

Attribution/License

- Original Materials developed by Mike Shah, Ph.D. (www.mshah.io)
- This slideset and associated source code may not be distributed without prior written notice from both authors

Please do not redistribute slides/source without prior written permission.

NYC++ Meetup

📍 New York, NY, USA

👥 1,376 members · Public group ⓘ

👤 Organized by **Daniel Katz** and **3 others**

The Little Talk of Semaphores -- and a Tour of C++ Concurrency

with Mike Shah

18:30 - 20:30 ET
Thur. October 17, 2024

60 minutes with Q&A
Introductory/Intermediate
Audience

Social: [@MichaelShah](https://twitter.com/MichaelShah)

Web: mshah.io

Courses: courses.mshah.io

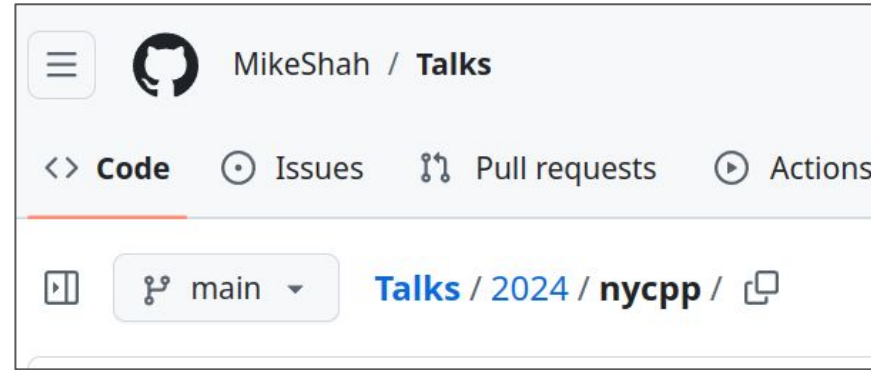


www.youtube.com/c/MikeShah

<http://tinyurl.com/mike-talks>

Code and Slides for the talk

- Code Located here:
<https://github.com/MikeShah/Talks/tree/main/2024/nycpp>
- Slides posted after talk at:
 - www.mshah.io
- Live coding the examples from this (if any) posted at:
 - www.youtube.com/c/MikeShah



Your Tour Guide for Today

Mike Shah

- **Current Role:** Teaching Faculty at **Yale University**
(Previously Teaching Faculty at Northeastern University)
 - **Teach/Research:** computer systems, graphics, geometry, game engine development, and software engineering.
- **Available for:**
 - **Contract work** in Gaming/Graphics Domains
 - e.g. tool building, plugins, code review
 - **Technical training** (virtual or onsite) in Modern C++, D, and topics in Performance or Graphics APIs
- **Fun:**
 - Guitar, running/weights, traveling, video games, and cooking are fun to talk to me about!



Web

www.mshah.io



YouTube

<https://www.youtube.com/c/MikeShah>

Non-Academic Courses

courses.mshah.io

Conference Talks

<http://tinyurl.com/mike-talks>

Your Tour Guide for Today

Mike Shah

- **Current Role**

(Previously Teaching Faculty)

- **Teach/Research** in
engine development

- **Available for**

- **Contract**

- e.g. tool building, plugins, code review

- **Technical training** (virtual or onsite) in Modern C++, D, and topics in Performance or Graphics APIs

- **Fun:**

- Guitar, running/weights, traveling, video games, and cooking are fun to talk to me about!

Thanks for spending your evening with me on a weeknight during the workweek!



Web

www.mshah.io



YouTube

<https://www.youtube.com/c/MikeShah>

Non-Academic Courses

courses.mshah.io

Conference Talks

<http://tinyurl.com/mike-talks>

Railroads and Trains

A snapshot of some trains in Paris, just outside of the French C++ User Group (FRUG) meeting location



Transport Tycoon (1/2)

- A favorite game I started playing when I was 5-6 years old,
- One of the goals of the games is to build trains that deliver goods between stations



Transport Tycoon

Note: Shout out to [OpenTTD](#) which is actively updated if you want to play for free!

Transport Tycoon (2/2)

- If you build more tracks, then you can have more trains
 - More trains means more goods delivered, and (potentially) more profit
 - The core mechanic of the game is simple to understand

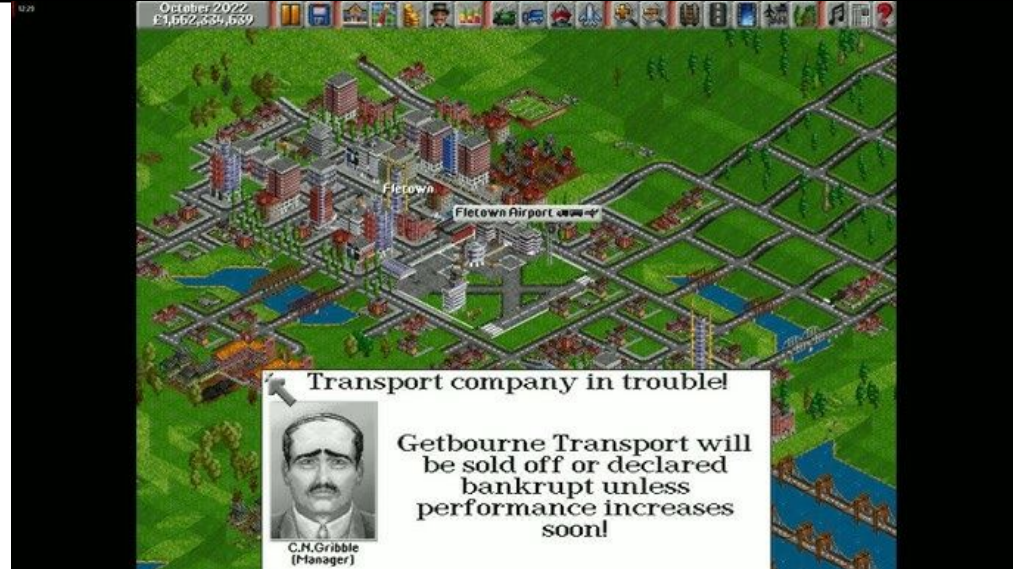


Transport Tycoon

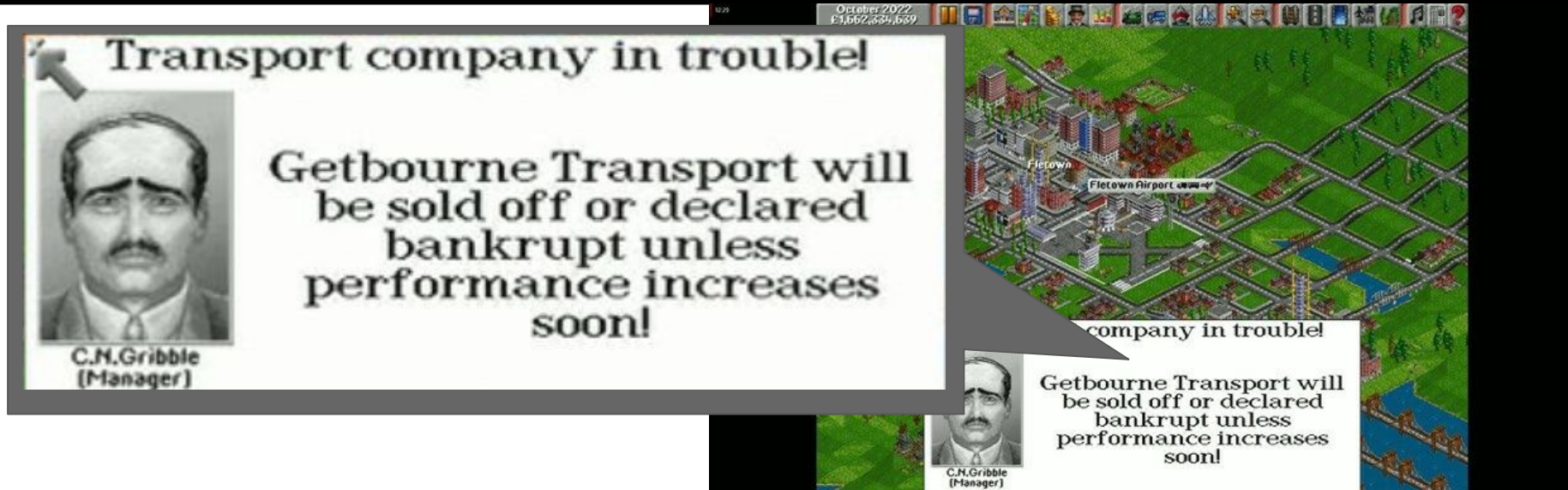
Note: Shout out to [OpenTTD](#) which is actively updated if you want to play for free!

Now of course... (1/2)

- You do have to manage your resources -- the train tracks especially can be costly
 - And you do not have unlimited money to build train tracks



Now of course... (2/2)



Oh no!

The screenshot shows a SimCity game interface. At the top, a status bar displays 'October 2022' and '£1,662,339,639'. A newspaper clipping is visible, featuring a photo of a man and the following text:

Transport company in trouble!

Getbourne Transport will be sold off or declared bankrupt unless performance increases soon!

C.H.Gribble (Manager)

The clipping is shown in two locations: a large version on the left and a smaller version on the right, which is partially overlapping a SimCity map. The map shows a city with various buildings, roads, and a river. A label 'Fletown Airport' is visible on the map.

One solution: Share the track with multiple trains (1/2)

- So a possible solution then is to share **one track** with **multiple trains**.
 - Our **station** can be shared otherwise **between multiple trains** because there are **multiple unique tracks within the station** to **deliver the goods**.



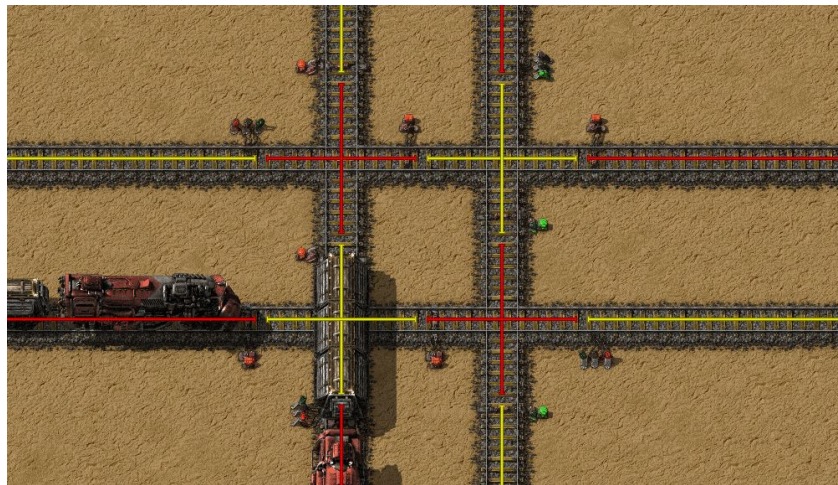
One solution: Share the track with multiple trains (2/2)

- So a possible solution then is to share **one track** with **multiple trains**.
 - Our **station** can be shared otherwise **between multiple trains** because there are **multiple unique tracks within the station** to **deliver the goods**.
- If you observe carefully, you'll notice signals exist to coordinate the trains
 - We don't want any accidents!



Coordinating (i.e. Synchronizing) trains

- Given more 'signals' we can create more intricate train networks to drop off more goods
 - We've taken care to optimize the amount of track we need to lay out as well.
 - In some cases we may add a few more tracks -- but the idea is useful.
- Of course -- we have to be a bit more careful as we add more signals, trains, and track
 - Synchronizing and coordinating all of these 'resources' takes careful thought!



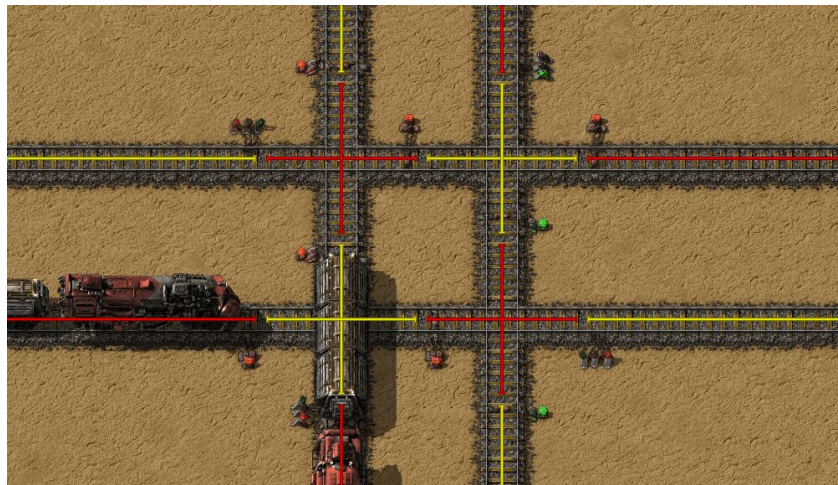
Top: Transport Tycoon

Bottom: A different game called Factorio

https://wiki.factorio.com/Tutorial:Train_signals

Into today's talk...

- Today we're going to do a tour of fundamental concurrent programming primitives
 - It's no more difficult to understand than playing a game like Transport Tycoon.
 - And in some cases we will just fundamentally want to avoid synchronization
- Okay -- let's dive into concurrency



Top: Transport Tycoon

Bottom: A different game called Factorio

https://wiki.factorio.com/Tutorial:Train_signals

Today's Talk

Abstract (1/2)

The abstract that you read and enticed you to join me here!

Talk Abstract: Our world is concurrent and often we need to write software that models our concurrent world. Luckily, the C++ Standard Template Library (STL) has with each new release provided several mechanisms for writing concurrent software. So whether you need the C++ STL concurrency library to model a concurrent problem, or otherwise utilize concurrency for performance, this talk will provide an overview of the concurrency support library. In this talk, I will give a tour with code examples of the concurrency primitives provided from C++11 to C++23. The audience will leave this talk knowing the difference between mechanisms such as thread, jthread, mutexes, semaphores, latches, and barriers. I'll also go one step further showing how to debug concurrency bugs by isolating and recording the bugs with tools like GDB and UDB. I Promise this talk in the Future will help you provide a foundation of C++ concurrency (oh yes, we'll talk about promises and futures as well!).

Abstract (2/2)

Talk Abstract: Our goal is to create a C++ standard library that models our concurrent world. The C++ Standard Library (STL) has with each version added support for concurrent software.

Whether you want to model a concurrent problem, or optimize concurrency for performance, this talk will provide an overview of the C++ concurrency support library. In this talk, I will give a tour with code examples of the concurrency primitives provided from C++11 to C++23. The audience will leave this talk knowing the difference between mechanisms such as **thread**, **jthread**, **mutexes**, **semaphores**, **latches**, and **barriers**. I'll also go one step further showing how to **debug concurrency** bugs by isolating and recording the bugs with tools like **GDB** and **UDB**. I **Promise** this talk in the **Future** will help you provide a foundation of C++ concurrency (oh yes, we'll talk about promises and futures as well!).

- Topics for today are a tour of foundational concurrency primitives
 - As mentioned, there will be lots of small code samples study after

Concurrency Motivation

- Today I'll assume you know some of the motivations of concurrency -- here's a short summary:
 - Potential gain in performance
 - Perhaps models your problem in a better/safer way
- If you want some more motivation, I have previous resources linked that provide a longer and gentle introduction.

Concurrency Mechanism - Thread

- One mechanism for achieving concurrency is a 'thread'
- A 'thread' allows us to execute two control flows at the same time
- The 'main thread' is where our program starts
 - We may then have 1 or more additional threads:
 - executing a block of code
 - executing other functions
 - And overall - sharing the same code, and the same data
 - (all while our main thread also executes.)

Mike Shah

Back to Basics: Concurrency

Parallelism vs Concurrency (programming context) (1/4)

Concurrency is often used interchangeably with parallelism--so let's separate those two terms.

1. Concurrency Definition: Multiple things can happen at once, the order matters, and sometimes tasks have to wait on shared resources.
2. Parallelism Definition: Everything happens at once, instantaneously

Top

Back to Basics: Concurrency - Mike Shah - CppCon 2021

<https://www.youtube.com/watch?v=pflC-kle4b0>

Bottom

The what and the why of concurrency | Introduction to Concurrency in Cpp

https://www.youtube.com/watch?v=Fn0xBsmact4&list=PLvv0ScY6vfd_ocTP2ZLicgoknvq50OCXM

(Aside) Better Code: Concurrency

- Now after today's talk, you can watch Sean Parent's Better Code: Concurrency talk
 - The talk **warns** about using raw synchronization primitives that I'm teaching you today
 - That's okay -- I need to show you the primitives, and then you can build safer higher level abstractions that your team can use.
 - (And that's sort of what Sean is saying too)



C++ Now Video: <https://www.youtube.com/watch?v=32f6JrQPv8c> (94 minutes)

Code::Dive Video: https://youtu.be/QIHv8pXbneI?si=YVT_Le7EBebY1Bzf&t=468 (78 minutes -- timestamp at 7:48)

Slides: <https://sean-parent.stlab.cc/presentations/2016-08-08-concurrency/2016-08-08-concurrency.pdf>

Topics Outline

- ~~Transport Tycoon~~
- `std::thread` and `std::jthread`
 - Launching and joining threads
 - Data Parallelism
- Synchronization Primitives
 - mutexes and mutex management
 - Condition Variables
 - semaphores
 - latches / barriers
- Promise and Futures
 - `async`
 - `packaged_tasks`
- Debugging concurrency
 - GDB and UDB.



Thread-Based Concurrency

Concurrency support library (C++11)

thread – jthread (C++20)

atomic – atomic_flag

atomic_ref (C++20) – memory_order

Mutual exclusion – Semaphores (C++20)

Condition variables – Futures

latch (C++20) – barrier (C++20)

Safe Reclamation (C++26)



```
#include <thread>  
std::thread
```

Concurrency support library (C++11)

thread – jthread (C++20)
atomic – atomic_flag
atomic_ref (C++20) – memory_order
Mutual exclusion – Semaphores (C++20)
Condition variables – Futures
latch (C++20) – barrier (C++20)
Safe Reclamation (C++26)

C++ Concurrency support library `std::thread`

`std::thread`

Defined in header `<thread>`

```
class thread;           (since C++11)
```

The class `thread` represents a single thread of execution . Threads allow multiple functions to execute concurrently.

Threads begin execution immediately upon construction of the associated thread object (pending any OS scheduling delays), starting at the top-level function provided as a [constructor argument](#). The return value of the top-level function is ignored and if it terminates by throwing an exception, `std::terminate` is called. The top-level function may communicate its return value or an exception to the caller via `std::promise` or by modifying shared variables (which may require synchronization, see `std::mutex` and `std::atomic`).

`std::thread` objects may also be in the state that does not represent any thread (after default construction, move from, [detach](#), or [join](#)), and a thread of execution may not be associated with any thread objects (after [detach](#)).

No two `std::thread` objects may represent the same thread of execution; `std::thread` is not [CopyConstructible](#) or [CopyAssignable](#), although it is [MoveConstructible](#) and [MoveAssignable](#).

Thread Example - Launching a thread (1/2)

- `#include <thread>`
 - `std::thread`
- Launching a `std::thread` is this idea of '[fork-join parallelism](#)' and with threads our memory is shared
 - Note: There's a key point of 'is the problem large enough' that it's worth separating out the work that we'll want to touch on later.

```
1 // @file thread1.cpp
2 // g++ -g -Wall -std=c++23 thread1.cpp -o prog -lpthread
3 #include <print>
4 #include <thread> // Include the thread library
5
6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::println("Hello from our thread!");
9     std::println("Argument passed in: {}", x);
10 }
11
12 int main() {
13     // Create a new thread and pass one parameter
14     std::thread myThread(&test, 100);
15     // Join with the main thread, which is the same as
16     // saying "hey, main thread--wait until myThread
17     // finishes before executing further."
18     myThread.join();
19
20     // Continue executing the main thread
21     std::println("Hello from the main thread!");
22
23     return 0;
24 }
```



```
mike@mike-MS-7B17:nycpp$ g++ -g -Wall -std=c++23 thread1.cpp -o prog -lpthread
mike@mike-MS-7B17:nycpp$ ./prog
Hello from our thread!
Argument passed in: 100
Hello from the main thread!
```

- `#include <thread>`
 - `std::thread`
- Launching a `std::thread` is this idea of 'fork-join parallelism' and with threads our memory is shared
 - Note: There's a key point of 'is the problem large enough' that it's worth separating out the work that we'll want to touch on later.

```
2 // g++ -g -Wall -std=c++23 thread1.cpp -o prog -lpthread
3 #include <print>
4 #include <thread> // Include the thread library
5
6 // Test function which we'll launch
7 void test(int x) {
8     std::println("Hello from our thread!");
9     std::println("Argument passed in: " + std::to_string(x));
10 }
11
12 int main() {
13     // Create a new thread and pass one parameter
14     std::thread myThread(&test, 100);
15     // Join with the main thread, which is the same as
16     // saying "hey, main thread--wait until myThread
17     // finishes before executing further."
18     myThread.join();
19
20     // Continue executing the main thread
21     std::println("Hello from the main thread!");
22
23     return 0;
24 }
```

Note: You may need to explicitly link in the pthread library on linux.

Visual execution of “Hello Thread” (1/13)

```
1 // @file thread1.cpp
2 // g++ -g -Wall -std=c++23 thread1.cpp -o prog -lpthread
3 #include <print>
4 #include <thread>    // Include the thread library
5
6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::println("Hello from our thread!");
9     std::println("Argument passed in: {}", x);
10 }
11
12 int main() {
13     // Create a new thread and pass one parameter
14     std::thread myThread(&test, 100);
15     // Join with the main thread, which is the same as
16     // saying "hey, main thread--wait until myThread
17     //         finishes before executing further."
18     myThread.join();
19
20     // Continue executing the main thread
21     std::println("Hello from the main thread!");
22
23     return 0;
24 }
```

Visual execution of “Hello Thread” (2/13)

Main Thread

main() function where all C++ programs start.

We have 1 thread in our program (the main thread)

```
1 // @file thread1.cpp
2 // g++ -g -Wall -std=c++23 thread1.cpp -o prog -lpthread
3 #include <print>
4 #include <thread> // Include the thread library
5
6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::println("Hello from our thread!");
9     std::println("Argument passed in: {}", x);
10 }
11
12 int main() {
13     // Create a new thread and pass one parameter
14     std::thread myThread(&test, 100);
15     // Join with the main thread, which is the same as
16     // saying "hey, main thread--wait until myThread
17     // finishes before executing further."
18     myThread.join();
19
20     // Continue executing the main thread
21     std::println("Hello from the main thread!");
22
23     return 0;
24 }
```

Visual execution of “Hello Thread” (3/13)

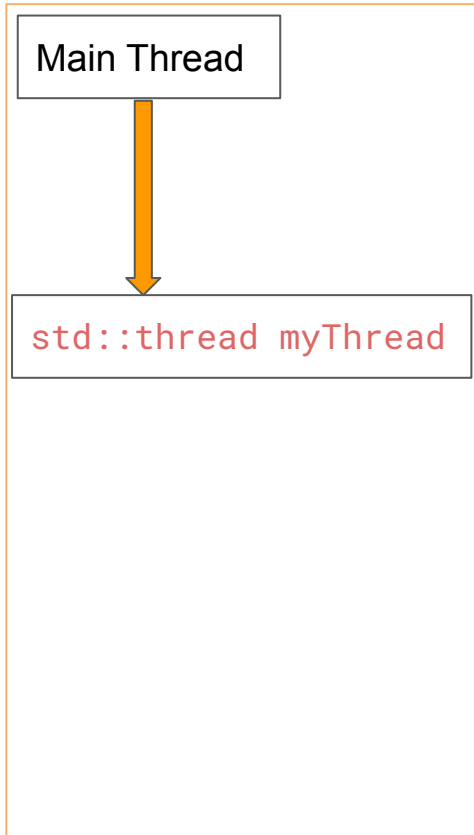
Main Thread



We begin constructing
`std::thread`
`myThread`

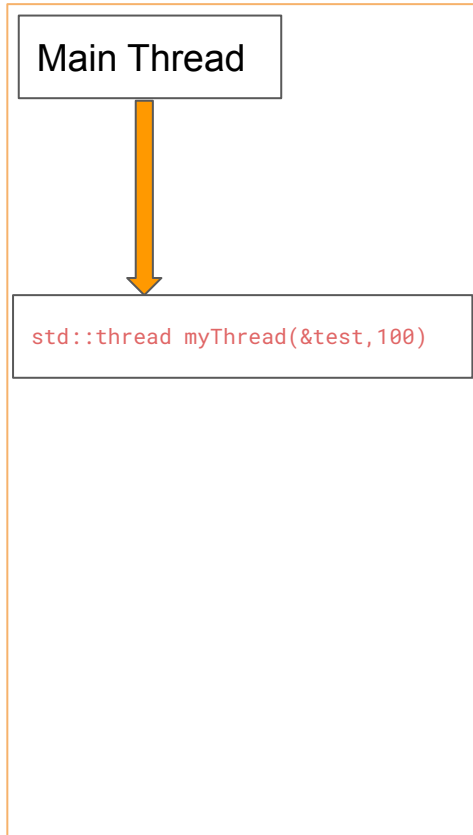
```
1 // @file thread1.cpp
2 // g++ -g -Wall -std=c++23 thread1.cpp -o prog -lpthread
3 #include <print>
4 #include <thread> // Include the thread library
5
6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::println("Hello from our thread!");
9     std::println("Argument passed in: {}", x);
10 }
11
12 int main() {
13     // Create a new thread and pass one parameter
14     std::thread myThread(&test, 100);
15     // Join with the main thread, which is the same as
16     // saying "hey, main thread--wait until myThread
17     // finishes before executing further."
18     myThread.join();
19
20     // Continue executing the main thread
21     std::println("Hello from the main thread!");
22
23     return 0;
24 }
```

Visual execution of “Hello Thread” (4/13)



```
1 // @file thread1.cpp
2 // g++ -g -Wall -std=c++23 thread1.cpp -o prog -lpthread
3 #include <print>
4 #include <thread> // Include the thread library
5
6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::println("Hello from our thread!");
9     std::println("Argument passed in: {}", x);
10 }
11
12 int main() {
13     // Create a new thread and pass one parameter
14     std::thread myThread(&test, 100);
15     // Join with the main thread, which is the same as
16     // saying "hey, main thread--wait until myThread
17     // finishes before executing further."
18     myThread.join();
19
20     // Continue executing the main thread
21     std::println("Hello from the main thread!");
22
23     return 0;
24 }
```

Visual execution of “Hello Thread” (5/13)



Our new thread will begin executing its logical control flow from the ‘test’ function. *separately* from main()

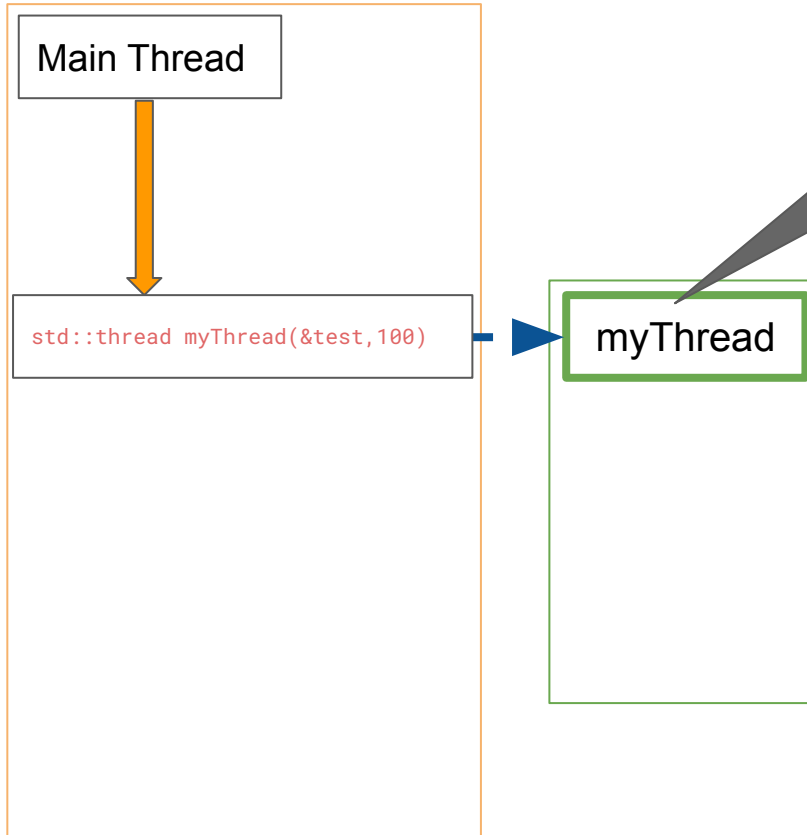
The thread will start executing immediately on construction

(Remember, threads shares code and the heap)

```
1 // @file thread1.cpp
2 // g++ -g -Wall -std=c++23 thread1.cpp -o prog -lpthread
3 #include <print>
4 #include <thread> // Include the thread library
5
6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::println("Hello from our thread!");
9     std::println("Argument passed in: {}", x);
10 }
11
12 int main() {
13     // Create a new thread and pass one parameter
14     std::thread myThread(&test, 100);
15     // Join with the main thread, which is the same as
16     // saying "hey, main thread--wait until myThread
17     // finishes before executing further."
18     myThread.join();
19
20     // Continue executing the main thread
21     std::println("Hello from the main thread!");
22
23     return 0;
24 }
```


Visual execution of “Hello Thread” (6/13)

So now we have two
“threads” executing

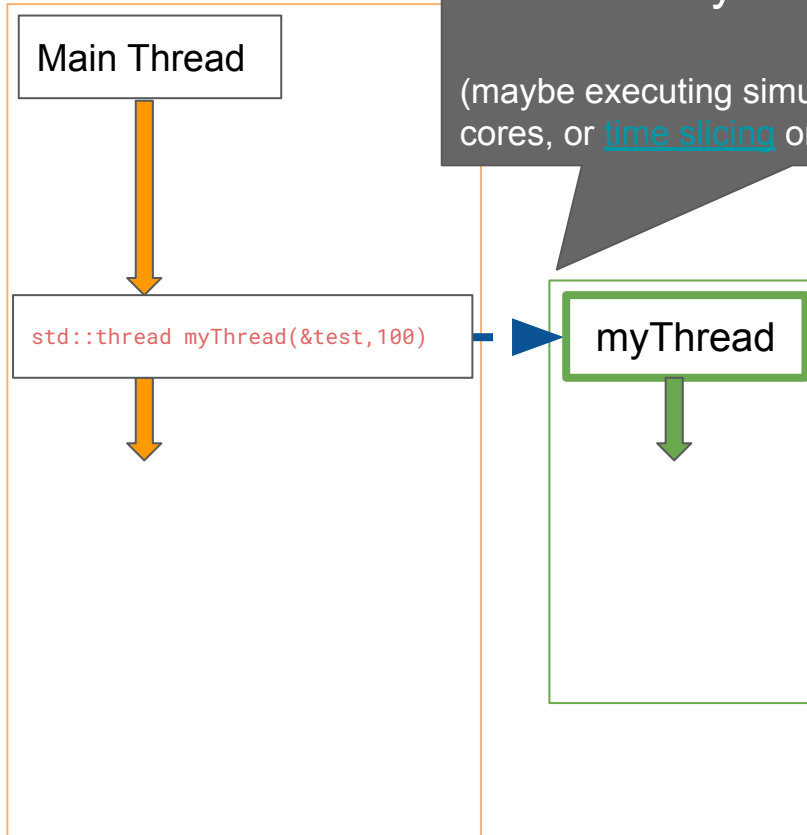


```
ad1.cpp -o prog -lpthread
4 #include <thread> // Include the thread library
5
6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::println("Hello from our thread!");
9     std::println("Argument passed in: {}", x);
10 }
11
12 int main() {
13     // Create a new thread and pass one parameter
14     std::thread myThread(&test, 100);
15     // Join with the main thread, which is the same as
16     // saying "hey, main thread--wait until myThread
17     // finishes before executing further."
18     myThread.join();
19
20     // Continue executing the main thread
21     std::println("Hello from the main thread!");
22
23     return 0;
24 }
```


Visual execution of “Hello Thread” (7/13)

Both threads of execution are concurrently alive!

(maybe executing simultaneously on separate cores, or time slicing on the same one)

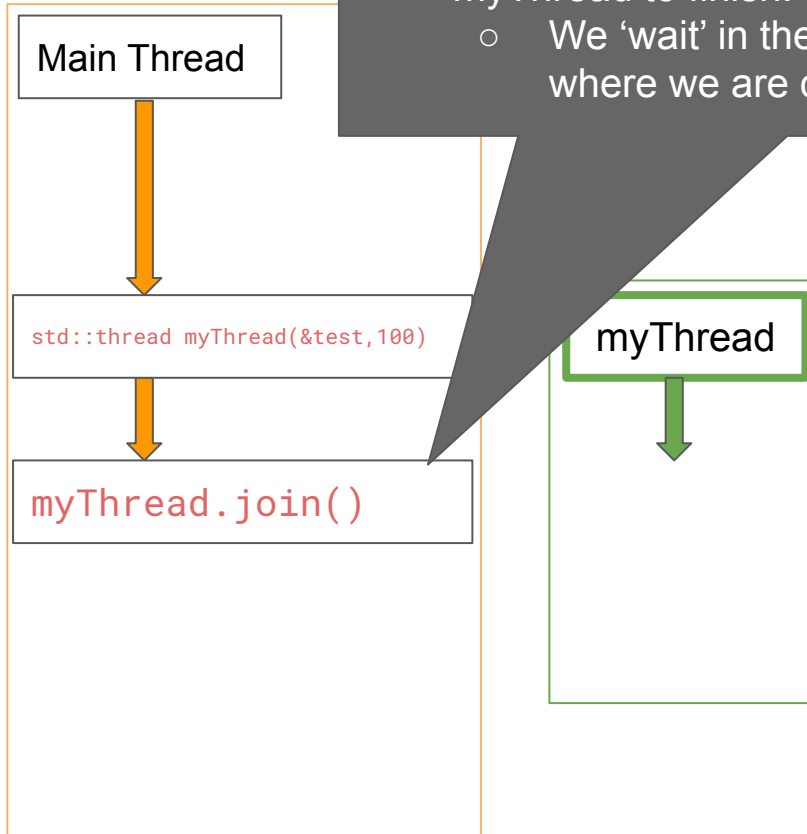


```
thread1.cpp -o prog -lpthread
#include the thread library

6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::println("Hello from our thread!");
9     std::println("Argument passed in: {}", x);
10 }
11
12 int main() {
13     // Create a new thread and pass one parameter
14     std::thread myThread(&test, 100);
15     // Join with the main thread, which is the same as
16     // saying "hey, main thread--wait until myThread
17     // finishes before executing further."
18     myThread.join();
19
20     // Continue executing the main thread
21     std::println("Hello from the main thread!");
22
23     return 0;
24 }
```

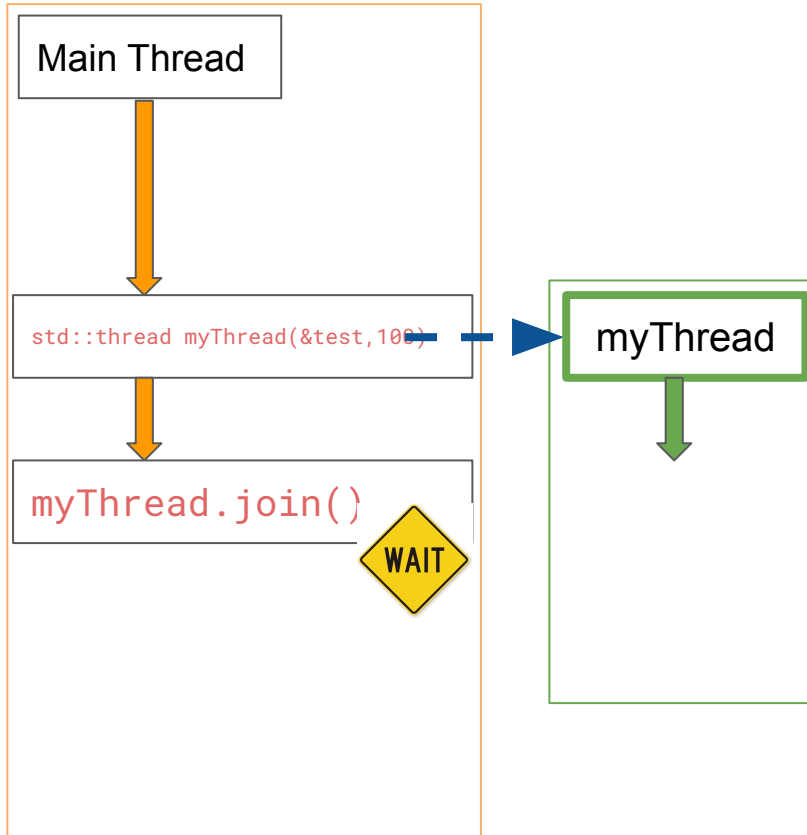
Visual execution

- We just happen to execute the next line in main thread
- `myThread.join()` tells the 'Main Thread' to wait on `myThread` to finish.
 - We 'wait' in the main thread, because this is where we are calling `join` from



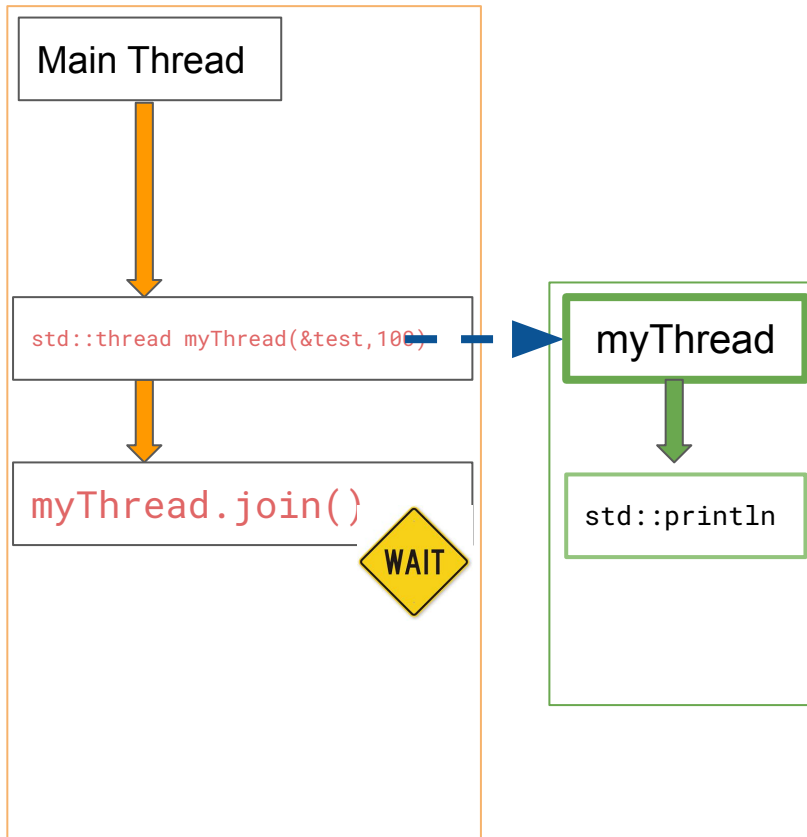
```
5 // Compile with g++ -std=c++11 -pthread -o prog -lpthread
6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::println("Hello from our thread!");
9     std::println("Argument passed in: {}", x);
10 }
11
12 int main() {
13     // Create a new thread and pass one parameter
14     std::thread myThread(&test, 100);
15     // Join with the main thread, which is the same as
16     // saying "hey, main thread--wait until myThread
17     // finishes before executing further."
18     myThread.join();
19
20     // Continue executing the main thread
21     std::println("Hello from the main thread!");
22
23     return 0;
24 }
```

Visual execution of “Hello Thread” (9/13)



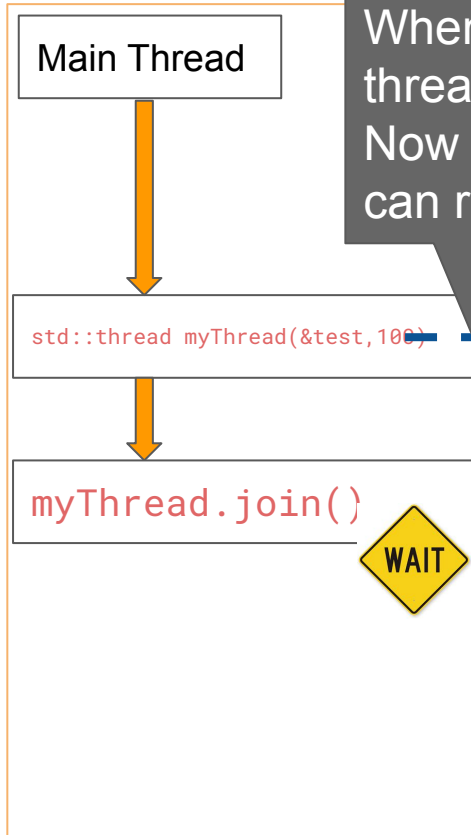
```
1 // @file thread1.cpp
2 // g++ -g -Wall -std=c++23 thread1.cpp -o prog -lpthread
3 #include <print>
4 #include <thread> // Include the thread library
5
6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::println("Hello from our thread!");
9     std::println("Argument passed in: {}", x);
10 }
11
12 int main() {
13     // Create a new thread and pass one parameter
14     std::thread myThread(&test, 100);
15     // Join with the main thread, which is the same as
16     // saying "hey, main thread--wait until myThread
17     // finishes before executing further."
18     myThread.join();
19
20     // Continue executing the main thread
21     std::println("Hello from the main thread!");
22
23     return 0;
24 }
```

Visual execution of “Hello Thread” (10/13)

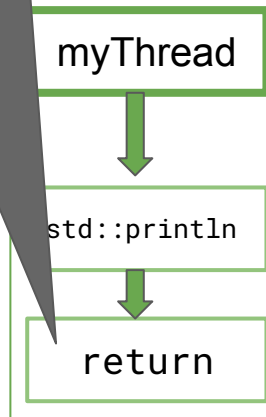


```
1 // @file thread1.cpp
2 // g++ -g -Wall -std=c++23 thread1.cpp -o prog -lpthread
3 #include <print>
4 #include <thread> // Include the thread library
5
6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::println("Hello from our thread!");
9     std::println("Argument passed in: {}", x);
10 }
11
12 int main() {
13     // Create a new thread and pass one parameter
14     std::thread myThread(&test, 100);
15     // Join with the main thread, which is the same as
16     // saying "hey, main thread--wait until myThread
17     // finishes before executing further."
18     myThread.join();
19
20     // Continue executing the main thread
21     std::println("Hello from the main thread!");
22
23     return 0;
24 }
```

Visual execution of “Hello Thread” (11/13)



When we return, our thread terminates. Now our 'main' thread can resume

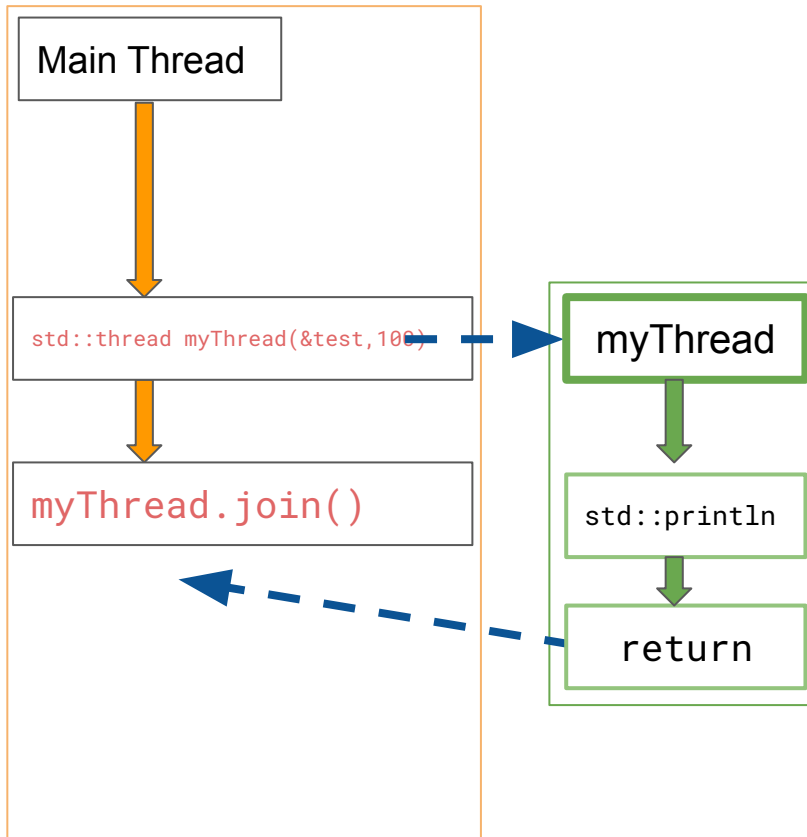


```
// @file thread1.cpp
// g++ -g -Wall -std=c++23 thread1.cpp -o prog -lpthread
#include <print>
#include <thread> // Include the thread library

// Test function which we'll launch threads from
void test(int x) {
    std::println("Hello from our thread!");
    std::println("Argument passed in: {}", x);
}

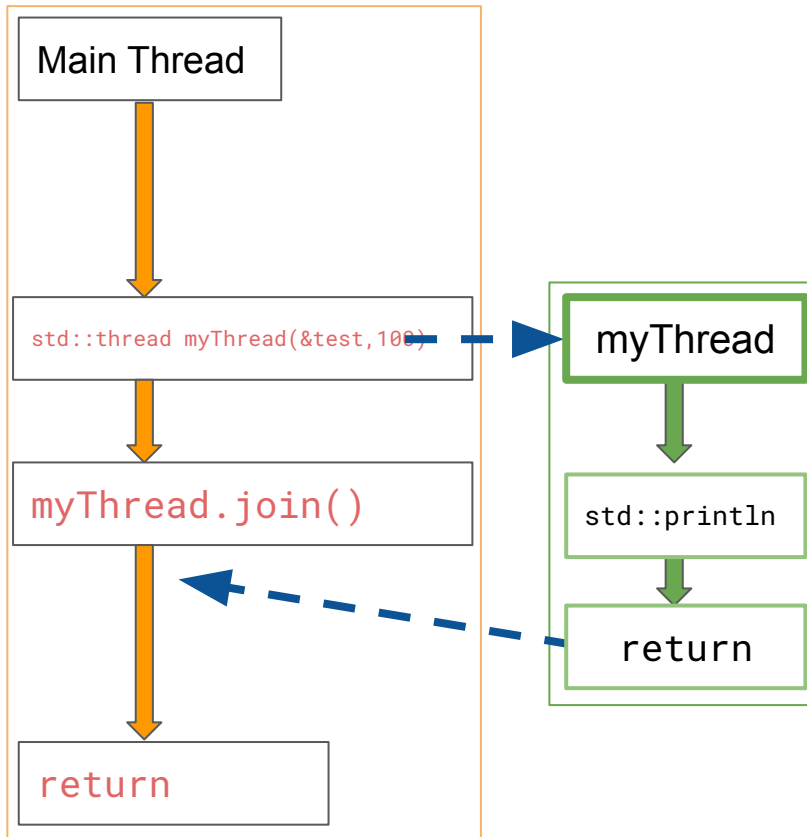
10 }
11
12 int main() {
13     // Create a new thread and pass one parameter
14     std::thread myThread(&test, 100);
15     // Join with the main thread, which is the same as
16     // saying "hey, main thread--wait until myThread
17     // finishes before executing further."
18     myThread.join();
19
20     // Continue executing the main thread
21     std::println("Hello from the main thread!");
22
23     return 0;
24 }
```


Visual execution of “Hello Thread” (12/13)



```
1 // @file thread1.cpp
2 // g++ -g -Wall -std=c++23 thread1.cpp -o prog -lpthread
3 #include <print>
4 #include <thread> // Include the thread library
5
6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::println("Hello from our thread!");
9     std::println("Argument passed in: {}", x);
10 }
11
12 int main() {
13     // Create a new thread and pass one parameter
14     std::thread myThread(&test, 100);
15     // Join with the main thread, which is the same as
16     // saying "hey, main thread--wait until myThread
17     // finishes before executing further."
18     myThread.join();
19
20     // Continue executing the main thread
21     std::println("Hello from the main thread!");
22
23     return 0;
24 }
```

Visual execution of “Hello Thread” (13/13)




```
1 // @file thread1.cpp
2 // g++ -g -Wall -std=c++23 thread1.cpp -o prog -lpthread
3 #include <print>
4 #include <thread> // Include the thread library
5
6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::println("Hello from our thread!");
9     std::println("Argument passed in: {}", x);
10 }
11
12 int main() {
13     // Create a new thread and pass one parameter
14     std::thread myThread(&test, 100);
15     // Join with the main thread, which is the same as
16     // saying "hey, main thread--wait until myThread
17     // finishes before executing further."
18     myThread.join();
19
20     // Continue executing the main thread
21     std::println("Hello from the main thread!");
22
23     return 0;
24 }
```

We can also move our function into a lambda

- `std::thread` takes any [callable](#) as the parameter--so:
 - lambdas, functions, functors, [std::function](#), [std::bind](#), etc. are all fine to call from thread

```
1 // @file thread1.cpp
2 // g++ -g -Wall -std=c++23 thread1.cpp -o prog -lpthread
3 #include <print>
4 #include <thread> // Include the thread library
5
6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::println("Hello from our thread!");
9     std::println("Argument passed in: {}", x);
10 }
11
12 int main() {
13     // Create a new thread and pass one parameter
14     std::thread myThread(&test, 100);
15     // Join with the main thread, which is the same as
16     // saying "hey, main thread--wait until myThread
17     // finishes before executing further."
18     myThread.join();
19
20     // Continue executing the main thread
21     std::println("Hello from the main thread!");
22
23     return 0;
24 }
```



Same example as before -- but with a lambda!

- Same logic as before, but instead of a function, I have a lambda with 1 parameter and a void return value
- Note:
 - If you want to pass a reference or some heap memory into a thread use `std::cref` or `std::ref`
 - [Tutorial](#)

```
1 // @file thread2.cpp
2 // g++ -g -Wall -std=c++23 thread2.cpp -o prog -lpthread
3 #include <print>
4 #include <thread> // Include the thread library
5
6 int main() {
7
8     // This time create a lambda function
9     auto lambda = [](int x){
10         std::println("Hello from our thread!");
11         std::println("Argument passed in: {}", x);
12     };
13
14     // Create a new thread and pass one parameter
15     std::thread myThread(lambda, 100);
16     // Join with the main thread, which is the same as
17     // saying "hey, main thread--wait until myThread
18     // finishes before executing further."
19     myThread.join();
20
21     // Continue executing the main thread
22     std::println("Hello from the main thread!");
23
24     return 0;
25 }
```

Now how about if we wanted 10 threads (0/5)

- Let's create a `std::vector<std::thread>`
 - Then we'll launch 10 threads from a loop
- It's important however, that we also join each of the threads!
 - Why?
 - Because our main thread of execution could complete before all `std::thread`'s otherwise complete their task.

```
1 // @file thread3.cpp
2 // g++ -std=c++17 thread3.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5 #include <vector>
6
7 int main() {
8
9     // This time create a lambda function
10    auto lambda = [](int x){
11        std::cout << "thread.get_id:" << std::this_thread::get_id() << std::endl;
12        std::cout << "Argument passed in:" << x << std::endl;
13    };
14
15    std::vector<std::thread> threads;
16    // Create a collection of threads
17    for(int i=0; i < 10; i++){
18        threads.push_back(std::thread(lambda,i));
19        threads[i].join();
20    }
21
22    // Continue executing the main thread
23    std::cout << "Hello from the main thread!" << std::endl;
24
25    return 0;
26 }
```


Now how about if we wanted 10 threads (1/5)

- So here we create each of our threads and join them
- Note:
 - `std::thread` is not copyable (kind of important if you think about it) -- so they are moved into the vector.
 - Using a `std::vector` is a common idiom to 'hold onto' threads.

```
1 // @file thread3.cpp
2 // g++ -std=c++17 thread3.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5 #include <vector>
6
7 int main() {
8
9     // This time create a lambda function
10    auto lambda = [](int x){
11        std::cout << "thread.get_id:" << std::this_thread::get_id() << std::endl;
12        std::cout << "Argument passed in:" << x << std::endl;
13    };
14
15    std::vector<std::thread> threads;
16    // Create a collection of threads
17    for(int i=0; i < 10; i++){
18        threads.push_back(std::thread(lambda,i));
19        threads[i].join();
20    }
21
22    // Continue executing the main thread
23    std::cout << "Hello from the main thread!" << std::endl;
24
25    return 0;
26 }
```

Now how about if we wanted 10 threads (2/5)

- So here we create each of our threads and join

```
mike:concurrency$ g++ -std=c++17 thread3.cpp -o prog -lpthread
mike:concurrency$ ./prog
thread.get_id:140658209871616
Argument passed in:0
thread.get_id:140658209871616
Argument passed in:1
thread.get_id:140658209871616
Argument passed in:2
thread.get_id:140658209871616
Argument passed in:3
thread.get_id:140658209871616
Argument passed in:4
thread.get_id:140658209871616
Argument passed in:5
thread.get_id:140658209871616
Argument passed in:6
thread.get_id:140658209871616
Argument passed in:7
thread.get_id:140658209871616
Argument passed in:8
thread.get_id:140658209871616
Argument passed in:9
Hello from the main thread!
```

```
1 // @file thread3.cpp
2 // g++ -std=c++17 thread3.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5 #include <vector>
6
7 int main() {
8
9     // This time create a lambda function
10    auto lambda = [](int x){
11        std::cout << "thread.get_id:" << std::this_thread::get_id() << std::endl;
12        std::cout << "Argument passed in:" << x << std::endl;
13    };
14
15    std::vector<std::thread> threads;
16    // Create a collection of threads
17    for(int i=0; i < 10; i++){
18        threads.push_back(std::thread(lambda,i));
19        threads[i].join();
20    }
21
22    // Continue executing the main thread
23    std::cout << "Hello from the main thread!" << std::endl;
24
25    return 0;
26 }
```

Now how about if we

- So here we create each of our threads and join

- The result seems a little strange...anyone see the problem?
 - (Hint: Look at arguments passed in output)

```
mike:concurrency$ g++ -std=c++17 thread3.cpp -pthread -lpthread
mike:concurrency$ ./prog
thread.get_id:140658209871616
Argument passed in:0
thread.get_id:140658209871616
Argument passed in:1
thread.get_id:140658209871616
Argument passed in:2
thread.get_id:140658209871616
Argument passed in:3
thread.get_id:140658209871616
Argument passed in:4
thread.get_id:140658209871616
Argument passed in:5
thread.get_id:140658209871616
Argument passed in:6
thread.get_id:140658209871616
Argument passed in:7
thread.get_id:140658209871616
Argument passed in:8
thread.get_id:140658209871616
Argument passed in:9
Hello from the main thread!
```

```
8
9 // This time create a lambda function
10 auto lambda = [](int x){
11     std::cout << "thread.get_id:" << std::this_thread::get_id() << std::endl;
12     std::cout << "Argument passed in:" << x << std::endl;
13 };
14
15 std::vector<std::thread> threads;
16 // Create a collection of threads
17 for(int i=0; i < 10; i++){
18     threads.push_back(std::thread(lambda,i));
19     threads[i].join();
20 }
21
22 // Continue executing the main thread
23 std::cout << "Hello from the main thread!" << std::endl;
24
25 return 0;
26 }
```

Now how about if

- So here we create
of our threads and

- By joining our threads immediately after launching our code, we've effectively made our program sequential (i.e. no performance gain)
- This is a form of over-synchronization
 - We have **serialized** something that we want to happen concurrently

```
mike:concurrency$ g++ -std=c++17 thread3.cpp
mike:concurrency$ ./prog
thread.get_id:140658209871616
Argument passed in:0
thread.get_id:140658209871616
Argument passed in:1
thread.get_id:140658209871616
Argument passed in:2
thread.get_id:140658209871616
Argument passed in:3
thread.get_id:140658209871616
Argument passed in:4
thread.get_id:140658209871616
Argument passed in:5
thread.get_id:140658209871616
Argument passed in:6
thread.get_id:140658209871616
Argument passed in:7
thread.get_id:140658209871616
Argument passed in:8
thread.get_id:140658209871616
Argument passed in:9
Hello from the main thread!
```

```
9
10 // Create a lambda function
11 auto lambda = [&x]{
12     std::cout << "Thread.get_id:" << std::this_thread::get_id() << std::endl;
13     std::cout << "Argument passed in:" << x << std::endl;
14 };
15 std::vector<std::thread> threads;
16 // Create a collection of threads
17 for(int i=0; i < 10; i++){
18     threads.push_back(std::thread(lambda,i));
19     threads[i].join();
20 }
21
22 // Continue executing the main thread
23 std::cout << "Hello from the main thread!" << std::endl;
24
25 return 0;
26 }
```

Let's try debugging our over-synchronization
error

Debugging the initial problem

- Debugging **concurrent programs is not always easy**
- I'm going to use live-recorder here
 - <https://undo.io/udb-free-trial/>
- Why?
 - It allows me to 'capture' one specific execution of my concurrent program run
 - This can be handy later on if we don't necessarily have deterministic execution
 - In this specific case of course, we just so happen to :)

Potential debugging workflow

1. First compile with debug symbols:

a. `g++-11 -g -std=c++23 ../thread3.cpp -o prog -lpthread`

2. Then run your program as normal

a. `sudo ~/Downloads/Undo-Suite-x86-8.0.0/live-record --recording-file recording.undo ./prog`

i. Note: Need sudo permissions to save recording

3. Then replay

a. `sudo ~/Downloads/Undo-Suite-x86-8.0.0/udb recording.undo`

b. Can debug as normal

i. e.g.

```
1. b main          // set a breakpoint at main
2. next            // step through code
3. info threads    // see how many threads are active
```

Debug / Live Recorder / Replay (Backup Animation)

```
1 // g++11 thread3.cpp
2 // g++-11 -g -std=c++23 thread3.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread>    // Include the thread library
5 #include <vector>
6
7 int main() {
8
9     // This time create a lambda function
10    "thread3.cpp" 29L, 716B written
```

2,13

Top

```
mike@system76-pc:~/Talks/2024/nycpp/undo$ g++-11 -g -std=c++23 ../thread3.cpp -o prog -lpthread
```

Debug / Live Recorder / Replay

Something about this just makes me truly happy :)

I can also store these replays too -- I gave some to students before and told them to find the bug.

```
1 // g++11 thread3.cpp
2 // g++-11 -g -std=c++23 thread3.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5 #include <vector>
6
7 int main() {
8
9     // This time create a lambda function
10    "thread3.cpp" 29L, 716B written
```

```
mike@system76-pc:~/Talks/2024/nycpp/undo$ g++-11 -g -std=c++23 ../thread3.cpp -o prog -lpthread
```

(Aside) Potential debugging workflow

1. Review:

- a. `g++-11 -g -std=c++23 ../thread3.cpp -o prog -lpthread`
- b. `sudo ~/Downloads/Undo-Suite-x86-8.0.0/live-record --recording-file recording.undo ./prog`
- c. `sudo ~/Downloads/Undo-Suite-x86-8.0.0/udb recording.undo`

2. Commands to try otherwise

- a. `layout src (tui mode)`
- b. `b main`
- c. `next`
- d. `reverse-next`

Now how about if we want

- So here we create each of our threads and join

```
mike:concurrency$ g++ -std=c++17 thread3_fix.cpp -o prog -lpthread
mike:concurrency$ ./prog
thread.get_id:139995667298048
Argument passed in:0
thread.get_id:139995507902208
Argument passed in:3
thread.get_id:thread.get_id:139995642119936
Argument passed in:4
thread.get_id:139995633727232
Argument passed in:5
139995650512640
Argument passed in:2
thread.get_id:139995658905344
Argument passed in:1
thread.get_id:139995608549120
Argument passed in:8
thread.get_id:139995532752640
Argument passed in:9
thread.get_id:139995616941824
Argument passed in:7
thread.get_id:139995625334528
Argument passed in:6
Hello from the main thread!
```

Here's the fix -- move 'join' to 'unblock' (i.e. avoid waiting) while spawning new threads

Observe the new output, the thread execution is out of order now (which is expected when 10 threads are simultaneously executed, the threads are scheduled according to OS)

```
1 //
2 //
3 #
4 #
5 #
6
7 int main() {
8
9     // This time we use a lambda function
10    auto lambda = [&x](){
11        std::cout << "Thread ID: " << std::this_thread::get_id() << std::endl;
12        std::cout << "Argument passed in: " << x << std::endl;
13    };
14
15    std::vector<std::thread> threads;
16    // Create a collection of threads
17    for(int i=0; i < 10; i++){
18        threads.push_back(std::thread(lambda,i));
19    }
20    // Join all of our threads here--
21    // one or more may have launched, but we'll have
22    // to wait in main until ALL threads finish.
23    for(int i=0; i < 10; i++){
24        threads[i].join();
25    }
26
27    // Continue executing the main thread
28    std::cout << "Hello from the main thread!" << std::endl;
29
30    return 0;
31 }
```

Now how

- So again -- remember what 'join' does
 - The calling thread is blocked, until all of threads[i] are complete at line 24

- So here we create
of our threads and join

```
mike:concurrency$ g++ -std=c++17 thread3_fix.cpp -o prog
mike:concurrency$ ./prog
thread.get_id:139995667298048
Argument passed in:0
thread.get_id:139995507902208
Argument passed in:3
thread.get_id:thread.get_id:139995642119936
Argument passed in:4
thread.get_id:13999563727232
Argument passed in:5
139995650512640
Argument passed in:2
thread.get_id:139995658905344
Argument passed in:1
thread.get_id:139995608549120
Argument passed in:8
thread.get_id:139995532752640
Argument passed in:9
thread.get_id:139995616941824
Argument passed in:7
thread.get_id:139995625334528
Argument passed in:6
Hello from the main thread!
```

```
file thread3_fix.cpp
++ -std=c++17 thread3_fix.cpp -o prog -lpthread
#include <iostream>
#include <thread> // Include the thread library
#include <vector>

int main() {
    // This time create a lambda function
    auto lambda = [](int x){
        std::cout << "thread.get_id:" << std::this_thread::get_id() << std::endl;
        std::cout << "Argument passed in:" << x << std::endl;
    };

    // Create a vector of threads
    std::vector<std::thread> threads;

    // Create a collection of threads
    for(int i=0; i < 10; i++){
        threads.push_back(std::thread(lambda,i));
    }

    // Join all of our threads here--
    // one or more may have launched, but we'll have
    // to wait in main until ALL threads finish.
    for(int i=0; i < 10; i++){
        threads[i].join();
    }

    // Continue executing the main thread
    std::cout << "Hello from the main thread!" << std::endl;

    return 0;
}
```



```
#include <thread>
std::jthread
```

Concurrency support library (C++11)

thread – **jthread** (C++20)

atomic – atomic_flag

atomic_ref (C++20) – memory_order

Mutual exclusion – Semaphores (C++20)

Condition variables – Futures

latch (C++20) – barrier (C++20)

Safe Reclamation (C++26)

C++ 20 - std::jthread (1/2)

- std::jthread is similar to std::thread, but an automatic [request](#) to join on destruction is made using RAII
 - Note: You can still 'join' manually if you want more explicit control
 - std::jthread helps ensure we do not forget to join otherwise!

```
1 // @file jthread.cpp
2 // g++ -std=c++23 jthread.cpp -o prog -lpthread
3 #include <print>
4 #include <vector>
5 #include <thread> // Include the thread library
6
7 int main() {
8     // This time create a lambda function
9     auto lambda = [](int x){
10         size_t tid = std::hash<std::thread::id>{}(std::this_thread::get_id());
11         std::println("thread.get_id:{}", tid);
12         std::println("Argument passed in:{}", x);
13     };
14
15     // Note: We use a 'jthread' this time.
16     //       Automatically rejoins on destruction.
17     std::vector<std::jthread> threads;
18     // Create a collection of threads
19     for(int i=0; i < 10; i++){
20         threads.push_back(std::jthread(lambda,i));
21     }
22
23     // Continue executing the main thread
24     std::println("Hello from the main thread!");
25
26     return 0;
27 }
```

C++ 20 - std::jthread (2/2)

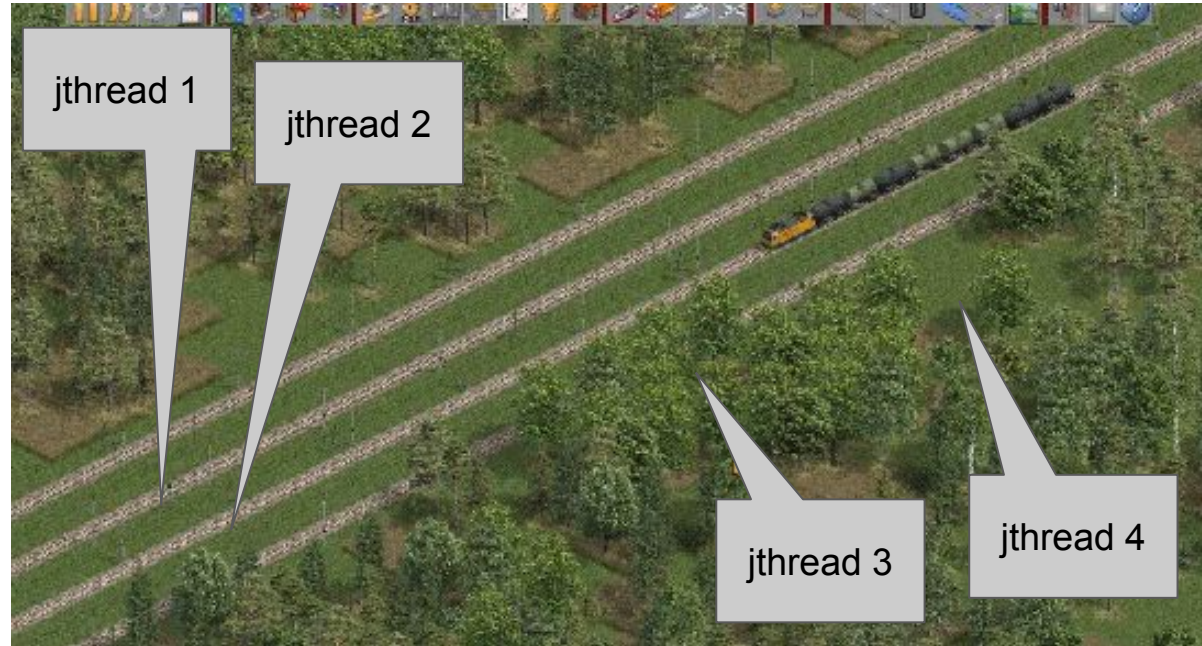
```
mike@mike-MS-7B17:nycpp$ g++ -std=c++23 jthread.cpp -o prog -lpthread && ./prog
thread.get_id:12461861030574371588
Argument passed in:0
thread.get_id:10223430736409425194
thread.get_id:9711323981635490620
Argument passed in:1
Argument passed in:2
thread.get_id:13201395818657114585
Argument passed in:4
thread.get_id:13903581504720946448
Argument passed in:3
thread.get_id:6465927851334123473
Argument passed in:5
thread.get_id:13123628352735147900
Argument passed in:6
Hello from the main thread!
thread.get_id:12826171936956292179
Argument passed in:7
thread.get_id:2071190819905647772
Argument passed in:8
thread.get_id:1575587564839372336
Argument passed in:9
```

```
1 // @file jthread.cpp
2 // g++ -std=c++23 jthread.cpp -o prog -lpthread
3 #include <print>
4 #include <vector>
5 #include <thread> // Include the thread library
6
7 int main() {
8     // This time create a lambda function
9     auto lambda = [](int x){
10         size_t tid = std::hash<std::thread::id>{}(std::this_thread::get_id());
11         std::println("thread.get_id:{}", tid);
12         std::println("Argument passed in:{}", x);
13     };
14
15     // Note: We use a 'jthread' this time.
16     //       Automatically rejoins on destruction.
17     std::vector<std::jthread> threads;
18     // Create a collection of threads
19     for(int i=0; i < 10; i++){
20         threads.push_back(std::jthread(lambda,i));
21     }
22
23     // Continue executing the main thread
24     std::println("Hello from the main thread!");
25
26     return 0;
27 }
```


Launching threads was fun -- let's launch 1000s of threads!

Teams of threads

Data Parallelism



The background of the slide is a repeating pattern of an aerial photograph of a forest. A dirt road runs diagonally from the bottom-left to the top-right. A train, composed of many dark-colored cars and a yellow locomotive, is traveling along this road. The forest is dense with green trees.

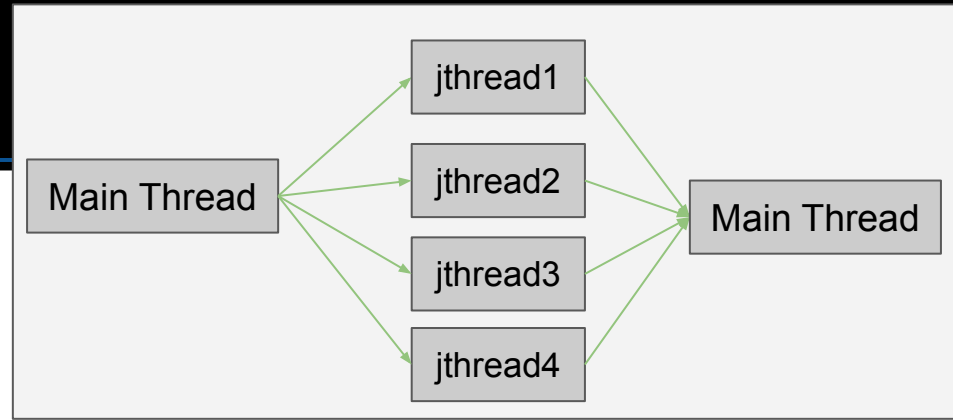
Launching threads was fun -- let's launch 1000s of threads!

Teams of
threads

Data Parallelism

Thread Parallelism

- We're in a pretty good spot in our introduction to threads
 - Each `std::thread` or `std::jthread` cherrily does some independent work
- This is effectively like each train executing on its own rail as we started out
- But what if we need to compute some final result?
 - We launch many threads -- and have them work together as a **team of threads**
 - (next slide)



Thread Team (0/9)

- Let's do a more interesting problem where we spawn multiple threads -- and have them 'collaborate' on a result

```
1  @file team.cpp
2  // g++ -std=c++23 team.cpp -o prog -lpthread
3  #include <iostream>
4  #include <thread>    // Include the thread library
5  #include <vector>
6  #include <array>
7  #include <cstring>
8
9  // Global data, or otherwise 'heap' allocated data
10 // is by default 'shareable' to every thread.
11 std::array<int,256> shared_global_data;
12
13 // Entry point to program
14 int main() {
15
16     // Zero-out memory
17     std::memset(shared_global_data.data(), 0, sizeof(int)*shared_global_data.size());
18
19     // Here is our diligent worker that will execute on some shared memory
20     // The 'index' (sometimes abbreviated 'idx' or just 'id') we will use
21     // in combination with 'jobSize' -- indicating how many bytes to increment.
22     auto AdditionWorker= [](size_t index, size_t jobSize){
23         // std::cout << "thread.get id:" << std::this_thread::get_id() << std::endl;
24         for(size_t i = index*jobSize; i < (index+1) * jobSize; i++){
25             shared_global_data[i] += 1;
26         }
27     };
28
29     // 'threads' vector enables us the ability to push in jthreads -- and execute
30     // multiple threads in parallel
31     std::vector<std::jthread> threads;
32     // Run many iterations of our simulation
33     for(int j=0; j < 5; j++){
34         // Create four threads at a time
35         // They will 'synchronize' and effectively work as a team of '4' at a time
36         for(int i=0; i < 4; i++){
37             threads.push_back(std::jthread(AdditionWorker,i,64));
38         }
39     }
40     std::cout << "threads.size: " << threads.size() << std::endl;
41
42     // Continue executing the main thread
43     std::cout << "Job completed -- in main thread and printing results" << std::endl;
44     // Write out data
45     for(size_t i=0; i < shared_global_data.size(); i++){
46         std::cout << shared_global_data[i] << " ";
47     }
48     std::cout << std::endl;
49     return 0;
50 }
```

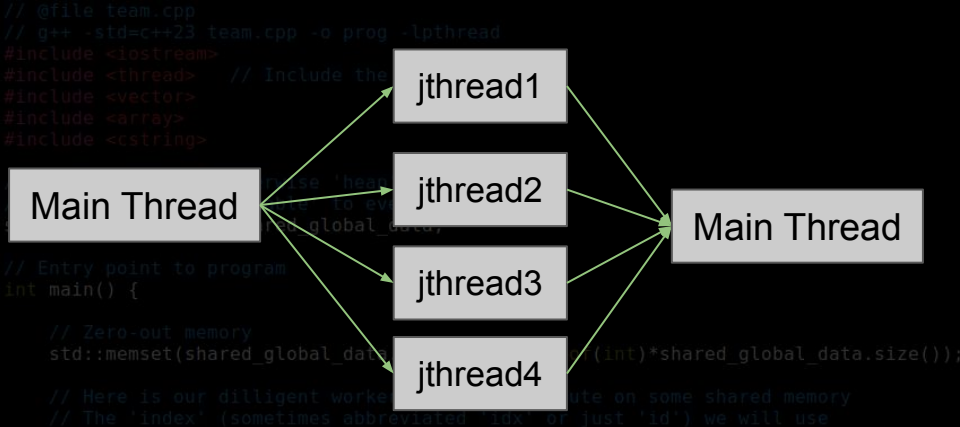
Thread Team (1/9)

- Let's do a more interesting problem where we spawn multiple threads -- and have them 'collaborate' on a result
- **The task:** increment unique indices in a global shared array.
- **Approach:** Launch multiple threads that work on separate indices

```
1  @file team.cpp
2  // g++ -std=c++23 team.cpp -o prog -lpthread
3  #include <iostream>
4  #include <thread>    // Include the thread library
5  #include <vector>
6  #include <array>
7  #include <string>
8
9  // Global data, or otherwise 'heap' allocated data
10 // is by default 'shareable' to every thread.
11 std::array<int,256> shared_global_data;
12
13 // Entry point to program
14 int main() {
15
16     // Zero-out memory
17     std::memset(shared_global_data.data(), 0, sizeof(int)*shared_global_data.size());
18
19     // Here is our diligent worker that will execute on some shared memory
20     // The 'index' (sometimes abbreviated 'idx' or just 'id') we will use
21     // in combination with 'jobSize' -- indicating how many bytes to increment.
22     auto AdditionWorker= [](size_t index, size_t jobSize){
23         // std::cout << "thread.get id:" << std::this_thread::get_id() << std::endl;
24         for(size_t i = index*jobSize; i < (index+1) * jobSize; i++){
25             shared_global_data[i] += 1;
26         }
27     };
28
29     // 'threads' vector enables us the ability to push in jthreads -- and execute
30     // multiple threads in parallel
31     std::vector<std::jthread> threads;
32     // Run many iterations of our simulation
33     for(int j=0; j < 5; j++){
34         // Create four threads at a time
35         // They will 'synchronize' and effectively work as a team of '4' at a time
36         for(int i=0; i < 4; i++){
37             threads.push_back(std::jthread(AdditionWorker,i,64));
38         }
39     }
40     std::cout << "threads.size: " << threads.size() << std::endl;
41
42     // Continue executing the main thread
43     std::cout << "Job completed -- in main thread and printing results" << std::endl;
44     // Write out data
45     for(size_t i=0; i < shared_global_data.size(); i++){
46         std::cout << shared_global_data[i] << " ";
47     }
48     std::cout << std::endl;
49     return 0;
50 }
```


Thread Team (2/9)

- Let's do a more interesting problem where we spawn multiple threads -- and have them 'collaborate' on a result
- **The task:** increment unique indices in a global shared array.
- **Approach:** Launch multiple threads that work on separate indices



From the main thread we'll 'spawn' 4 threads in a loop -- push them into a vector (like previous) and have them work **on separate blocks of shared memory**

```

29 // 'threads' vector enables us the ability to push in jthreads -- and execute
30 // multiple threads in parallel
31 std::vector<std::jthread> threads;
32 // Run many iterations of our simulation
33 for(int j=0; j < 5; j++){
34     // Create four threads at a time
35     // They will 'synchronize' and effectively work as a team of '4' at a time
36     for(int i=0; i < 4; i++){
37         threads.push_back(std::jthread(AdditionWorker,i,64));
38     }
39 }
40 std::cout << "threads.size: " << threads.size() << std::endl;
41
42 // Continue executing the main thread
43 std::cout << "Job completed -- in main thread and printing results" << std::endl;
44 // Write out data
45 for(size_t i=0; i < shared_global_data.size(); i++){
46     std::cout << shared_global_data[i] << " ";
47 }
48 std::cout << std::endl;
49 return 0;
50 }

```

Thread Team (3/9)

- Let's do a more interesting problem where we spawn multiple threads -- and have them 'collaborate' on a result
- The task:** increment unique indices in a global shared array.
- Approach:** Launch multiple threads that work on separate indices

```
1  @file team.cpp
2  // g++ -std=c++23 team.cpp -o prog -lpthread
3  #include <iostream>
4  #include <thread> // Include the
5  #include <vector>
6  #include <array>
7  #include <string>
8
9
10 // Use 'shared' to create a global variable,
11 // which can be accessed by all threads.
12
13 // Entry point to program
14 int main() {
15
16     // Zero-out memory
17     std::memset(shared_global_data, 0, (int)*shared_global_data.size());
18
19     // Here is our diligent worker thread that will write on some shared memory
20     // The 'index' (sometimes abbreviated 'idx' or just 'id') we will use
21
22
23
24
25
26
27
28
29 // 'threads' vector enables us the ability to push in jthreads -- and execute
30 // multiple threads in parallel
31 std::vector<std::jthread> threads;
32 // Run many iterations of our simulation
33 for(int j=0; j < 5; j++){
34     // Create four threads at a time
35     // They will 'synchronize' and ef
36     for(int i=0; i < 4; i++){
37         threads.push_back(std::jthread
38     }
39 }
40 std::cout << "threads.size: " << threads.size() << endl;
41
42 // Continue executing the main thread
43 std::cout << "Job completed -- in mai
44 // Write out data
45 for(size_t i=0; i < shared_global_data.size(); i++)
46     std::cout << shared_global_data[i] << " ";
47 }
48 std::cout << std::endl;
49 return 0;
50 }
```

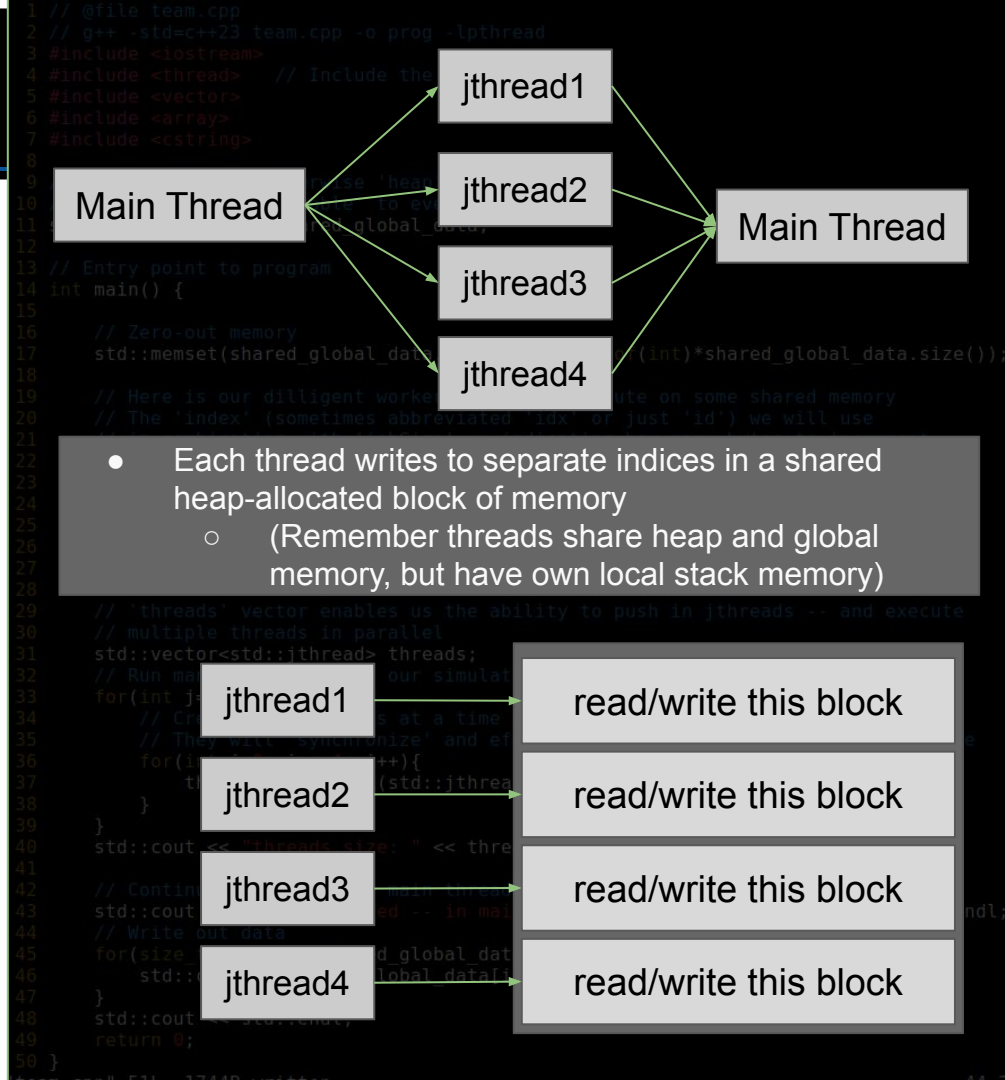
```
graph LR
    MainThread[Main Thread] --> jthread1[jthread1]
    MainThread --> jthread2[jthread2]
    MainThread --> jthread3[jthread3]
    MainThread --> jthread4[jthread4]
    jthread1 --> MainThread2[Main Thread]
    jthread2 --> MainThread2
    jthread3 --> MainThread2
    jthread4 --> MainThread2
```

Below is an example of 'shared memory'

Shared memory (i.e. a big array)

Thread Team (4/9)

- Let's do a more interesting problem where we spawn multiple threads -- and have them 'collaborate' on a result
- The task:** increment unique indices in a global shared array.
- Approach:** Launch multiple threads that work on separate indices



Thread Team (5/9)

- Here is full resulting code
 - We'll look at each chunk in the next few slides

```
1 // @file team.cpp
2 // g++ -std=c++23 team.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread>    // Include the thread library
5 #include <vector>
6 #include <array>
7 #include <cstring>
8
9 // Global data, or otherwise 'heap' allocated data
10 // is by default 'shareable' to every thread.
11 std::array<int,256> shared_global_data;
12
13 // Entry point to program
14 int main() {
15
16     // Zero-out memory
17     std::memset(shared_global_data.data(), 0, sizeof(int)*shared_global_data.size());
18
19     // Here is our diligent worker that will execute on some shared memory
20     // The 'index' (sometimes abbreviated 'idx' or just 'id') we will use
21     // in combination with 'jobSize' -- indicating how many bytes to increment.
22     auto AdditionWorker= [](size_t index, size_t jobSize){
23         // std::cout << "thread.get_id:" << std::this_thread::get_id() << std::endl;
24         for(size_t i = index*jobSize; i < (index+1) * jobSize; i++){
25             shared_global_data[i] += 1;
26         }
27     };
28
29     // 'threads' vector enables us the ability to push in jthreads -- and execute
30     // multiple threads in parallel
31     std::vector<std::jthread> threads;
32     // Run many iterations of our simulation
33     for(int j=0; j < 5; j++){
34         // Create four threads at a time
35         // They will 'synchronize' and effectively work as a team of '4' at a time
36         for(int i=0; i < 4; i++){
37             threads.push_back(std::jthread(AdditionWorker,i,64));
38         }
39     }
40     std::cout << "threads.size: " << threads.size() << std::endl;
41
42     // Continue executing the main thread
43     std::cout << "Job completed -- in main thread and printing results" << std::endl;
44     // Write out data
45     for(size_t i=0; i < shared_global_data.size(); i++){
46         std::cout << shared_global_data[i] << " ";
47     }
48     std::cout << std::endl;
49     return 0;
50 }
```

Thread Team (6/9)

- Here is the resulting code

```
9 // Global data, or otherwise 'heap' allocated data
10 // is by default 'shareable' to every thread.
11 std::array<int,256> shared_global_data;
12
13 // Entry point to program
14 int main() {
15
16     // Zero-out memory
17     std::memset(shared_global_data.data(), 0, sizeof(int)*shared_global_data.size());
18 }
```

Here we initialize a chunk of shared memory

```
1 @file team.cpp
2 // g++ -std=c++23 team.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5 #include <vector>
6 #include <array>
7 #include <string>
8
9 // Global data, or otherwise 'heap' allocated data
10 // is by default 'shareable' to every thread.
11 std::array<int,256> shared_global_data;
12
13 // Entry point to program
14 int main() {
15
16     // Zero-out memory
17     std::memset(shared_global_data.data(), 0, sizeof(int)*shared_global_data.size());
18
19     // Here is our diligent worker that will execute on some shared memory
20
21     // ...
22
23     // ...
24
25     // ...
26
27     // ...
28
29     // ...
30
31     // ...
32
33     // ...
34
35     // ...
36
37     // ...
38
39     // ...
40
41     // ...
42
43     // ...
44
45     // ...
46     std::cout << shared_global_data[i] << " ";
47 }
48 std::cout << std::endl;
49 return 0;
50 }
```


T

```
1 // @file team.cpp
2 // g++ -std=c++23 team.cpp -o prog -lpthread
3 #include <iostream>
19 // Here is our dilligent worker that will execute on some shared memory
20 // The 'index' (sometimes abbreviated 'idx' or just 'id') we will use
21 // in combination with 'jobSize' -- indicating how many bytes to increment.
22 auto AdditionWorker= [](size_t index, size_t jobSize){
23     // std::cout << "thread.get_id:" << std::this_thread::get_id() << std::endl;
24     for(size_t i = index*jobSize; i < (index+1) * jobSize; i++){
25         shared_global_data[i] += 1;
26     }
27 };
```

```
23 // std::cout << "thread.get_id:" << std::this_thread::get_id() << std::endl;
24 for(size_t i = index*jobSize; i < (index+1) * jobSize; i++){
25     shared_global_data[i] += 1;
26 }
27 };
```

Next we create a ‘worker thread’ that will execute -- observe:

- An ‘index’ and ‘jobSize’ provides the ‘range’ (start and finish) of where we’ll access the array.
 - Care is taken so we do not overlap

We then do '5' iterations with '4' worker threads

```
29 // 'threads' vector enables us the ability to push in jthreads -- and execute
30 // multiple threads in parallel
31 std::vector<std::jthread> threads;
32 // Run many iterations of our simulation
33 for(int j=0; j < 5; j++){
34     // Create four threads at a time
35     // They will 'synchronize' and effectively work as a team of '4' at a time
36     for(int i=0; i < 4; i++){
37         threads.push_back(std::jthread(AdditionWorker,i,64));
38     }
39 }
```

```
ared_global_data.size());
e shared memory
) we will use
bytes to increment.

et_id() << std::endl;
e; i++){
```

```
29 // 'threads' vector enables the ability to push in jthreads -- and execute
30 // multiple threads in parallel
31 std::vector<std::jthread> threads;
32 // Run many iterations of our simulation
33 for(int j=0; j < 5; j++){
34     // Create four threads at a time
35     // They will 'synchronize' and effectively work as a team of '4' at a time
36     for(int i=0; i < 4; i++){
37         threads.push_back(std::jthread(AdditionWorker,i,64));
38     }
39 }
40 std::cout << "threads.size: " << threads.size() << std::endl;
41
42 // Continue executing the main thread
43 std::cout << "Job completed -- in main thread and printing results" << std::endl;
44 // Write out data
45 for(size_t i=0; i < shared_global_data.size(); i++){
46     std::cout << shared_global_data[i] << " ";
47 }
48 std::cout << std::endl;
49 return 0;
50 }
```

Thread Team (9/9)

- The program works as expected
 - i.e. We successfully increment each value '5' times (Printing out the 256, fives sequentially at the end)

```
1 // @file team.cpp
2 +-- 9 lines: g++ -std=c++23 team.cpp -o prog -lpthread-----
11 std::array<int,256> shared_global_data;
12
13 // Entry point to program
14 int main() {
15
16     mike@mike-MS-7B17:nycpp$ g++ -std=c++23 team.cpp -o prog -lpthread
17     mike@mike-MS-7B17:nycpp$ time ./prog
18     threads.size: 20
19     Job completed -- in main thread and printing results
20     [ 0] 5
21     [ 1] 5 [ 2] 5 [ 3] 5 [ 4] 5 [ 5] 5 [ 6] 5 [ 7] 5 [ 8] 5 [ 9] 5 [10] 5
22     [11] 5 [12] 5 [13] 5 [14] 5 [15] 5 [16] 5 [17] 5 [18] 5 [19] 5 [20] 5
23     [21] 5 [22] 5 [23] 5 [24] 5 [25] 5 [26] 5 [27] 5 [28] 5 [29] 5 [30] 5
24     [31] 5 [32] 5 [33] 5 [34] 5 [35] 5 [36] 5 [37] 5 [38] 5 [39] 5 [40] 5
25     [41] 5 [42] 5 [43] 5 [44] 5 [45] 5 [46] 5 [47] 5 [48] 5 [49] 5 [50] 5
26     [51] 5 [52] 5 [53] 5 [54] 5 [55] 5 [56] 5 [57] 5 [58] 5 [59] 5 [60] 5
27     [61] 5 [62] 5 [63] 5 [64] 5 [65] 5 [66] 5 [67] 5 [68] 5 [69] 5 [70] 5
28     [71] 5 [72] 5 [73] 5 [74] 5 [75] 5 [76] 5 [77] 5 [78] 5 [79] 5 [80] 5
29     [81] 5 [82] 5 [83] 5 [84] 5 [85] 5 [86] 5 [87] 5 [88] 5 [89] 5 [90] 5
30     [91] 5 [92] 5 [93] 5 [94] 5 [95] 5 [96] 5 [97] 5 [98] 5 [99] 5 [100] 5
31     [101] 5 [102] 5 [103] 5 [104] 5 [105] 5 [106] 5 [107] 5 [108] 5 [109] 5 [110] 5
32     [111] 5 [112] 5 [113] 5 [114] 5 [115] 5 [116] 5 [117] 5 [118] 5 [119] 5 [120] 5
33     [121] 5 [122] 5 [123] 5 [124] 5 [125] 5 [126] 5 [127] 5 [128] 5 [129] 5 [130] 5
34     [131] 5 [132] 5 [133] 5 [134] 5 [135] 5 [136] 5 [137] 5 [138] 5 [139] 5 [140] 5
35     [141] 5 [142] 5 [143] 5 [144] 5 [145] 5 [146] 5 [147] 5 [148] 5 [149] 5 [150] 5
36     [151] 5 [152] 5 [153] 5 [154] 5 [155] 5 [156] 5 [157] 5 [158] 5 [159] 5 [160] 5
37     [161] 5 [162] 5 [163] 5 [164] 5 [165] 5 [166] 5 [167] 5 [168] 5 [169] 5 [170] 5
38     [171] 5 [172] 5 [173] 5 [174] 5 [175] 5 [176] 5 [177] 5 [178] 5 [179] 5 [180] 5
39     [181] 5 [182] 5 [183] 5 [184] 5 [185] 5 [186] 5 [187] 5 [188] 5 [189] 5 [190] 5
40     [191] 5 [192] 5 [193] 5 [194] 5 [195] 5 [196] 5 [197] 5 [198] 5 [199] 5 [200] 5
41     [201] 5 [202] 5 [203] 5 [204] 5 [205] 5 [206] 5 [207] 5 [208] 5 [209] 5 [210] 5
42     [211] 5 [212] 5 [213] 5 [214] 5 [215] 5 [216] 5 [217] 5 [218] 5 [219] 5 [220] 5
43     [221] 5 [222] 5 [223] 5 [224] 5 [225] 5 [226] 5 [227] 5 [228] 5 [229] 5 [230] 5
44     [231] 5 [232] 5 [233] 5 [234] 5 [235] 5 [236] 5 [237] 5 [238] 5 [239] 5 [240] 5
45     [241] 5 [242] 5 [243] 5 [244] 5 [245] 5 [246] 5 [247] 5 [248] 5 [249] 5 [250] 5
46     [251] 5 [252] 5 [253] 5 [254] 5 [255] 5
47
48     real    0m0.002s
49     user    0m0.000s
50     sys     0m0.003s
51
52     std::print("{0:3} {1:2} ",i,shared_global_data[i]);
53     if(i%10==0){std::println();}
54 }
55
56     std::println();
57     return 0;
58 }
```

Thread Team Round 2 (1/5)

- Great -- now let's do a real test on a “real” workload -- I've modified the program to now run ‘50000’ times
 - and ... (next slide)

```
1 // @file team50000.cpp
2 +-- 6 lines: g++ -std=c++23 team.cpp -o prog -pthread-----
3
4 // Global data, or otherwise 'heap' allocated data
5 // is by default 'shareable' to every thread.
6 std::array<int,256> shared_global_data;
7
8 // Entry point to program
9 int main() {
10
11     // Zero-out memory
12     std::memset(shared_global_data.data(), 0, sizeof(int)*shared_global_data.size());
13
14     // Here is our dilligent worker that will execute on some shared memory
15     // The 'index' (sometimes abbreviated 'idx' or just 'id') we will use
16     // in combination with 'jobSize' -- indicating how many bytes to increment.
17     auto AdditionWorker= [](size_t index, size_t jobSize){
18         std::println << "thread.get id:" << std::this_thread::get_id() << std::endl;
19         for(size_t i = index*jobSize; i < (index+1) * jobSize; i++){
20             shared_global_data[i] += 1;
21         }
22     };
23
24     // 'threads' vector enables us the ability to push in jthreads -- and execute
25     // multiple threads in parallel
26     std::vector<std::jthread> threads;
27     // Run many iterations of our simulation
28     for(int j=0; j < 50'000; j++){
29         // Create four threads at a time
30         // They will 'synchronize' and effectively work as a team of '4' at a time
31         for(int i=0; i < 4; i++){
32             threads.push_back(std::jthread(AdditionWorker,i,64));
33         }
34     }
35     std::println("threads.size: {}", threads.size());
36
37     // Ensure all threads have been joined
38     for(int i=0; i < threads.size();i++){ threads[i].join(); }
39
40     // Continue executing the main thread
41     std::println("Job completed -- in main thread and printing results");
42     // Write out data
43     for(size_t i=0; i < shared_global_data.size(); i++){
44         std::print("[{:3}] {:2} ",i,shared_global_data[i]);
45         if(i%10==0){std::println();}
46     }
47     std::println();
48     return 0;
49 }
```


Thread Team Round 2 (2/5)

- Great -- now let's do a real test on a "real" workload -- I've modified the program to now run '50000' times
 - and ... (next slide)
 - CRASH**

```
mike@mike-MS-7B17:nycpp$ g++ -std=c++23 team50000.cpp -o prog -lpthread
mike@mike-MS-7B17:nycpp$ time ./prog
```

```
terminate called after throwing an instance of 'std::system_error'
  what(): Resource temporarily unavailable
Aborted (core dumped)
```

```
real    0m0.658s
user    0m0.094s
sys     0m0.809s
mike@mike-MS-7B17:nycpp$
```

```
1 // @file team50000.cpp
2 +-- 6 lines: g++ -std=c++23 team.cpp -o prog -lpthread-----
3
4 // Global data, or otherwise 'heap' allocated data
5 // is by default 'shareable' to every thread.
6 std::array<int,256> shared_global_data;
7
8 // Entry point to program
9 int main() {
10
11     // Zero-out memory
12     std::memset(shared_global_data.data(), 0, sizeof(int)*shared_global_data.size());
13
14     // Here is our dilligent worker that will execute on some shared memory
15     // The 'index' (sometimes abbreviated 'idx' or just 'id') we will use
16     // in combination with 'jobSize' -- indicating how many bytes to increment.
17     auto AdditionWorker= [](size_t index, size_t jobSize){
18         // std::println << "thread.get id:" << std::this_thread::get_id() << std::endl;
19         for(size_t i = index*jobSize; i < (index+1) * jobSize; i++){
20             shared_global_data[i] += 1;
21         }
22     };
23
24     // 'threads' vector enables us the ability to push in jthreads -- and execute
25     // multiple threads in parallel
26     std::vector<std::jthread> threads;
27     // Run many iterations of our simulation
28     for(int j=0; j < 50'000; j++){
29         // Create four threads at a time
30         // They will 'synchronize' and effectively work as a team of '4' at a time
31         for(int i=0; i < 4; i++){
32             threads.push_back(std::jthread(AdditionWorker,i,64));
33         }
34     }
35     std::println("threads.size: {}", threads.size());
36
37     // Ensure all threads have been joined
38     for(int i=0; i < threads.size();i++){ threads[i].join(); }
39
40     // Continue executing the main thread
41     std::println("Job completed -- in main thread and printing results");
42     // Write out data
43     for(size_t i=0; i < shared_global_data.size(); i++){
44         std::print("{0:3} {1:2} ",i,shared_global_data[i]);
45         if(i%10==0){std::println();}
46     }
47     std::println();
48     return 0;
49 }
50
51
52
53
54 }
```

Thread Team Round 2 (3/5)

- Question to Audience:
 - What is the issue? (Hint highlighted)

```
mike@system76-pc:~/Talks/2024/french_cpp_user_group_frug$ g++ -g -W
all -std=c++23 team50000.cpp -o prog -lpthread
mike@system76-pc:~/Talks/2024/french_cpp_user_group_frug$ time ./pr
og
terminate called after throwing an instance of 'std::system_error'
what(): Resource temporarily unavailable
Aborted (core dumped)
```

```
real    0m0.562s
user    0m0.060s
sys     0m0.667s
```

```
mike@system76-pc:~/Talks/2024/french_cpp_user_group_frug$
```

```
1 // @file team50000.cpp
2 +-- 6 lines: g++ -std=c++23 team.cpp -o prog -lpthread-----
3
4 // Global data, or otherwise 'heap' allocated data
5 // is by default 'shareable' to every thread.
6 std::array<int,256> shared_global_data;
7
8 // Entry point to program
9 int main() {
10
11     // Zero-out memory
12     std::memset(shared_global_data.data(), 0, sizeof(int)*shared_global_data.size());
13
14     // Here is our dilligent worker that will execute on some shared memory
15     // The 'index' (sometimes abbreviated 'idx' or just 'id') we will use
16     // in combination with 'jobSize' -- indicating how many bytes to increment.
17     auto AdditionWorker= [](size_t index, size_t jobSize){
18         // std::println << "thread.get id:" << std::this_thread::get_id() << std::endl;
19         for(size_t i = index*jobSize; i < (index+1) * jobSize; i++){
20             shared_global_data[i] += 1;
21         }
22     };
23
24     // 'threads' vector enables us the ability to push in jthreads -- and execute
25     // multiple threads in parallel
26     std::vector<std::jthread> threads;
27     // Run many iterations of our simulation
28     for(int j=0; j < 50'000; j++){
29         // Create four threads at a time
30         // They will 'synchronize' and effectively work as a team of '4' at a time
31         for(int i=0; i < 4; i++){
32             threads.push_back(std::jthread(AdditionWorker,i,64));
33         }
34     }
35     std::println("threads.size: {}", threads.size());
36
37     // Ensure all threads have been joined
38     for(int i=0; i < threads.size();i++){ threads[i].join(); }
39
40     // Continue executing the main thread
41     std::println("Job completed -- in main thread and printing results");
42     // Write out data
43     for(size_t i=0; i < shared_global_data.size(); i++){
44         std::print("{}[{}:3] {}{1:2} ",i,shared_global_data[i]);
45         if(i%10==0){std::println();}
46     }
47     std::println();
48     return 0;
49 }
50 }
```


Thread Team Round 2 (4/5)

- Question to Audience:
 - What is the issue? (Hint highlighted)
 - Answer: Too many threads created on stack at once for my cpu
 - I have created **50,000*4** threads for one process.
 - The threads don't terminate after all, until 'vector' destructor is called
 - (And that is end of program)
 - Note: With other thread libraries, we also need to be aware of what could happen when resizing containers
 - (std::threads are non-copyable, which is good and prevents weird behavior).

```
1 // @file team50000.cpp
2 +-- 6 lines: g++ -std=c++23 team.cpp -o prog -lpthread-----
3
4 // Global data, or otherwise 'heap' allocated data
5 // is by default 'shareable' to every thread.
6 std::array<int,256> shared_global_data;
7
8 // Entry point to program
9 int main() {
10
11     // Zero-out memory
12     std::memset(shared_global_data.data(), 0, sizeof(int)*shared_global_data.size());
13
14     // Here is our dilligent worker that will execute on some shared memory
15     // The 'index' (sometimes abbreviated 'idx' or just 'id') we will use
16     // in combination with 'jobSize' -- indicating how many bytes to increment.
17     auto AdditionWorker= [](size_t index, size_t jobSize){
18         // std::println << "thread.get id:" << std::this_thread::get_id() << std::endl;
19         for(size_t i = index*jobSize; i < (index+1) * jobSize; i++){
20             shared_global_data[i] += 1;
21         }
22     };
23
24     // 'threads' vector enables us the ability to push in jthreads -- and execute
25     // multiple threads in parallel
26     std::vector<std::jthread> threads;
27     // Run many iterations of our simulation
28     for(int j=0; j < 50'000; j++){
29         // Create four threads at a time
30         // They will 'synchronize' and effectively work as a team of '4' at a time
31         for(int i=0; i < 4; i++){
32             threads.push_back(std::jthread(AdditionWorker,i,64));
33         }
34     }
35     std::println("threads.size: {}", threads.size());
36
37     // Ensure all threads have been joined
38     for(int i=0; i < threads.size();i++){ threads[i].join(); }
39
40     // Continue executing the main thread
41     std::println("Job completed -- in main thread and printing results");
42     // Write out data
43     for(size_t i=0; i < shared_global_data.size(); i++){
44         std::print("{}[{}:3] {}[{}:2] ",i,shared_global_data[i]);
45         if(i%10==0){std::println();}
46     }
47     std::println();
48     return 0;
49 }
```

Thread Team Round 2 (5/5)

- **Live GDB Session:**

- Launch GDB
- [set mi-async on](#)
 - Then load executable: file ./prog
- b 37 if j > 15
 - Observe that 'threads vector' 'never shrinks'!
 - Note: threads are 'moved' instead of copied, but we still have a large 'move' to do -- plus our stack of 'functions' potentially grows very fast!
- set scheduler-locking on
 - Mode needs to be 'on'
 - This pauses all threads when one stops -- easier to debug
- display threads.size()
 - Updates when we push into size
- Press 'c' for continue a few times
- call malloc_stats()
 - Gives us some idea of memory allocations (at least for the heap allocations with threads)

```
1 // @file team50000.cpp
2 +-- 6 lines: g++ -std=c++23 team.cpp -o prog -lpthread-----
3
4
5 // Global data, or otherwise 'heap' allocated data
6 // is by default 'shareable' to every thread.
7 std::array<int,256> shared_global_data;
8
9 // Entry point to program
10 int main() {
11
12     // Zero-out memory
13     std::memset(shared_global_data.data(), 0, sizeof(int)*shared_global_data.size());
14
15     // Here is our dilligent worker that will execute on some shared memory
16     // The 'index' (sometimes abbreviated 'idx' or just 'id') we will use
17     // in combination with 'jobSize' -- indicating how many bytes to increment.
18     auto AdditionWorker= [](size_t index, size_t jobSize){
19         std::println << "thread.get id:" << std::this_thread::get_id() << std::endl;
20         for(size_t i = index*jobSize; i < (index+1) * jobSize; i++){
21             shared_global_data[i] += 1;
22         }
23     };
24
25     // 'threads' vector enables us the ability to push in jthreads -- and execute
26     // multiple threads in parallel
27     std::vector<std::jthread> threads;
28     // Run many iterations of our simulation
29     for(int j=0; j < 50'000; j++){
30         // Create four threads at a time
31         // They will 'synchronize' and effectively work as a team of '4' at a time
32         for(int i=0; i < 4; i++){
33             threads.push_back(std::jthread(AdditionWorker,i,64));
34         }
35     }
36     std::println("threads.size: {}", threads.size());
37
38     // Ensure all threads have been joined
39     for(int i=0; i < threads.size();i++){ threads[i].join(); }
40
41     // Continue executing the main thread
42     std::println("Job completed -- in main thread and printing results");
43     // Write out data
44     for(size_t i=0; i < shared_global_data.size(); i++){
45         std::print("[{:3}] {:2} ",i,shared_global_data[i]);
46         if(i%10==0){std::println();}
47     }
48     std::println();
49     return 0;
50 }
```

Thread Team Fixed (1/2)

- The fix itself was quite simple -- but could be tricky to find!
 - Idea is to move 'threads' into scope of each iteration
 - Would I have found this bug if I only launched 50 threads? How about 1000?
 - The answer is it's system dependent on the thread limits

```
1 // @file team50000.cpp
2 +-- 6 lines: g++ -std=c++23 team.cpp -o prog -lpthread-----
3
4
5
6
7
8
9 // Global data, or otherwise 'heap' allocated data
10 // is by default 'shareable' to every thread.
11 std::array<int,256> shared_global_data;
12
13 // Entry point to program
14 int main() {
15
16     // Zero-out memory
17     std::memset(shared_global_data.data(), 0, sizeof(int)*shared_global_data.size());
18
19     // Here is our dilligent worker that will execute on some shared memory
20     // The 'index' (sometimes abbreviated 'idx' or just 'id') we will use
21     // in combination with 'jobSize' -- indicating how many bytes to increment.
22     auto AdditionWorker= [](size_t index, size_t jobSize){
23         // std::println << "thread.get_id:" << std::this_thread::get_id() << std::endl;
24         for(size_t i = index*jobSize; i < (index+1) * jobSize; i++){
25             shared_global_data[i] += 1;
26         }
27     };
28
29     // Run many iterations of our simulation
30     for(int i=0; i < 50'000; i++){
31         // 'threads' vector enables us the ability to push in jthreads -- and execute
32         // multiple threads in parallel
33         std::vector<std::jthread> threads;
34         // Create four threads at a time
35         // They will 'synchronize' and effectively work as a team of '4' at a time
36         for(int i=0; i < 4; i++){
37             threads.push_back(std::jthread(AdditionWorker,i,64));
38         }
39         std::println("threads.size each iteration: {}", threads.size());
40     }
41
42     // Ensure all threads have been joined
43     for(int i=0; i < threads.size();i++){ threads[i].join(); } // Don't need this anymore --
44                                     // threads destroyed in scope
45
46     // Continue executing the main thread
47     std::println("Job completed -- in main thread and printing results");
48     // Write out data
49     for(size_t i=0; i < shared_global_data.size(); i++){
50         std::print("[{0:3}] {1:2} ",i,shared_global_data[i]);
51         if(i%10==0){std::println();}
52     }
53     std::println();
54     return 0;
55 }
```

Thread Team Fixed (2/2)

- There are a finite number of threads available on your operating system
 - As well as stack size (ulimit -s indicates 8mb on my machine)
 - (See 'ulimit -a' for more info)

```
mike@system76-pc:~$ cat /proc/sys/kernel/threads-max
512511
mike@system76-pc:~$ ulimit -s
8192
mike@system76-pc:~$
```

```
1 // @file team50000_fix.cpp
2 // g++ -std=c++23 team50000_fix.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5 #include <vector>
6 #include <array>
7 #include <cstring>
8
9 // Global data, or otherwise 'heap' allocated data
10 // is by default 'shareable' to every thread.
11 std::array<int,256> shared_global_data;
12
13 // Entry point to program
14 int main() {
15
16     // Zero-out memory
17     std::memset(shared_global_data.data(), 0, sizeof(int)*shared_global_data.size());
18
19     // Here is our dilligent worker that will execute on some shared memory
20     // The 'index' (sometimes abbreviated 'idx' or just 'id') we will use
21     // in combination with 'jobSize' -- indicating how many bytes to increment.
22     auto AdditionWorker= [](size_t index, size_t jobSize){
23         // std::cout << "thread.get_id:" << std::this_thread::get_id() << std::endl;
24         for(size_t i = index*jobSize; i < (index+1) * jobSize; i++){
25             shared_global_data[i] += 1;
26         }
27     };
28
29     // Run many iterations of our simulation
30     for(int j=0; j < 50'000; j++){
31         // 'threads' vector enables us the ability to push in jthreads -- and execut
32         // multiple threads in parallel
33         std::vector<std::jthread> threads;
34         // Create four threads at a time
35         // They will 'synchronize' and effectively work as a team of '4' at a time
36         for(int i=0; i < 4; i++){
37             threads.push_back(std::jthread(AdditionWorker,i,64));
38         }
39     }
40     //std::cout << "threads.size: " << threads.size() << std::endl;
41
42     // Continue executing the main thread
43     std::cout << "Job completed -- in main thread and printing results" << std::endl;
44     // Write out data
45     for(size_t i=0; i < shared_global_data.size(); i++){
46         std::cout << shared_global_data[i] << " ";
47     }
48     std::cout << std::endl;
49     return 0;
50 }
```

Can I launch 50,000 threads with my limit?

- Searching: `nl /etc/systemd/system.conf`
 - I'm allowed to have 15% of my maximum allowable threads allocated to a process on Ubuntu 22.04
 - (This seems reasonable -- I could for instance launch 25,000 threads no problem -- probably way too many though!)
- Probably not a good idea to launch this many on your desktop CPU in 2024
 - 2 threads per 1 core is a 'metric' used by some
 - Threads have a cost to start and to join
 - Generally this is considered 'costly'
- This brings up two interesting ideas
 - The first is whether 'sequential' execution is actually better in some cases
 - The second is -- how can we avoid 'recreation' of threads
 - i.e. the idea of a thread pool

(Aside) Sequential Execution is Sometimes
Better (and False Sharing)

Sequential (1/2)

- Comparing the sequential performance
 - Get the correct answer (useful for unit testing!)
 - Hmm, seems to run quite fast!
 - Less complicated code even

[illegible]

```

1 // @file sequential.cpp
2 // g++ -std=c++23 sequential.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5 #include <vector>
6 #include <array>
7 #include <cstring>
8
9 // Global data, or otherwise 'heap' allocated data
10 // is by default 'shareable' to every thread.
11 std::array<int,256> shared_global_data;
12
13 // Entry point to program
14 int main() {
15
16     // Zero-out memory
17     std::memset(shared_global_data.data(), 0, sizeof(int)*shared_global_data.size());
18
19     // Here is our diligent worker that will execute on some shared memory
20     // The 'index' (sometimes abbreviated 'idx' or just 'id') we will use
21     // in combination with 'jobSize' -- indicating how many bytes to increment.
22     auto AdditionWorker= [](size_t index, size_t jobSize){
23         // std::cout << "thread.get id:" << std::this_thread::get_id() << std::endl;
24         for(size_t i = index*jobSize; i < (index+1) * jobSize; i++){
25             shared_global_data[i] += 1;
26         }
27     };
28
29     // Run many iterations of our simulation
30     for(int j=0; j < 50'000; j++){
31         // Create four threads at a time
32         // They will 'synchronize' and effectively work as a team of '4' at a time
33         for(int i=0; i < 4; i++){
34             AdditionWorker(i,64);
35         }
36     }
37
38     // Continue executing the main thread
39     std::cout << "Job completed -- in main thread and printing results" << std::endl;
40     // Write out data
41     for(size_t i=0; i < shared_global_data.size(); i++){
42         std::cout << shared_global_data[i] << " ";
43     }
44     std::cout << std::endl;
45     return 0;
46 }

```

- In my benchmarks why does the sequential benchmark win?
 - Less time spinning up threads which could take 100s or 1000s of cycles
 - Better cache locality

- Less time spinning up threads which could take 100s or 1000s of cycles
- Better cache locality

```
real    0m1.567s
user    0m0.304s
sys     0m1.804s
```

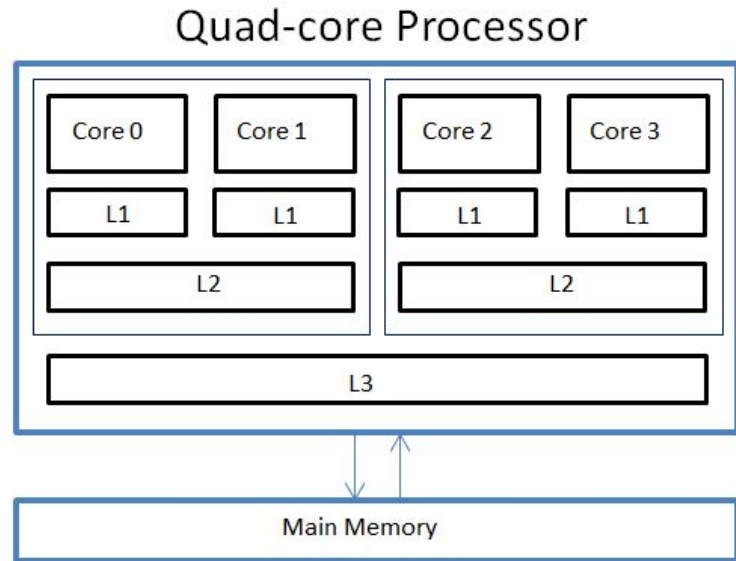
4 threads, constantly spinning up new threads

```
real    0m0.028s
user    0m0.027s
sys     0m0.000s
```

1 thread sequentially calling accumulate function

(Aside) How many threads to work together? (0/2)

- We can query with `std::thread::hardware_concurrency()` a 'good' number of threads for our hardware.
 - Conventional wisdom is 1-2 active threads per core -- measure for your system
- We also have to consider our 'cache'
 - Basically -- we want to access (for my specific architecture) no more than 64 bytes on independent threads.
 - Accessing more than that 'shares' data that must be evicted at least to the L3 cache, and then 'kept coherent' amongst other cores.
 - This creates a great slow down!
 - <https://devblogs.microsoft.com/oldnewthing/20230424-00/?p=108085>
 - https://en.cppreference.com/w/cpp/thread/hardware_destructive_interference_size



<https://www.researchgate.net/publication/322994264/figure/fig4/AS:599034075029513@1519832261760/A-three-level-shared-cache-quad-core-architecture.png>

(Aside) How many threads to work together? (1/2)

- Okay -- so I made the fix in regards to accessing '64 bytes' (16 ints, 4 bytes each) per thread
 - But we're still slower!
 - (In fact, ~10 times slower now than previous threads example, and several orders of magnitude slower than simple sequential code)



```
1 // @file team50000_locality_fix.cpp
2 // g++ -std=c++23 Team50000_locality_fix.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5 #include <vector>
6 #include <array>
7 #include <cstring>
8
9 // Global data, or otherwise 'heap' allocated data
10 // is by default 'shareable' to every thread.
11 std::array<int,256> shared_global_data;
12
13 void helper(){
14     std::cout << std::thread::hardware_concurrency() << " # of concurrent threads supported.\n";
15 }
16
17 // Entry point to program
18 int main() {
19
20     helper();
21
22     // Zero-out memory
23     std::memset(shared_global_data.data(), 0, sizeof(int)*shared_global_data.size());
24
25     // Here is our diligent worker that will execute on some shared memory
26     // The 'index' (sometimes abbreviated 'idx' or just 'id') we will use
27     // in combination with 'jobSize' -- indicating how many bytes to increment.
28     auto AdditionWorker= [](size_t index, size_t jobSize){
29         // std::cout << "thread.get id:" << std::this_thread::get_id() << std::endl;
30         for(size_t i = index*jobSize; i < (index+1) * jobSize; i++){
31             shared_global_data[i] += 1;
32         }
33     };
34
35     // Run many iterations of our simulation
36     for(int j=0; j < 50'000; j++){
37         // 'threads' vector enables us the ability to push in jthreads -- and execute
38         // multiple threads in parallel
39         std::vector<std::thread> threads;
40         // Create four threads at a time
41         // They will 'synchronize' and effectively work as a team of '4' at a time
42         for(int i=0; i < 16; i++){
43             threads.push_back(std::thread(AdditionWorker,i,16));
44         }
45     }
46     //std::cout << "threads.size: " << threads.size() << std::endl;
47
48     // Continue executing the main thread
49     std::cout << "Job completed -- in main thread and printing results" << std::endl;
50     // Write out data
```

```
real    0m15.204s
user    0m1.995s
sys     0m18.234s
```

(Aside) How many threads to work together? (2/2)

- Note: Slight confession -- the amount of work in our 'thread' is so trivial we should never have used threads in the first place
 - BUT -- I have to introduce these ideas to you somehow in a slideshow :)



```
1 // @file team50000_locality_fix.cpp
2 // g++ -std=c++23 Team50000_locality_fix.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5 #include <vector>
6 #include <array>
7 #include <cstring>
8
9 // Global data, or otherwise 'heap' allocated data
10 // is by default 'shareable' to every thread.
11 std::array<int,256> shared_global_data;
12
13 void helper(){
14     std::cout << std::thread::hardware_concurrency() << " # of concurrent threads supported.\n";
15 }
16
17 // Entry point to program
18 int main() {
19
20     helper();
21
22     // Zero-out memory
23     std::memset(shared_global_data.data(), 0, sizeof(int)*shared_global_data.size());
24
25     // Here is our diligent worker that will execute on some shared memory
26     // The 'index' (sometimes abbreviated 'idx' or just 'id') we will use
27     // in combination with 'jobSize' -- indicating how many bytes to increment.
28     auto AdditionWorker= [](size_t index, size_t jobSize){
29         // std::cout << "thread.get id:" << std::this_thread::get_id() << std::endl;
30         for(size_t i = index*jobSize; i < (index+1) * jobSize; i++){
31             shared_global_data[i] += 1;
32         }
33     };
34
35     // Run many iterations of our simulation
36     for(int j=0; j < 50'000; j++){
37         // 'threads' vector enables us the ability to push in jthreads -- and execute
38         // multiple threads in parallel
39         std::vector<std::thread> threads;
40         // Create four threads at a time
41         // They will 'synchronize' and effectively work as a team of '4' at a time
42         for(int i=0; i < 16; i++){
43             threads.push_back(std::thread(AdditionWorker,i,16));
44         }
45     }
46     //std::cout << "threads.size: " << threads.size() << std::endl;
47
48     // Continue executing the main thread
49     std::cout << "Job completed -- in main thread and printing results" << std::endl;
50     // Write out data
```

```
real    0m15.204s
user    0m1.995s
sys     0m18.234s
```

First Attempt at Reusable Threads with a Thread Pool

Removing issue of thread creation

Thread Pools

- A thread pool is a ‘pool’ of threads that are allocated at startup
 - The ‘pool’ of threads is long lived, and ‘grab’ work as needed.
- I’m going to show a first attempt of a thread pool, but we’ll need some mechanism to ensure our ‘threads remain alive’
 - Remember-- a `std::thread` (or `std::jthread`) is meant to execute one time.

Thread Pool - First Attempt (1/2)

- A first attempt to create a 'struct ThreadPool' on the right
 - The end result is the same
 - The result is correct, but we get similar performance when compared to our prior data-parallel example
 - **But** we've not yet solved our problem of thread creation -- but we are getting closer, and getting some encapsulation.
- It's important to notice however, that in our array we are still creating new threads (using 'move assignment')
- **But we can do better**

```
real    0m2.262s
user    0m0.344s
sys     0m2.710s
```

```
1 // @file pool_almost.cpp
2 // g++ -std=c++23 pool_almost.cpp -o prog -lpthread
3 +-- 6 lines: #include <iostream>-----
9
10 // Global data, or otherwise 'heap' allocated data
11 // is by default 'shareable' to every thread.
12 std::array<int,256> shared_global_data;
13
14 template <size_t threadcount>
15 struct ThreadPool{
16
17     ThreadPool(std::function<void(int,int)> func){
18         command = func;
19     }
20
21     void executeAll(size_t iterations, size_t jobSize){
22         size_t count = 0;
23
24         // Execute our '50000' iterations
25         while(count < iterations){
26             for(size_t i=0; i < threadcount; i++){
27                 // Assign ahead of time the thread you want to execute
28                 threads[i] = std::jthread(command, i, jobSize);
29             }
30             count++;
31         }
32     }
33
34     std::function<void(int,int)> command;
35     std::array<std::jthread,threadcount> threads;
36 };
37
38 +-- 10 lines: Entry point to program-----
49 auto AdditionWorker= [](size_t index, size_t jobSize){
50     for(size_t i = index*jobSize; i < (index+1) * jobSize; i++){
51         shared_global_data[i] += 1;
52     }
53 };
54 auto threadPool = ThreadPool<4>(AdditionWorker);
55
56 threadPool.executeAll(50000,64);
```

Thread Pool - First Attempt (2/2)

- A first attempt to create a 'struct ThreadPool' on the right

So our goal is to figure out how to keep threads alive, and then communicate (i.e. signal) to them that we have meaningful work to do

- It's important to notice however, that in our array we are still creating new threads (using 'move assignment')
- But we can do better**

```
real    0m2.262s
user    0m0.344s
sys     0m2.710s
```

```
1 // @file pool_almost.cpp
2 // g++ -std=c++23 pool_almost.cpp -o prog -lpthread
3 +-- 6 lines: #include <iostream>-----
4
5
6
7
8
9
10 // Global data, or otherwise 'heap' allocated data
11 // is by default 'shareable' to every thread.
12 std::array<int,256> shared_global_data;
13
14 template <size_t threadcount>
15 struct ThreadPool{
16
17     ThreadPool(std::function<void(int,int)> func){
18         command = func;
19     }
20
21     void executeAll(size_t iterations, size_t jobSize){
22         size_t count = 0;
23
24         // Execute our '50000' iterations
25         while(count < iterations){
26             for(size_t i=0; i < threadcount; i++){
27                 // Assign ahead of time the thread you want to execute
28                 threads[i] = std::jthread(command, i, jobSize);
29             }
30             count++;
31         }
32     }
33
34     std::function<void(int,int)> command;
35     std::array<std::jthread,threadcount> threads;
36 };
37
38 +-- 10 lines: Entry point to program-----
39
40 auto AdditionWorker= [](size_t index, size_t jobSize){
41     for(size_t i = index*jobSize; i < (index+1) * jobSize; i++){
42         shared_global_data[i] += 1;
43     }
44 };
45
46 auto threadPool = ThreadPool<4>(AdditionWorker);
47 threadPool.executeAll(50000,64);
```

Topics Outline

- ~~Transport Tycoon~~
- ~~std::thread and std::jthread~~
 - ~~Launching and joining threads~~
 - ~~Data Parallelism~~
- Synchronization Primitives
 - mutexes and mutex management
 - Condition Variables
 - semaphores
 - latches / barriers
- Promise and Futures
 - async
 - packaged_tasks
- Debugging concurrency
 - GDB and UDB.



Synchronization Primitives

Concurrency support library (C++11)

thread – jthread (C++20)

atomic – atomic_flag

atomic ref (C++20) – memory order

Mutual exclusion – Semaphores (C++20)

Condition variables – Futures

latch (C++20) – barrier (C++20)

Safe Reclamation (C++26)

Synchronization

- Not all of our problems are going to be 'data parallel'
- And even in the cases where we do have data parallelism, we cannot afford to keep spawning new threads
- This is where we are going to need more **fine-grained synchronization**!



Data Parallel - Each train/thread independent



Concurrent - Needs synchronization at station



Mutual Exclusion

Concurrency support library (C++11)

thread – jthread (C++20)

atomic – atomic_flag

atomic ref (C++20) – memory_order

Mutual exclusion – Semaphores (C++20)

Condition variables – Futures

latch (C++20) – barrier (C++20)

Safe Reclamation (C++26)

Mutual Exclusion

- A `std::mutex` is our first synchronization tool.
 - It helps us synchronize in the sense that only 1 operation can happen while the lock is held.
 - (Thus making that operation atomic)
- Thus a `std::mutex` enables ‘mutual exclusion’ to a block of code.
 - Thus, the operation is ‘atomic’ *Analogy:*
 - *Think about having exactly 1 key to your home, and you always carry the key with you.*
 - *Only the person who has the key can access the house.*
 - *When the person enters, they lock the door*
 - *When the person leaves, they can pass on the key to someone else to enter, who will also lock the door when they enter.*

std::mutex (1/2)

- Four new lines of code
 - #include <mutex> library
 - A global lock with std::mutex
 - A lock() and unlock() call on our global lock

```
1 // @file thread6.cpp
2 // g++ -std=c++17 thread6.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5 #include <vector>
6 #include <mutex> // New library
7
8 // Some shared value
9 static int shared_value = 0;
10 std::mutex gLock; // A global lock
11
12 void increment_shared_value(){
13     gLock.lock();
14     shared_value = shared_value + 1;
15     gLock.unlock();
16 }
17
18 int main() {
19     std::vector<std::thread> threads;
20     // Create a collection of threads
21     for(int i=0; i < 1000; i++){
22         threads.push_back(std::thread(increment_shared_value));
23     }
24     // Join our threads
25     for(int i=0; i < 1000; i++){
26         threads[i].join();
27     }
28     // Retrieve our result
29     std::cout << "Result = " << shared_value << std::endl;
30
31     return 0;
32 }
```

std::mutex (2/2)

- Four new lines of code
 - `#include <mutex>` library
 - A global lock with `std::mutex`
 - A `lock()` and `unlock()` call on our global lock

```
mike:concurrency$ g++-10 -std=c++17 thread6.cpp -o prog -lpthread
mike:concurrency$ ./prog
Result = 1000
mike:concurrency$ ./prog
Result = 1000
mike:concurrency$ ./prog
Result = 1000
mike:concurrency$ ./prog
Result = 1000
mike:concurrency$ ./prog
Result = 1000
```

```
1 // @file thread6.cpp
2 // g++ -std=c++17 thread6.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5 #include <vector>
6 #include <mutex> // New library
7
8 // Some shared value
9 static int shared_value = 0;
10 std::mutex gLock; // A global lock
11
12 void increment_shared_value(){
13     gLock.lock();
14     shared_value = shared_value + 1;
15     gLock.unlock();
16 }
17
18 int main() {
19     std::vector<std::thread> threads;
20     // Create a collection of threads
21     for(int i=0; i < 1000; i++){
22         threads.push_back(std::thread(increment_shared_value));
23     }
24     // Join our threads
25     for(int i=0; i < 1000; i++){
26         threads[i].join();
27     }
28     // Retrieve our result
29     std::cout << "Result = " << shared_value << std::endl;
30
31     return 0;
32 }
```

std::mutex and mutual exclusion (1/4)

- So what our lock is doing is providing access to only one thread at a time (mutually exclusive access).

```
12 void increment_shared_value(){  
13     gLock.lock();  
14     shared_value = shared_value + 1;  
15     gLock.unlock();  
16 }
```


std::mutex and mutual exclusion (2/4)

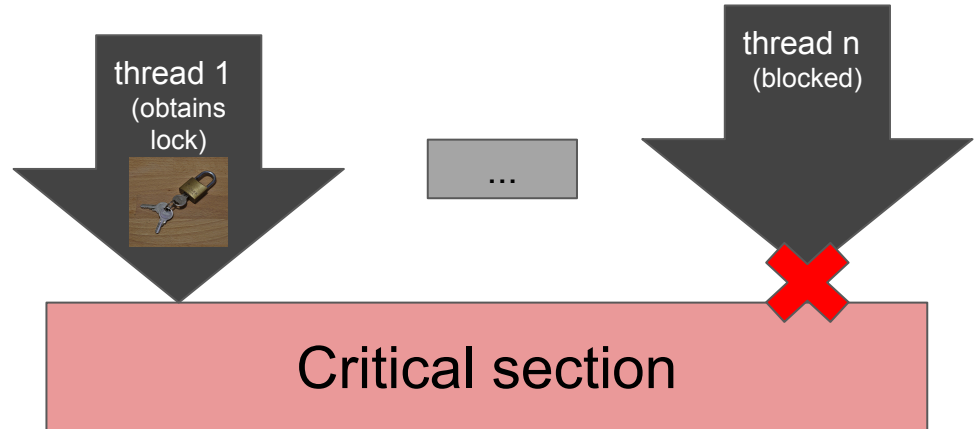
- So what our lock is doing is providing access to only one thread at a time (mutually exclusive access).
 - This region is called the 'critical section' that is protected by the lock.
 - Critical because we only want one thread at a time to enter and modify the shared state in the program.

```
12 void increment_shared_value(){  
13     gLock.lock();  
14     shared_value = shared_value + 1;  
15     gLock.unlock();  
16 }
```

std::mutex and mutual exclusion (3/4)

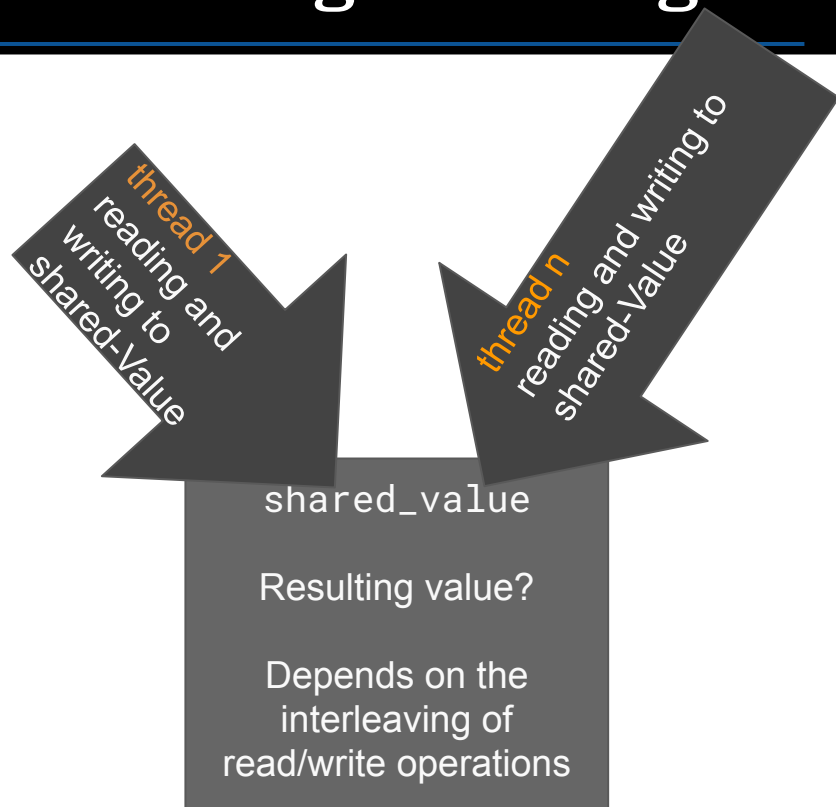
- So what our lock is doing is providing access to only one thread at a time (mutually exclusive access).
 - This region is called the 'critical section' that is protected by the lock.
 - Critical because we only want one thread at a time to enter and modify the shared state in the program.

```
12 void increment_shared_value(){  
13     gLock.lock();  
14     shared_value = shared_value + 1;  
15     gLock.unlock();  
16 }
```



(Aside) Problem with Threads -- Reading & Writing

- Data Race (or race condition)
 - Because data is shared--one or more thread could be writing to the same piece of memory at the same time
 - One thread may have read a 'stale' value right before the new 'write' to the value
 - The thread that then writes will update +1 to a stale value, overwriting the other threads update
 - This makes the operation non-deterministic
 - i.e. We may get unexpected or undefined results regarding the final value based on a non-deterministic order of operations



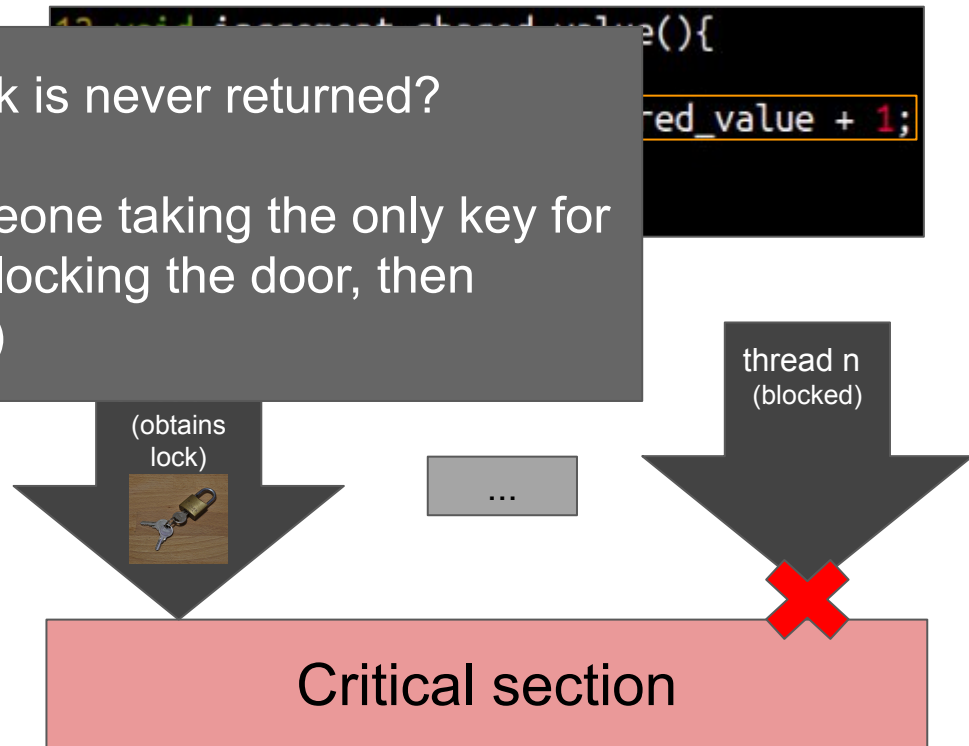
std::mutex and mutual exclusion (4/4)

- So what is provided by one thread (mutual access)

What happens if the lock is never returned?

(e.g., equivalent to someone taking the only key for your house, walking in, locking the door, then flushing key down toilet)

- The 'critical section' that is protected by the lock.
- Critical because we only want one thread at a time to enter and modify the shared state in the program.



Deadlock - lack of *any* progress for a thread (1/2)

- Deadlock
 - Is the prevention of a thread from ever acquiring a resource
 - Thus, no forward progress can be made (the thread waits forever)
 - This typically happens when a thread does not release a lock, and goes out of scope or otherwise terminates before releasing the lock

```
1 // @file thread7_deadlock.cpp
2 // g++ -std=c++17 thread7_deadlock.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5 #include <vector>
6 #include <mutex> // New library
7
8 // Some shared value
9 static int shared_value = 0;
10 std::mutex gLock; // A global lock
11
12 void increment_shared_value(){
13     gLock.lock();
14     shared_value = shared_value + 1;
15     // gLock.unlock(); // Oops, never return lock
16 }
```


Deadlock - lack of *any* progress for a thread (2/2)

- Fixing deadlock
 - Re-run code, and see if you are missing a pair of lock/unlock
 - Static analysis techniques (i.e. thread sanitizers) may detect deadlock before compilation.
 - Otherwise deadlock has to be carefully detected at run-time and fixed.
- Note: Deadlock is the most extreme form of starvation
 - Starvation is when a thread cannot fairly acquire access to a resource

```
1 // @file thread7_deadlock.cpp
2 // g++ -std=c++17 thread7_deadlock.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread>    // Include the thread library
5 #include <vector>
6 #include <mutex>    // New library
7
8 // Some shared value
9 static int shared_value = 0;
10 std::mutex gLock; // A global lock
11
12 void increment_shared_value(){
13     gLock.lock();
14     shared_value = shared_value + 1;
15 //     gLock.unlock(); // Oops, never return lock
16 }
```

Careful with std::mutex (1/2)


- So let's make sure we have a lock for every unlock
 - Our code is fixed right?
 - (I agree this looks correct)
- The problem is if another programmer comes and updates line 14

```
1 // @file thread7_deadlock.cpp
2 // g++ -std=c++17 thread7_deadlock.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread>    // Include the thread library
5 #include <vector>
6 #include <mutex>    // New library
7
8 // Some shared value
9 static int shared_value = 0;
10 std::mutex gLock; // A global lock
11
12 void increment_shared_value(){
13     gLock.lock();
14     shared_value = shared_value + 1;
15     gLock.unlock(); // Fixed right?
16 }
```

Careful with std::mutex (2/2)

- So let's make sure we have a lock for every unlock
 - Our code is fixed right?
 - (I agree this looks correct)

```
1 // @file thread7_deadlock.cpp
2 // g++ -std=c++17 thread7_deadlock.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5 #include <vector>
6 #include <mutex> // New library
7
8 // Some shared value
9 static int shared_value = 0;
10 std::mutex gLock; // A global lock
11
12 void increment_shared_value(){
13     gLock.lock();
14     shared_value = shared_value + 1;
15     gLock.unlock(); // Fixed right?
16 }
```



```
12 void increment_shared_value(){
13     gLock.lock();
14     try{
15         shared_value = shared_value + 1;
16         throw "Dangerous exception abort";
17     }catch(...){
18         std::cout << "handle exception by returning from thread\n";
19         return;
20     }
21     gLock.unlock(); // Oops, never return lock
22 }
```

Maybe our object can throw an exception, or a programmer updates to the following

So this code will also deadlock! Consider the more complex case where some 'exception' is thrown and we 'forget' to also release the lock in catch.

You *could still* remember to use a lock, but we have a better tool

Prefer lock_guard (C++11) over lock/unlock (1/2)

- We instead of a `lockGuard` that can 'wrap' an individual `std::mutex`
 - The destructor of `lock_guard` will take care of releasing the lock
- `std::lock_guard` is a good example of RAII
 - `lock_guard` takes ownership of the lock, and when we leave scope the mutex is released (and the `lock_guard` destroyed)

```

1 // @file thread9.cpp
2 // g++ -std=c++17 thread9.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5 #include <vector>
6 #include <mutex> // New library
7
8 // Some shared value
9 static int shared_value = 0;
10 std::mutex gLock; // A global lock
11
12 void increment_shared_value(){
13     // lock_guard follows RAII principles and will
14     // release lock after leaving scope.
15     // This includes if an exception is thrown.
16     std::lock_guard<std::mutex> lockGuard(gLock);
17     try{
18         shared_value = shared_value + 1;
19         throw "Dangerous exception abort";
20     }catch(...){
21         std::cout << "handle exception by returning from thread\n";
22         return;
23     }
24 }

```

[illegible]

Prefer lock_guard (C++11) over lock/unlock (2/2)

- We instead of a lockGuard that can 'wrap' an individual std::mutex
 - The destructor of lock_guard will take care of releasing the lock

```
1 // @file thread9.cpp
2 // g++ -std=c++17 thread9.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5 #include <vector>
6 #include <mutex> // New library
7
8 // Some shared value
9 static int shared_value = 0;
10 std::mutex gLock; // A global lock
11
12 void increment_shared_value(){
13     // lock_guard follows RAII principles and will
14     // release lock after leaving scope.
15     // This includes if an exception is thrown.
16     std::lock_guard<std::mutex> lockGuard(gLock);
17     try{
18         shared_value = shared_value + 1;
19         throw "Dangerous exception abort";
20     }
21 }
```

[std::lock_guard](#) is only 3 member functions

Member functions

(constructor)	constructs a lock_guard, optionally locking the given mutex (public member function)
(destructor)	destructs the lock_guard object, unlocks the underlying mutex (public member function)
operator=[deleted]	not copy-assignable (public member function)

```
handle exception by returning from thread
Result = 1000
```


std::scoped_lock - Other mechanisms

- [std::scoped_lock](#) (C++17) -
 - An update to `lock_guard`, but can acquire multiple locks at once
 - i.e., `std::scoped_lock(mutex1, mutex2);`
 - Prefer `scoped_lock` (over `lock_guard`) if you are able to utilize C++17.

Member functions

(constructor)	constructs a <code>scoped_lock</code> , optionally locking the given mutexes (public member function)
(destructor)	destructs the <code>scoped_lock</code> object, unlocks the underlying mutexes (public member function)
operator= [deleted]	not copy-assignable (public member function)

std::unique_lock - Other mechanisms

- [std::unique_lock](#) (C++11) -
 - A bit more powerful than `lock_guard` and `scoped_guard` in that we can control locking and unlocking
 - Used in `condition_variable` (coming up)
 - Also follows RAII so we can use it safely.

Member functions

(constructor)	constructs a <code>unique_lock</code> , optionally (public member function)
(destructor)	unlocks (i.e., releases ownership) (public member function)
operator=	unlocks (i.e., releases ownership) (public member function)

Locking

lock	locks (i.e., takes ownership of) the mutex (public member function)
try_lock	tries to lock (i.e., takes ownership) (public member function)
try_lock_for	attempts to lock (i.e., takes ownership) if the mutex has been unavailable for at least the specified duration (public member function)
try_lock_until	tries to lock (i.e., takes ownership) if the mutex has been unavailable until specified time (public member function)
unlock	unlocks (i.e., releases ownership) (public member function)

Modifiers

swap	swaps state with another <code>std::unique_lock</code> (public member function)
release	disassociates the associated mutex (public member function)

Observers

mutex	returns a pointer to the associated mutex (public member function)
owns_lock	tests whether the lock owns (i.e., holds) the mutex (public member function)
operator bool	tests whether the lock owns (i.e., holds) the mutex (public member function)

There exist several other primitives you can find here

- https://en.cppreference.com/w/cpp/thread#Mutual_exclusion
 - e.g. `timed_lock`, `recursive_mutex`, `shared_mutex`, etc.

Mutual exclusion	
Mutual exclusion algorithms prevent multiple threads from simultaneously accessing shared resources. This prevents data races and provides support for synchronization between threads.	
Defined in header <code><mutex></code>	
<code>mutex</code> (C++11)	provides basic mutual exclusion facility (class)
<code>timed_mutex</code> (C++11)	provides mutual exclusion facility which implements locking with a timeout (class)
<code>recursive_mutex</code> (C++11)	provides mutual exclusion facility which can be locked recursively by the same thread (class)
<code>recursive_timed_mutex</code> (C++11)	provides mutual exclusion facility which can be locked recursively by the same thread and implements locking with a timeout (class)
Defined in header <code><shared_mutex></code>	
<code>shared_mutex</code> (C++17)	provides shared mutual exclusion facility (class)
<code>shared_timed_mutex</code> (C++14)	provides shared mutual exclusion facility and implements locking with a timeout (class)
Generic mutex management	
Defined in header <code><mutex></code>	
<code>lock_guard</code> (C++11)	implements a strictly scope-based mutex ownership wrapper (class template)
<code>scoped_lock</code> (C++17)	deadlock-avoiding RAIL wrapper for multiple mutexes (class template)
<code>unique_lock</code> (C++11)	implements movable mutex ownership wrapper (class template)
<code>shared_lock</code> (C++14)	implements movable shared mutex ownership wrapper (class template)
<code>defer_lock_t</code> (C++11) <code>try_to_lock_t</code> (C++11) <code>adopt_lock_t</code> (C++11)	tag type used to specify locking strategy (class)
<code>defer_lock</code> (C++11) <code>try_to_lock</code> (C++11) <code>adopt_lock</code> (C++11)	tag constants used to specify locking strategy (constant)

Topics Outline

- ~~Transport Tycoon~~
- ~~std::thread and std::jthread~~
 - ~~Launching and joining threads~~
 - ~~Data Parallelism~~
- Synchronization Primitives
 - ~~mutexes and mutex management~~
 - Condition Variables
 - semaphores
 - latches
 - barriers
- Promise and Futures
 - async
 - packaged_tasks
- Debugging concurrency
 - GDB and UDB.

Condition variables

A condition variable is a synchronization primitive that allows multiple threads to communicate with each other. It allows some number of threads to wait (possibly with a timeout) for notification from another thread that they may proceed. A condition variable is always associated with a mutex.

Defined in header `<condition_variable>`

<code>condition_variable</code> (C++11)	provides a condition variable associated with a <code>std::unique_lock</code> (class)
<code>condition_variable_any</code> (C++11)	provides a condition variable associated with any lock type (class)
<code>notify_all_at_thread_exit</code> (C++11)	schedules a call to <code>notify_all</code> to be invoked when this thread is completely finished (function)
<code>cv_status</code> (C++11)	lists the possible results of timed waits on condition variables (enum)

Condition Variables

A way to signal an event between 2 or more threads

Concurrency support library (C++11)

`thread` – `jthread` (C++20)

`atomic` – `atomic_flag`

`atomic_ref` (C++20) – `memory_order`

Mutual exclusion – Semaphores (C++20)

Condition variables – Futures

`latch` (C++20) – `barrier` (C++20)

Safe Reclamation (C++26)

Introducing Condition Variables

- Condition variables
 - Allows us to keep threads alive -- **without having to spawn new threads**,
 - We have discovered that otherwise creating threads is expensive
- The use case for condition variables
 - Dispatch work to worker threads periodically in order to do work on a subset of data.
- Condition variables have two main methods:
 - wait and notify.
 - wait will stop the calling thread and put it to sleep
 - notify will awaken the thread
- Thus a condition variable is often used as an efficient **‘signaling pattern’**

Condition Variables Example

- A condition variable allows us to otherwise ‘signal’ from one function to the other when there is work to be done.
 - A common pattern is the **producer/consumer** pattern
 - When data is ‘produced’ then a signal is made that work is ready to be acquired and processed by a ‘consumer’ thread.

```
13 // Global state that can be accessed
14 // (Otherwise could be in a struct/class)
15 std::mutex          shared_lock_between_producerconsumer;
16 std::condition_variable cv;
17 bool               ready {false};
18 std::queue<int>     shared_queue;
```

Observe that we need three parts:

1. a mutex (for synchronization)
2. a condition_variable
 - a. To ‘awaken’ another thread
3. a ‘variable’ (e.g. ready)
 - a. This variable will be protected by the lock

Condition Variables Example (producer)

- The job of the producer is to do some work on a protected piece of data
 - (Note `std::lock_guard` with locking safely through RAI)
- It's worth noting also at this point that our 'consumer' will be blocked until 'notified' (See `notify_all`)

```
13 // Global state that can be accessed
14 // (Otherwise could be in a struct/class)
15 std::mutex          shared_lock_between_producerconsumer;
16 std::condition_variable cv;
17 bool               ready {false};
18 std::queue<int>     shared_queue;
```

```
20
21 // Producers goal is to otherwise 'add' or 'modify' data
22 static void producer() {
23
24     for(int i=0; i < 5; i++)
25     {
26         std::this_thread::sleep_for(250ms);
27         {
28             std::lock_guard<std::mutex> lk {shared_lock_between_producerconsumer};
29             // Do some interesting work here
30             // Note: We have 'locked' the 'shared' portion of data
31             shared_queue.push(i);
32         }
33         // Something interesting has happened, so notify the conditoinal variable
34         // Effectively -- wake all threads
35         cv.notify_all();
36     }
37
38     {
39         std::lock_guard<std::mutex> lk {shared_lock_between_producerconsumer};
40         ready = true;
41     }
42     cv.notify_all();
43 }
```

Condition Variables Example (consumer)

- Here's the consumer side
- The consumer 'diligently waits' to acquire the lock
- The 'wait' portion otherwise is where we awaken when we are notified by the producer.
 - If 'ready' is 'false' then we release the lock and wait here blocked -- but 'we fall asleep' (rather than have a spin lock)
 - If the condition is otherwise true, then we awaken when we're notified and acquire the lock to proceed forward.

```
13 // Global state that can be accessed
14 // (Otherwise could be in a struct/class)
15 std::mutex          shared_lock_between_producerconsumer;
16 std::condition_variable cv;
17 bool                ready {false};
18 std::queue<int>      shared_queue;
```

```
45 // Consumer thread
46 // Usual goal to 'consume' data
47 static void consumer() {
48     while (!ready) {
49         std::unique_lock<std::mutex> l {shared_lock_between_producerconsumer};
50         cv.wait(l, [] { return !shared_queue.empty() || ready; });
51     }
52     std::print("Consuming new value from shared_queue: {}", shared_queue.front());
53     shared_queue.pop();
54 }
55 }
```

Condition_variable with thread pool -- what's the point?

- We went from a data parallel problem to a more efficient 'thread pool'
 - The 'data parallel' problem *may* or *may not* need to reuse threads -- perhaps crunching numbers is just fine
 - However -- it's useful to know how to reimplement some of these systems.
- The point of the mechanism (i.e. a conditional variable) is to understand this 'signal pattern' is going to be we now have a mechanism to 'block' our threads when executing
 - They can then 'pick up' work, or be assigned new work when needed.
 - We could have implemented this with a 'mutex' and 'while-loop' as well, but it gets clunky, and it's inefficient to constantly check if we should advance (i.e. this is a 'spin-lock')

(Aside) `condition_variable`

- Tempting to just use a lock and flags, however that often results in you implementing a 'spin lock'
- `condition_variable` is more efficient than spin locks
 - A spin lock wastes lots of CPU cycles constantly running a loop

Topics Outline

- ~~Transport Tycoon~~
- ~~std::thread and std::jthread~~
 - ~~Launching and joining threads~~
 - ~~Data Parallelism~~
- Synchronization Primitives
 - ~~mutexes and mutex management~~
 - ~~Condition Variables~~
 - semaphores
 - latches / barriers
- Promise and Futures
 - async
 - packaged_tasks
- Debugging concurrency
 - GDB and UDB.

Semaphores

A semaphore is a lightweight synchronization primitive used to constrain concurrent access to a shared resource. When either would suffice, a semaphore can be more efficient than a condition variable.

Defined in header `<semaphore>`

counting_semaphore (C++20)	semaphore that models a non-negative resource count (class template)
binary_semaphore (C++20)	semaphore that has only two states (typedef)

Semaphores

An even more primitive way to signal

Concurrency support library (C++11)

thread – jthread (C++20)

atomic – atomic_flag

atomic_ref (C++20) – memory_order

Mutual exclusion – Semaphores (C++20)

Condition variables – Futures

latch (C++20) – barrier (C++20)

Safe Reclamation (C++26)

(1/2) Let's look at that word semaphore that was in the talk today :)

The Little Talk of Semaphores -- and a Tour of C++ Concurrency

with Mike Shah

(2/2) And we'd probably do well to define the word as well.

The Little Talk of **Semaphores** -- and a Tour of C++ Concurrency

with Mike Shah

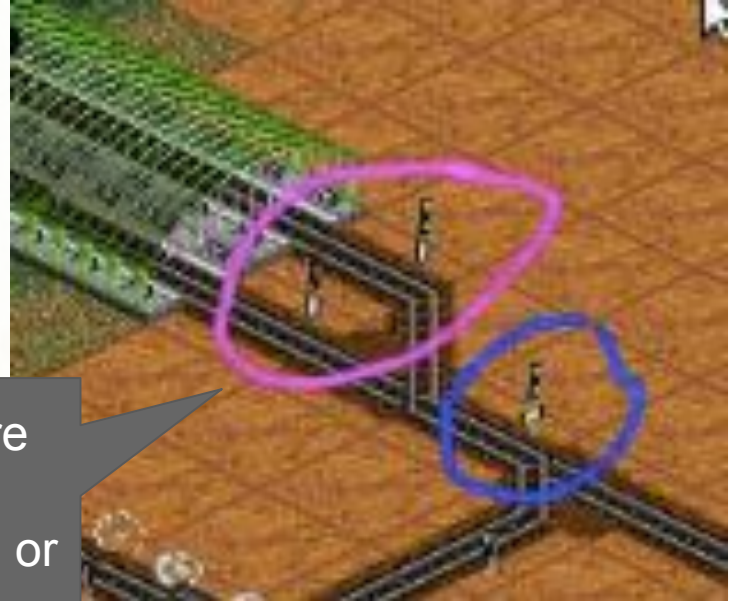
Semaphore Wiki Definition

Semaphore (lit. 'apparatus for signalling'; from **Ancient Greek** **σῆμα** (*sêma*) 'mark, sign, token' and **Greek** **-φόρος** (*-phóros*) 'bearer, carrier')^[1] is the use of an apparatus to create a visual **signal** transmitted over distance.^{[2][3]} A semaphore can be performed with devices including: fire, lights, **flags**, **sunlight**, and moving arms.^{[2][3][4]} Semaphores can be used for **telegraphy** when arranged in **visually connected networks**, or for traffic signalling such as in **railway systems**, or **traffic lights** in cities.^[5]



A semaphore is a signal -- just like what we saw before

Semaphore (lit. '**apparatus for signalling**'; from Ancient Greek *σῆμα* (*sêma*) 'mark, sign, token' and Greek *-φόρος* (*-phóros*) 'bearer, carrier')^[1] is the use of an apparatus to create a visual signal transmitted over distance.^{[2][3]} A semaphore can be performed with devices including: fire, lights, flags, sunlight, and moving arms. It can be used for telegraph or connected networks in railway systems, or as a signal indicating that a train can or cannot pass through.



Each signal here is a semaphore -- 'raising' and 'lowering' of the signal indicating that a train can or cannot pass through

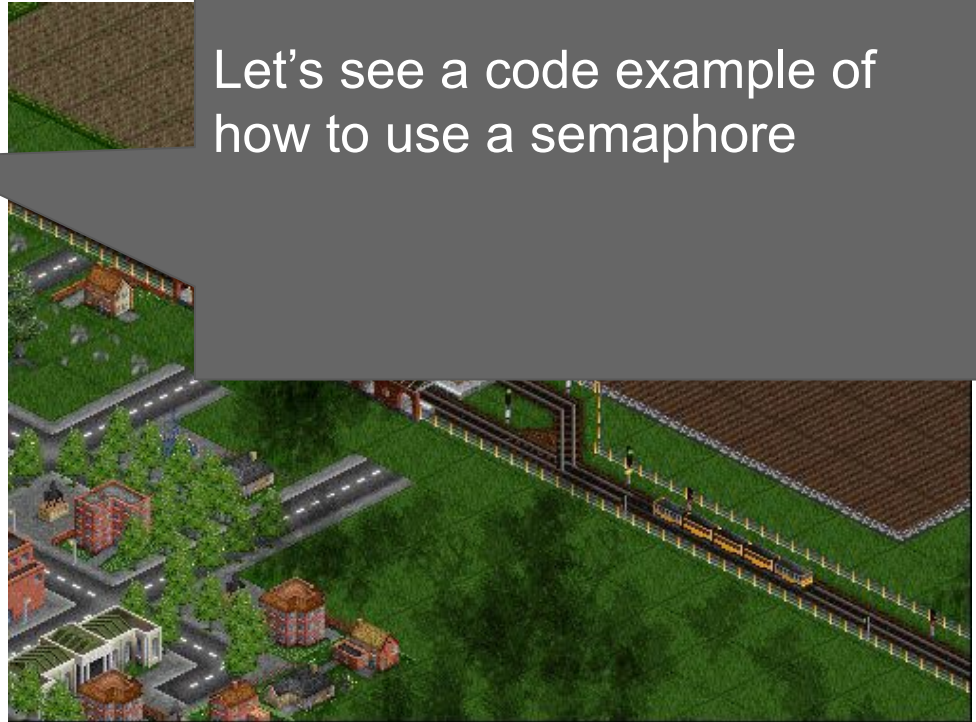
In our analogy (1/2)

- **trains === worker threads**
 - Trains spend time doing something useful (transportation)
- **station === shared memory**
 - This is where we 'drop off' (write) or 'pick up' (read) data
- **signal === semaphore**
 - A primitive for making sure our shared resource (train tracks) are owned by only 1 train at a time.



In our analogy (2/2)

- **trains === worker threads**
 - Trains spend time doing something useful (transportation)
- **station === shared memory**
 - This is where we 'drop off' (write) or 'pick up' (read) data
- **signal === semaphore**
 - A primitive for making sure our shared resource (train tracks) are owned by only 1 train at a time.



binary_semaphore

- Provided is an example of a binary_semaphore
- binary_semaphore holds a value of 1 or 0 (available or unavailable)
 - It looks a lot like a std::mutex, in that we acquire/release -- but it's just a signal
 - No real concept of a thread 'owning a lock' for a scope
 - *Could* be used in similar manner, but is more lightweight.

```
1 // @file semaphore1.cpp
2 // g++ -std=c++23 semaphore1.cpp -o prog -lpthread
3 #include <vector>
4 #include <chrono>
5 #include <print>
6 #include <thread> // Include the thread library
7 #include <semaphore> // semaphore for synchronization
8
9 std::binary_semaphore gSem(0);
10
11 void WorkerThread(int arg){
12     // Acquire 'decrements' the counter associated
13     // with the semaphore, such that '1' less
14     // piece of work can pass by.
15     gSem.acquire();
16     size_t tid = std::hash<std::thread::id>{}(std::this_thread::get_id())
17     std::println("thread.get id:{}", tid);
18     std::println("Argument passed in:{}", arg);
19     gSem.release();
20 }
21
22 int main() {
23     // Launch a thread
24     std::jthread j(WorkerThread,10);
25     // Proceed to do any useful work
26     std::println("Hello from the main thread!");
27
28     // Our worker thread is always blocked until
29     // we 'release' or 'signal' that we are ready.
30     // 'release' increments our counter, making available one more
31     // 'slot' to acquire from the semaphore.
32     std::this_thread::sleep_for(std::chrono::seconds(1));
33     gSem.release();
34
35     gSem.acquire();
36     // Continue executing the main thread
37     std::println("Program ending");
38     gSem.release();
39
40     return 0;
41 }
```

Thread Safe Data Structures

ThreadSafe Queue

- At this point, it's probably a good idea to start building some abstractions
- Here's a *very trivial* thread-safe queue we can make it safer or faster later
 - e.g. Perhaps make this lockless
 - e.g. Perhaps 'reader/writer' pattern
 - e.g. Move any work that does not require synchronization to before the lock
- Note: This creates a 'shared data structure'
 - Depending on the task, we might have to be careful that this introduces a bottleneck

[illegible]

Topics Outline

- ~~Transport Tycoon~~
- ~~std::thread and std::jthread~~
 - ~~Launching and joining threads~~
 - ~~Data Parallelism~~
- Synchronization Primitives
 - ~~mutexes and mutex management~~
 - ~~Condition Variables~~
 - ~~semaphores~~
 - latches / barriers
- Promise and Futures
 - async
 - packaged_tasks
- Debugging concurrency
 - GDB and UDB.

Latches and Barriers

Grouping threads

Concurrency support library (C++11)

thread – jthread (C++20)

atomic – atomic_flag

atomic_ref (C++20) – memory_order

Mutual exclusion – Semaphores (C++20)

Condition variables – Futures

latch (C++20) – barrier (C++20)

Safe Reclamation (C++26)

Latch

- I'll demonstrate now a `std::latch` with our thread safe queue
- A latch is like a semaphore, but it only counts down
 - It's a 'one time only' data structure otherwise (i.e. not reusable)
- A `std::barrier` on the other hand is reusable

```
1 // @file thread_safe_queue.cpp
2 // g++ -std=c++23 thread_safe_queue.cpp -o prog -lpthread
3 #include <print>
4 #include <vector>
5 #include <thread> // Include the thread library
6 #include <mutex>
7 #include <queue>
8 #include <latch>
9
10 //std::counting_semaphore latch(100); // No need to manage counting semaphore.
11 // This example uses a latch which 'counts down' until
12 // finished.
13
14 +--- 32 lines: Thread-Safe namespace for versions of containers.-----
46
47 int main() {
48     std::println();
49
50     const int problemSize = 100;
51     ThreadSafe::Queue q;
52     std::latch jobsToComplete(problemSize);
53
54     // This time create a lambda function
55     auto lambda = [&](int x){
56         q.push(x);
57         // Decrement each time we finish a job just before
58         // termination.
59         jobsToComplete.count_down();
60     };
61
62     // Note: We now have a jthread
63     // No joins in the program
64     std::vector<std::jthread> threads;
65     // Create a collection of threads
66     for(size_t i=0; i < problemSize; ++i){
67         threads.push_back(std::jthread(lambda,i));
68     }
69
70     // Block until we have finished all of our jobs
71     jobsToComplete.wait();
72
73     for(size_t i=0; i < problemSize; ++i){
74         std::print("{},".q.front());
75         q.pop();
76     }
77
78     // Continue executing the main thread
79     std::println("\n\nEnd of main thread!");
80
81     return 0;
82 }
```

Topics Outline

- ~~Transport Tycoon~~
- ~~std::thread and std::jthread~~
 - ~~Launching and joining threads~~
 - ~~Data Parallelism~~
- ~~Synchronization Primitives~~
 - ~~mutexes and mutex management~~
 - ~~Condition Variables~~
 - ~~semaphores~~
 - ~~latches~~
 - ~~barriers~~
- Promise and Futures
 - async
 - packaged_tasks
- Debugging concurrency
 - GDB and UDB.

Futures

The standard library provides facilities to obtain values that are returned and to catch exceptions that are thrown by asynchronous tasks (i.e. functions launched in separate threads). These values are communicated in a *shared state*, in which the asynchronous task may write its return value or store an exception, and which may be examined, waited for, and otherwise manipulated by other threads that hold instances of `std::future` or `std::shared_future` that reference that shared state.

Futures

A way to signal an event between 2 or more threads

Concurrency support library (C++11)

`thread` – `jthread` (C++20)

`atomic` – `atomic_flag`

`atomic_ref` (C++20) – `memory_order`

Mutual exclusion – Semaphores (C++20)

Condition variables – **Futures**

`latch` (C++20) – `barrier` (C++20)

Safe Reclamation (C++26)

Asynchronous Programming

Another form concurrency where execution can happen independently of the main program flow

Asynchronous means that events happen ‘without synchronicity’ or ‘without order’.

std::async

- We can throw away much of the signaling with std::async
- The idea is we execute a callable asynchronously, and we are only blocked if we are awaiting the result (in a future)
 - Note: We will still need locks for shared resources however -- that remains true!
- Good way to use threads
 - Partition to I/O bound tasks and CPU bound tasks (or perhaps even GPU)
 - Goal is to avoid blocking

std::async

Defined in header <future>

```
template< class F, class... Args >
std::future< /* see below */> async( F&& f, Args&&... args );
```

(1) (since C++11)

```
template< class F, class... Args >
std::future< /* see below */> async( std::launch policy, F&& f, Args&&... args );
```

(2) (since C++11)

The function template std::async runs the function `f` asynchronously (potentially in a separate thread which might be a part of a thread pool) and returns a `std::future` that will eventually hold the result of that function call.

std::async example

- `#include <future>`
- `std::async`
 - [Promise](#) and Future
 - Promise - Will hold the result
 - Future - Where the future result will be stored
 - We are blocked at `a.get()` until the value is returned.

```
1 // @file thread10.cpp
2 // g++ -std=c++17 thread10.cpp -o prog -lpthread
3 #include <iostream>
4 #include <future>
5
6 int square(int x){
7     return x*x;
8 }
9
10 int main() {
11     // asyncFunction is a 'future'
12     // type = std::future<int>
13     auto asyncFunction = std::async(&square,12);
14     // .... some time passes
15     int result = asyncFunction.get(); // We are blocked here if
16                                     // our value has not been
17                                     // computed. Otherwise, the
18                                     // value from get() (which
19                                     // is wrapped in a promise
20                                     // is returned.
21
22     std::cout << "The async thread has returned! " << result
23               << std::endl;
24
25     return 0;
26 }
27
```

1,1

```
mike:concurrency$ g++-10 -std=c++17 thread10.cpp -o prog -lpthread
mike:concurrency$ ./prog
The async thread has returned! 144
```


A Concrete Example for `std::async`

- Blocking Input/Output (I/O)
 - I/O is any task where we are reading or writing data.
 - e.g. network connection (e.g. downloading data), disk load (e.g. opening a file)
 - We can use a 'background thread' (i.e., `std::async`) execute to start loading that data.
 - The application can then proceed unblocked until it needs that data.
 - If we do not have the data ready when we need, we are thus 'blocked' -- hence the term Blocking I/O



Async I/O Simulation (1/5)

- “mocked” version of using an async thread to load data
 - We spawn a ‘background thread’ asynchronously using `std::async`
 - Then in our ‘main loop’ we continuously query to see if our function has returned
 - (I have added a few artificial sleeps to make it more interesting)

```
1 // @file async.cpp
2 // g++ -std=c++17 async.cpp -o prog -lpthread
3 #include <iostream>
4 #include <future>
5 #include <thread>
6 #include <chrono>
7
8 // Toy example - This will be our 'background thread'
9 //               that executes asynchronously
10 bool bufferedFileLoading(){
11     size_t bytesLoaded= 0;
12     while(bytesLoaded < 20000){
13         std::cout << "Loading File..." << std::endl;
14         std::this_thread::sleep_for(std::chrono::milliseconds(250));
15         bytesLoaded+= 1000;
16     }
17     return true;
18 }
19
20 int main() {
21     // Launch thread asynchronously, and this will execute in background
22     std::future<bool> backgroundThread = std::async(std::launch::async,
23                                                    bufferedFileLoading());
24
25     // Store status of our future
26     std::future_status status;
27     // Meanwhile, we have our main thread of execution
28     while(true){
29         std::cout << "Main thread running" << std::endl;
30         // artificial pause
31         std::this_thread::sleep_for(std::chrono::milliseconds(50));
32         // Check if our
33         status = backgroundThread.wait_for(std::chrono::milliseconds(1));
34         // If our data is ready--do something
35         // we'll just terminate for now
36         if(status == std::future_status::ready){
37             std::cout << "data ready..." << std::endl;
38             break;
39         }
40     }
41
42     return 0;
43 }
```

Asy

Here we'll create a background thread that will execute with `std::async`

I've been explicit in setting up the parameters and types.

Also, there is a 'status' that we'll keep track of so we know when a value has been returned

```
1 // @file async.cpp
2 // g++ -std=c++17 async.cpp -o prog -lpthread
3 #include <iostream>
4 #include <future>
5 #include <thread>
6 #include <chrono>
7
8 // Toy example - This will be our 'background thread'
9 // that executes asynchronously
10 bool bufferedFileLoading(){
11     size_t bytesLoaded= 0;
12     while(bytesLoaded < 20000){
13         std::cout << "Loading File..." << std::endl;
14         std::this_thread::sleep_for(std::chrono::milliseconds(250));
15         bytesLoaded+= 1000;
16     }
17     return true;
18 }
19
20 int main() {
21     // Launch thread asynchronously, and this will execute in background
22     std::future<bool> backgroundThread = std::async(std::launch::async,
23                                                    bufferedFileLoading());
24     // Store status of our future
25     std::future_status status;
26     // Meanwhile, we have our main thread of execution
27     while(true){
28         std::cout << "Main thread running" << std::endl;
29         // artificial pause
30         std::this_thread::sleep_for(std::chrono::milliseconds(50));
31         // Check if our
32         status = backgroundThread.wait_for(std::chrono::milliseconds(1));
33         // If our data is ready--do something
34         // we'll just terminate for now
35         if(status == std::future_status::ready){
36             std::cout << "data ready..." << std::endl;
37             break;
38         }
39     }
40
41     return 0;
42 }
```

Here we are reading in 'bytes' from a file.

Perhaps we are 'streaming' in some # of bytes from a data source

- thread' asynchronously using `std::async`
- Then in our 'main loop' we continuously query to see if our function has returned
- (I have added a few artificial sleeps to make it more interesting)

```
1 // @file async.cpp
2 // g++ -std=c++17 async.cpp -o prog -lthread
3 #include <iostream>
4 #include <future>
5 #include <thread>
6 #include <chrono>
7
8 // Toy example - This will be our 'background thread'
9 // that executes asynchronously
10 bool bufferedFileLoading(){
11     size_t bytesLoaded= 0;
12     while(bytesLoaded < 20000){
13         std::cout << "Loading File..." << std::endl;
14         std::this_thread::sleep_for(std::chrono::milliseconds(250));
15         bytesLoaded+= 1000;
16     }
17     return true;
18 }
19
20 int main() {
21     // Launch thread asynchronously, and this will execute in background
22     std::future<bool> backgroundThread = std::async(std::launch::async,
23                                                    bufferedFileLoading());
24
25     // Store status of our future
26     std::future_status status;
27     // Meanwhile, we have our main thread of execution
28     while(true){
29         std::cout << "Main thread running" << std::endl;
30         // artificial pause
31         std::this_thread::sleep_for(std::chrono::milliseconds(50));
32         // Check if our
33         status = backgroundThread.wait_for(std::chrono::milliseconds(1));
34         // If our data is ready--do something
35         // we'll just terminate for now
36         if(status == std::future_status::ready){
37             std::cout << "data ready..." << std::endl;
38             break;
39         }
40     }
41
42     return 0;
43 }
```

Async I/O Simulation (4/5)

- “mocked” version of using an async thread to load data
 - We spawn a ‘background thread’ asynchronously using

In our main loop we will check every 1 millisecond the status of our future value (which is wrapped in a promise object)

```
1 // @file async.cpp
2 // g++ -std=c++17 async.cpp -o prog -lthread
3 #include <iostream>
4 #include <future>
5 #include <thread>
6 #include <chrono>
7
8 // Toy example - This will be our 'background thread'
9 // that executes asynchronously
10 bool bufferedFileLoading(){
11     size_t bytesLoaded= 0;
12     while(bytesLoaded < 20000){
13         std::cout << "Loading File..." << std::endl;
14         std::this_thread::sleep_for(std::chrono::milliseconds(250));
15         bytesLoaded+= 1000;
16     }
17     return true;
18 }
19
20 int main() {
21     // Launch thread asynchronously, and this will execute in background
22     std::future<bool> backgroundThread = std::async(std::launch::async,
23                                                    bufferedFileLoading());
24
25     // Store status of our future
26     std::future_status status;
27     // Meanwhile, we have our main thread of execution
28     while(true){
29         std::cout << "Main thread running" << std::endl;
30         // artificial pause
31         std::this_thread::sleep_for(std::chrono::milliseconds(50));
32         // Check if our
33         status = backgroundThread.wait_for(std::chrono::milliseconds(1));
34         // If our data is ready--do something
35         // we'll just terminate for now
36         if(status == std::future_status::ready){
37             std::cout << "data ready..." << std::endl;
38             break;
39         }
40     }
41
42     return 0;
43 }
```


Async I/O Simulation (5/5)

Here's what our execution looks like

(Again--code available from github)

1

```
1 // @file async.cpp
2 // g++ -std=c++11 -pthread -std::filesystem
3 #include <iostream>
4 #include <future>
5 #include <thread>
6 #include <chrono>
```

```
example
```

```
ffectedEi
```

```
std::
std::ti
bytesLo
```

```
urn true;
```

```
n() {
```

```
Launch thread asynchronously, and this will execute in background
::future<bool> backgroundThread = std::async(std::launch::async,
                                             bufferedFileLoading);
```

```
Store status of our future
```

```
::future_status status;
```

```
Meanwhile, we have our main thread of execution
```

```
le(true){
```

```
std::cout << "Main thread running" << std::endl;
```

```
// artificial pause
```

```
std::this_thread::sleep_for(std::chrono::milliseconds(50));
```

```
// Check if our
```

```
status = backgroundThread.wait_for(std::chrono::milliseconds(1));
```

```
// If our data is ready--do something
```

```
// we'll just terminate for now
```

```
if(status == std::future_status::ready){
```

```
std::cout << "data ready..." << std::endl;
```

```
break;
```

```
}
```

```
41 return 0;
```

```
42 }
```


packaged_task

- A packaged task is a ‘generalization’ or building block for `std::async`
- Provided in this example I ‘package up’ some work to be done in a ‘task’
 - The task is invoked, and starts executing concurrently
 - We’re blocked until the result is otherwise ready

```
1 // @file packaged_task.cpp
2 +-- 11 lines: // Demonstrates packaged_task, which effectively is what you co
13
14 // This is the 'work unit' that we'll wrap in a packaged_task later on
15 int SomeWork(int x, int y){
16     std::this_thread::sleep_for(std::chrono::seconds(3));
17     return x*x + y*y;
18 }
19
20 int main() {
21     std::println("\nHello from the main thread!");
22
23     // Start up some other work
24     std::packaged_task<int> task(std::bind(SomeWork,1,2));
25
26     // Prepa
27     std::future<int> result = task.get_future();
28
29 +-- 8 lines: Call the task-----
37     std::thread myThread(std::move(task));
38
39     std::println("Proceeding to do some work");
40     std::println("Proceeding to do some work");
41     std::println("Proceeding to do some work");
42     std::println("Proceeding to do some work");
43
44     // Blocked here until we get result
45     // Note: It's important that we invoke (i.e. call the 'task()' somewhere
46     //       before we try to retrieve our future.
47     std::println("Got result {}",result.get());
48
49     // Make sure we join our thread
50     myThread.join();
51
52     std::println("End of program");
53
54     return 0;
55 }
```

Topics Outline

- ~~Transport Tycoon~~
- ~~std::thread and std::jthread~~
 - ~~Launching and joining threads~~
 - ~~Data Parallelism~~
- ~~Synchronization Primitives~~
 - ~~mutexes and mutex management~~
 - ~~Condition Variables~~
 - ~~semaphores~~
 - ~~latches / barriers~~
- ~~Promise and Futures~~
 - ~~asynce~~
 - ~~packaged_tasks~~
- Debugging concurrency
 - GDB and UDB.

Troubleshooting and Debugging Concurrency

Let's see the program run!

Live GDB: Conditional Variable Demonstration

- Build Command
 - `g++ -g -Wall -std=c++23 simple_cv.cpp -o prog -lpthread`
- Execute
 - `./prog`
- Debug
 - `gdb --tui ./prog`
 - (Can try 'info threads') to see the threads
 - (Still a good idea to setup 'set scheduler-lock on' as well)

Live Live Recorder and UDB: Demonstration

- Run and create a recording from Undo -- if you prefer instead of gdb
 - `/home/mike/Downloads/undo-7.2.1/live-record ./prog`
 - Then use 'rr' or 'udb' to replay
 - `/home/mike/Downloads/undo-7.2.1/udb`
`prog-3008963-2024-05-14T10-37-40.324.undo`
 - Try 'start' 'layout src' and then using 'n' to step through
 - 'info threads' and other GDB knowledge works as well
 - Neat way to debug these things is with 'live recorder'
 - <https://docs.undo.io/UsingTheLiveRecorderTool.html>

Wrap-Up

We've done a quick tour of C++ concurrency

- Given the topics below there's much to learn -- and we have not even gotten to measuring performance.
- Concurrency is a deep and important topic
 - The “free lunch [really] is over” (Herb Sutter 20 years ago)

- Transport Tycoon
- `std::thread` and `std::jthread`
 - Launching and joining threads
 - Data Parallelism
- Synchronization Primitives
 - mutexes and mutex management
 - Condition Variables
 - semaphores
 - latches / barriers
- Promise and Futures
 - `async`
 - `packaged_tasks`
- Debugging concurrency
 - GDB and UDB.

More Resources

Operating Systems: Three Easy Pieces

- Free book chapters on concurrency.
- <https://pages.cs.wisc.edu/~remzi/OSTEP/>

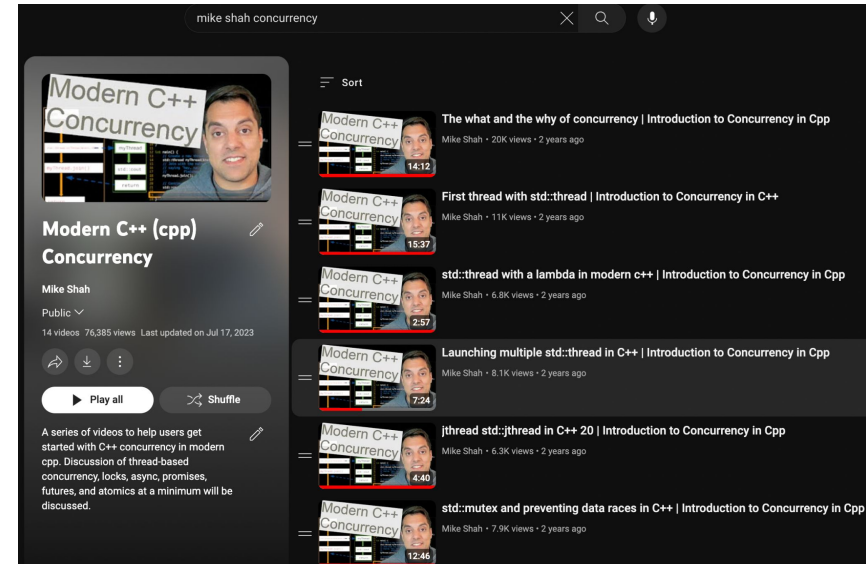
Concurrency
25 <u>Dialogue</u>
26 <u>Concurrency and Threads</u> <small>code</small>
27 <u>Thread API</u> <small>code</small>
28 <u>Locks</u> <small>code</small>
29 <u>Locked Data Structures</u>
30 <u>Condition Variables</u> <small>code</small>
31 <u>Semaphores</u> <small>code</small>
32 <u>Concurrency Bugs</u>
33 <u>Event-based Concurrency</u>
34 <u>Summary</u>

Further resources and training materials

- Debugging Cheatsheet with UDB
 - <https://undo.io/resources/undo-cheat-sheet/>
- Time Travel Debugging - Greg Law - Meeting C++ 2023
 - <https://www.youtube.com/watch?v=qyGdk6QMpMY>
- Back to Basics: Debugging in Cpp - Greg Law - CppCon 2023
 - <https://www.youtube.com/watch?v=qgszy9GquRs>
- Back to Basics: Debugging in C++ - Mike Shah - CppCon 2022
 - <https://www.youtube.com/watch?v=YzIBwqWC6EM>
- Cool New Stuff in Gdb 9 and Gdb 10 - Greg Law - CppCon 2021
 - <https://www.youtube.com/watch?v=xSnetY3eoIk>
- CppCon 2018: Greg Law “Debugging Linux C++”
 - <https://www.youtube.com/watch?v=V1t6faOKjuQ>
- CppCon 2016: Greg Law “GDB - A Lot More Than You Knew”
 - <https://www.youtube.com/watch?v=-n9Fkq1e6sg>

Further resources and training materials

- Playlist on C++ concurrency on YouTube:
 - https://www.youtube.com/playlist?list=PLvv0ScY6vfd_ocTP2ZLicgqKnvq50OCXM
- Slides from this talk will be added to my website shortly.



Further resources and training materials

Some useful talks on concurrency

- GCAP 2016: Parallel Game Engine Design - Brooke Hodgman
 - <https://www.youtube.com/watch?v=JpmK0zu4Mts>
- The MAW: Safely Multithreading the Deterministic Gameplay of 'Age of Empires IV'
 - (Slideshow below -- talk may be available on YouTube or with GDC vault access)
 - <https://www.gdcvault.com/play/1027610/The-MAW-Safely-Multithreading-the>
- Multithreading the Entire Destiny Engine (GDC 2015)
 - https://www.youtube.com/watch?v=v2Q_zHG3vqg
- Sean Parent: Better Code Concurrency
 - <https://www.youtube.com/watch?v=zULU6Hhp42w>

Today's Talk

- Today's talk was inspired by a book I read in graduate school around 2015/2016
- It's a free book by Allen Downey on synchronization
 - <https://greenteapress.com/wp/semaphores/>
- The only primitive needed is a semaphore

The Little Book of Semaphores

by Allen B. Downey

[Download *The Little Book of Semaphores* in PDF.](#)

The Little Book of Semaphores is a free (in both senses of the word) textbook that introduces the principles of synchronization for concurrent programming.

In most computer science curricula, synchronization is a module in an Operating Systems class. OS textbooks present a standard set of problems with a standard set of solutions, but most students don't get a good understanding of the material or the ability to solve similar problems.

The approach of this book is to identify patterns that are useful for a variety of synchronization problems and then show how they can be assembled into solutions. After each problem, the book offers a hint before showing a solution, giving students a better chance of discovering solutions on their own.

The book covers the classical problems, including "Readers-writers," "Producer-consumer", and "Dining Philosophers." In addition, it collects a number of not-so-classical problems, some written by the author and some by other teachers and textbook writers. Readers are invited to create and submit new problems.

NYC++ Meetup

📍 New York, NY, USA

👥 1,376 members · Public group ⓘ

👤 Organized by **Daniel Katz** and **3 others**

Thank you NYC++ 2024!
(Daniel, Nick, and Neel)

The Little Talk of Semaphores -- and a Tour of C++ Concurrency

with Mike Shah

18:30 - 20:30 ET
Thur. October 17, 2024

60 minutes with Q&A
Introductory/Intermediate
Audience

Social: [@MichaelShah](https://twitter.com/MichaelShah)
Web: mshah.io
Courses: courses.mshah.io
 **YouTube**
www.youtube.com/c/MikeShah
<http://tinyurl.com/mike-talks>

This slide is intentionally blank

Extra Slides

Our original motivation was about performance (1/2)

- An interesting reality is that many applications we write are I/O

bound

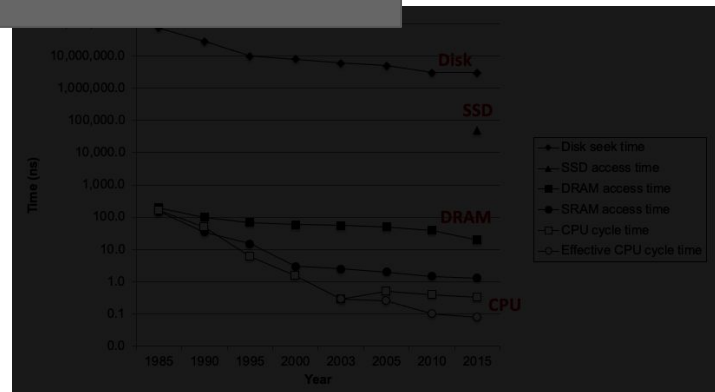
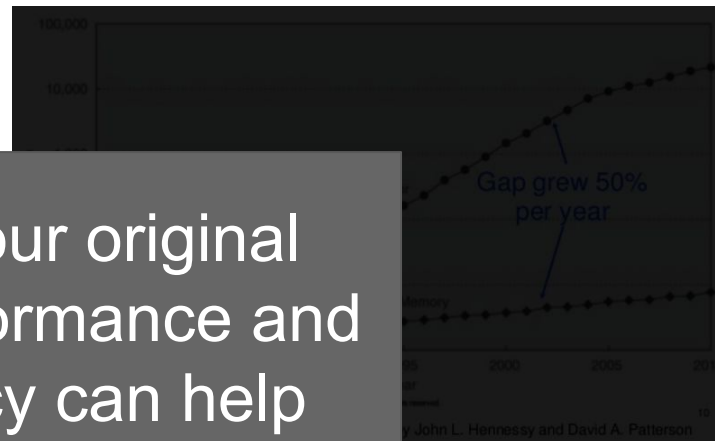
- That means memory
- The figure

while on
faster on

remains orders of magnitude slower

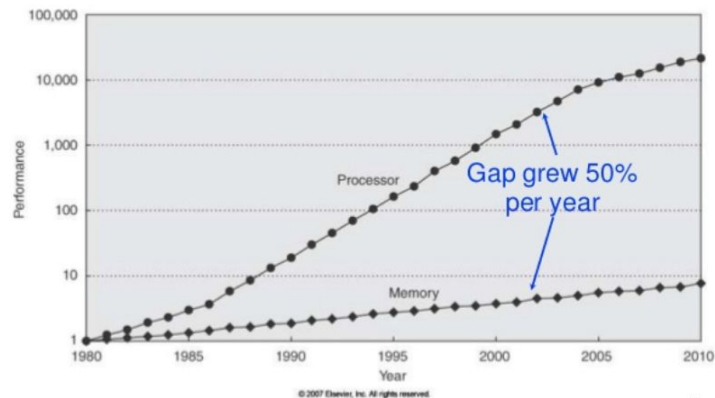
- Thus, we often cannot fetch data at the rate that we process it.

Let's try to answer our original motivation about performance and see how concurrency can help



Our original motivation was about performance (2/2)

- An interesting reality is that many applications we write are I/O bound
 - That means that we are waiting on memory operations
 - The figure to the right shows that while our processors have gotten faster over time, accessing memory remains orders of magnitude slower
 - Thus, we often cannot fetch data at the rate that we process it.



Source: Computer Architecture, A Quantitative Approach by John L. Hennessy and David A. Patterson

