Concurrency in C++20 and beyond

Anthony Williams

Woven Planet
`https://www.woven-planet.global`

November 2021

Concurrency in C++20 and beyond

- New Concurrency Features in C++20
- New Concurrency Features for Future Standards

New Concurrency Features in C++20

woven planet

New Concurrency Features in C++20

C++20 is a **huge** release, with lots of new features, including Concurrency facilities:

- Support for cooperative cancellation of threads
- A new thread class that automatically joins
- New synchronization facilities
- Updates to atomics
- Coroutines

Cooperative Cancellation

Cooperative Cancellation

- GUIs often have “Cancel” buttons for long-running operations.
- You don’t need a GUI to want to cancel an operation.
- Forcibly stopping a thread is undesirable

woven planet

Cooperative Cancellation II

C++20 provides `std::stop_source` and `std::stop_token` to handle cooperative cancellation.

Purely cooperative: if the target task doesn't check, nothing happens.

Cooperative Cancellation III

1. Create a `std::stop_source`

Cooperative Cancellation III

1. Create a `std::stop_source`
2. Obtain a `std::stop_token` from the `std::stop_source`

Cooperative Cancellation III

1. Create a `std::stop_source`
2. Obtain a `std::stop_token` from the `std::stop_source`
3. Pass the `std::stop_token` to a new thread or task

Cooperative Cancellation III

1. Create a `std::stop_source`
2. Obtain a `std::stop_token` from the `std::stop_source`
3. Pass the `std::stop_token` to a new thread or task
4. When you want the operation to stop call `source.request_stop()`

Cooperative Cancellation III

1. Create a `std::stop_source`
2. Obtain a `std::stop_token` from the `std::stop_source`
3. Pass the `std::stop_token` to a new thread or task
4. When you want the operation to stop call `source.request_stop()`
5. Periodically call `token.stop_requested()` to check
   ⇒ Stop the task if stopping requested

woven planet

Cooperative Cancellation III

1. Create a `std::stop_source`
2. Obtain a `std::stop_token` from the `std::stop_source`
3. Pass the `std::stop_token` to a new thread or task
4. When you want the operation to stop call `source.request_stop()`
5. Periodically call `token.stop_requested()` to check
   ⇒ Stop the task if stopping requested
6. **If you do not check `token.stop_requested()`, nothing happens**

Cooperative Cancellation IV

`std::stop_token` integrates with `std::condition_variable_any`, so if your code is waiting for something to happen, the wait can be interrupted by a stop request.

woven planet

Cooperative Cancellation V

```
std::mutex m;
std::queue<Data> q;
std::condition_variable_any cv;

Data wait_for_data(std::stop_token st){
  std::unique_lock lock(m);
  if(!cv.wait_until(lock,[]{return !q.empty();},st))
    throw op_was_cancelled();
  Data res=q.front();
  q.pop_front();
  return res;
}
```

Cooperative Cancellation VI

You can also use `std::stop_callback` to provide your own cancellation mechanism. e.g. to cancel some async IO.

```
Data read_file(
    std::stop_token st,
    std::filesystem::path filename ){
  auto handle=open_file(filename);
  std::stop_callback cb(st,[&]{ cancel_io(handle);});
  return read_data(handle); // blocking
}
```

New thread class

New thread class: `std::jthread`

`std::jthread` integrates with `std::stop_token` to support cooperative cancellation.

Destroying a `std::jthread` calls `source.request_stop()` and `thread.join()`.

The thread still needs to check the stop token passed in to the thread function.

New thread class II

```
void thread_func(
    std::stop_token st,
    std::string arg1,int arg2){
  while(!st.stop_requested()){
    do_stuff(arg1,arg2);
  }
}

void foo(std::string s){
  std::jthread t(thread_func,s,42);
  do_stuff();
} // destructor requests stop and joins
```

woven planet

New synchronization facilities

New synchronization facilities

- Latches
- Barriers
- Semaphores

Latches

Latches

`std::latch` is a single-use counter that allows threads to wait for the count to reach zero.

1. Create the latch with a non-zero count
2. One or more threads decrease the count
3. Other threads may wait for the latch to be signalled.
4. When the count reaches zero it is permanently signalled and all waiting threads are woken.

Waiting for tasks with a latch

```
void foo(){
  unsigned const thread_count=...;
  std::latch done(thread_count);
  my_data data[thread_count];
  std::vector<std::jthread> threads;
  for(unsigned i=0;i<thread_count;++i)
    threads.push_back(std::jthread([&,i]{
      data[i]=make_data(i);
      done.count_down();
      do_more_stuff();
    }));
  done.wait();
  process_data();
}
```

Synchronizing Tests with Latches

Using a latch is great for multithreaded tests:

1. Set up the test data
2. Create a latch
3. Create the test threads
   ⇒ The first thing each thread does is
   `test_latch.arrive_and_wait()`
4. When all threads have reached the latch they are unblocked to run their code

Barriers

Barriers

`std::barrier<>` is a reusable barrier.

Synchronization is done in **phases**:

1. Construct a barrier, with a non-zero count and a **completion function**
2. One or more threads arrive at the barrier
3. These or other threads wait for the barrier to be signalled
4. When the count reaches zero, the barrier is signalled, the **completion function** is called and the count is reset

woven planet

Barriers II

Barriers are great for loop synchronization between parallel tasks.

The **completion function** allows you to do something between loops: pass the result on to another step, write to a file, etc.

Barriers III

```
unsigned const num_threads=...;
void finish_task();

std::barrier<std::function<void()>> b(
  num_threads,finish_task);

void worker_thread(std::stop_token st,unsigned i){
  while(!st.stop_requested()){
    do_stuff(i);
    b.arrive_and_wait();
  }
}
```

woven planet

Semaphores

Semaphores

A semaphore represents a number of available “slots”. If you **acquire** a slot on the semaphore then the count is decreased until you **release** the slot.

Attempting to acquire a slot when the count is zero will either block or fail.

A thread may release a slot without acquiring one and vice versa.

Semaphores II

Semaphores can be used to build just about any synchronization mechanism, including latches, barriers and mutexes.

A **binary semaphore** has 2 states: 1 slot free or no slots free. It can be used as a mutex.

Semaphores in C++20

C++20 has `std::counting_semaphore<max_count>`
`std::binary_semaphore` is an alias for `std::counting_semaphore<1>`.

As well as **blocking** `sem.acquire()`, there are also `sem.try_acquire()`, `sem.try_acquire_for()` and `sem.try_acquire_until()` functions that fail instead of blocking.

Semaphores in C++20 II

```
std::counting_semaphore<5> slots(5);

void func(){
  slots.acquire();
  do_stuff(); // at most 5 threads can be here
  slots.release();
}
```

Updates to Atomics

Updates to Atomics

- Low-level waiting for atomics
- Atomic Smart Pointers
- `std::atomic_ref`

Low-level waiting for atomics

`std::atomic<T>` now provides a `var.wait()` member function to wait for it to change.

`var.notify_one()` and `var.notify_all()` wake one or all threads blocked in `wait()`.

Like a low level `std::condition_variable`.

Atomic smart pointers

C++20 provides std::atomic<std::shared_ptr<T>> and std::atomic<std::weak_ptr<T>> specializations.

- May or may not be **lock-free**
- If lock-free, can simplify lock-free algorithms.
- If not lock-free, a better replacement for std::shared_ptr<T> and a mutex.
- Can be slow under high contention.

atomic<shared_ptr<T>> Stack

```
template<typename T> class stack{
  struct node{
    T value;
    shared_ptr<node> next;
    node(){} node(T&& nv):value(std::move(nv)){}
  };
  std::atomic<shared_ptr<node>> head;
public:
  stack():head(nullptr){}
  ~stack(){ while(head.load()) pop(); }
  void push(T);
  T pop();
};
```

atomic<shared_ptr<T>> Stack II

```
template<typename T>
void stack<T>::push(T val){
  auto new_node=std::make_shared<node>(
    std::move(val));
  new_node->next=head.load();
  while(!head.compare_exchange_weak(
    new_node->next,new_node)){}
}
```

woven planet

atomic<shared_ptr<T>> Stack III

```
template<typename T>
T stack<T>::pop(){
  auto old_head=head.load();
  while(old_head){
    if(head.compare_exchange_strong(
        old_head,old_head->next))
      return std::move(old_head->value);
  }
  throw std::runtime_error("Stack empty");
}
```

woven planet

`std::atomic_ref`

`std::atomic_ref` allows you to perform atomic operations on non-atomic objects.

This can be important when sharing headers with C code, or where a `struct` needs to match a specific binary layout so you can't use `std::atomic`.

If you use `std::atomic_ref` to access an object, all accesses to that object must use `std::atomic_ref`.

std::atomic_ref

```
struct my_c_struct{
  int count;
  data* ptr;
};

void do_stuff(my_c_struct* p){

  std::atomic_ref<int> count_ref(p->count);
  ++count_ref;
  // ...
}
```

Coroutines

What is a Coroutine?

A **coroutine** is a function that can be **suspended** mid execution and **resumed** at a later time.

Resuming a coroutine continues from the suspension point; local variables have their values from the original call.

Stackless Coroutines

C++20 provides **stackless coroutines**

- Only the locals for the current function are saved
- Everything is localized
- Minimal memory allocation — can have millions of in-flight coroutines
- Whole coroutine overhead can be eliminated by the compiler — Gor's "disappearing coroutines"

Waiting for others

```
future<remote_data>
async_get_data(key_type key);

future<data> retrieve_data(
  key_type key){
  auto rem_data=
    co_await async_get_data(key);
  co_return process(rem_data);
}
```

What C++20 coroutines are missing

C++20 has no library support for coroutines:

$\Longrightarrow$ you need to write your own support code (hard) or use a third party library.

e.g.
`https://github.com/lewissbaker/cppcoro`
`https://github.com/David-Haim/concurrencpp`

New Concurrency Features for Future Standards

woven planet

New Features for Future Standards

Additional concurrency facilities are under development for future standards. These include:

- A synchronization wrapper for ordinary objects
- Executors — thread pools and more
- Coroutine library support for concurrency
- Concurrent Data Structures
- Safe Memory Reclamation Facilities

woven planet

A synchronization wrapper for ordinary objects

A synchronization wrapper

`synchronized_value` encapsulates a mutex and a value.

- Cannot forget to lock the mutex
- It’s easy to lock across a whole operation
- Multi-value operations are just as easy

woven planet

A synchronization wrapper II

```
synchronized_value<std::string> sv;

std::string get_value(){
  return apply([](std::string& s){
    return s;
  },sv);
}

void append_string(std::string extra){
  apply([&](std::string& s){
    s+=extra;
  },sv);
}
```

A synchronization wrapper III

```
synchronized_value<std::string> sv;
synchronized_value<std::string> sv2;

std:string combine_strings(){
  return apply(
    [&](std::string& s,std::string & s2){
      return s+s2;
    },sv,sv2);
}
```

Executors

woven planet

Executors

Executor

An object that controls how, where and when a task is executed

Thread pools are a special case of **Executors.**

Executors ⇒ Senders and Receivers

Executor as a concept combines too many responsibilities. The `std::execution` proposal splits them into 3:

Scheduler

Controls **where** a task is to be run

Sender

Controls **what** the task is

Receiver

Controls **what** to do with the result

Senders and Receivers

Asynchronous operation are pipelines: each sender is chained to a receiver, which can then initiate another sender, or just store the result somewhere.

Initial sender ⇒ receiver ⇒ sender ⇒ receiver ⇒ sender ⇒ … ⇒ … ⇒ final receiver

The scheduler runs the pipeline.

Senders and Receivers

Schedulers are things like thread pools and GPU schedulers.

Receivers are usually internal to algorithms like `std::execution::then` and `std::this_thread::sync_wait`.

Application-level code usually focuses on constructing **Senders** from the **tasks** that need to be done.

Scheduling work

If you have a task that needs to be run, the simplest mechanism is just to call `std::execution::execute`.

```
// Assumed for all subsequent examples
namespace execution=std::execution;

execution::execute(some_scheduler,[]{
    do_something();
  });
```

This **detaches** the work, so you can't wait for it.

Scheduling work

`execution::execute` can be split into multiple steps:

```
auto initial_sender=execution::schedule(my_scheduler);

auto work_sender=execution::then(initial_sender, []{
    do_something();
  });

execution::start_detached(work_sender);
```

Waiting

You can start execution on a scheduler, and then wait for the result.

```
auto done=execution::ensure_started(work_sender);

do_other_stuff();

auto result = std::this_thread::sync_wait(done);
```

Scheduling work

You can chain operations together with `execution::then`:

```
auto initial_sender=execution::schedule(my_scheduler);
auto middle_sender=execution::then(initial_sender, []{
    return find_the_answer();
  });
auto work_sender=execution::then(
  middle_sender, [](int answer){
     return find_the_question(answer);
  });
auto result = std::this_thread::sync_wait(work_sender);
```

Pipelines

Code can be simplified using pipes (|) rather than named variables.

```
auto work_sender=execution::schedule(my_scheduler) |
  execution::then(find_the_answer) |
  execution::then(find_the_question);

auto result = std::this_thread::sync_wait(work_sender);
```

Handling errors

By default, exceptions are propagated down the pipeline, and rethrown from `sync_wait`.

`execution::upon_error` can be used to handle errors **within** the pipeline.

```
auto work_sender=execution::schedule(my_scheduler) |
  execution::then(do_something) |
  execution::upon_error(handle_error) |
  execution::then(do_something_else);

auto result = std::this_thread::sync_wait(work_sender);
```

woven planet

Libunifex

```
https://github.com/facebookexperimental/libunifex
```

Provides a sample implementation of the executor model and extensive documentation.

Coroutine support for concurrency

Coroutine support for concurrency

I hope to see things like `task<T>` that allows you to write a coroutine intended to run as an async task:

```
task<int> task1();
task<int> task2();

task<int> sum(){
  int r1=co_await task1();
  int r2=co_await task2();
  co_return r1+r2;
}
```

Coroutines support for concurrency

All awaitables are senders:

```
task<int> coroutine_task();

auto foo() {
  return execution::sync_wait(coroutine_task());
}
```

Coroutine support for concurrency

Some senders are awaitable:

```
task<int> other_coro(){
  auto sender = execution::schedule(my_scheduler) |
    execution::then(find_the_answer);
  co_return co_wait sender;
}
```

Concurrent Data Structures

Concurrent Data Structures

Developers commonly need data structures that allow concurrent access.

Proposals for standardization include:

- Concurrent Queues
- Concurrent Hash Maps

Concurrent Queues

Queues are a core mechanism for communicating between threads.

```
concurrent_queue<MyData> q;

void producer_thread(){
  q.push(generate_data());
}
void consumer_thread(){
  process_data(q.value_pop());
}
```

woven planet

Concurrent Hash Maps

- Hash maps are often used for fast look-up of data
- Using a mutex for synchronization can hurt performance
- Special implementations designed for concurrent access can be better

Safe Memory Reclamation Facilities

Safe Memory Reclamation Facilities

Lock-free algorithms need a way to delete data when no other thread is accessing it.

RCU provides a lock-free read side. Deletion is either blocking or deferred on a background thread.

Hazard pointers defer deletion, and provide a different set of performance trade-offs.

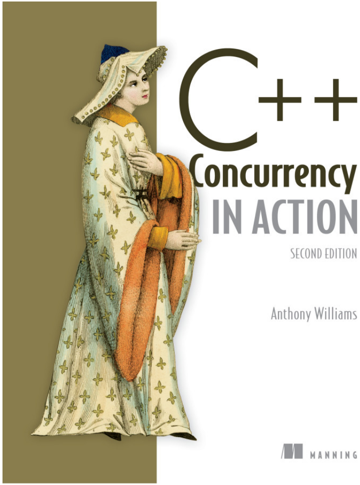
Both mechanisms are in the second Concurrency TS for future C++ standardization.

Proposals

Here are the papers for those future things that have proposals:

- Synchronized Value: P0290
- Senders and Receivers: P2300
- Concurrency TS2 draft (Hazard pointers and RCU): N4895
- Concurrent Queues: P0260
- Concurrent Hash Map: P0652 P1761

My Book



C++ Concurrency in Action
Second Edition

Covers C++17 and the
first Concurrency TS
C++20 Addendum coming soon!

cplusplusconcurrencyinaction.com

woven planet

Questions?