An introduction to multithreading in C++20

Anthony Williams

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

November 2022

Assumptions

- New project
- C++20 compiler and library

An introduction to multithreading in C++20

- Choosing your Concurrency Model
- Starting and Managing Threads
- Synchronizing Data

woven planet

Choosing your Concurrency Model

woven planet

Choosing your Concurrency Model

We want to use multithreading in our applications for 2 fundamental reasons:

Choosing your Concurrency Model

We want to use multithreading in our applications for 2 fundamental reasons:

- Scalability

Choosing your Concurrency Model

We want to use multithreading in our applications for 2 fundamental reasons:

- Scalability
- Separation of Concerns

Choosing your Concurrency Model

We want to use multithreading in our applications for 2 fundamental reasons:

- Scalability
- Separation of Concerns

These reasons inform our choice of model.

Multithreading for Scalability

If you want Scalability, then Amdahl's law applies:

$$S = \frac{1}{1 - p + \frac{p}{n}}$$

S = Maximum speedup multiplier
p = Fraction of program that can be parallelized
n = Number of processors

Multithreading for Scalability

If you want Scalability, then Amdahl's law applies:

$$S = \frac{1}{1 - 0.9 + \frac{0.9}{n}}$$

S = Maximum speedup multiplier
p = Fraction of program that can be parallelized
n = Number of processors

Multithreading for Scalability

If you want Scalability, then Amdahl's law applies:

$$S = \frac{1}{1 - 0.9 + \frac{0.9}{1000}}$$

S = Maximum speedup multiplier

p = Fraction of program that can be parallelized

n = Number of processors

Multithreading for Scalability

If you want Scalability, then Amdahl's law applies:

$$9.91 = \frac{1}{1 - 0.9 + \frac{0.9}{1000}}$$

S = Maximum speedup multiplier

p = Fraction of program that can be parallelized

n = Number of processors

Multithreading for Scalability

If you want Scalability, then Amdahl's law applies:

$$9.999 = \frac{1}{1 - 0.9 + \frac{0.9}{100000}}$$

S = Maximum speedup multiplier

p = Fraction of program that can be parallelized

n = Number of processors

Multithreading for Scalability

If you want Scalability, then Amdahl's law applies:

$$99.9 = \frac{1}{1 - 0.99 + \frac{0.99}{100000}}$$

S = Maximum speedup multiplier

p = Fraction of program that can be parallelized

n = Number of processors

Parallel Algorithms

Many standard library algorithms have parallel versions:

```
std::vector<MyData> data= ...;
std::sort(
  std::execution::par,
  data.begin(),data.end(),
  MyComparator{});
```

woven planet

Parallel Algorithms

See if you can combine consecutive calls:

```
std::transform(std::execution::par,...);
std::reduce(std::execution::par,...);
```

Parallel Algorithms

See if you can combine consecutive calls:

```
std::transform(std::execution::par,...);
std::reduce(std::execution::par,...);
```

```
std::transform_reduce(std::execution::par,...);
```

Independent Tasks

Split your work into many independent tasks and use a (non-standard) thread pool.

```
thread_pool tp;

void foo(){
  execute(tp,[]{ do_work(); });
  execute(tp,[]{ do_other_work(); });
}
```

Separation of Concerns

- Raw performance not a priority
- Large sequential tasks that can run concurrently “in the background”

Dedicated Threads

Run each long-running task on its own thread.

```
std::jthread gui{[]{ run_gui(); }};
std::jthread printing{[]{ do_printing(); }};
```

Starting and Managing Threads

~~Starting and Managing Threads~~

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 Types

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 Usage

1. Create a `std::stop_source`

Cooperative Cancellation Usage

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

Cooperative Cancellation Usage

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 Usage

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()`

woven planet

Cooperative Cancellation Usage

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
   $\Rightarrow$ Stop the task if stopping requested

Cooperative Cancellation Usage

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**

woven planet

Cancellation Example

```
void stoppable_func(std::stop_token st){
  while(!st.stop_requested()){
    do_stuff();
  }
}

void stopper(std::stop_source source){
  while(!done()){
    do_something();
  }
  source.request_stop();
}
```

Custom Cancellation

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
}
```

woven planet

Starting and Managing Threads

Starting and Managing Threads

To manage threads, use the `std::jthread` class in 99% of cases.

Starting and Managing Threads

To manage threads, use the `std::jthread` class in 99% of cases.

`std::async` can be used where you want a result.

Starting and Managing Threads

To manage threads, use the `std::jthread` class in 99% of cases.

`std::async` can be used where you want a result.

`std::thread` should only be used if you have no choice.

`std::jthread` — Overview

Creating a `std::jthread` object starts the thread.

```
std::jthread t{my_func,arg1,arg2};
```

Runs `my_func(stop_token,arg1,arg2)` on the new thread.

`std::jthread` — Overview

Creating a `std::jthread` object starts the thread.

```
std::jthread t{my_func,arg1,arg2};
```

Runs `my_func(stop_token,arg1,arg2)` on the new thread.

Or runs `my_func(arg1,arg2)` on the new thread.

`std::jthread` — Basic API

`std::jthread` default constructor

Create an empty object with no thread

`std::jthread x{Callable,`*Args...*`}`

Create a new `std::stop_source` — `src`

Create a new thread running *Callable(*`src.get_token(),`*Args...)*

or *Callable(Args...)*

`std::jthread` destructor

Calls `src.request_stop()` and waits for the owned thread to finish

`x.get_id()`

Obtains the thread ID of the owned thread

`x.join()`

Wait for the owned thread to finish

`std::jthread` is a value type

`std::jthread` is a **handle**.

It is **movable** ⇒

- Ownership can be transferred
- Can be stored in containers (e.g. `std::vector<std::jthread>`)
- no need to use `new`

Threads: Callables and Arguments

The callable and arguments are **copied** into storage local to the new thread.

This helps avoid dangling references and race conditions.

Use `std::ref` when you really want a reference. Or use a lambda.

std::jthread destructor semantics

The destructor will request stop and wait for the thread to finish:

```
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
```

Cancellation and `std::jthread`

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

Cancellation and `std::jthread`

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

- Starting a thread with `std::jthread` implicitly creates a `std::stop_source`.

Cancellation and `std::jthread`

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

- Starting a thread with `std::jthread` implicitly creates a `std::stop_source`.
- A stop token obtained from `source.get_token()` is passed to your thread function as an **optional** first parameter.

Cancellation and `std::jthread`

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

- Starting a thread with `std::jthread` implicitly creates a `std::stop_source`.
- A stop token obtained from `source.get_token()` is passed to your thread function as an **optional** first parameter.
- Destroying a `std::jthread` calls `source.request_stop()` and `thread.join()`.

woven planet

Cancellation and `std::jthread`

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

- Starting a thread with `std::jthread` implicitly creates a `std::stop_source`.
- A stop token obtained from `source.get_token()` is passed to your thread function as an **optional** first parameter.
- 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.

`std::jthread` — Cancellation API

Given

```
std::jthread x{some_callable};
```

`x.get_stop_source()`

obtain the stop source for the thread

`x.get_stop_token()`

obtain a stop token for the thread

`x.request_stop()`

equivalent to `x.get_stop_source().request_stop()`

Synchronization facilities

Synchronization facilities

Most multithreaded programs need to share state between threads.

woven planet

Synchronization facilities

Most multithreaded programs need to share state between threads.

Data Race

> Unsynchronized access to a memory location from more than thread, where at least one thread is writing.

Synchronization facilities

Most multithreaded programs need to share state between threads.

Data Race

Unsynchronized access to a memory location from more than thread, where at least one thread is writing.

All data races are undefined behaviour $\Rightarrow$ we need synchronization.

Synchronization facilities

C++ provides a bunch of synchronization facilities:

- Latches
- Barriers
- Futures
- Mutexes
- Semaphores
- Atomics

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.

Latch API

`std::latch x{count}`

Create a latch with the specified count

`x.count_down()`

Decrease the count. Trigger latch if count reaches zero

`x.wait()`

Wait for the latch to be triggered.

`x.arrive_and_wait()`

`x.count_down()` then `x.wait()`

Waiting for tasks with a latch

```
void foo(){
  unsigned const thread_count=...;
  std::latch done(thread_count);
  std::vector<std::optional<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(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. Some of these 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

Barrier API

`std::barrier<task_type> x{count,task}`

Create a barrier with the specified count and completion function

`auto arrival_token=x.arrive()`

Decrease the count. Trigger completion phase if count reaches zero

`x.wait(arrival_token)`

Wait for the completion phase to be complete.

`x.arrive_and_wait()`

`x.wait(x.arrive())`

`x.arrive_and_drop()`

Decrease the count permanently (and potentially trigger completion phase) **without waiting**.

Barriers and Loops

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. It is run on **one of the participating threads**.

Barrier Example

```
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();
  }
}
```

Futures

Futures

Futures provide a mechanism for a one-shot transfer of data between threads.

- `std::async` — launch a task that returns a value
- `std::promise` — explicitly set a value
- `std::packaged_task` — wrap a task that returns a value

All of these give you a `std::future<T>` for the result.

`std::future<T>` — Basic API

`std::future<T>` default constructor

    Create an empty object with no state

`f.valid()`

    Check if the future has state

`f.wait()`

    Wait for the data to be ready

`f.wait_for(duration)`

    Wait for the data to be ready for the specified duration

`f.wait_until(time_point)`

    Wait for the data to be ready until the specified time

`x.get()`

    Wait for the data and retrieve it

Using futures

Futures provide blocking waits and polling for data:

```
void blocking(std::future<int> f){
  f.wait(); // can be omitted
  do_stuff(f.get()); // blocks until ready
}

void polling(std::future<int> f){
  if(f.wait_for(0s) == std::future_status::ready){
    do_stuff(f.get());
  }
}
```

woven planet

`std::promise<T>` — Basic API

`std::promise<T>` default constructor

Create an object with an empty state

`p.valid()`

Check if the promise has state

`p.set_value()`

Set the value in the state

`p.set_exception(ex_ptr)`

Set the exception in the state

`p.get_future()`

Get the `std::future<T>` for the state

woven planet

Passing data with Futures

```
std::promise<MyData> prom;
std::future<MyData> f=prom.get_future();

std::jthread thread1{[f=std::move(f)]{
  do_stuff(f.get());
}};

std::jthread thread2{[&prom]{
  prom.set_value(make_data());
}};
```

Passing exceptions with Futures

```
std::promise<MyData> prom;
std::future<MyData> f=prom.get_future();

std::jthread thread1{[f=std::move(f)]{
  do_stuff(f.get()); // throws my_exception
}};

std::jthread thread2{[&prom]{
  prom.set_exception(
    std::make_exception_ptr(my_exception{}));
}};
```

Launching tasks with `std::async`

`std::async` can be used to create threads.

```
// Call func(arg1,arg2) on a new thread
auto f=std::async(std::launch::async,
  func,arg1,arg2);
```

- `f.get()` will return the result of the call to `func`
- `f` owns the thread. Similar to `jthread` — the destructor will wait for the thread to exit.

`std::future<T>` is one-shot

After calling `f.get()`, a `std::future` no longer holds a value.

```
std::promise<MyData> prom;
std::future<MyData> f=prom.get_future();

do_stuff(f.get());
assert(!f.valid());
f.get(); // error, will throw
```

woven planet

std::shared_future<T>

`std::shared_future` allows multiple threads to receive the same result.

```
std::promise<MyData> prom;
std::shared_future<MyData> f=
  prom.get_future().share();

std::jthread thread1{[f]{ do_stuff(f.get()); }};
std::jthread thread2{[f]{ do_stuff(f.get()); }};
```

Mutexes

Mutexes

Mutex: **Mut**ual **Ex**clusion

A mutex is a means of **preventing** concurrent execution.

Mutexes

Mutex: **Mu**tual **Ex**clusion

A mutex is a means of **preventing** concurrent execution.

$\Rightarrow$ we should use them as sparingly as possible.

C++ Mutexes

C++ provides 6 mutex types. For most code that is 5 too many.

C++ Mutexes

C++ provides 6 mutex types. For most code that is 5 too many.

- **`std::mutex`** ⇐ **Use this one**
- `std::timed_mutex`
- `std::recursive_mutex`
- `std::recursive_timed_mutex`
- `std::shared_mutex`
- `std::shared_timed_mutex`

Locking mutexes

Locking and unlocking is done via RAII types.

- **`std::scoped_lock`** ⇐ **use this one**
- `std::unique_lock`
- `std::lock_guard`
- `std::shared_lock`

Locking example

```
int some_data;
std::mutex some_data_mutex;

void add_to_data(int delta){
  std::scoped_lock lock(some_data_mutex); // locks
  some_data+=delta;
} // unlocks mutex in lock destructor
```

woven planet

Locking multiple mutexes

```
class account
{
  std::mutex m;
  currency_value balance;
public:
  friend void transfer(account& from,account& to,
    currency_value amount)
  {
    std::scoped_lock lock_from(from.m);
    std::scoped_lock lock_to(to.m);
    from.balance -= amount;
    to.balance += amount;
  }
};
```

Locking multiple mutexes

Thread 1	**Thread 2**
Calls `transfer(a1,a2,v1)`	Calls `transfer(a2,a1,v2)`
Locks `a1.m`	Locks `a2.m`
Tries to lock `a2.m` *Blocks*	Tries to lock `a1.m` *Blocks*

DEADLOCK

Locking multiple mutexes

```
class account
{
  std::mutex m;
  currency_value balance;
public:
  friend void transfer(account& from,account& to,
    currency_value amount)
  {
    std::scoped_lock locks(from.m,to.m);
    from.balance -= amount;
    to.balance += amount;
  }
};
```

Waiting for Data

Waiting for Data

How do you wait for data to be ready?

Busy wait?

```
std::mutex m;
std::optional<Data> data;

void busy_wait(){
  while(true){
    std::scoped_lock lock(m);
    if(data.has_value()) break;
  }
  process_data();
}
```

Busy waiting is bad

Busy waiting:

- Consumes CPU time waiting
- Wastes electricity
- Delays the notification

Busy waiting is bad

Busy waiting:

- Consumes CPU time waiting
- Wastes electricity
- Delays the notification

`std::condition_variable` provides notifications to avoid busy waiting.

Condition Variables optimize waiting

```
std::mutex m;
std::condition_variable cond;
std::optional<Data> data;

void cv_wait(){
  std::unique_lock lock(m);
  cond.wait(lock,[]{return data.has_value();});
  process_data();
}
```

woven planet

Condition Variable notifications

`std::condition_variable` must be notified.

```
void cv_notify(){
  {
    std::scoped_lock lock(m);
    data = make_data();
  }
  cond.notify_one();
}
```

woven planet

Cancelling Waits

Handling cancellation with busy waits is easy:

```
void busy_wait(std::stop_token token){
  while(true){
    if(token.stop_requested()) return;
    std::scoped_lock lock(m);
    if(data.has_value()) break;
  }
  process_data();
}
```

For condition variables we need `std::condition_variable_any`.

Cancelling Condition Variable Waits

```
std::condition_variable_any cond;

void cv_wait(std::stop_token token){
  std::unique_lock lock(m);
  if(!cond.wait(lock,token,
    []{return data.has_value();}))
    return;
  process_data();
}
```

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.

woven planet

Semaphores II

Semaphores can be used to build just about any synchronization mechanism, including latches, barriers and mutexes.
See The Little Book Of Semaphores.

Mostly you are better off using the higher level structures.

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

woven planet

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.

Semaphore Example

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

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

Atomics

Atomics

Atomic variables are the lowest level of synchronization primitive.

In C++ they are written `std::atomic<T>`.

`T` must be **Trivially copyable**, and **Bitwise comparable**.

Atomics

Atomic variables are the lowest level of synchronization primitive.

In C++ they are written `std::atomic<T>`.

`T` must be **Trivially copyable**, and **Bitwise comparable**.
Except `T` can also be `std::shared_ptr<U>` or `std::weak_ptr<U>`.

woven planet

Lock free or not?

`std::atomic<T>` may not be lock-free — may use an internal mutex.

`std::atomic_flag`, `std::atomic_signed_lock_free` and `std::atomic_unsigned_lock_free` are only guaranteed lock-free types.

Only most platforms `std::atomic<`*integral-type*`>` and `std::atomic<T*>` are lock-free.

Can query with `std::atomic<T>::is_always_lock_free`.

Summary

Summary

- Avoid managing your own threads if you can
- Use `std::jthread` for threads
- Use `std::stop_token` for cancellation
- Use `std::future`, `std::latch` and `std::barrier` where you can
- Use `std::mutex` almost everywhere else
- Use `std::atomic` in rare cases

Questions?

We're hiring

woven planet

3 global offices and remote

Arene at Woven Planet is the platform that will allow for programmable mobility - autonomy, navigation & games!