3::pop(T& value) {
  auto pushCursor = pushCursor_.load(std::memory_order_acquire);
  auto popCursor = popCursor_.load(std::memory_order_relaxed);
  if (empty(pushCursor, popCursor)) {
    return false;
  }
  value = *element(popCursor);
  element(popCursor)->~T();
  popCursor_.store(popCursor + 1, std::memory_order_release);
  return true;
}
```

```
cfrasch@Charles-PC:~/cppcon2023/build/debug$ ./unitTests --gtest_filter=FifoTest/2.*
Running main() from ./googletest/src/gtest_main.cc
Note: Google Test filter = FifoTest/2.*
[==========] Running 6 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 6 tests from FifoTest/2, where TypeParam = Fifo3<int>
…
[----------] 6 tests from FifoTest/2 (0 ms total)
[----------] Global test environment tear-down
[==========] 6 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 6 tests.
cfrasch@Charles-PC:~/cppcon2023/build/debug$ ./fifo3.tsan
Fifo3:  1,159,822 ops/s
cfrasch@Charles-PC:~/cppcon2023/build/release$ ./fifo3
Fifo3: 49,430,743 ops/s
```

```
Fifo2:  12,817,132 ops/s
```

```
template<typename T, typename Alloc = std::allocator<T>>
class Fifo4 : private Alloc
{
  std::size_t capacity_;
  T* ring_;
  static_assert<std::atomic<std::size_t>::is_always_lock_free);
  alignas(std::hardware_destructive_interference_size) atomic<std::size_t> pushCursor_;

  /// Exclusive to the push thread
  alignas(std::hardware_destructive_interference_size) std::size_t cachedPopCursor_{};

  alignas(std::hardware_destructive_interference_size) atomic<std::size_t> popCursor_;

  /// Exclusive to the pop thread
  alignas(std::hardware_destructive_interference_size) std::size_t cachedPushCursor_{};

  auto full(std::size_t pushCursor, std::size_t pop Cursor) const …
  static auto empty(std::size_t pushCursor, std::size_t popCursor) …
  auto element(std::size_t cursor) …

public:
  explicit Fifo4(std::size_t size, Alloc const& alloc = Alloc{}) …
  ~Fifo4() …

  auto capacity() const …
  auto size() const …
  auto empty() const …
  auto full() const …

  auto push(T const& value);
  auto pop(TY value);
};
```

```
auto Fifo4::push(T const& value) {
  auto pushCursor = pushCursor_.load(std::memory_order_relaxed);
  if (full(pushCursor, cachedPopCursor_)) {
    cachedPopCursor_ = popCursor_.load(std::memory_order_acquire);
    if (full(pushCursor, cachedPopCursor_)) {
      return false;
    }
  }
  new (&ring_[pushCursor % capacity_]) T(value);
  pushCursor_.store(pushCursor + 1, std::memory_order_release);
  return true;
}
```

```
auto Fifo4::pop(T& value) {
  auto popCursor = popCursor_.load(std::memory_order_relaxed);
  if (empty(cachedPushCursor_, popCursor)) {
    cachedPushCursor_ = pushCursor_.load(std::memory_order_acquire);
    if (empty(cachedPushCursor_, popCursor)) {
      return false;
    }
  }
  value = ring_[popCursor % capacity_];
  ring_[popCursor % capacity_].~T();
  popCursor_.store(popCursor + 1, std::memory_order_release);
  return true;
}
```

```
cfrasch@Charles-PC:~/cppcon2023/build/debug$ ./unitTests --gtest_filter=FifoTest/3.*
Running main() from ./googletest/src/gtest_main.cc
Note: Google Test filter = FifoTest/3.*
[==========] Running 6 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 6 tests from FifoTest/3, where TypeParam = Fifo4<int>
…
[----------] 6 tests from FifoTest/3 (0 ms total)
[----------] Global test environment tear-down
[==========] 6 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 6 tests.
cfrasch@Charles-PC:~/cppcon2023/build/debug$ ./fifo4.tsan
Fifo4:  1,159,822 ops/s
cfrasch@Charles-PC:~/cppcon2023/build/release$ ./fifo4
Fifo4: 165,926,288 ops/s
```

```
Fifo2: 12,817,132 ops/s
Fifo3: 49,430,743 ops/s
```

```
struct Foo { int bar; int blat; ...};
static_assert(std::is_implicit_lifetime_v<Foo>);  // See P2674R0
Fifo4<Foo> fifo(1024);

// called in thread 1
void readAndPushFoo(int sock, Fifo4<Foo>& fifo) {
  alignas(Foo) unsigned char buffer[sizeof(Foo)];
  ::read(sock, buffer, sizeof(Foo));                    ← memcpy
  auto foo = std::start_lifetime_as<Foo>(buffer);
  if (foo->bar != someValue) throw std::runtime_error("bad bar");
  fifo.push(*foo);                                      ← memcpy
}


// called in thread 2
void popAndSendFoo(int sock, Fifo4<Foo>& fifo) {
   Foo foo;
   if (!fifo.pop(foo)) return;                          ← memcpy
   foo.blat = someBlatValue;
   ::write(sock, std::addressof(foo), sizeof(Foo));     ← memcpy
}
```

```
struct Foo {int bar; int blat; ...};
static_assert(std::is_trivial_v<Foo>);
Fifo5<Foo> fifo(42);

// called in thread 1
void readAndPushFoo(int rsock, Fifo5<Foo>& fifo) {
  auto pusher = fifo.push();
  ::read(rsock, pusher.get(), sizeof(Foo));          ← memcpy
  if (pusher->bar != someValue) {
    pusher.release();
    throw std::runtime_error("bad bar");
  }
}


// called in thread 2
void popAndSendFoo(int wsock, Fifo5<Foo>& fifo) {
   auto popper = fifo.pop();
   if (!popper) return;
   popper->blat = someBlatValue;
   ::write(wsock, popper.get(), sizeof(Foo));        ← memcpy
}
```

```
template<typename T, typename Alloc = std::allocator<T>>
class Fifo5 : private Alloc requires std::is_trivial_v<T>
{
  size_t mask_;
  T* ring_;
  static_assert<std::atomic<std::size_t>::is_always_lock_free);
  alignas(hardware_destructive_interference_size) atomic<std::size_t> pushCursor_;
  alignas(hardware_destructive_interference_size) std::size_t cachedPopCursor_{};
  alignas(hardware_destructive_interference_size) atomic<std::size_t> popCursor_;
  alignas(hardware_destructive_interference_size) std::atomic<std::size_t> popCursor_;
  auto element(std::size_t cursor) ...
  auto full(std::size_t pushCursor, std::size_t popCursor) const ...
  static auto empty(std::size_t pushCursor, std::size_t popCursor) ...

public:
  explicit Fifo5(std::size_t size, Alloc const& alloc = Alloc{}) ...
  ~Fifo5() ...

  auto capacity() const ...
  auto size() const ...
  auto empty() const ...
  auto full() const ...

  class pusher_t;
  pusher_t push();
  bool push(T const& value);

  class popper_t;
  popper_t pop();
  bool pop(T* value);
};
```

```
class Fifo5::pusher_t {
  Fifo5* fifo_{};
  std::size_t cursor_;

public:
  pusher_t() = default;
  explicit pusher_t(Fifo5* fifo, std::size_t cursor) : fifo_{fifo}, cursor_{cursor} {}

  // Not copyable; is movable

  ~pusher_t() {
    if (fifo_) {
      fifo_->pushCursor_.store(cursor_ + 1, std::memory_order_release);
    }
  }
  explicit operator bool() const { return fifo_; }
  void release() { fifo_ = {}; }

  T* get() { return fifo_->element(cursor_); }
  T const* get() const { return fifo_->element(cursor_); }
  T* operator->() { return get(); }
  T const* operator->() const { return get(); }
};
```

```
Fifo5::pusher_t Fifo5::push() {
  auto pushCursor = pushCursor_.load(std::memory_order_relaxed);
  if (full(pushCursor, cachedPopCursor_)) {
    cachedPopCursor_ = popCursor_.load(std::memory_order_acquire);
    if (full(pushCursor, cachedPopCursor_)) {
      return pusher_t{};
    }
  }
  return pusher_t(this, pushCursor);
}

bool Fifo5::push(T const& value) {
  auto pusher = push();
  if (pusher) {
    *pusher = value;
    return true;
  }
  return false;
}
```

```
class Fifo5::popper_t {
  Fifo5* fifo_{};
  std::size_t cursor_;

public:
  popper_t() = default;
  explicit popper_t(Fifo5* fifo, std::size_t cursor) : fifo_{fifo}, cursor_{cursor} {}

  // Not copyable; is movable

  ~popper_t() {
    if (fifo_) {
      fifo_->popCursor_.store(cursor_ + 1, std::memory_order_release);
    }
  }
  explicit operator bool() const { return fifo_; }
  void release() { fifo_ = {}; }

  T* get() { return fifo_->element(cursor_); }
  T const* get() const { return fifo_->element(cursor_); }
  T* operator->() { return get(); }
  T const* operator->() const { return get(); }
};
```

```
Fifo5::popper_t Fifo5::pop() {
  auto popCursor = popCursor_.load(std::memory_order_relaxed);
  if (empty(cachedPushCursor_, popCursor)) {
    cachedPushCursor_ = pushCursor_.load(std::memory_order_acquire);
    if (empty(cachedPushCursor_, popCursor)) {
      return popper_t{};
    }
  }
  return popper_t{this, popCursor};
}


bool Fifo5::pop(T& value) {
  auto popper = pop();
  if (popper) {
    value = *popper;
    return true;
  }
  return false;
}
```

```
cfrasch@Charles-PC:~/cppcon2023/build/debug$ ./unitTests --gtest_filter=FifoTest/4.*
Running main() from ./googletest/src/gtest_main.cc
Note: Google Test filter = FifoTest/4.*
[==========] Running 6 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 6 tests from FifoTest/4, where TypeParam = Fifo5<int>
…
[----------] 6 tests from FifoTest/4 (0 ms total)
[----------] Global test environment tear-down
[==========] 6 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 6 tests.
cfrasch@Charles-PC:~/cppcon2023/build/debug$ ./fifo5.tsan
Fifo5:  2,549,464 ops/s
cfrasch@Charles-PC:~/cppcon2023/build/release$ ./fifo5
Fifo5: 165,383,212 ops/s
```

```
Fifo2:  12,817,132 ops/s
Fifo3:  49,430,743 ops/s
Fifo4: 165,926,288 ops/s
```

Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz 8 cores, 12MB L3 cache
Ubuntu-22.04 on wsl version 2
Fifo size 131,072 (2^17); int64_t; 100 runs; 400’000’000 iterations

Ops/second

	minimum	maximum	mean	median	
Fifo2[1]	12,233,984	12,817,132	12,569,860	12,621,200	Naive
Fifo3	39,768,640	49,430,743	42,133,450	41,859,740	Relaxed atomics, no false sharing
Fifo4	156,198,885	165,926,288	164,687,821	165,138,887	Cached cursors
Fifo5	158,045,126	165,383,212	162,540,387	162,768,763	Proxies
boost	93,128,252	107,257,150	101,470,133	101,850,213	Fixed size
rigtorp	222,322,284	393,366,196	321,693,552	330,680,215	Cached cursors
mutex[1]	5,498,095	5,847,872	5,756,232	5,816,369	Not lock free

[1]10 runs; 10’000’000 iterations

Ops/second

	minimum	maximum	mean	median	isns/cycle
Fifo4	156,198,885	165,926,288	164,687,821	165,138,887	remainder 0.63
Fifo4a	**643,016,342**	**726,088,682**	**691,178,417**	**690,662,074**	**AND with mask 2.61**
Fifo5	158,045,126	165,383,212	162,540,387	162,768,763	remainder 0.67
boost	93,128,252	107,257,150	101,470,133	101,850,213	add + comparison 0.51
rigtorp	222,322,284	393,366,196	321,693,552	330,680,215	add + comparison 1.89

Resources

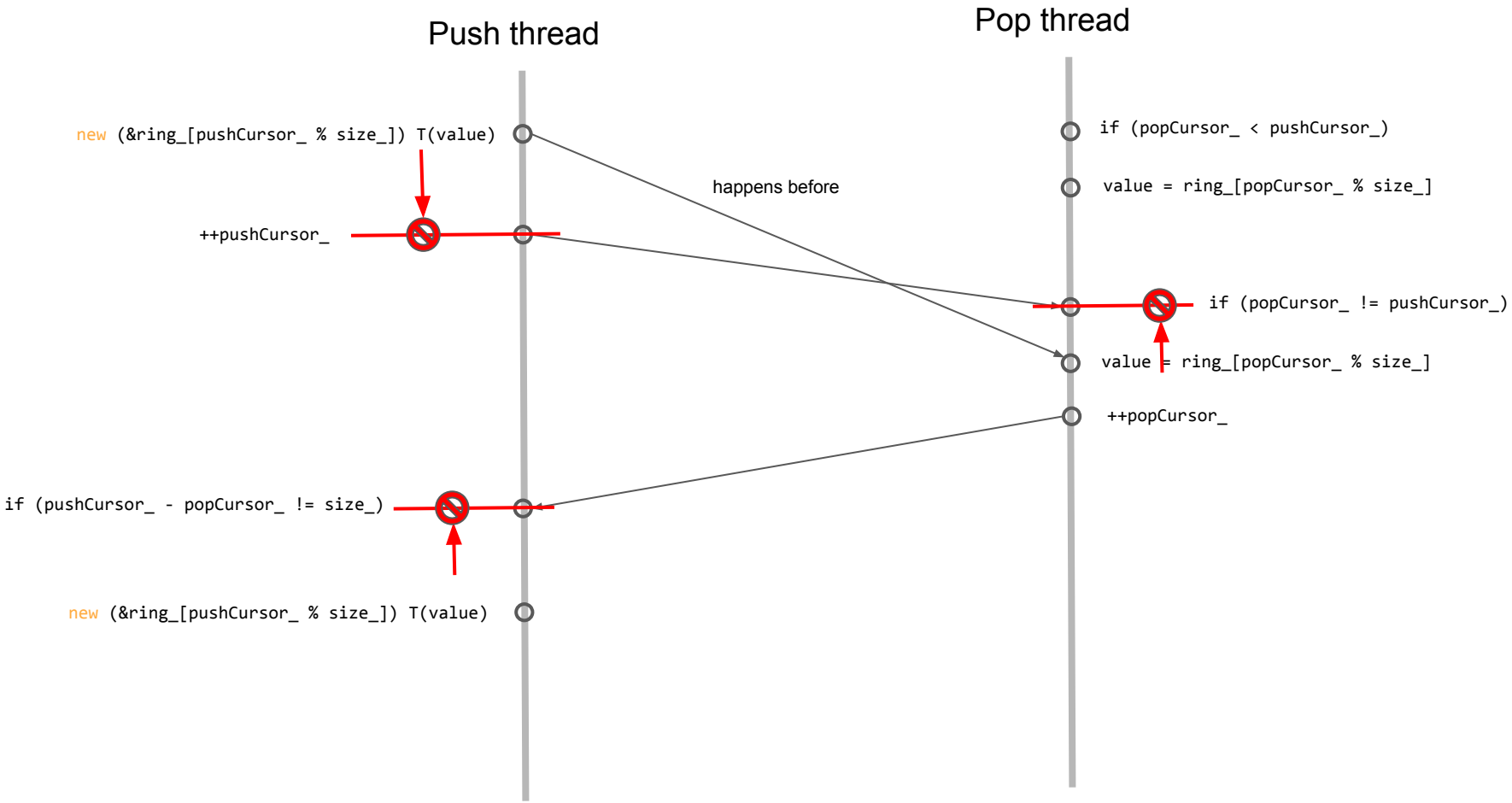
C++ atomics, from basic to advanced. What do they really do? - Fedor Pikus

Taking a Byte Out of C++ - Avoiding Punning by Starting Lifetimes - Robert Leahy

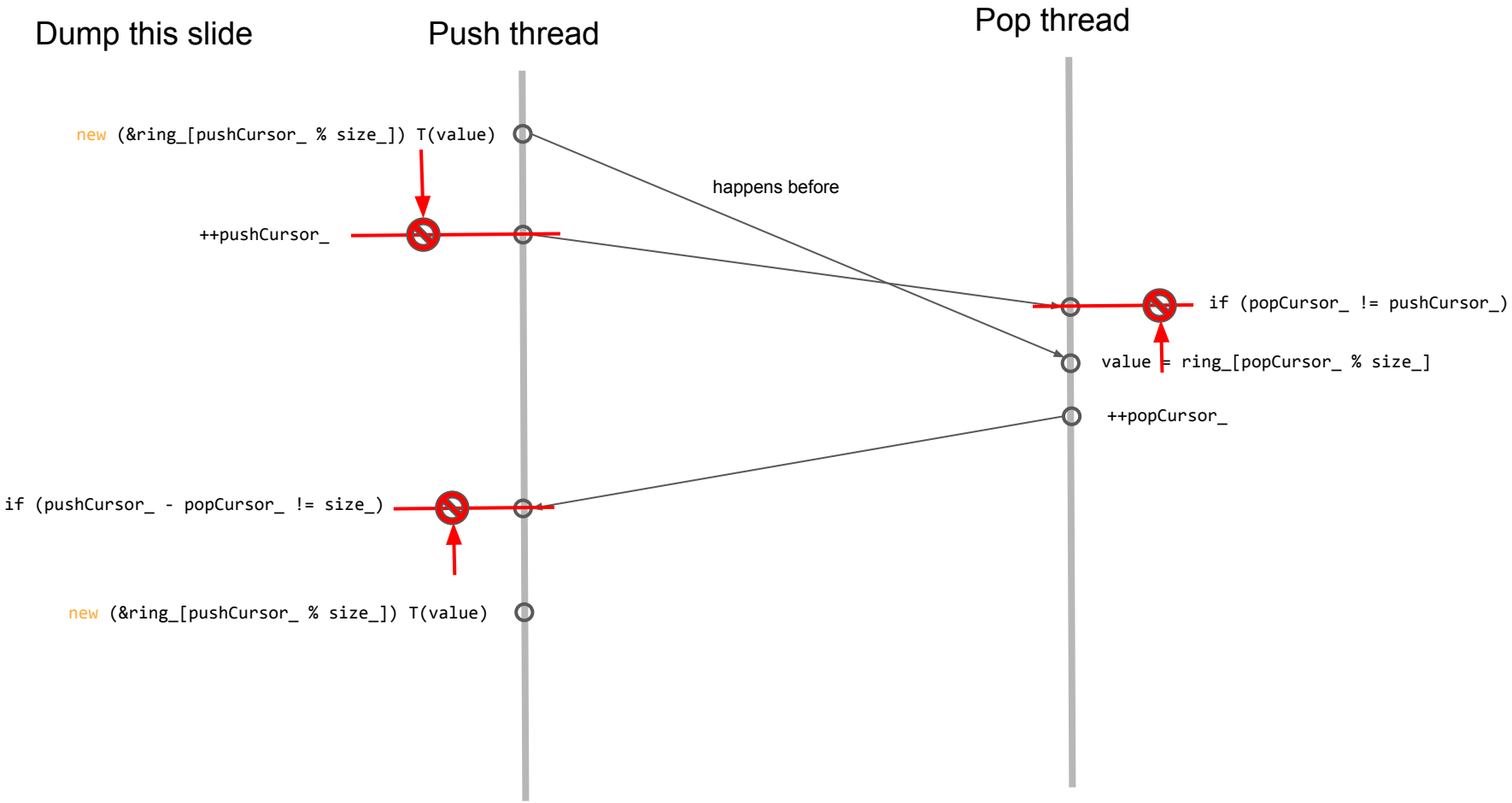
Awesome Lock-Free - Erik Rigtorp

What is Low Latency C++? (Part 1) - Timur Doumler - CppNow 2023

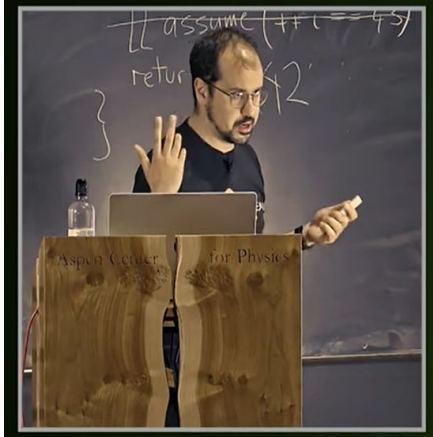
What is Low Latency C++? (Part 2) - Timur Doumler - CppNow 2023


Push thread
Pop thread
new (&ring_[pushCursor_ % size_]) T(value)
++pushCursor_
if (pushCursor_ - popCursor_ != size_)
new (&ring_[pushCursor_ % size_]) T(value)
happens before
if (popCursor_ < pushCursor_)
value = ring_[popCursor_ % size_]
if (popCursor_ != pushCursor_)
value = ring_[popCursor_ % size_]
++popCursor_

Dump this slide



C++ now | 2023 MAY 8-12 Aspen, Colorado, USA



Timur Doumler

What is Low Latency C++?
Part 2 of 2

The cost of various lock-free queues

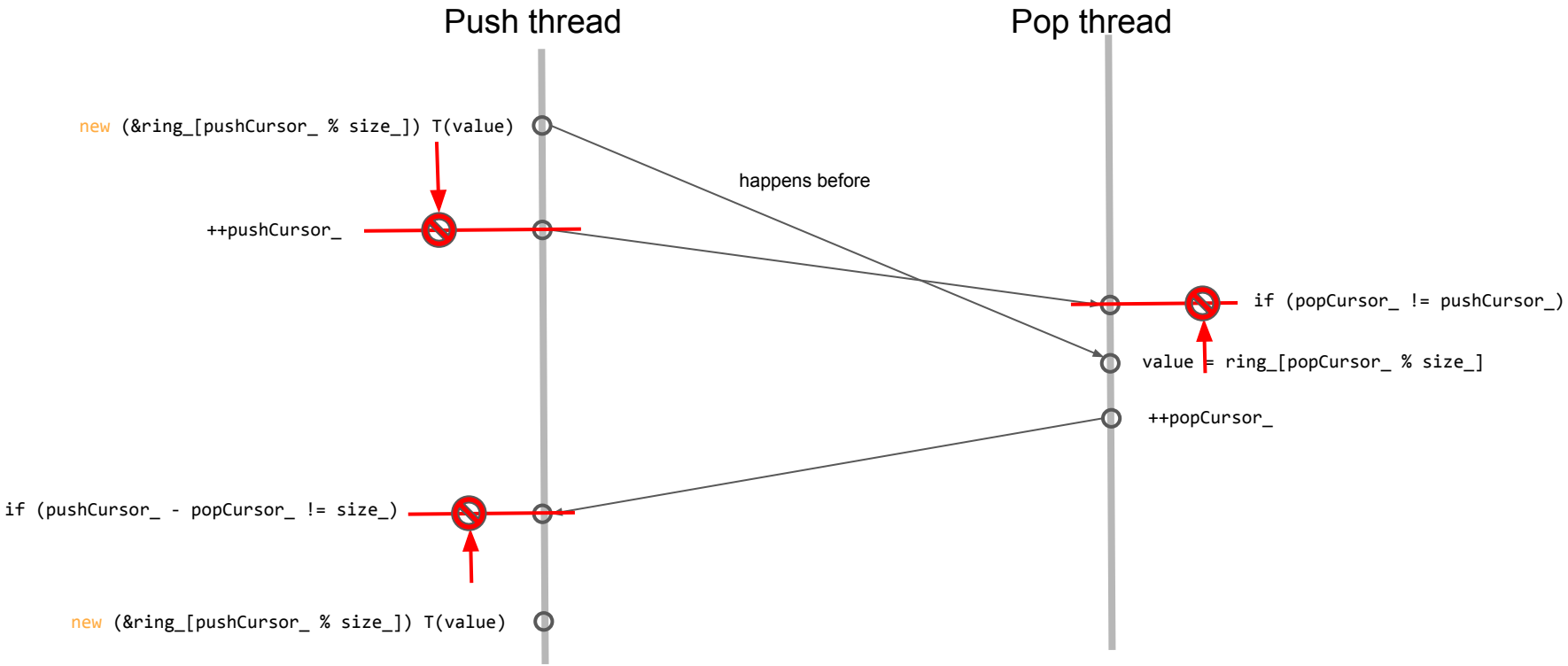
(from "Real-time 101" by Fabian Renn-Giles & Dave Rowland)

Consumer \ Producer		Single Producer		Multiple Producer	
		Report Full	Overwrite on Full	Report Full	Overwrite on Full
Single Consumer	Report Empty	Wait free on read and write	Wait free on read and write	Wait free on read	Wait free on read and write
Single Consumer	"null" on Empty	Wait free on read and write	Wait free on read and write	Wait free on read	Wait free on read and write
Multiple Consumer	Report Empty	Wait free on write	Wait free on write	Not wait free on write or read	Wait free on write
Multiple Consumer	"null" on Empty	Wait free on read and write	Wait free on read and write	Wait free on read	Wait free on read and write

Wait free on read | Wait free on write | Wait free on read and write | Not wait free on write or read

Copyright (c) Timur Doumler | @timur_audio | https://timur.audio 59

Maybe maybe not this slide
We can do better with relaxed atomics

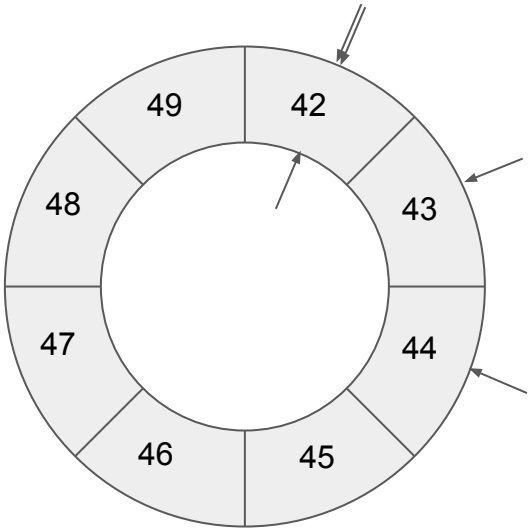


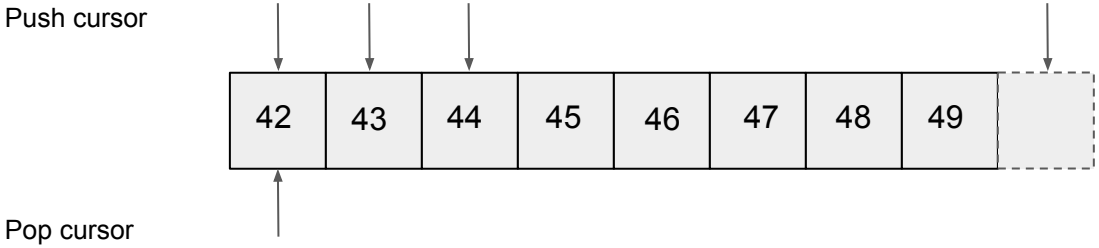
Ops/second

	minimum	maximum	mean	median	isns/cycle
Fifo4	156,198,885	165,926,288	164,687,821	165,138,887	remainder 0.63
Fifo4a	**643,016,342**	**726,088,682**	**691,178,417**	**690,662,074**	**AND with mask 2.61**
Fifo4b	**653,505,550**	**690,579,021**	**683,464,371**	**685,053,739**	**Constrained 2.24**
Fifo5	158,045,126	165,383,212	162,540,387	162,768,763	remainder 0.67
boost	93,128,252	107,257,150	101,470,133	101,850,213	constrained 0.51
rigtorp	222,322,284	393,366,196	321,693,552	330,680,215	constrained 1.89

[1] 10 runs; 10’000’000 iterations

Circular Fifo - Push





Circular Fifo - Pop