The C++20 Synchronization Library

Bryce Adelstein Lelbach

```
unique_future<std::uint64_t>
fibonacci(boost_blockable_submitter auto&& s, std::uint64_t n) {
  if (n < 2) co_return n;

  auto n1 = async(s, fibonacci<decltype(s)>, s, n - 1);
  auto n2 = fibonacci(s, n - 2);

  co_return co_await n1 + co_await n2;
}
```

THE C++20 SYNCHRONIZATION LIBRARY

Bryce Adelstein Lelbach

@blelbach

CUDA C++ Core Libraries Lead

ISO C++ Library Evolution Incubator Chair, ISO C++ Tooling Study Group Chair

```
namespace stdr = std::ranges;
namespace stdv = std::views;
```

Recipe For a Tasking Runtime

- Worker threads.
- Multi-consumer, multi-producer concurrent queue.
- Termination detection mechanism.
- Parallel algorithms.

Recipe For a Tasking Runtime

- Worker threads.
- Multi-consumer, multi-producer concurrent queue.
- Termination detection mechanism.
- Parallel algorithms.

```
struct thread_group {
private:
  std::vector<std::thread> members;

public:
  thread_group(std::uint64_t n, std::invocable auto&& f) {
    for (auto i : stdv::iota(0, n)) members.emplace_back(f);
  }
};
```

```
struct thread_group {
private:
  std::vector<std::thread> members;

public:
  thread_group(std::uint64_t n, std::invocable auto&& f) {
    for (auto i : stdv::iota(0, n)) members.emplace_back(f);
  }
};

int main() {
  std::atomic<std::uint64_t> count(0);

  {
    thread_group tg(6, [&] { ++count; });
  }

  std::cout << count << "\n";
}
```

```
struct thread_group {
private:
  std::vector<std::thread> members;

public:
  thread_group(std::uint64_t n, std::invocable auto&& f) {
    for (auto i : stdv::iota(0, n)) members.emplace_back(f);
  }

  ~thread_group() {
    stdr::for_each(members, [] (std::thread& t) { t.join(); });
  }
};
```

```cpp
int main() {
  std::atomic<std::uint64_t> count(0);

  {


    thread_group tg(6,
      [&] {
        while (true)
          ++count;
      }
    );


  }

  std::cout << count << "\n";
}
```

```
int main() {
  std::atomic<std::uint64_t> count(0);

  {
    std::atomic<bool> done(false);

    thread_group tg(6,
      [&] {
        while (!done.load(std::memory_order_relaxed))
          ++count;
      }
    );

    done.store(true, std::memory_order_relaxed);
  }

  std::cout << count << "\n";
}
```

```
struct thread_group {
private:
  std::vector<std::jthread> members;

public:
  thread_group(std::uint64_t n, std::invocable<std::stop_token> auto&& f) {
    for (auto i : stdv::iota(0, n)) members.emplace_back(f);
  }
};
```

`std::jthread`

- Just like `std::thread`, except:
- If the thread is joinable, it joins with it instead of calling `terminate`.

```
struct thread_group {
private:
  std::vector<std::jthread> members;

public:
  thread_group(std::uint64_t n, std::invocable<std::stop_token> auto&& f) {
    for (auto i : stdv::iota(0, n)) members.emplace_back(f);
  }

  auto size() { return members.size(); }

  void request_stop() {
    stdr::for_each(members, [] { t.request_stop(); });
  }
};
```

`std::jthread`

- Just like `std::thread`, except:
- If the thread is joinable, it joins with it instead of calling `terminate`.
- It supports interruption.
  - `std::thread` invocable parameter: takes no arguments.
  - `std::jthread` invocable parameter: takes either no arguments, or a `std::stop_token`.
- Interruption API:

```
[[nodiscard]] stop_source std::jthread::get_stop_source() noexcept;
[[nodiscard]] stop_token std::jthread::get_stop_token() const noexcept;
bool std::jthread::request_stop() noexcept;
```

std::stop_*

- std::stop_source (analogous to a promise)
  - Producer of stop requests.
  - Owns the shared state (if any).
- std::stop_token (analogous to future)
  - Handle to a std::stop_source.
  - Consumer only; can query for stop requests, but can’t make them.
- std::stop_callback (analogous to future::then)
  - Mechanism for registering invocables to be run upon receiving a stop request.

CV Interruption Support

```
struct condition_variable_any {
  template <typename Lock, typename Predicate>
    bool wait_until(Lock& lock, Predicate pred, stop_token stoken);
  template <typename Lock, class Clock, typename Duration, typename Predicate>
    bool wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time
                    Predicate pred, stop_token stoken);
  template <typename Lock, typename Rep, typename Period, typename Predicate>
    bool wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time,
                  Predicate pred, stop_token stoken);
};
```

```
int main() {
  std::atomic<std::uint64_t> count(0);

  {
    thread_group tg(6,
      [&] (std::stop_token s) {
        while (!s.stop_requested())
          ++count;
      }
    );
  }

  std::cout << count << "\n";
}
```

Recipe For a Tasking Runtime

- Worker threads.
- Multi-consumer, multi-producer concurrent queue.
- Termination detection mechanism.
- Parallel algorithms.

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  T pop();

public:
  constexpr concurrent_bounded_queue() = default;

  void enqueue(std::convertible_to<T> auto&& u);

  T dequeue();
  std::optional<T> try_dequeue();
};
```

```cpp
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  T pop();

public:
  constexpr concurrent_bounded_queue() = default;

  void enqueue(std::convertible_to<T> auto&& u);

  T dequeue();
  std::optional<T> try_dequeue();
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  T pop();

public:
  constexpr concurrent_bounded_queue() = default;

  void enqueue(std::convertible_to<T> auto&& u);

  T dequeue();
  std::optional<T> try_dequeue();
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  T pop();

public:
  constexpr concurrent_bounded_queue() = default;

  void enqueue(std::convertible_to<T> auto&& u);

  T dequeue();
  std::optional<T> try_dequeue();
};
```

std::counting_semaphore

```
template <ptrdiff_t least_max_value = implementation-defined>
struct counting_semaphore {
  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit counting_semaphore(ptrdiff_t desired);

  void release(ptrdiff_t update = 1);

  void acquire();
  bool try_acquire() noexcept;
  template <typename Rep, typename Period>
    bool try_acquire_for(const chrono::duration<Rep, Period>& rel_time);
  template <typename Clock, typename Duration>
    bool try_acquire_until(const chrono::time_point<Clock, Duration>& abs_time);
};
```

std::counting_semaphore

```
template <ptrdiff_t least_max_value = implementation-defined>
struct counting_semaphore {
  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit counting_semaphore(ptrdiff_t desired);

  void release(ptrdiff_t update = 1);

  void acquire();
  bool try_acquire() noexcept;
  template <typename Rep, typename Period>
    bool try_acquire_for(const chrono::duration<Rep, Period>& rel_time);
  template <typename Clock, typename Duration>
    bool try_acquire_until(const chrono::time_point<Clock, Duration>& abs_time);
};
```

std::counting_semaphore

```
template <ptrdiff_t least_max_value = implementation-defined>
struct counting_semaphore {
  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit counting_semaphore(ptrdiff_t desired);

  void release(ptrdiff_t update = 1);

  void acquire();
  bool try_acquire() noexcept;
  template <typename Rep, typename Period>
    bool try_acquire_for(const chrono::duration<Rep, Period>& rel_time);
  template <typename Clock, typename Duration>
    bool try_acquire_until(const chrono::time_point<Clock, Duration>& abs_time);
};
```

std::counting_semaphore

```
template <ptrdiff_t least_max_value = implementation-defined>
struct counting_semaphore {
  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit counting_semaphore(ptrdiff_t desired);

  void release(ptrdiff_t update = 1);

  void acquire();
  bool try_acquire() noexcept;
  template <typename Rep, typename Period>
    bool try_acquire_for(const chrono::duration<Rep, Period>& rel_time);
  template <typename Clock, typename Duration>
    bool try_acquire_until(const chrono::time_point<Clock, Duration>& abs_time);
};
```

std::counting_semaphore

```
template <ptrdiff_t least_max_value = implementation-defined>
struct counting_semaphore {
  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit counting_semaphore(ptrdiff_t desired);

  void release(ptrdiff_t update = 1);

  void acquire();
  bool try_acquire() noexcept;
  template <typename Rep, typename Period>
    bool try_acquire_for(const chrono::duration<Rep, Period>& rel_time);
  template <typename Clock, typename Duration>
    bool try_acquire_until(const chrono::time_point<Clock, Duration>& abs_time);
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  T pop();

public:
  constexpr concurrent_bounded_queue() = default;

  void enqueue(std::convertible_to<T> auto&& u);

  T dequeue();
  std::optional<T> try_dequeue();
};
```

```cpp
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  T pop() {
    std::optional<T> tmp;
    std::scoped_lock l(items_mtx);
    tmp = std::move(items.front());
    items.pop();
    return std::move(*tmp);
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  T pop() {
    std::optional<T> tmp;
    std::scoped_lock l(items_mtx);
    tmp = std::move(items.front());
    items.pop();
    return std::move(*tmp);
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  T pop() {
    std::optional<T> tmp;
    std::scoped_lock l(items_mtx);
    tmp = std::move(items.front());
    items.pop();
    return std::move(*tmp);
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

public:
  void enqueue(std::convertible_to<T> auto&& u) {
    remaining_space.acquire();
    {
      std::scoped_lock l(items_mtx);
      items.emplace(std::forward<decltype(u)>(u));
    }
    items_produced.release();
  }
};
```

```cpp
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

public:
  void enqueue(std::convertible_to<T> auto&& u) {
    remaining_space.acquire();
    {
      std::scoped_lock l(items_mtx);
      items.emplace(std::forward<decltype(u)>(u));
    }
    items_produced.release();
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

public:
  void enqueue(std::convertible_to<T> auto&& u) {
    remaining_space.acquire();
    {
      std::scoped_lock l(items_mtx);
      items.emplace(std::forward<decltype(u)>(u));
    }
    items_produced.release();
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

public:
  void enqueue(std::convertible_to<T> auto&& u) {
    remaining_space.acquire();
    {
      std::scoped_lock l(items_mtx);
      items.emplace(std::forward<decltype(u)>(u));
    }
    items_produced.release();
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

public:
  T dequeue() {
    items_produced.acquire();
    T tmp = pop();
    remaining_space.release();
    return std::move(tmp);
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

public:
  T dequeue() {
    items_produced.acquire();
    T tmp = pop();
    remaining_space.release();
    return std::move(tmp);
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

public:
  T dequeue() {
    items_produced.acquire();
    T tmp = pop();
    remaining_space.release();
    return std::move(tmp);
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

public:
  T dequeue() {
    items_produced.acquire();
    T tmp = pop();
    remaining_space.release();
    return std::move(tmp);
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

public:
  std::optional<T> try_dequeue() {
    if (!items_produced.try_acquire()) return {};
    T tmp = pop();
    remaining_space.release();
    return std::move(tmp);
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

public:
  std::optional<T> try_dequeue() {
    if (!items_produced.try_acquire()) return {};
    T tmp = pop();
    remaining_space.release();
    return std::move(tmp);
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  T pop();

public:
  constexpr concurrent_bounded_queue() = default;

  void enqueue(std::convertible_to<T> auto&& u);

  T dequeue();
  std::optional<T> try_dequeue();
};
```

```
struct spin_mutex {
private:
  std::atomic<bool> flag = ATOMIC_VAR_INIT(false);

public:
  void lock() {
    while (flag.exchange(true, std::memory_order_acquire))
      ;
  }

  void unlock() {
    flag.store(false, std::memory_order_release);
  }
};
```

```
struct spin_mutex {
private:
  std::atomic_flag flag = ATOMIC_FLAG_INIT;

public:
  void lock() {
    while (flag.test_and_set(std::memory_order_acquire))
      ;
  }

  void unlock() {
    flag.clear(std::memory_order_release);
  }
};
```

```
struct spin_mutex {
private:
  std::atomic_flag flag = ATOMIC_FLAG_INIT;

public:
  void lock() {
    for (std::uint64_t k = 0; !flag.test_and_set(std::memory_order_acquire); ++k) {
      if (k < 4) ;
      else if (k < 16) __asm__ __volatile__( "rep; nop" : : : "memory" );
      else if (k < 64) sched_yield();
      else {
        timespec rqtp = { 0, 0 };
        rqtp.tv_sec = 0; rqtp.tv_nsec = 1000;
        nanosleep(&rqtp, nullptr);
      }
    }
  }

  void unlock() {
    flag.clear(std::memory_order_release);
  }
};
```

```
struct spin_mutex {
private:
  std::atomic_flag flag = ATOMIC_FLAG_INIT;

public:
  void lock() {
    for (std::uint64_t k = 0; !flag.test_and_set(std::memory_order_acquire); ++k) {
      if (k < 4) ;
      else if (k < 16) __asm__ __volatile__( "rep; nop" : : : "memory" );
      else if (k < 64) sched_yield();
      else {
        timespec rqtp = { 0, 0 };
        rqtp.tv_sec = 0; rqtp.tv_nsec = 1000;
        nanosleep(&rqtp, nullptr);
      }
    }
  }

  void unlock() {
    flag.clear(std::memory_order_release);
  }
};
```

```
struct spin_mutex {
private:
  std::atomic_flag flag = ATOMIC_FLAG_INIT;

public:
  void lock() {
    for (std::uint64_t k = 0; !flag.test_and_set(std::memory_order_acquire); ++k) {
      if (k < 4) ;
      else if (k < 16) __asm__ __volatile__( "rep; nop" : : : "memory" );
      else if (k < 64) sched_yield();
      else {
        timespec rqtp = { 0, 0 };
        rqtp.tv_sec = 0; rqtp.tv_nsec = 1000;
        nanosleep(&rqtp, nullptr);
      }
    }
  }

  void unlock() {
    flag.clear(std::memory_order_release);
  }
};
```

```
struct spin_mutex {
private:
  std::atomic_flag flag = ATOMIC_FLAG_INIT;

public:
  void lock() {
    for (std::uint64_t k = 0; !flag.test_and_set(std::memory_order_acquire); ++k) {
      if (k < 4) ;
      else if (k < 16) __asm__ __volatile__( "rep; nop" : : : "memory" );
      else if (k < 64) sched_yield();
      else {
        timespec rqtp = { 0, 0 };
        rqtp.tv_sec = 0; rqtp.tv_nsec = 1000;
        nanosleep(&rqtp, nullptr);
      }
    }
  }

  void unlock() {
    flag.clear(std::memory_order_release);
  }
};
```

```
struct spin_mutex {
private:
  std::atomic_flag flag = ATOMIC_FLAG_INIT;

public:
  void lock() {
    for (std::uint64_t k = 0; !flag.test_and_set(std::memory_order_acquire); ++k) {
      if (k < 4) ;
      else if (k < 16) __asm__ __volatile__( "rep; nop" : : : "memory" );
      else if (k < 64) sched_yield();
      else {
        timespec rqtp = { 0, 0 };
        rqtp.tv_sec = 0; rqtp.tv_nsec = 1000;
        nanosleep(&rqtp, nullptr);
      }
    }
  }

  void unlock() {
    flag.clear(std::memory_order_release);
  }
};
```

```
struct spin_mutex {
private:
  std::atomic_flag flag = ATOMIC_FLAG_INIT;

public:
  void lock() {
    for (std::uint64_t k = 0; !flag.test_and_set(std::memory_order_acquire); ++k) {
      if (k < 4) ;
      else if (k < 16) __asm__ __volatile__( "rep; nop" : : : "memory" );
      else if (k < 64) sched_yield();
      else {
        timespec rqtp = { 0, 0 };
        rqtp.tv_sec = 0; rqtp.tv_nsec = 1000;
        nanosleep(&rqtp, nullptr);
      }
    }
  }

  void unlock() {
    flag.clear(std::memory_order_release);
  }
};
```

```
struct spin_mutex {
private:
  std::atomic_flag flag = ATOMIC_FLAG_INIT;

public:
  void lock() {
    while (flag.test_and_set(std::memory_order_acquire))
      flag.wait(true, std::memory_order_relaxed);
  }

  void unlock() {
    flag.clear(std::memory_order_release);
    flag.notify_one();
  }
};
```

```
struct spin_mutex {
private:
  std::atomic<bool> flag = ATOMIC_VAR_INIT(false);

public:
  void lock() {
    while (flag.exchange(true, std::memory_order_acquire))
      flag.wait(true, std::memory_order_relaxed);
  }

  void unlock() {
    flag.store(false, std::memory_order_release);
    flag.notify_one();
  }
};
```

std::atomic{_flag} wait and notify

```
template <typename T>
struct atomic {
  void wait(T, memory_order = memory_order::seq_cst) const volatile noexcept;
  void wait(T, memory_order = memory_order::seq_cst) const noexcept;
  void notify_one() volatile noexcept;
  void notify_one() noexcept;
  void notify_all() volatile noexcept;
  void notify_all() noexcept;
};
```

std::atomic{_flag} wait and notify

```
template <typename T>
struct atomic {
  void wait(T, memory_order = memory_order::seq_cst) const volatile noexcept;
  void wait(T, memory_order = memory_order::seq_cst) const noexcept;
  void notify_one() volatile noexcept;
  void notify_one() noexcept;
  void notify_all() volatile noexcept;
  void notify_all() noexcept;
};
```

std::atomic{_flag} wait and notify

```
template <typename T>
struct atomic {
  void wait(T, memory_order = memory_order::seq_cst) const volatile noexcept;
  void wait(T, memory_order = memory_order::seq_cst) const noexcept;
  void notify_one() volatile noexcept;
  void notify_one() noexcept;
  void notify_all() volatile noexcept;
  void notify_all() noexcept;
};
```

`std::atomic{_flag}` wait and notify

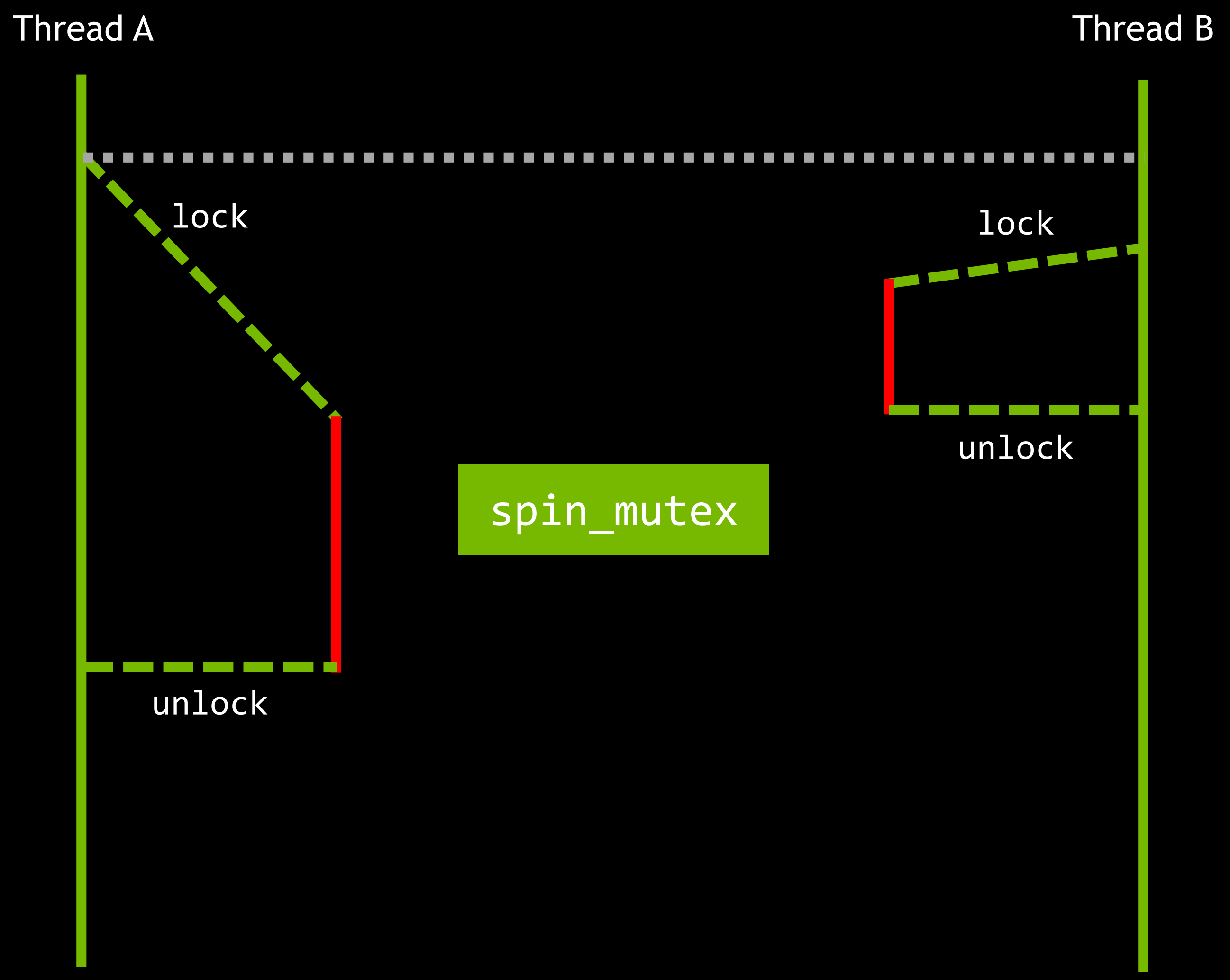
Some possible implementation strategies

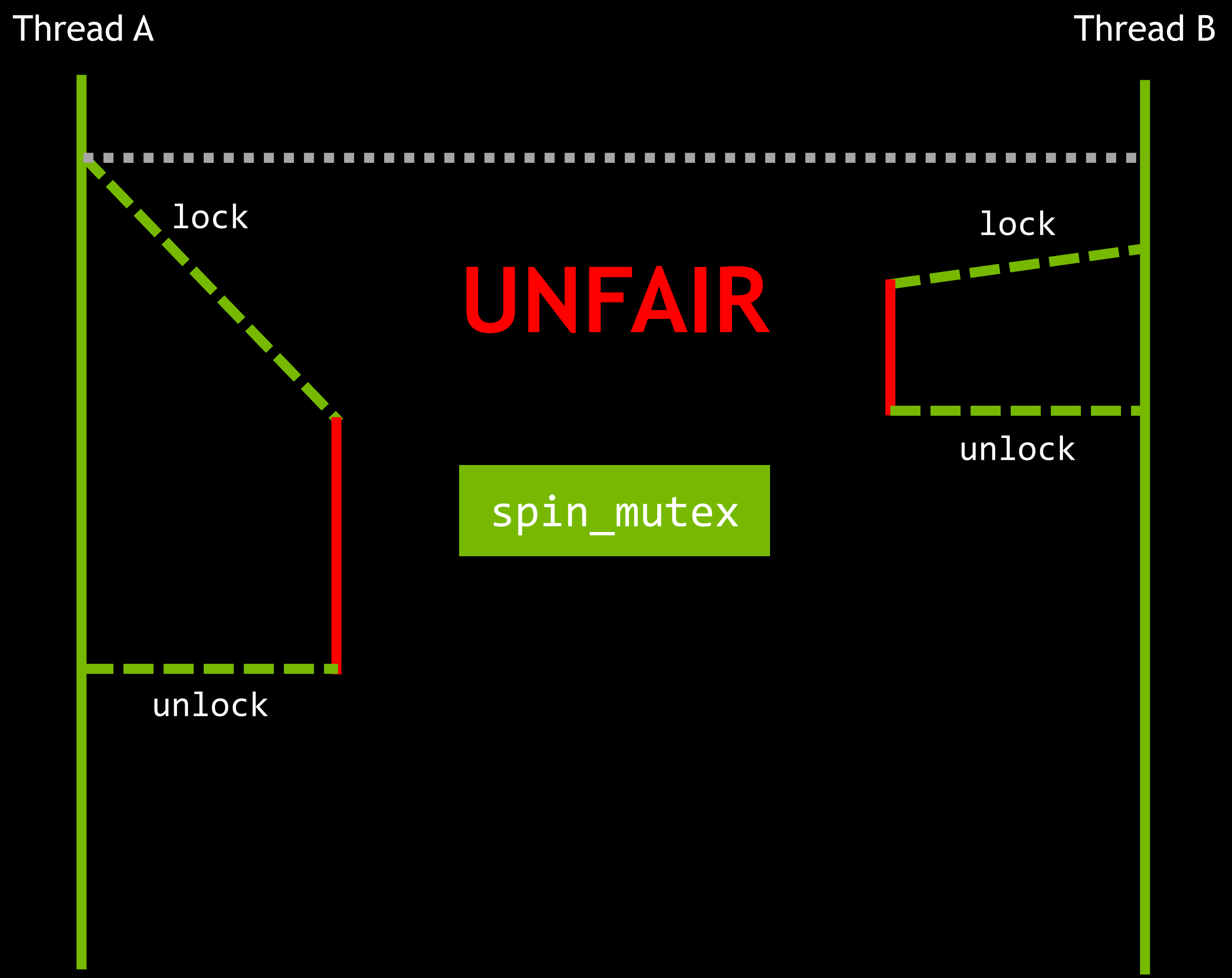
- Futex. Supported for certain size objects on Linux and Windows.
- Condition Variables. Supported for certain size objects on Linux and Mac.
- Contention Table. Used to optimize futex notify or to hold CVs.
- Timed back-off. Supported on everything.
- Spinlock. Supported on everything. Note: performance is terrible.

std::atomic_flag test

```
struct atomic_flag {
  bool test(memory_order = memory_order::seq_cst) const volatile noexcept;
  bool test(memory_order = memory_order::seq_cst) const noexcept;
  bool test_and_set(memory_order = memory_order::seq_cst) volatile noexcept;
  bool test_and_set(memory_order = memory_order::seq_cst) noexcept;
  void clear(memory_order = memory_order::seq_cst) volatile noexcept;
  void clear(memory_order = memory_order::seq_cst) noexcept;

  void wait(bool, memory_order = memory_order::seq_cst) const volatile noexcept;
  void wait(bool, memory_order = memory_order::seq_cst) const noexcept;
  void notify_one() volatile noexcept;
  void notify_one() noexcept;
  void notify_all() volatile noexcept;
  void notify_all() noexcept;
};
```

```cpp
struct spin_mutex {
private:
  std::atomic_flag flag = ATOMIC_FLAG_INIT;

public:
  void lock() {
    while (flag.test_and_set(std::memory_order_acquire))
      flag.wait(true, std::memory_order_relaxed);
  }

  void unlock() {
    flag.clear(std::memory_order_release);
    flag.notify_one();
  }
};
```


Thread A
Thread B
lock
lock
unlock
spin_mutex
unlock


Thread A
Thread B
lock
lock
UNFAIR
unlock
spin_mutex
unlock

```
struct ticket_mutex {
private:
  std::atomic<int> in = ATOMIC_VAR_INIT(0);
  std::atomic<int> out = ATOMIC_VAR_INIT(0);



public:
  void lock() {
    auto const my = in.fetch_add(1, std::memory_order_acquire);
    while (true) {
      auto const now = out.load(std::memory_order_acquire);
      if (now == my) return;
      out.wait(now, std::memory_order_relaxed);
    }
  }

  void unlock() {
    out.fetch_add(1, std::memory_order_release);
    out.notify_all();
  }
};
```

```
struct ticket_mutex {
private:
  std::atomic<int> in = ATOMIC_VAR_INIT(0);
  std::atomic<int> out = ATOMIC_VAR_INIT(0);



public:
  void lock() {
    auto const my = in.fetch_add(1, std::memory_order_acquire);
    while (true) {
      auto const now = out.load(std::memory_order_acquire);
      if (now == my) return;
      out.wait(now, std::memory_order_relaxed);
    }
  }

  void unlock() {
    out.fetch_add(1, std::memory_order_release);
    out.notify_all();
  }
};
```

```
struct ticket_mutex {
private:
  std::atomic<int> in = ATOMIC_VAR_INIT(0);
  std::atomic<int> out = ATOMIC_VAR_INIT(0);



public:
  void lock() {
    auto const my = in.fetch_add(1, std::memory_order_acquire);
    while (true) {
      auto const now = out.load(std::memory_order_acquire);
      if (now == my) return;
      out.wait(now, std::memory_order_relaxed);
    }
  }

  void unlock() {
    out.fetch_add(1, std::memory_order_release);
    out.notify_all();
  }
};
```

```
struct ticket_mutex {
private:
  std::atomic<int> in = ATOMIC_VAR_INIT(0);
  std::atomic<int> out = ATOMIC_VAR_INIT(0);



public:
  void lock() {
    auto const my = in.fetch_add(1, std::memory_order_acquire);
    while (true) {
      auto const now = out.load(std::memory_order_acquire);
      if (now == my) return;
      out.wait(now, std::memory_order_relaxed);
    }
  }

  void unlock() {
    out.fetch_add(1, std::memory_order_release);
    out.notify_all();
  }
};
```

```
struct ticket_mutex {
private:
  std::atomic<int> in = ATOMIC_VAR_INIT(0);
  std::atomic<int> out = ATOMIC_VAR_INIT(0);



public:
  void lock() {
    auto const my = in.fetch_add(1, std::memory_order_acquire);
    while (true) {
      auto const now = out.load(std::memory_order_acquire);
      if (now == my) return;
      out.wait(now, std::memory_order_relaxed);
    }
  }

  void unlock() {
    out.fetch_add(1, std::memory_order_release);
    out.notify_all();
  }
};
```

```
struct ticket_mutex {
private:
  std::atomic<int> in = ATOMIC_VAR_INIT(0);
  std::atomic<int> out = ATOMIC_VAR_INIT(0);



public:
  void lock() {
    auto const my = in.fetch_add(1, std::memory_order_acquire);
    while (true) {
      auto const now = out.load(std::memory_order_acquire);
      if (now == my) return;
      out.wait(now, std::memory_order_relaxed);
    }
  }

  void unlock() {
    out.fetch_add(1, std::memory_order_release);
    out.notify_all();
  }
};
```


NOW SERVING
52
PLEASE
Take A
Number

```
struct ticket_mutex {
private:
  std::atomic<int> in = ATOMIC_VAR_INIT(0);
  std::atomic<int> out = ATOMIC_VAR_INIT(0);



public:
  void lock() {
    auto const my = in.fetch_add(1, std::memory_order_acquire);
    while (true) {
      auto const now = out.load(std::memory_order_acquire);
      if (now == my) return;
      out.wait(now, std::memory_order_relaxed);
    }
  }

  void unlock() {
    out.fetch_add(1, std::memory_order_release);
    out.notify_all();
  }
};
```

```
struct ticket_mutex {
private:
  alignas(std::hardware_destructive_interference_size) std::atomic<int> in
    = ATOMIC_VAR_INIT(0);
  alignas(std::hardware_destructive_interference_size) std::atomic<int> out
    = ATOMIC_VAR_INIT(0);

public:
  void lock() {
    auto const my = in.fetch_add(1, std::memory_order_acquire);
    while (true) {
      auto const now = out.load(std::memory_order_acquire);
      if (now == my) return;
      out.wait(now, std::memory_order_relaxed);
    }
  }

  void unlock() {
    out.fetch_add(1, std::memory_order_release);
    out.notify_all();
  }
};
```

```cpp
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  ticket_mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  T pop();

public:
  constexpr concurrent_bounded_queue() = default;

  void enqueue(std::convertible_to<T> auto&& u);

  T dequeue();
  std::optional<T> try_dequeue();
};
```

Recipe For a Tasking Runtime

- Worker threads.
- Multi-consumer, multi-producer concurrent queue.
- **Termination detection mechanism.**
- Parallel algorithms.

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  concurrent_bounded_queue<any_invocable<void()>, QueueDepth> tasks;
  thread_group threads;

  void process_tasks(std::stop_token s);

public:
  bounded_depth_task_manager(std::uint64_t n);

  void submit(std::invocable auto&& f);
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  concurrent_bounded_queue<any_invocable<void()>, QueueDepth> tasks;
  thread_group threads;

  void process_tasks(std::stop_token s);

public:
  bounded_depth_task_manager(std::uint64_t n);

  void submit(std::invocable auto&& f);
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  concurrent_bounded_queue<any_invocable<void()>, QueueDepth> tasks;
  thread_group threads;

  void process_tasks(std::stop_token s);

public:
  bounded_depth_task_manager(std::uint64_t n);

  void submit(std::invocable auto&& f);
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  concurrent_bounded_queue<any_invocable<void()>, QueueDepth> tasks;
  thread_group threads;

  void process_tasks(std::stop_token s);

public:
  bounded_depth_task_manager(std::uint64_t n);

  void submit(std::invocable auto&& f);
};
```

```cpp
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  concurrent_bounded_queue<any_invocable<void()>, QueueDepth> tasks;
  thread_group threads;

  void process_tasks(std::stop_token s) {
    while (!s.stop_requested())
      tasks.dequeue()();
  }

public:
  bounded_depth_task_manager(std::uint64_t n)
    : threads(n, [&] (std::stop_token s) { process_tasks(s); })
  {}
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  concurrent_bounded_queue<any_invocable<void()>, QueueDepth> tasks;
  thread_group threads;

  void process_tasks(std::stop_token s) {
    while (!s.stop_requested())
      tasks.dequeue()();
  }

public:
  bounded_depth_task_manager(std::uint64_t n)
    : threads(n, [&] (std::stop_token s) { process_tasks(s); })
  {}
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  concurrent_bounded_queue<any_invocable<void()>, QueueDepth> tasks;
  thread_group threads;

public:
  void submit(std::invocable auto&& f) {
    tasks.enqueue(std::forward<decltype(f)>(f));
  }
};
```

```
int main() {
  std::atomic<std::uint64_t> count(0);

  {
    bounded_depth_task_manager<64> tm(6);

    for (auto i : stdv::iota(0, 256))
      tm.submit([&] { ++count; });
  }

  std::cout << count << "\n";
}
```

```
int main() {
  std::atomic<std::uint64_t> count(0);

  {
    bounded_depth_task_manager<64> tm(6);

    for (auto i : stdv::iota(0, 256))
      tm.submit([&] { ++count; });
  }

  std::cout << count << "\n";
}
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  concurrent_bounded_queue<any_invocable<void()>, QueueDepth> tasks;
  thread_group threads;

  void process_tasks(std::stop_token s) {
    while (!s.stop_requested())
      tasks.dequeue()();
  }

public:
  bounded_depth_task_manager(std::uint64_t n)
    : threads(n, [&] (std::stop_token s) { process_tasks(s); })
  {}
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  concurrent_bounded_queue<any_invocable<void()>, QueueDepth> tasks;
  thread_group threads;

  void process_tasks(std::stop_token s) {
    while (!s.stop_requested())
      tasks.dequeue()();
    while (true) {
      if (auto f = tasks.try_dequeue()) std::move(*f)();
      else break;
    }
  }

public:
  bounded_depth_task_manager(std::uint64_t n)
    : threads(n, [&] (std::stop_token s) { process_tasks(s); })
  {}
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  void process_tasks(std::stop_token s) {
    while (!s.stop_requested())
      tasks.dequeue()();
    while (true) {
      if (auto f = tasks.try_dequeue()) std::move(*f)();
      else break;
    }
  }
public:
 ~bounded_depth_task_manager() {
    std::latch l(threads.size() + 1);
    for (auto i : stdv::iota(0, threads.size()))
      submit([&] { l.arrive_and_wait(); });
    threads.request_stop();
    l.count_down();
  }
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  void process_tasks(std::stop_token s) {
    while (!s.stop_requested())
      tasks.dequeue()();
    while (true) {
      if (auto f = tasks.try_dequeue()) std::move(*f)();
      else break;
    }
  }
public:
 ~bounded_depth_task_manager() {
    std::latch l(threads.size() + 1);
    for (auto i : stdv::iota(0, threads.size()))
      submit([&] { l.arrive_and_wait(); });
    threads.request_stop();
    l.count_down();
  }
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  void process_tasks(std::stop_token s) {
    while (!s.stop_requested())
      tasks.dequeue()();
    while (true) {
      if (auto f = tasks.try_dequeue()) std::move(*f)();
      else break;
    }
  }
public:
 ~bounded_depth_task_manager() {
    std::latch l(threads.size() + 1);
    for (auto i : stdv::iota(0, threads.size()))
      submit([&] { l.arrive_and_wait(); });
    threads.request_stop();
    l.count_down();
  }
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  void process_tasks(std::stop_token s) {
    while (!s.stop_requested())
      tasks.dequeue()();
    while (true) {
      if (auto f = tasks.try_dequeue()) std::move(*f)();
      else break;
    }
  }
public:
 ~bounded_depth_task_manager() {
    std::latch l(threads.size() + 1);
    for (auto i : stdv::iota(0, threads.size()))
      submit([&] { l.arrive_and_wait(); });
    threads.request_stop();
    l.count_down();
  }
};
```

```cpp
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  void process_tasks(std::stop_token s) {
    while (!s.stop_requested())
      tasks.dequeue()();
    while (true) {
      if (auto f = tasks.try_dequeue()) std::move(*f)();
      else break;
    }
  }
public:
 ~bounded_depth_task_manager() {
    std::latch l(threads.size() + 1);
    for (auto i : stdv::iota(0, threads.size()))
      submit([&] { l.arrive_and_wait(); });
    threads.request_stop();
    l.count_down();
  }
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  void process_tasks(std::stop_token s) {
    while (!s.stop_requested())
      tasks.dequeue()();
    while (true) {
      if (auto f = tasks.try_dequeue()) std::move(*f)();
      else break;
    }
  }
public:
 ~bounded_depth_task_manager() {
    std::latch l(threads.size() + 1);
    for (auto i : stdv::iota(0, threads.size()))
      submit([&] { l.arrive_and_wait(); });
    threads.request_stop();
    l.count_down();
  }
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  void process_tasks(std::stop_token s) {
    while (!s.stop_requested())
      tasks.dequeue()();
    while (true) {
      if (auto f = tasks.try_dequeue()) std::move(*f)();
      else break;
    }
  }
public:
 ~bounded_depth_task_manager() {
    std::latch l(threads.size() + 1);
    for (auto i : stdv::iota(0, threads.size()))
      submit([&] { l.arrive_and_wait(); });
    threads.request_stop();
    l.count_down();
  }
};
```

std::latch

```
struct latch {
  constexpr explicit latch(ptrdiff_t expected);

  latch(const latch&) = delete;
  latch& operator=(const latch&) = delete;

  void count_down(ptrdiff_t update = 1);
  bool try_wait() const noexcept;
  void wait() const;
  void arrive_and_wait(ptrdiff_t update = 1);
};
```

std::latch

```
struct latch {
  constexpr explicit latch(ptrdiff_t expected);

  latch(const latch&) = delete;
  latch& operator=(const latch&) = delete;

  void count_down(ptrdiff_t update = 1);
  bool try_wait() const noexcept;
  void wait() const;
  void arrive_and_wait(ptrdiff_t update = 1);
};
```

std::latch

```
struct latch {
  constexpr explicit latch(ptrdiff_t expected);

  latch(const latch&) = delete;
  latch& operator=(const latch&) = delete;

  void count_down(ptrdiff_t update = 1);
  bool try_wait() const noexcept;
  void wait() const;
  void arrive_and_wait(ptrdiff_t update = 1);
};
```

std::latch

```
struct latch {
  constexpr explicit latch(ptrdiff_t expected);

  latch(const latch&) = delete;
  latch& operator=(const latch&) = delete;

  void count_down(ptrdiff_t update = 1);
  bool try_wait() const noexcept;
  void wait() const;
  void arrive_and_wait(ptrdiff_t update = 1);
};
```

std::latch

```
struct latch {
  constexpr explicit latch(ptrdiff_t expected);

  latch(const latch&) = delete;
  latch& operator=(const latch&) = delete;

  void count_down(ptrdiff_t update = 1);
  bool try_wait() const noexcept;
  void wait() const;
  void arrive_and_wait(ptrdiff_t update = 1);
};
```

```cpp
void submit_tree(auto& tm, std::atomic<std::uint64_t>& count, std::uint64_t level) {
  ++count;
  if (0 != level) {
    tm.submit([&tm, &count, level] { submit_tree(tm, count, level - 1); });
    tm.submit([&tm, &count, level] { submit_tree(tm, count, level - 1); });
  }
}

int main() {
  std::atomic<std::uint64_t> count(0);

  {
    bounded_depth_task_manager<64> tm(6);

    submit_tree(tm, count, 8);
  }

  std::cout << count << "\n";
}
```

```
void submit_tree(auto& tm, std::atomic<std::uint64_t>& count, std::uint64_t level) {
  ++count;
  if (0 != level) {
    tm.submit([&tm, &count, level] { submit_tree(tm, count, level - 1); });
    tm.submit([&tm, &count, level] { submit_tree(tm, count, level - 1); });
  }
}

int main() {
  std::atomic<std::uint64_t> count(0);

  {
    bounded_depth_task_manager<64> tm(6);

    submit_tree(tm, count, 8);
  }

  std::cout << count << "\n";
}
```

```cpp
void submit_tree(auto& tm, std::atomic<std::uint64_t>& count, std::uint64_t level) {
  ++count;
  if (0 != level) {
    tm.submit([&tm, &count, level] { submit_tree(tm, count, level - 1); });
    tm.submit([&tm, &count, level] { submit_tree(tm, count, level - 1); });
  }
}

int main() {
  std::atomic<std::uint64_t> count(0);

  {
    bounded_depth_task_manager<64> tm(6);

    submit_tree(tm, count, 8);
  }

  std::cout << count << "\n";
}
```

```cpp
void submit_tree(auto& tm, std::atomic<std::uint64_t>& count, std::uint64_t level) {
  ++count;
  if (0 != level) {
    tm.submit([&tm, &count, level] { submit_tree(tm, count, level - 1); });
    tm.submit([&tm, &count, level] { submit_tree(tm, count, level - 1); });
  }
}

int main() {
  std::atomic<std::uint64_t> count(0);

  {
    bounded_depth_task_manager<64> tm(6);

    submit_tree(tm, count, 8);
  }

  std::cout << count << "\n";
}
```

```
void submit_tree(auto& tm, std::atomic<std::uint64_t>& count, std::uint64_t level) {
  ++count;
  if (0 != level) {
    tm.submit([&tm, &count, level] { submit_tree(tm, count, level - 1); });
    tm.submit([&tm, &count, level] { submit_tree(tm, count, level - 1); });
  }
}

int main() {
  std::atomic<std::uint64_t> count(0);

  {
    bounded_depth_task_manager<64> tm(6);

    submit_tree(tm, count, 8);
  }

  std::cout << count << "\n";
}
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  concurrent_bounded_queue<any_invocable<void()>, QueueDepth> tasks;
  thread_group threads;

public:
  void submit(std::invocable auto&& f) {
    tasks.enqueue(std::forward<decltype(f)>(f));
  }
};
```

```cpp
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  concurrent_bounded_queue<any_invocable<void()>, QueueDepth> tasks;
  thread_group threads;

public:
  void submit(std::invocable auto&& f) {
    while (!tasks.try_enqueue(std::forward<decltype(f)>(f)))
      make_progress();
  }

  void make_progress() {
    if (auto f = tasks.try_dequeue()) std::move(*f)();
  }
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  concurrent_bounded_queue<any_invocable<void()>, QueueDepth> tasks;
  thread_group threads;

public:
  void submit(std::invocable auto&& f) {
    while (!tasks.try_enqueue(std::forward<decltype(f)>(f)))
      make_progress();
  }

  void make_progress() {
    if (auto f = tasks.try_dequeue()) std::move(*f)();
  }
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  concurrent_bounded_queue<any_invocable<void()>, QueueDepth> tasks;
  thread_group threads;

public:
  void submit(std::invocable auto&& f) {
    while (!tasks.try_enqueue(std::forward<decltype(f)>(f)))
      make_progress();
  }

  void make_progress() {
    if (auto f = tasks.try_dequeue()) std::move(*f)();
  }
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  concurrent_bounded_queue<any_invocable<void()>, QueueDepth> tasks;
  thread_group threads;

public:
  void submit(std::invocable auto&& f) {
    while (!tasks.try_enqueue(std::forward<decltype(f)>(f)))
      make_progress();
  }

  void make_progress() {
    if (auto f = tasks.try_dequeue()) std::move(*f)();
  }
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  concurrent_bounded_queue<any_invocable<void()>, QueueDepth> tasks;
  thread_group threads;

  void process_tasks(std::stop_token s);

public:
  bounded_depth_task_manager(std::uint64_t n);

  ~bounded_depth_task_manager();

  void submit(std::invocable auto&& f);

  void make_progress();
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  concurrent_bounded_queue<any_invocable<void()>, QueueDepth> tasks;
  thread_group threads;

  void process_tasks(std::stop_token s);

public:
  bounded_depth_task_manager(std::uint64_t n);

  ~bounded_depth_task_manager();

  void submit(std::invocable auto&& f);

  void make_progress();
};
```

Recipe For a Tasking Runtime

- Worker threads.
- Multi-consumer, multi-producer concurrent queue.
- Termination detection mechanism.
- **Parallel algorithms.**

```
template <stdr::range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
void histogram(I&& input, O output, T inc, OP op) {
  stdr::for_each(input, [&] (auto&& t) { output[op(t)] += inc; });
}
```

```
template <stdr::range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
void histogram(I&& input, O output, T inc, OP op) {
  stdr::for_each(input, [&] (auto&& t) { output[op(t)] += inc; });
}
```

```
template <stdr::range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
void histogram(I&& input, O output, T inc, OP op) {
  stdr::for_each(input, [&] (auto&& t) { output[op(t)] += inc; });
}
```

```
template <stdr::range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
void histogram(I&& input, O output, T inc, OP op) {
  stdr::for_each(input, [&] (auto&& t) { output[op(t)] += inc; });
}
```

```
template <execution_policy EP,
          stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
 requires /* ... */
void histogram(EP&& exec, I&& input, O output, T inc, OP op);
```

```
template <execution_policy EP,
          stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
 requires /* ... */
void histogram(EP&& exec, I&& input, O output, T inc, OP op);
```

```
template <execution_policy EP,
          stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
 requires /* ... */
void histogram(EP&& exec, I&& input, O output, T inc, OP op);
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  std::uint64_t const elements = stdr::distance(input);
  std::uint64_t const chunks   = (1 + ((elements - 1) / exec.chunk_size()));

  // ...
}
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  std::uint64_t const elements = stdr::distance(input);
  std::uint64_t const chunks   = (1 + ((elements - 1) / exec.chunk_size()));

  // ...
}
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  std::uint64_t const elements = stdr::distance(input);
  std::uint64_t const chunks   = (1 + ((elements - 1) / exec.chunk_size()));

  std::latch l(chunks);

 // ...
}
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit(
      [&, =chunk] {
        // ...
      }
    );
}
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit(
      // ...
    );
}
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit(
      [=, &exec, &input, &l] {
        auto const my_begin = chunk * exec.chunk_size();
        auto const my_end   = std::min(elements, (chunk + 1) * exec.chunk_size());

        // ...
      }
    );
}
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit(
      [=, &exec, &input, &l] {
        auto const my_begin = chunk * exec.chunk_size();
        auto const my_end   = std::min(elements, (chunk + 1) * exec.chunk_size());

        // ...
      }
    );
}
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit(
      [=, &exec, &input, &l] {
        auto const my_begin = chunk * exec.chunk_size();
        auto const my_end   = std::min(elements, (chunk + 1) * exec.chunk_size());

        std::for_each(stdr::begin(input) + my_begin,
                      stdr::begin(input) + my_end,
                      [&] (auto&& t) {
                        output[op(t)] += inc;
                      });

        // ...
      }
    );
}
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit(
      [=, &exec, &input, &l] {
        auto const my_begin = chunk * exec.chunk_size();
        auto const my_end   = std::min(elements, (chunk + 1) * exec.chunk_size());

        std::for_each(stdr::begin(input) + my_begin,
                      stdr::begin(input) + my_end,
                      [&] (auto&& t) {
                        output[op(t)] += inc;
                      });

        // ...
      }
    );
}
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit(
      [=, &exec, &input, &l] {
        auto const my_begin = chunk * exec.chunk_size();
        auto const my_end   = std::min(elements, (chunk + 1) * exec.chunk_size());

        std::for_each(stdr::begin(input) + my_begin,
                      stdr::begin(input) + my_end,
                      [&] (auto&& t) {
                        std::atomic_ref r(output[op(t)]);
                        r.fetch_add(inc, std::memory_order_relaxed);
                      });

        // ...
      }
    );
}
```

std::atomic_ref<T>

std::atomic<T> holds a T.

```
template <struct T>
struct atomic {
private:
  T data; // exposition only
public:
  // ...
};
```

std::atomic<T> does not hold a T.

```
template <struct T>
struct atomic_ref {
private:
  T* ptr; // exposition only
public:
  explicit atomic_ref(T&);

  // Otherwise, same API as std::atomic.
};
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit(
      [=, &exec, &input, &l] {
        auto const my_begin = chunk * exec.chunk_size();
        auto const my_end   = std::min(elements, (chunk + 1) * exec.chunk_size());

        std::for_each(stdr::begin(input) + my_begin,
                      stdr::begin(input) + my_end,
                      [&] (auto&& t) {
                        std::atomic_ref r(output[op(t)]);
                        r.fetch_add(inc, std::memory_order_relaxed);
                      });

        // ...
      }
    );
}
```

std::atomic<*floating-point*>

```
template<> struct atomic<floating-point> {
  floating-point fetch_add(floating-point,
                           memory_order = memory_order_seq_cst) volatile noexcept;
  floating-point fetch_add(floating-point,
                           memory_order = memory_order_seq_cst) noexcept;
  floating-point fetch_sub(floating-point,
                           memory_order = memory_order_seq_cst) volatile noexcept;
  floating-point fetch_sub(floating-point,
                           memory_order = memory_order_seq_cst) noexcept;
};
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit(
      [=, &exec, &input, &l] {
        auto const my_begin = chunk * exec.chunk_size();
        auto const my_end   = std::min(elements, (chunk + 1) * exec.chunk_size());

        std::for_each(stdr::begin(input) + my_begin,
                      stdr::begin(input) + my_end,
                      [&] (auto&& t) {
                        std::atomic_ref r(output[op(t)]);
                        r.fetch_add(inc, std::memory_order_relaxed);
                      });

        l.count_down();
        while (!l.try_wait()) exec.make_progress();
      }
    );
}
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit(
      [=, &exec, &input, &l] {
        auto const my_begin = chunk * exec.chunk_size();
        auto const my_end   = std::min(elements, (chunk + 1) * exec.chunk_size());

        std::for_each(stdr::begin(input) + my_begin,
                      stdr::begin(input) + my_end,
                      [&] (auto&& t) {
                        std::atomic_ref r(output[op(t)]);
                        r.fetch_add(inc, std::memory_order_relaxed);
                      });

        l.count_down();
        while (!l.try_wait()) exec.make_progress();
      }
    );
}
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  // ...

  std::latch l(chunks);

  for (std::uint64_t chunk = 0; chunk < chunks; ++chunk)
    exec.submit(
      // ...
    );

  while (!l.try_wait())
    exec.make_progress();
}
```

```
template <stdr::range I, std::weakly_incrementable O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
O exclusive_scan(I&& input, O output, T init, OP op);
```

Exclusive Scan

a	b	c	d	e	f	g	h	i

Exclusive Scan

a	b	c	d	e	f	g	h	i

X	Xa	Xab	Xabc	Xabcd	Xabcde	Xabcdef	Xabcdefg	Xabcdefgh

Exclusive Scan

a	b	c	d	e	f	g	h	i

X	Xa	Xab	Xabc	Xabcd	Xabcde	Xabcdef	Xabcdefg	Xabcdefgh

Exclusive Scan

a	b	c	d	e	f	g	h	i

	a	ab

`std::exclusive_scan`

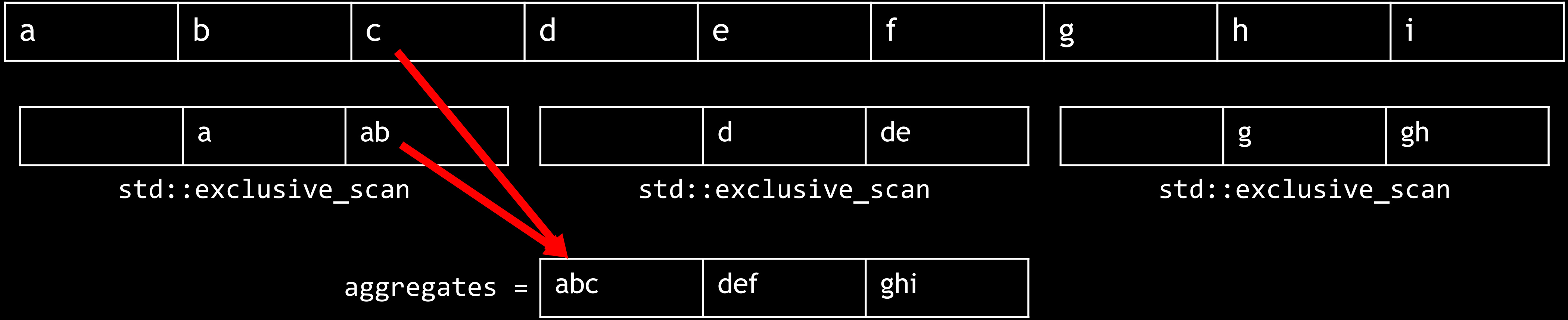
	d	de

`std::exclusive_scan`

	g	gh

`std::exclusive_scan`

Exclusive Scan



Exclusive Scan

a	b	c	d	e	f	g	h	i

	a	ab

`std::exclusive_scan`

	d	de

`std::exclusive_scan`

	g	gh

`std::exclusive_scan`

aggregates =

abc	def	ghi

`std::exclusive_scan` (init = X)

Exclusive Scan

a	b	c	d	e	f	g	h	i

	a	ab

`std::exclusive_scan`

	d	de

`std::exclusive_scan`

	g	gh

`std::exclusive_scan`

`aggregates =`

X	Xabc	Xabcdef

`std::exclusive_scan` (init = X)

Exclusive Scan

a	b	c	d	e	f	g	h	i

	a	ab

`std::exclusive_scan`

	d	de

`std::exclusive_scan`

	g	gh

`std::exclusive_scan`

`aggregates =`

X	Xabc	Xabcdef

`std::exclusive_scan` (init = X)

X	Xa	Xab

Increment by aggregates[0]

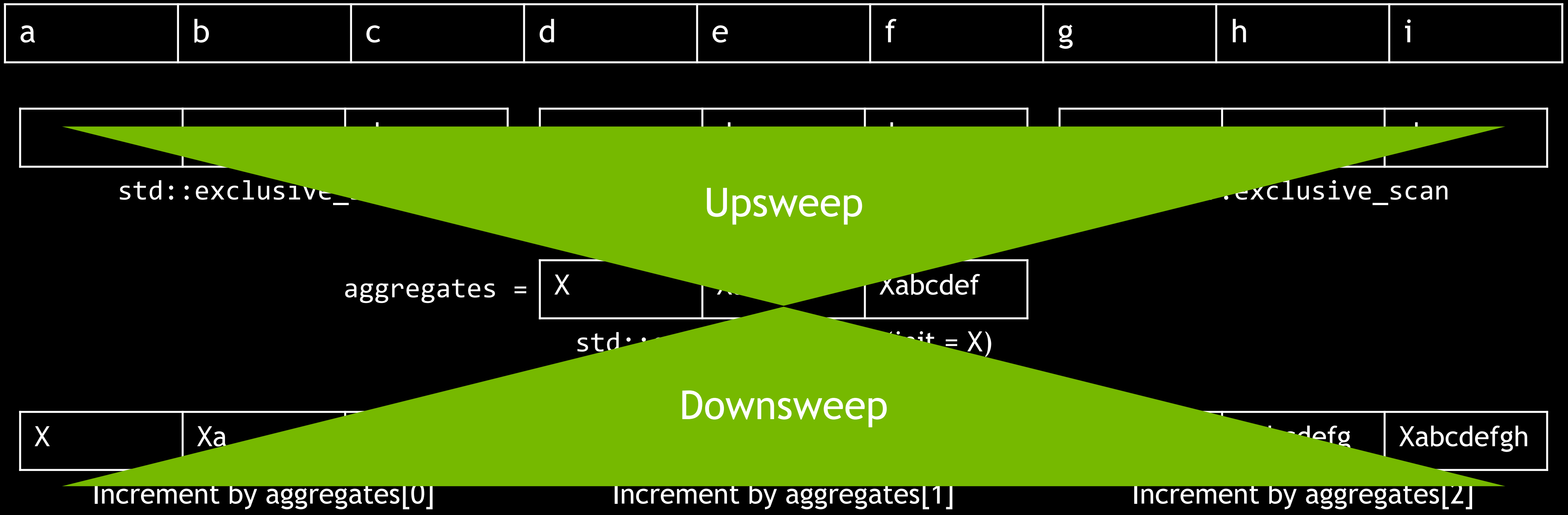
Xabc	Xabcd	Xabcde

Increment by aggregates[1]

Xabcdef	Xabcdefg	Xabcdefgh

Increment by aggregates[2]

Exclusive Scan



```
template <stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
struct exclusive_scanner {
  // ...
};
```

```
template <stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
struct exclusive_scanner {
private:
  I  input;
  O  output;
  T  init;
  BO op;

  // ...
};
```

```
template <stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
struct exclusive_scanner {
private:
  I  input;
  O  output;
  T  init;
  BO op;

  std::uint64_t const concurrency;
  std::uint64_t const elements;
  std::uint64_t const chunk_size;

  // ...
};
```

```
template <stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
struct exclusive_scanner {
private:
  I  input;
  O  output;
  T  init;
  BO op;

  std::uint64_t const concurrency;
  std::uint64_t const elements;
  std::uint64_t const chunk_size;

  std::vector<T>                    aggregates;
  std::barrier<std::function<void()>> upsweep_barrier;
  std::latch                        downsweep_latch;

  // ...
};
```

```
template <stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
struct exclusive_scanner {
private:
  I  input;
  O  output;
  T  init;
  BO op;

  std::uint64_t const concurrency;
  std::uint64_t const elements;
  std::uint64_t const chunk_size;

  std::vector<T>                     aggregates;
  std::barrier<std::function<void()>> upsweep_barrier;
  std::latch                         downsweep_latch;

  // ...
};
```

std::barrier

```
template <typename CompletionFunction = see below>
struct barrier {
  using arrival_token = see below;

  constexpr explicit barrier(ptrdiff_t expected,
                             CompletionFunction f = CompletionFunction());

  [[nodiscard]] arrival_token arrive(ptrdiff_t update = 1);
  void wait(arrival_token&& arrival) const;

  void arrive_and_wait();
  void arrive_and_drop();
};
```

std::barrier

```
template <typename CompletionFunction = see below>
struct barrier {
  using arrival_token = see below;

  constexpr explicit barrier(ptrdiff_t expected,
                             CompletionFunction f = CompletionFunction());

  [[nodiscard]] arrival_token arrive(ptrdiff_t update = 1);
  void wait(arrival_token&& arrival) const;

  void arrive_and_wait();
  void arrive_and_drop();
};
```

std::barrier

```
template <typename CompletionFunction = see below>
struct barrier {
  using arrival_token = see below;

  constexpr explicit barrier(ptrdiff_t expected,
                             CompletionFunction f = CompletionFunction());

  [[nodiscard]] arrival_token arrive(ptrdiff_t update = 1);
  void wait(arrival_token&& arrival) const;

  void arrive_and_wait();
  void arrive_and_drop();
};
```

std::barrier

```
template <typename CompletionFunction = see below>
struct barrier {
  using arrival_token = see below;

  constexpr explicit barrier(ptrdiff_t expected,
                             CompletionFunction f = CompletionFunction());

  [[nodiscard]] arrival_token arrive(ptrdiff_t update = 1);
  void wait(arrival_token&& arrival) const;

  void arrive_and_wait();
  void arrive_and_drop();
};
```

std::barrier

```
template <typename CompletionFunction = see below>
struct barrier {
  using arrival_token = see below;

  constexpr explicit barrier(ptrdiff_t expected,
                             CompletionFunction f = CompletionFunction());

  [[nodiscard]] arrival_token arrive(ptrdiff_t update = 1);
  void wait(arrival_token&& arrival) const;

  void arrive_and_wait();
  void arrive_and_drop();
};
```

std::latch

```
struct latch {
  constexpr explicit latch(ptrdiff_t expected);

  latch(const latch&) = delete;
  latch& operator=(const latch&) = delete;

  void count_down(ptrdiff_t update = 1);
  bool try_wait() const noexcept;
  void wait() const;
  void arrive_and_wait(ptrdiff_t update = 1);
};
```

`std::latch` vs `std::barrier`

std::latch

- Supports asynchronous arrival.
- Single phase.
- No thread identity:
  - Threads may arrive multiple times.
  - Any thread may wait on a latch.
- No completion function.

std::barrier

- Supports asynchronous arrival.
- Multi phase.
- Thread identity:
  - A thread may arrive only once per phase.
  - Only a thread who has arrived may wait.
- Supports completion functions.

```
template <stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
struct exclusive_scanner {
private:
  I  input;
  O  output;
  T  init;
  BO op;

  std::uint64_t const concurrency;
  std::uint64_t const elements;
  std::uint64_t const chunk_size;

  std::vector<T>                     aggregates;
  std::barrier<std::function<void()>> upsweep_barrier;
  std::latch                         downsweep_latch;

  // ...
};
```

```
template <stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
struct exclusive_scanner {
private:
  I  input;
  O  output;
  T  init;
  BO op;

  std::uint64_t const concurrency;
  std::uint64_t const elements;
  std::uint64_t const chunk_size;

  std::vector<T>                     aggregates;
  std::barrier<std::function<void()>> upsweep_barrier;
  std::latch                         downsweep_latch;

  // ...
};
```

```
template <stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
struct exclusive_scanner {
private:
  I  input;
  O  output;
  T  init;
  BO op;

  std::uint64_t const concurrency;
  std::uint64_t const elements;
  std::uint64_t const chunk_size;

  std::vector<T>                    aggregates;
  std::barrier<std::function<void()>> upsweep_barrier;
  std::latch                         downsweep_latch;

  void scan_aggregates();
public:
  exclusive_scanner(std::uint64_t concurrency_, I input_, O output_, T init_, BO op_);

  void launch(execution_policy auto&& exec);

  void process_chunk(std::uint64_t chunk, stdr::random_access_range auto&& my_input, O my_output);
};
```

```
template <stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
struct exclusive_scanner {
public:
  exclusive_scanner(std::uint64_t concurrency_, I input_, O output_, T init_, BO op_);
    : input(std::move(input_))
    , output(std::move(output_))
    , init(std::move(init_))
    , op(std::move(op_))
    , concurrency(concurrency_)
    , elements(stdr::distance(input_))
    , chunk_size((elements + concurrency - 1) / concurrency)
    , aggregates(concurrency)
    , upsweep_barrier(concurrency, [&] { scan_aggregates(); })
    , downsweep_latch(concurrency)
  {}
};
```

```
template <stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
struct exclusive_scanner {
public:
  exclusive_scanner(std::uint64_t concurrency_, I input_, O output_, T init_, BO op_);
    : input(std::move(input_))
    , output(std::move(output_))
    , init(std::move(init_))
    , op(std::move(op_))
    , concurrency(concurrency_)
    , elements(stdr::distance(input_))
    , chunk_size((elements + concurrency - 1) / concurrency)
    , aggregates(concurrency)
    , upsweep_barrier(concurrency, [&] { scan_aggregates(); })
    , downsweep_latch(concurrency)
  {}
};
```

```
template <stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
struct exclusive_scanner {
private:
  std::vector<T> aggregates;

  void scan_aggregates() {
    std::exclusive_scan(aggregates, aggregates.begin(), std::move(init), op);
  }
};
```

```
template <stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
struct exclusive_scanner {
public:
  void launch(execution_policy auto&& exec) {
    for (auto chunk : stdv::iota(0, concurrency))
      exec.submit(
        [&, chunk] {
          auto const my_begin = chunk * chunk_size;
          auto const my_end   = std::min(elements, (chunk + 1) * chunk_size);
          process_chunk(chunk,
                        std::span(stdr::begin(input) + my_begin,
                                  stdr::begin(input) + my_end)
                        output + my_begin);
        }
      );
    }

    while (!downsweep_latch.try_wait())
      exec.make_progress();
  }
};
```

```
template <stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
struct exclusive_scanner {
public:
  void launch(execution_policy auto&& exec) {
    for (auto chunk : stdv::iota(0, concurrency))
      exec.submit(
        [&, chunk] {
          auto const my_begin = chunk * chunk_size;
          auto const my_end   = std::min(elements, (chunk + 1) * chunk_size);
          process_chunk(chunk,
                        std::span(stdr::begin(input) + my_begin,
                                  stdr::begin(input) + my_end)
                        output + my_begin);
        }
      );
    }

    while (!downsweep_latch.try_wait())
      exec.make_progress();
  }
};
```

```
template <stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
struct exclusive_scanner {
public:
  void launch(execution_policy auto&& exec) {
    for (auto chunk : stdv::iota(0, concurrency))
      exec.submit(
        [&, chunk] {
          auto const my_begin = chunk * chunk_size;
          auto const my_end   = std::min(elements, (chunk + 1) * chunk_size);
          process_chunk(chunk,
                        std::span(stdr::begin(input) + my_begin,
                                  stdr::begin(input) + my_end)
                        output + my_begin);
        }
      );
    }

    while (!downsweep_latch.try_wait())
      exec.make_progress();
  }
};
```

```
template <stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
struct exclusive_scanner {
public:
  void process_chunk(std::uint64_t chunk, stdr::random_access_range auto&& my_input, O my_output) {
    // Upsweep.
    auto const& last_element = *(last - 1);
    aggregates[chunk] = op(*--std::exclusive_scan(my_input, my_output, T{}, op), last_element);

    // Barrier completion function does the aggregate sum.
    upsweep_barrier.arrive_and_wait();

    // Downsweep.
    std::for_each(my_output, my_output + stdr::distance(my_input),
                  [&, chunk] (auto& t) { t = op(std::move(t), aggregates[chunk]); });

    downsweep_latch.count_down();
  }
};
```

```
template <stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
struct exclusive_scanner {
public:
  void process_chunk(std::uint64_t chunk, stdr::random_access_range auto&& my_input, O my_output) {
    // Upsweep.
    auto const& last_element = *(last - 1);
    aggregates[chunk] = op(*--std::exclusive_scan(my_input, my_output, T{}, op), last_element);

    // Barrier completion function does the aggregate sum.
    upsweep_barrier.arrive_and_wait();

    // Downsweep.
    std::for_each(my_output, my_output + stdr::distance(my_input),
                  [&, chunk] (auto& t) { t = op(std::move(t), aggregates[chunk]); });

    downsweep_latch.count_down();
  }
};
```

```
template <stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
struct exclusive_scanner {
public:
  void process_chunk(std::uint64_t chunk, stdr::random_access_range auto&& my_input, O my_output) {
    // Upsweep.
    auto const& last_element = *(last - 1);
    aggregates[chunk] = op(*--std::exclusive_scan(my_input, my_output, T{}, op), last_element);

    // Barrier completion function does the aggregate sum.
    upsweep_barrier.arrive_and_wait();

    // Downsweep.
    std::for_each(my_output, my_output + stdr::distance(my_input),
                  [&, chunk] (auto& t) { t = op(std::move(t), aggregates[chunk]); });

    downsweep_latch.count_down();
  }
};
```

```
template <stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
struct exclusive_scanner {
public:
  void process_chunk(std::uint64_t chunk, stdr::random_access_range auto&& my_input, O my_output) {
    // Upsweep.
    auto const& last_element = *(last - 1);
    aggregates[chunk] = op(*--std::exclusive_scan(my_input, my_output, T{}, op), last_element);

    // Barrier completion function does the aggregate sum.
    upsweep_barrier.arrive_and_wait();

    // Downsweep.
    std::for_each(my_output, my_output + stdr::distance(my_input),
                  [&, chunk] (auto& t) { t = op(std::move(t), aggregates[chunk]); });

    downsweep_latch.count_down();
  }
};
```

```
template <stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
struct exclusive_scanner {
public:
  void process_chunk(std::uint64_t chunk, stdr::random_access_range auto&& my_input, O my_output) {
    // Upsweep.
    auto const& last_element = *(last - 1);
    aggregates[chunk] = op(*--std::exclusive_scan(my_input, my_output, T{}, op), last_element);

    // Barrier completion function does the aggregate sum.
    upsweep_barrier.arrive_and_wait();

    // Downsweep.
    std::for_each(my_output, my_output + stdr::distance(my_input),
                  [&, chunk] (auto& t) { t = op(std::move(t), aggregates[chunk]); });

    downsweep_latch.count_down();
  }
};
```

`std::barrier`

Synchronization

N x arrives

happens-before

1 x completion

happens-before

N x waits complete

```
template <stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
struct exclusive_scanner {
public:
  void process_chunk(std::uint64_t chunk, stdr::random_access_range auto&& my_input, O my_output) {
    // Upsweep.
    auto const& last_element = *(last - 1);
    aggregates[chunk] = op(*--std::exclusive_scan(my_input, my_output, T{}, op), last_element);

    // Barrier completion function does the aggregate sum.
    upsweep_barrier.arrive_and_wait();

    // Downsweep.
    std::for_each(my_output, my_output + stdr::distance(my_input),
                  [&, chunk] (auto& t) { t = op(std::move(t), aggregates[chunk]); });

    downsweep_latch.count_down();
  }
};
```

```
template <stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
struct exclusive_scanner {
public:
  void process_chunk(std::uint64_t chunk, stdr::random_access_range auto&& my_input, O my_output) {
    // Upsweep.
    auto const& last_element = *(last - 1);
    aggregates[chunk] = op(*--std::exclusive_scan(my_input, my_output, T{}, op), last_element);

    // Barrier completion function does the aggregate sum.
    upsweep_barrier.arrive_and_wait();

    // Downsweep.
    std::for_each(my_output, my_output + stdr::distance(my_input),
                  [&, chunk] (auto& t) { t = op(std::move(t), aggregates[chunk]); });

    downsweep_latch.count_down();
  }
};
```

C++20 Synchronization Library

- std::atomic<T> et al
  - wait/notify interface
  - std::atomic_ref<T>
  - test interface for std::atomic_flag
  - Floating-point specializations
- std::latch & std::barrier
- std::counting_semaphore
- std::jthread
  - Joining destructor
  - std::stop_* interruption mechanism

@blelbach

NVIDIA