

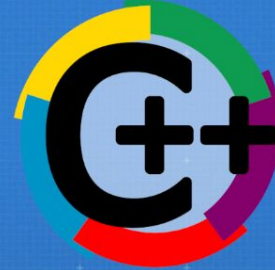
Attribution/License

- 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

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

C++ French User Group

Développeurs C++ de tous les pays, rencontrez-vous!



Fundamentals of Concurrency Threads, Pools, and Patterns

-- in C++
with Mike Shah

19:00 - 21:00 Tue, May 14, 2024

~60 minutes | Introductory/Advanced
Audience

Social: [@MichaelShah](#)

Web: [mshah.io](#)

Courses: [courses.mshah.io](#)

 **YouTube**

www.youtube.com/c/MikeShah

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

What you're going to learn today

- Recap of `#include <thread>`
 - `std::thread`, `std::jthread`
 - Data Parallel Problem (no synchronization)
- Some Basic Patterns with Threads
 - Thread Pools
 - Producer/Consumer
- How to observe behavior of threaded programs
 - Using `gdb` and `udb`



Pretend these seats are filled :)

<https://pixnio.com/free-images/2017/03/11/2017-03-11-16-47-11-550x413.jpg>

Your Tour Guide for Today

by Mike Shah

- **Associate Teaching Professor** at Northeastern University in Boston, Massachusetts.
 - I **love teaching**: courses in computer systems, computer graphics, geometry, and game engine development.
 - My **research** is divided into computer graphics (geometry) and software engineering and computer systems.
- I am **available for contract work** or **technical training** on Modern C++, DLang, Concurrency, OpenGL, and Vulkan projects
- **Outside of work**: guitar, running/weights, traveling and cooking are fun to talk about



Web

www.mshah.io



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

Non-Academic Courses

courses.mshah.io

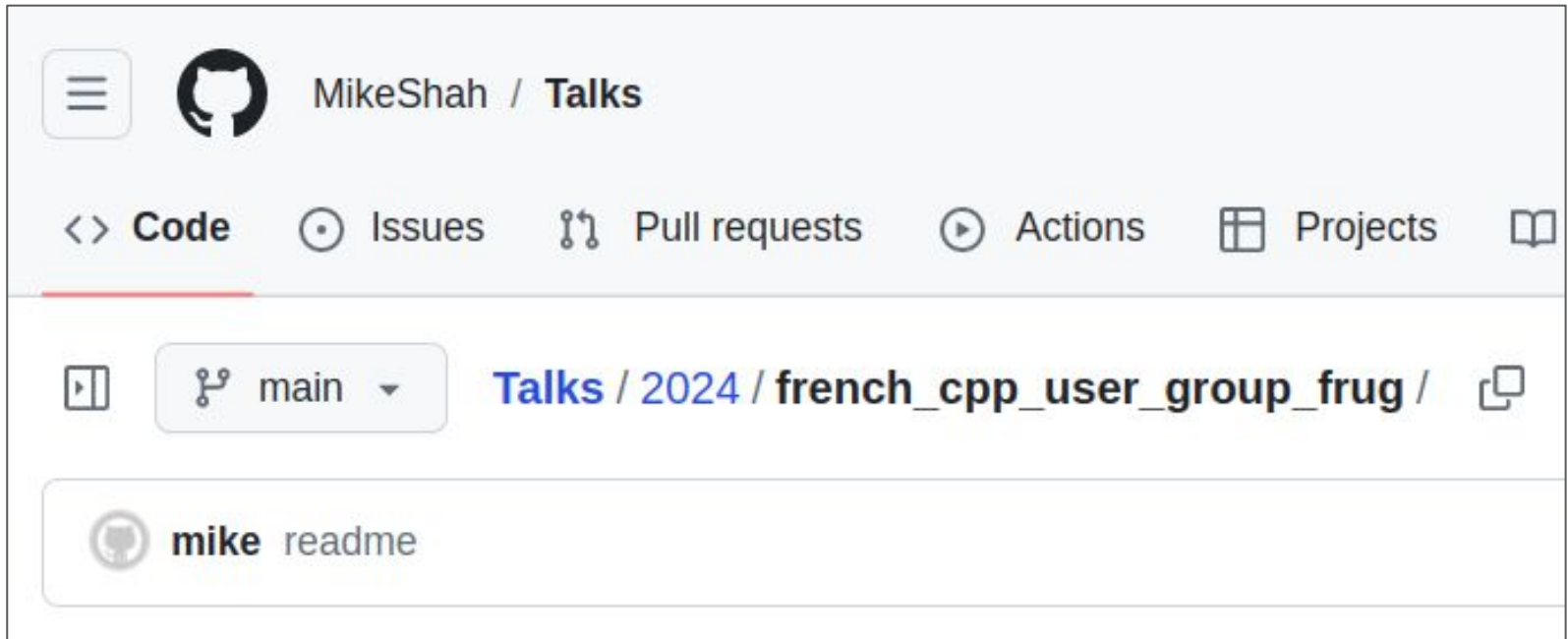
Conference Talks

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

Code for the talk

- Located here:

https://github.com/MikeShah/Talks/tree/main/2024/french_cpp_user_group_frug



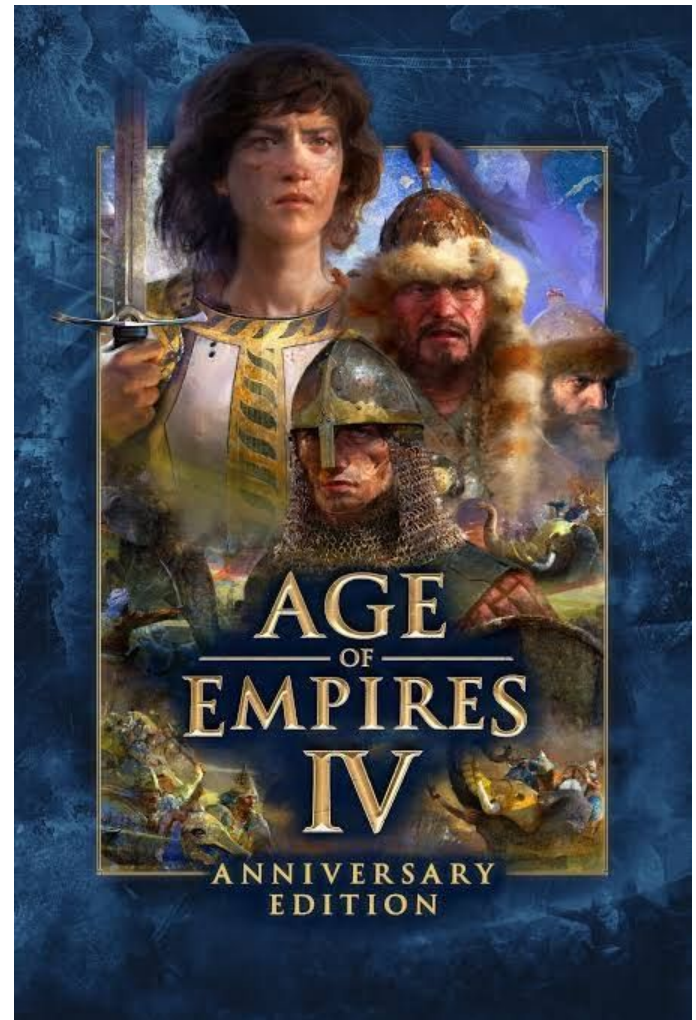
Abstract



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

With the addition of `<thread>` in C++ 11, programmers now have an interface for launching a `std::thread`(or `std::jthread`) with relative ease. But how do we use threads effectively? Often we hear about scary things like ‘deadlock’ or ‘data races’, and you are further warned to be very cautious with threads, while simultaneously told that ‘threads are the answer to performance’. In this talk, we will go over the fundamentals of threads, and approach your ‘second day’ with `std::thread`, and showcase practical examples of how to use threads and use them safely. We’ll cover some ‘pitfalls’, but the goal is to leave this talk understanding more about threads so we can be more comfortable eventually ‘architecting software’ using multiple threads. We’ll analyze and even investigate how to implement thread-pools, producer-consumer, and other common patterns used in threading and found in the real world. After leaving this talk, you should feel more comfortable to try your own experiments, and consider architecting your software with threads to maximize your performance.

Age of Empires 4



(Disclaimed: I did not work on the game, but it would have been cool if I did!)

Computers are Incredibly Powerful (Age of Empires IV)

- I'm very fascinated by how powerful our computers are!
- My fascination is often in game programming
 - Look at the hundreds of individual AI agents running around!
 - The physics simulation of a crumbling castle
 - The beautiful graphics and animations
- There's a lot of interesting 'stuff' being computed every millisecond!



Age of Empires 4 <https://media2.giphy.com/media/11mV0tBosR61ac3m1i/200.gif>

Engineering Challenges

- Now of course -- there's lots of interesting engineering going on
 - Much of that engineering is in the name of **performance**
- The image to the right is a full talk about the 'multithreading' that was needed to enable the previous animation you saw from Age of Empires IV.
 - (Notes included below for some context)
- In short -- today we're going to want to learn a bit about the primitives that enable us engineer performant systems

The Game



AGE OF EMPIRES IV

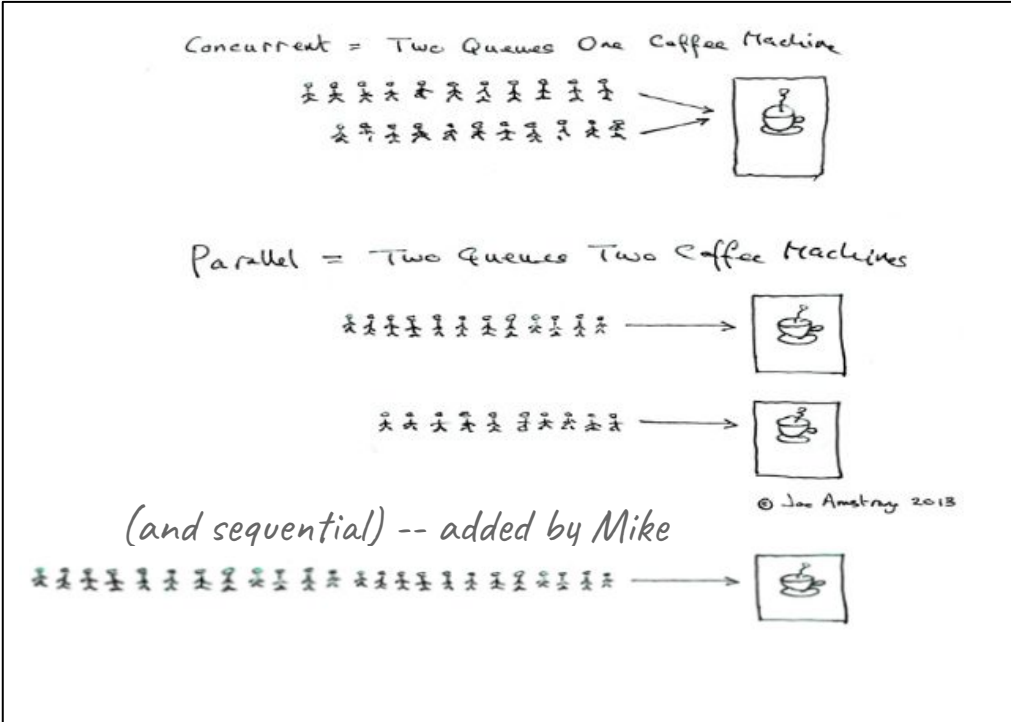
March 21-25, 2021 | San Francisco, CA #GDC21

GDC

Each circle here is a simulation island, an independent group of units where each individual unit, in theory, only is looking at other units in the same island during its the update phases that require modifying unit state. Across the rest of the map, there are probably dozens, if not hundreds, more. Plenty enough islands to spread across the cores of most PCs these days.

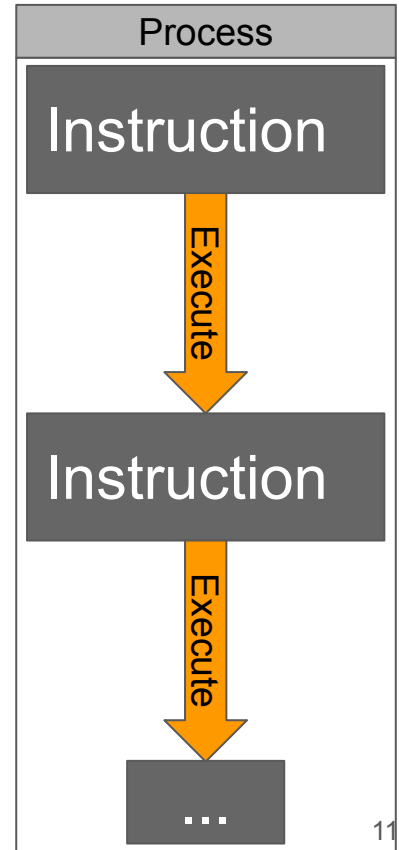
Serving Coffee

(2 lines are better than 1)



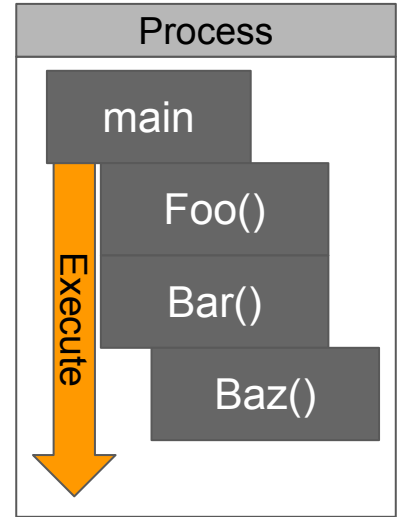
Sequential Software Construction (1/4)

- We learn software construction writing programs that execute one instruction at a time
 - i.e. We have one main ‘thread of execution’ in our process running
 - Note: We use the terms “serial” or “sequential” to describe this execution



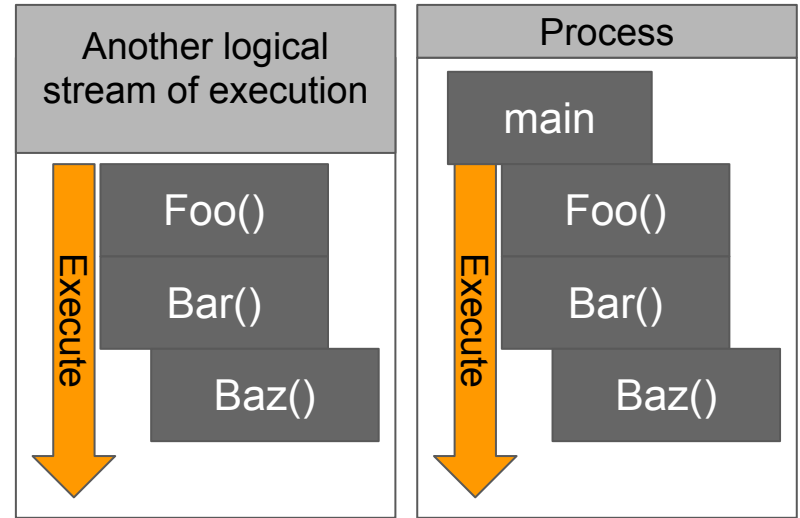
Sequential Software Construction (2/4)

- We can abstract our visualization, and show the call stack.
 - (One function calling the other, with the indentation indicating the call stack)

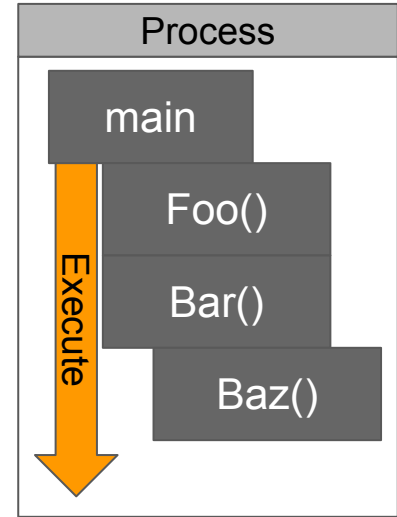
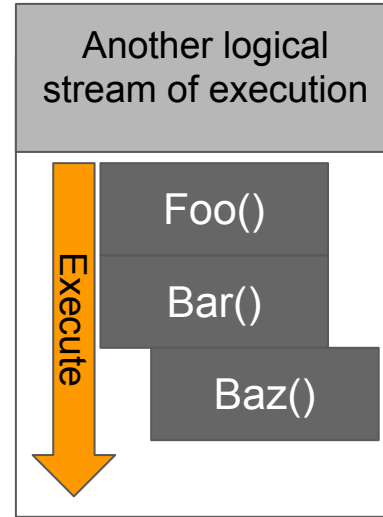
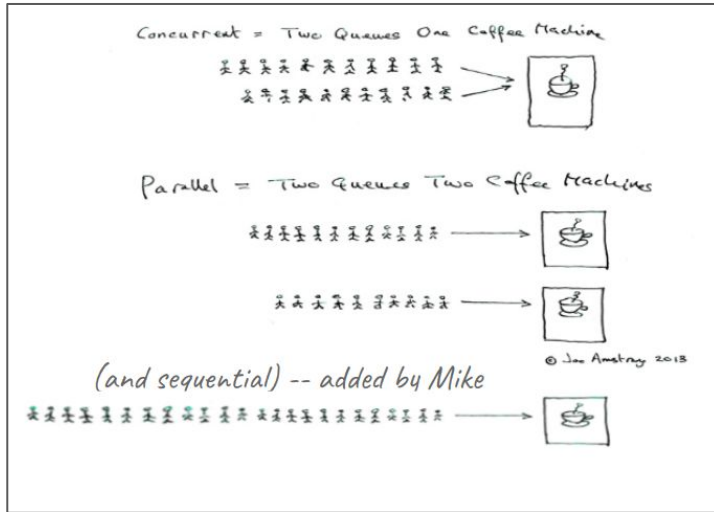


Sequential Software Construction (3/4)

- We can abstract our visualization, and show the call stack.
 - (One function calling the other, with the indentation indicating the call stack)
- As you might imagine -- having two or more streams of execution could speed things up!
 - Or otherwise -- just make solving a problem more easy to reason about



Sequential Software Construction (4/4)



- The motivation for allowing a program to have '2' (or more) execution paths is exactly what is shown on the illustration on the left with 'coffee machines' (which one would you line up in?)
 - I think we all understand the idea that if we have two lines we can do things faster
 - (top picture -- perhaps two people will always have their wallets ready, rather than only the first person in the line to save 'overall time')
 - (middle picture -- two coffee machines, should be about twice as fast service)
 - (bottom picture - Slowest line)

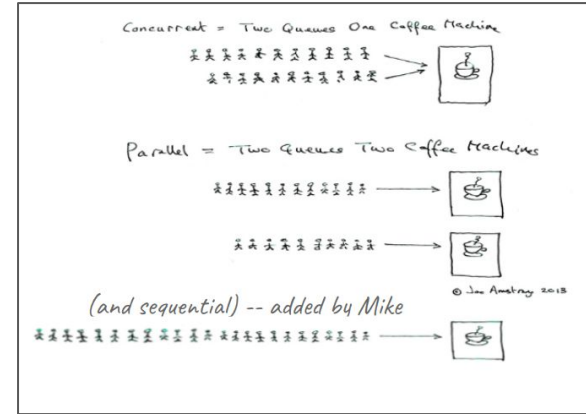
Concurrency

Definitions

Parallelism vs Concurrency (programming context) (1/3)

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

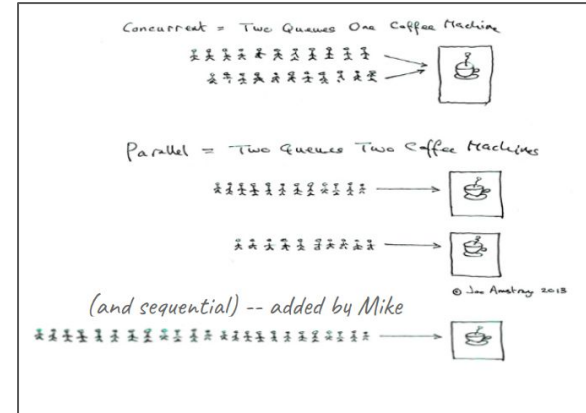
1. **Concurrency Definition:** Multiple things can happen at once, the order matters, and sometimes tasks have to wait on shared resources.
2. **Parallelism Definition:** Everything happens at once, instantaneously



Parallelism vs Concurrency (programming context) (2/3)

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

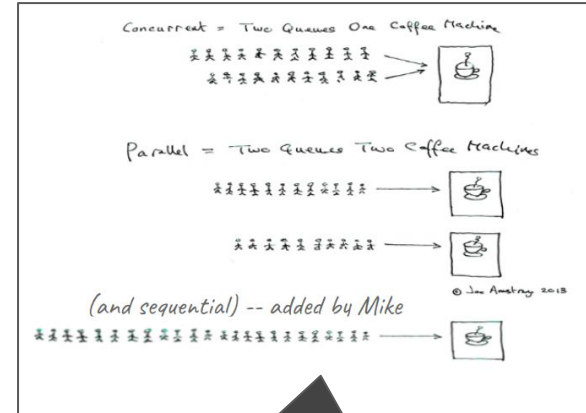
1. **Concurrency Definition:** Multiple things can happen at once, **the order matters, and sometimes tasks have to wait on shared resources.**
2. **Parallelism Definition:** Everything happens at once, instantaneously



Parallelism vs Concurrency (programming context) (3/3)

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



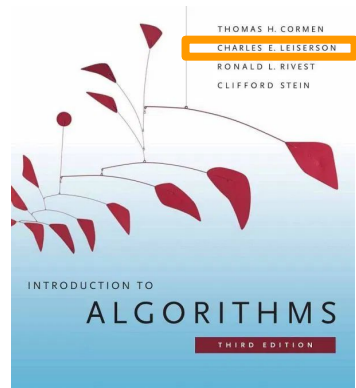
Both concurrency and parallelism can be utilized to yield *better software construction* -- often times meaning better performance.

(Another View) Concurrency versus Parallel

- **Concurrency** -- There are multiple flows of control on a (potentially) shared piece of data
 - More interested in structuring a problem when writing concurrent software
- **Parallelism** -- This is achieved by a hardware mechanism where operations are done simultaneously.
 - The operations are *potentially* related
 - You are doing many things at once.
 - More interested in executing operations fast
- Both ideas often motivated by increased **performance**
 - *The potential* for more tasks to happen at once can thus increase performance
 - Typically if we have multiple cores on our machine
 - Sometimes concurrency/parallelism available on other pieces of hardware
 - e.g. disk fetching memory can be a non-blocking operation (asynchronous) until data is needed (concurrency)
 - e.g. disk fetching multiple pieces of memory at once (parallelism)

“Performance is the currency of computing.”

“Performance is the currency of computing. You can often “buy” needed properties [of software] with performance” - [Charles Leiserson](#)




The Free Lunch is Over - Herb Sutter (1/2)

- Question to Audience:
 - How many folks have read this article written by Herb Sutter?

The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software

Home | Blog | Talks | Books & Articles | Training & Consulting

On the blog  November 4: Other Concurrency Sessions at PDC
November 3: PDC'09: Tutorial & Panel

October 26: Hoare on Testing
October 23: Deprecating export Considered for ISO C++0x

The Free Lunch Is Over

A Fundamental Turn Toward Concurrency in Software

By Herb Sutter

The biggest sea change in software development since the OO revolution is knocking at the door, and its name is Concurrency.

This article appeared in [Dr. Dobbs's Journal](#), 30(3), March 2005. A much briefer version under the title "The Concurrency Revolution" appeared in [C/C++ Users Journal](#), 23(2), February 2005.

Update note: The CPU trends graph last updated August 2009 to include current data and show the trend continues as predicted. The rest of this article including all text is still original as first posted here in December 2004.

Your free lunch will soon be over. What can you do about it? What are you doing about it?

The major processor manufacturers and architectures, from Intel and AMD to Sparc and PowerPC, have run out of room with most of their traditional approaches to boosting CPU performance. Instead of driving clock speeds and straight-line instruction throughput ever higher, they are instead turning *en masse* to hyperthreading and multicore architectures. Both of these features are already available on chips today; in particular, multicore is available on current PowerPC and Sparc IV processors, and is coming in 2005 from Intel and AMD. Indeed, the big theme of the 2004 In-Stat/MDR Fall Processor Forum was multicore devices, as many companies showed new or updated multicore processors. Looking back, it's not much of a stretch to call 2004 the year of multicore.

And that puts us at a fundamental turning point in software development, at least for the next few years and for applications targeting general-purpose desktop computers and low-end servers (which happens to account for the vast bulk of the dollar value of software sold today). In this article, I'll describe the changing face of hardware, why it suddenly does matter to software, and how specifically the concurrency revolution matters to you and is going to change the way you will likely be writing software in the


<http://www.gotw.ca/publications/concurrency-ddj.htm>

The Free Lunch is Over - Herb Sutter (2/2)

- **Question to Audience:**
 - How many folks have read this article written by Herb Sutter?
 - The reality is -- our CPU architectures will continue to adopt multicore architectures
 - We don't (as much) get 'free performance' from CPU speeds anymore
 - Why? **Next slide!**

The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software

Home | Blog | Talks | Books & Articles | Training & Consulting

On the blog  November 4: Other Concurrency Sessions at PDC
November 3: PDC'09: Tutorial & Panel

October 26: Hoare on Testing
October 23: Deprecating export Considered for ISO C++0x

The Free Lunch Is Over

A Fundamental Turn Toward Concurrency in Software

By Herb Sutter

The biggest sea change in software development since the OO revolution is knocking at the door, and its name is Concurrency.

This article appeared in [Dr. Dobbs's Journal](#), 30(3), March 2005. A much briefer version under the title "The Concurrency Revolution" appeared in [C/C++ Users Journal](#), 23(2), February 2005.

Update note: The CPU trends graph last updated August 2009 to include current data and show the trend continues as predicted. The rest of this article including all text is still original as first posted here in December 2004.

Your free lunch will soon be over. What can you do about it? What are you doing about it?

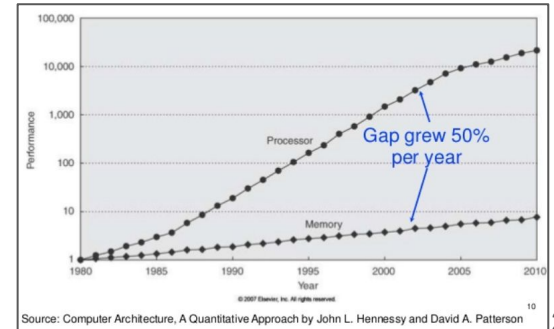
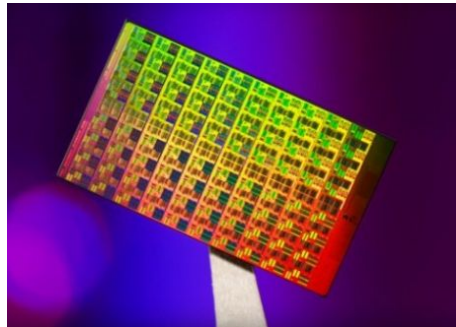
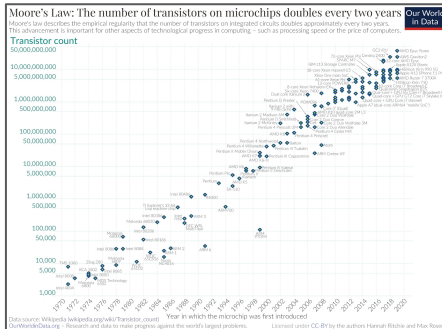
The major processor manufacturers and architectures, from Intel and AMD to Sparc and PowerPC, have run out of room with most of their traditional approaches to boosting CPU performance. Instead of driving clock speeds and straight-line instruction throughput ever higher, they are instead turning *en masse* to hyperthreading and multicore architectures. Both of these features are already available on chips today; in particular, multicore is available on current PowerPC and Sparc IV processors, and is coming in 2005 from Intel and AMD. Indeed, the big theme of the 2004 In-Stat/MDR Fall Processor Forum was multicore devices, as many companies showed new or updated multicore processors. Looking back, it's not much of a stretch to call 2004 the year of multicore.

And that puts us at a fundamental turning point in software development, at least for the next few years and for applications targeting general-purpose desktop computers and low-end servers (which happens to account for the vast bulk of the dollar value of software sold today). In this article, I'll describe the changing face of hardware, why it suddenly does matter to software, and how specifically the concurrency revolution matters to you and is going to change the way you will likely be writing software in the

<http://www.gotw.ca/publications/concurrency-ddj.htm>

Computer Software and Architecture Trends

A few basic ideas and 'laws'



Three long held Software Trends

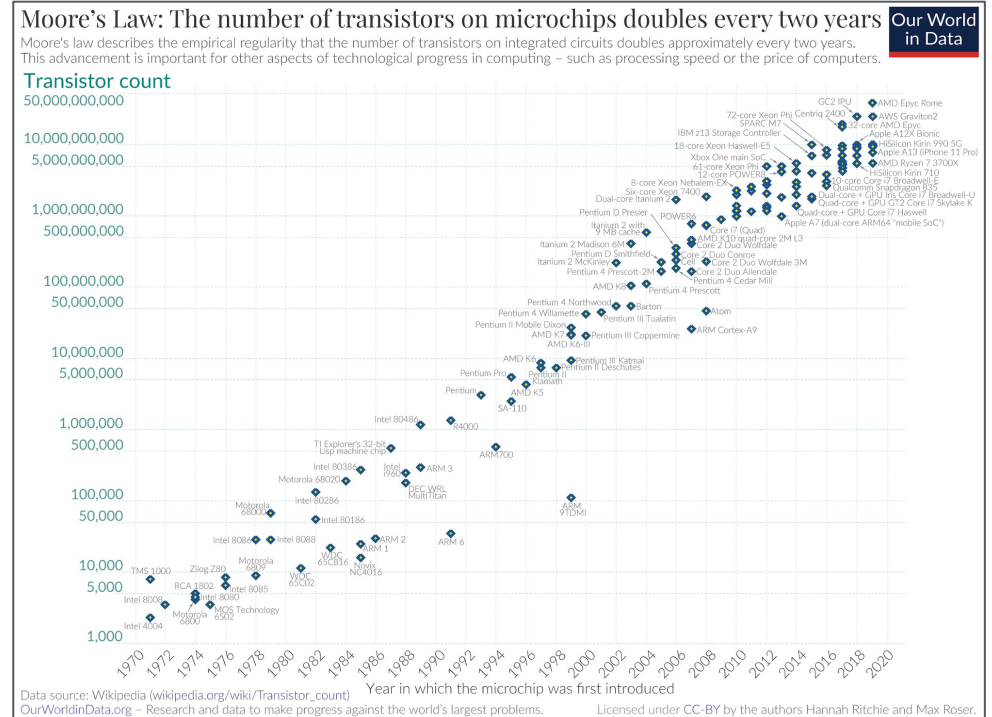
1. Multicore hardware architecture will continue to shape how we write software
2. Cores will come in different form factors (e.g. smaller)
 - a. (Or even a mix of small and large cores on a single processor)
3. Processing speed (GPU or CPU) will likely continue to outpace 'reading speed' (i.e. I/O from disk)

Moore's Law (1/2)

"The number of transistors incorporated in a chip will approximately double every 24 months."

--Gordon Moore, Intel co-founder

- Around 1965 Gordon Moore predicted the number of transistors would roughly double every 18-24 months
 - And largely this held true!



Moore's Law (2/2)

"The number of transistors incorporated in a chip will approximately double every 24 months."

--Gordon Moore, Intel co-founder

- Around 1965 Gordon

Moore predicted that the number of transistors on a chip would double every 18 months. And

- The problem is as transistors get closer and closer (i.e. transistor density increases)
 - More heat is generated
 - Faster clock speeds demand more power
 - And memory speeds did not keep up with the rate at which we can compute

Moore's Law: The number of transistors on microchips doubles every two years. Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Our World in Data

- AMD Ryzen 9000
- AWS Graviton2
- Apple A12X Bionic
- HiSilicon Kirin 990 5G
- Apple A13 (iPhone 11 Pro)
- AMD Ryzen 7 3700X
- HiSilicon Kirin 710
- Apple Core 2 Duo E6700
- Intel Core i7-9700
- Apple A11 Bionic
- Intel Core i5-7260U
- Apple A10 Fusion
- Intel Core i7-8550U
- Apple A12 Bionic
- ARMv8 mobile SoC

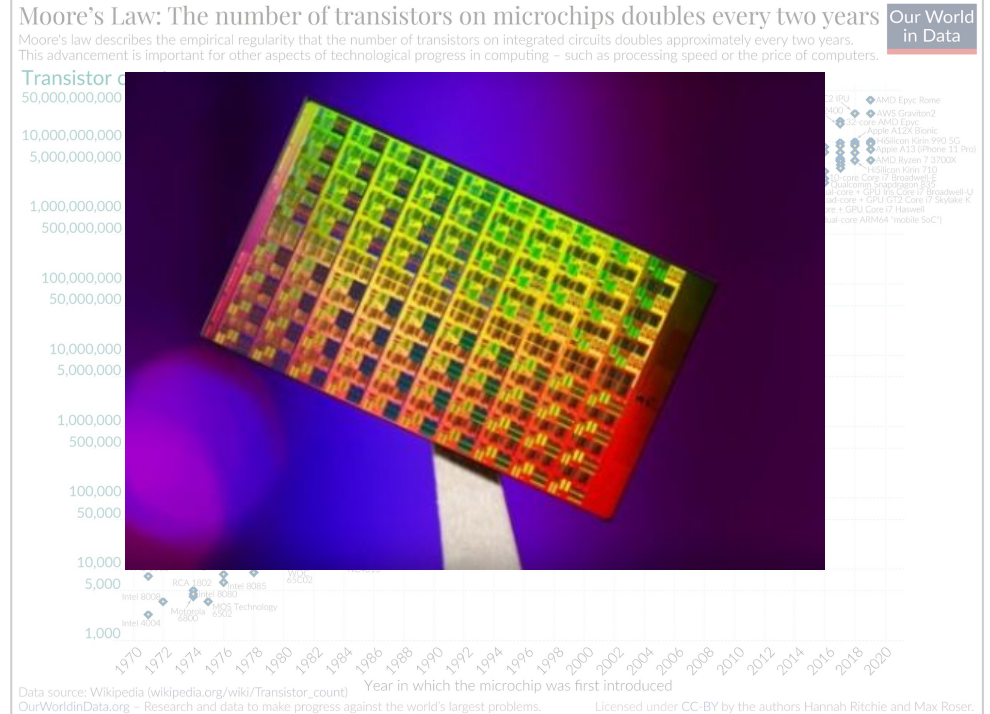


Dennard Scaling (1/3)

"The number of transistors incorporated in a chip will approximately double every 24 months."

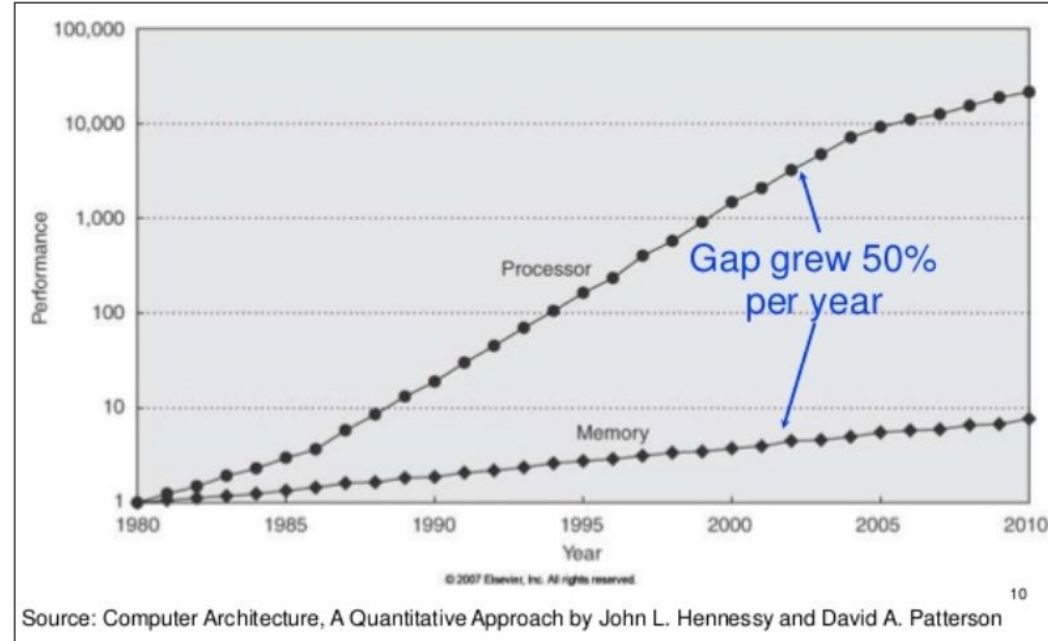
--Gordon Moore, Intel co-founder

- Physically (on the atomic scale) transistors are packed very tightly together
- Heat becomes a problem
- Energy consumption increases
 - (i.e. Dennard Scaling)



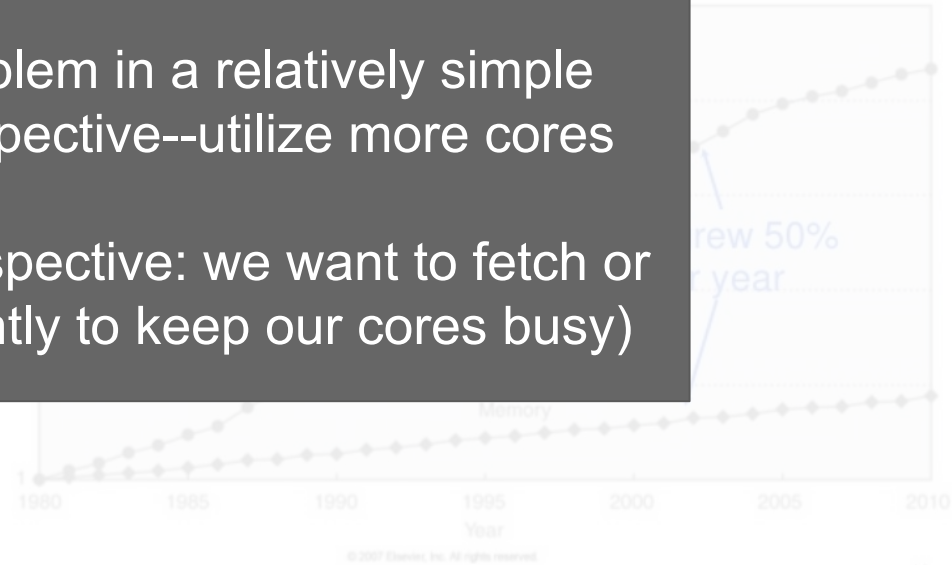
Another problem--Processor Memory Gap (1/2)

- The processor memory gap in particular continues to grow!
 - So even as cpus get faster, other technologies cannot keep up.
 - This *tends* to make our applications I/O bound
 - (i.e. we are waiting on the data to be read/written from memory)



Another problem--Processor Memory Gap (2/2)

- The processor memory gap is continuing to grow
 - Since the mid-1990s, processor performance has increased at a rate of approximately 50% per year
 - This has led to a significant gap between processor performance and memory bandwidth
 - Today we mitigate the problem in a relatively simple way from a hardware perspective--utilize more cores (From an engineering perspective: we want to fetch or be fetching data concurrently to keep our cores busy)
 - This has led to applications being I/O bound
 - (i.e. we are waiting on the data to be read/written from memory)



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

Concurrency

(One more time)

$$S_{\text{latency}}(s) = \frac{1}{(1 - p) + \frac{p}{s}}$$

Concurrency -- Is it worth it?

- There is one other law that I want to briefly introduce on the next slide (Amdahl's Law)
- In short -- it tries to answer the question of
 - “Can I split up my software into different jobs that could execute either concurrently or in parallel”
 - “And if I go through that effort -- will I get a reasonable speedup”
 - (i.e. How ‘serial’ is my program)

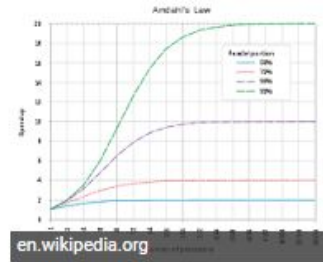
(Note: There are other questions to ask and other variants of Amdahl's law:
https://accu.org/journals/overload/28/157/teodorescu_2795/)

Amdahl's Law

- Performance (execution speed)
- But how much performance?

Amdahl's law is a formula used to find the maximum improvement possible by improving a particular part of a system. In parallel computing, **Amdahl's law** is mainly used to predict the theoretical maximum speedup for program processing using multiple processors. ... This term is also known as **Amdahl's argument**.

What is Amdahl's Law? - Definition from Techopedia
<https://www.techopedia.com/definition/17035/amdahls-law>



$$S_{\text{latency}}(s) = \frac{1}{(1 - p) + \frac{p}{s}}$$

s = speedup of task that benefits from improved resources

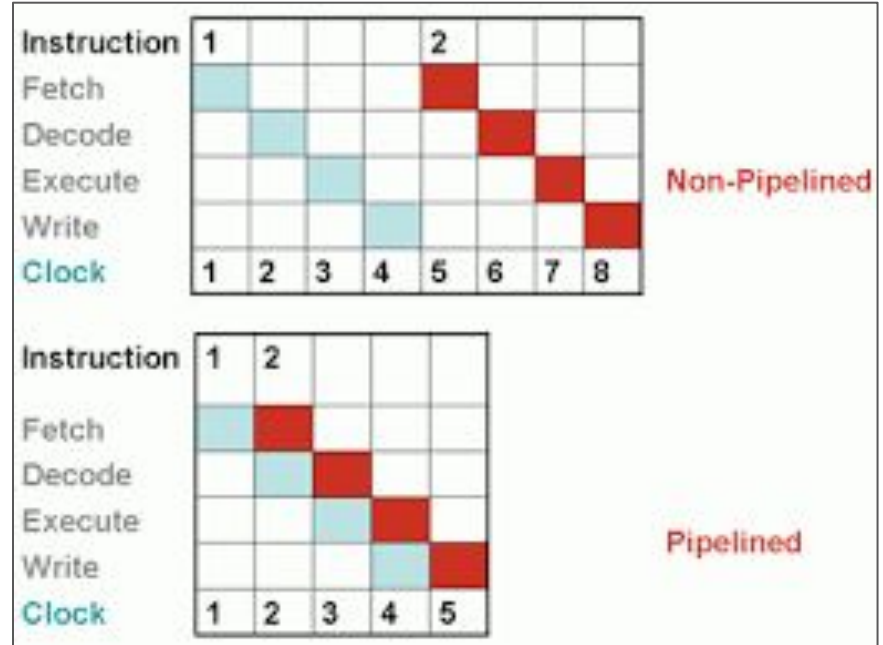
p = portion of execution time benefiting from improved speedup

https://en.wikipedia.org/wiki/Amdahl%27s_law

Applied example: <http://web.cs.iastate.edu/~prabhu/Tutorial/CACHE/CompPerf.pdf>

(Aside: Some parallelism for free (implicit parallelism))

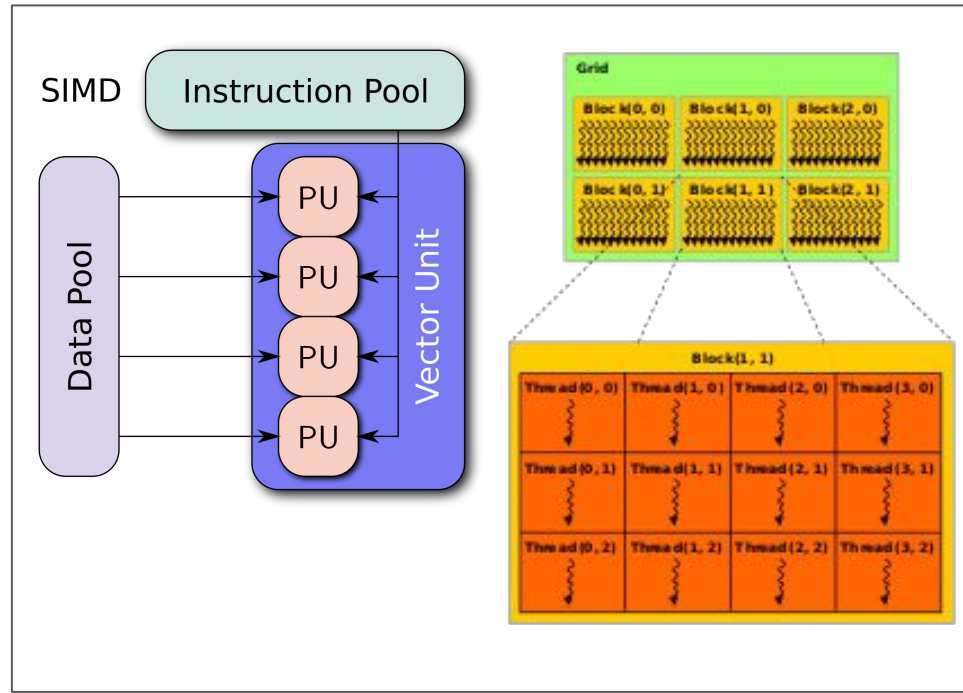
- CPU Pipelining is an example of *parallelism* we typically get for free
 - (i.e. implicit parallelism)
- Potential compiler optimizations to automatically vectorize code.



<https://s0.stackpointer.io/wp-content/uploads/2009/02/pipelining.png>

(Aside: Some parallelism not for free (explicit) parallelism)

- Using the GPU
 - Whether CUDA or OpenCL for general purpose GPU programming
 - Or perhaps a shader language like GLSL or HLSL
- SIMD Instructions
 - Our SSE or AVX instructions





Threads

(i.e. “lightweight processes”)

The Necessity of Concurrency

- In general, concurrency (like parallelism) is used because it is necessary for a system to function.
 - Concurrency
 - Real world concurrency examples
 - e.g. an orchestra, a subway transit system, cars at a traffic stop
 - Computer Science examples
 - e.g. A memory allocator, File I/O, Network requests (awaiting data)
 - e.g. A server trying to handle millions of users
 - Parallelism
 - Real world example:
 - Highway with multiple lanes, multiple elevators in an apartment all going up
 - Computer Science example
 - fragment shader in a computer game running in parallel so we can render at 60 FPS

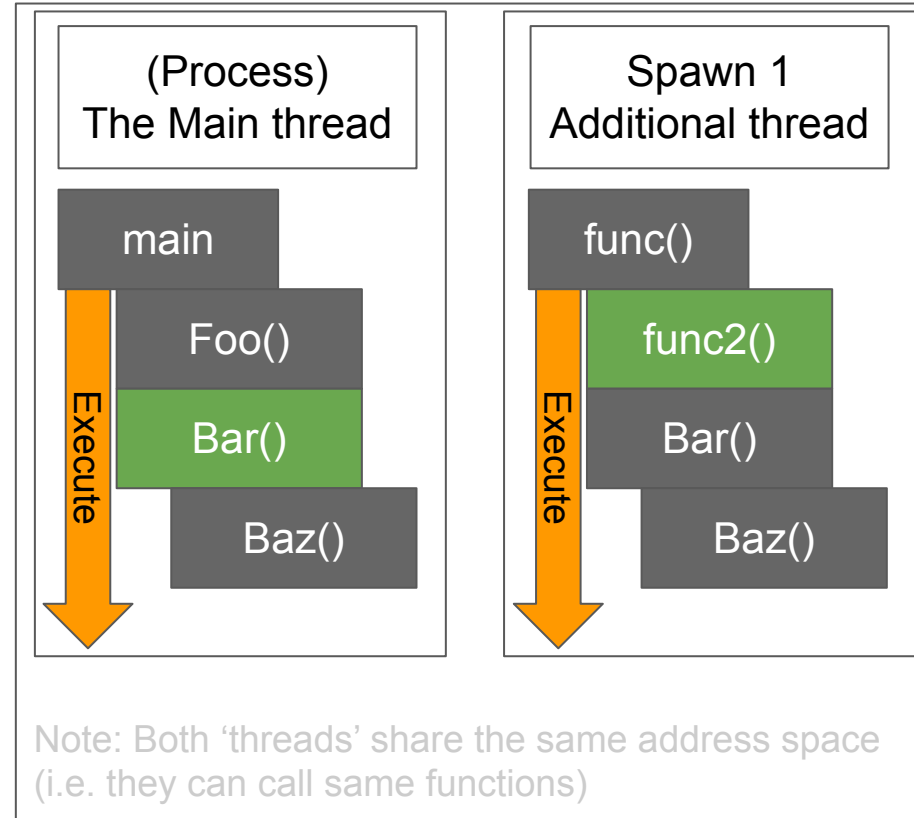
(Live Concurrency Example)

- (Trains are a great example of concurrency and parallelism by the way)
- (One of my favorite parts of Paris is taking the trains everywhere)



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 coordinates with this thread)



What is a thread? (1/2)

- A ‘thread’ is often defined as a ‘**lightweight process**’
- A thread has its own ‘thread-control block’ with:
 - A thread id (TID)
 - A its own logical control flow
 - (e.g. instruction pointer)
 - Its own stack for local variables

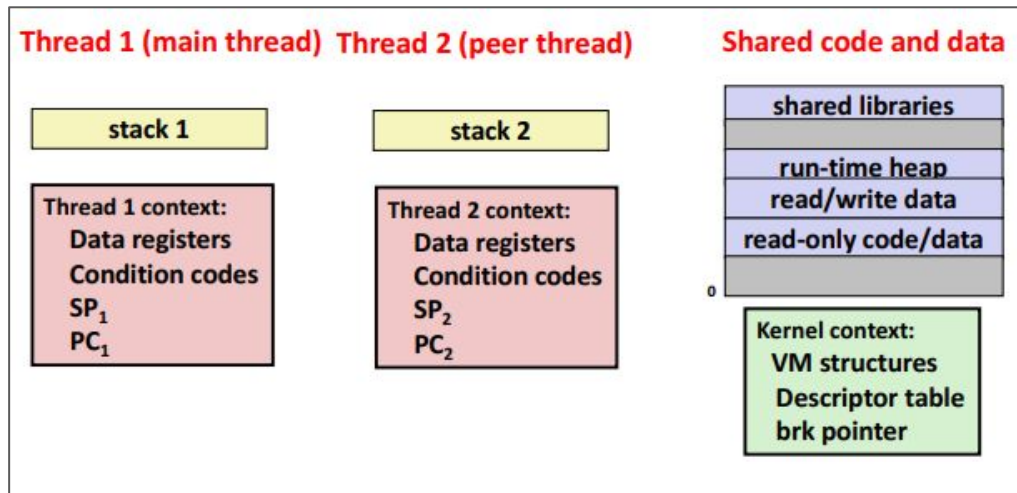


Figure from: Computer Systems a Programmer's Perspective 3rd Edition

What is a thread? (2/2)

- 1 Process (i.e. your application) can have many threads:
 - Each thread shares the same code, data, and kernel context
 - But each thread can execute separately within the same process (i.e. address space) independently.

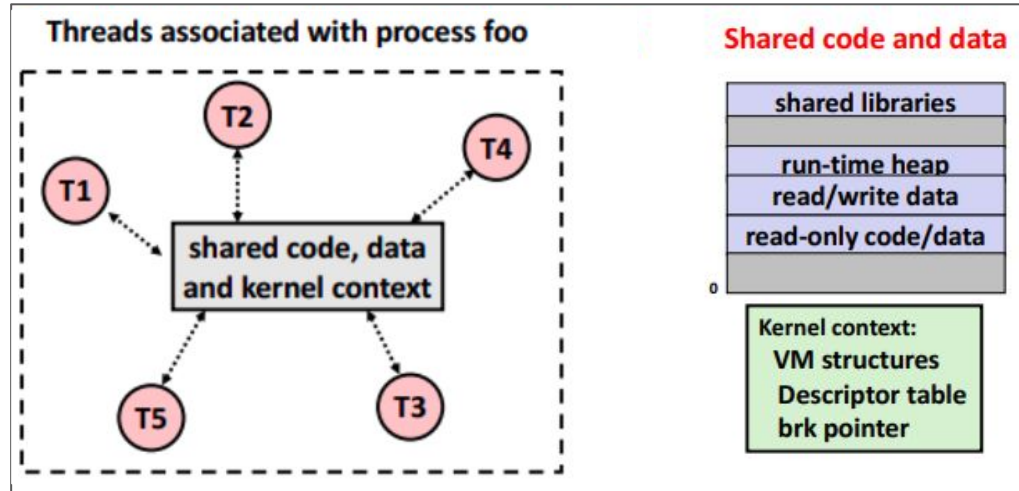


Figure from: Computer Systems a Programmer's Perspective 3rd Edition

(Aside: Thread vs Process -- What's the difference?)

- A 'process' can contain multiple threads
 - Threads exist within the process
- The advantage of threads is that they require fewer system resources
 - Organizing a group of threads to 'cooperatively' work together is *likely* cheaper than organizing multiple processes to work together
- Threads can be scheduled (e.g. by priority, round robin, etc.), and usually your thread API provides often provides some control over this.
 - <https://www.ibm.com/docs/en/aix/7.3?topic=threads-thread-scheduling>

When to use threads

- Heavy Computations
 - Use threads to work on a heavy computation
 - The most common case is actually using threads on your GPU for graphics
 - GPUs have 100s or 1000s of threads that are good for massively parallel tasks.
 - (You could also use things like CUDA to take advantage of your graphics hardware)
 - You may need to use a series of threads to otherwise resolve complex computations on your CPU where decisions may need to be made.
- Using threads to separate work
 - Gives performance (Same as above)
 - But also simplifies the logic of your problem
- (If it's useful -- you can visualize 'threads' like workers being hired in a factory, ideally working together to solve some problem, and balancing the right number of helpers)

Threads associated with process foo

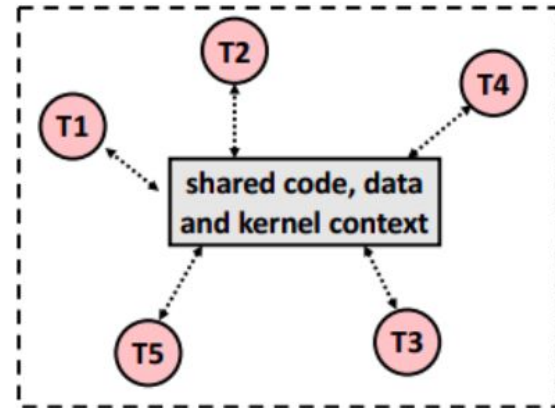


Figure from: Computer Systems a Programmer's Perspective 3rd Edition

(Aside) The term ‘thread’

- There’s some confusion when it comes to the term ‘thread’
 - Operating system-thread
 - Also called a ‘kernel thread’ [\[link\]](#)
 - These are threads that the operating system gets to schedule and assign to do work.
 - Number of kernel threads != number of CPU cores
 - But there are some number of kernel threads
 - user-space threads
 - These are what we ‘spawn’ from a process
 - Operating system *may* assign a user-thread to be run on a kernel thread (i.e. we *may* on some architectures think of this as a 1:1 model)
 - GPU threads
 - Perhaps many grouped up together to do some computation in a ‘thread block’
 - Usually 100s or 1000s of these ‘small threads’ executing a ‘kernel’ (usually a small program) or ‘shader’ (for graphics)
 - These are usually meant for ‘data parallel’ computations



Threads in Modern C++

The `std::thread` - Since C++11, we have a standard interface to threading

Thread Libraries

- Before C++11/14/17/20/23, there existed threading libraries with different semantics
 - Libraries like “Boost”, Intel “Thread Building Blocks”, or “pthread” were used
 - Perhaps you have used pthread at least in C
 - (std::thread I believe is implemented with pthread most posix systems)
- Typically today *I would personally recommend* using the standard C++ threading library for portability reasons as the default choice.

C++ Thread support library	
<h2>Thread support library</h2>	
C++ includes built-in support for threads, mutual exclusion, condition variables, and futures.	
<h3>Threads</h3>	
Threads enable programs to execute across several processor cores.	
Defined in header <code><thread></code>	
<code>thread</code> (C++11)	manages a separate thread (class)
<code>jthread</code> (C++20)	<code>std::thread</code> with support for auto-joining and cancellation (class)
<h3>Functions managing the current thread</h3>	
Defined in namespace <code>this_thread</code>	
<code>yield</code> (C++11)	suggests that the implementation reschedule execution of threads (function)
<code>get_id</code> (C++11)	returns the thread id of the current thread (function)
<code>sleep_for</code> (C++11)	stops the execution of the current thread for a specified time duration (function)
<code>sleep_until</code> (C++11)	stops the execution of the current thread until a specified time point (function)

We actually have a good number of primitives (mostly low level) for concurrency support.

Understanding how to use them is part of the goal of today's talk.

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)

Standard library (headers)

Named requirements

Feature test macros (C++20)

Language support library

- Program utilities
- source_location (C++20)
- Coroutine support (C++20)
- Three-way comparison (C++20)
- Type support
- numeric_limits – type_info
- initializer_list (C++11)

Concepts library (C++20)

Diagnostics library

- exception – System error
- basic_stacktrace (C++23)

Strings library

- basic_string – char_traits
- basic_string_view (C++17)
- Null-terminated strings:
- byte – multibyte – wide

Containers library

- vector – deque – array (C++11)
- list – forward_list (C++11)
- map – multimap – set – multiset
- unordered_map (C++11)
- unordered_multimap (C++11)
- unordered_set (C++11)
- unordered_multiset (C++11)
- Container adaptors
- span (C++20) – mdspan (C++23)

C++20, C++23, C++26

Algorithms library

Complex numbers library (C++20)

Mathematics library

- Execution policies (C++17)
- Constrained algorithms (C++20)

Generics library

- Common math functions
- Mathematical special functions (C++17)
- Mathematical constants (C++20)
- Basic linear algebra algorithms (C++26)
- Numeric algorithms
- Pseudo-random number generation
- Floating-point environment (C++11)
- complex – valarray

Date and time library

- Calendar (C++20) – Time zone (C++20)

Localization library

- locale – Character classification
- text_encoding (C++26)

Input/output library

- Print functions (C++23)
- Stream-based I/O – I/O manipulators
- basic_istream – basic_ostream
- Synchronized output (C++20)
- File systems (C++17)

Regular expressions library (C++11)

- basic_regex – Algorithms
- Default regular expression grammar

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>
```

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`](#)
- (Aside: For those familiar, this is essentially going to do 'fork-join' parallelism)

```
1 // @file thread1.cpp
2 // g++ -std=c++17 thread1.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5
6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::cout << "Hello from our thread!" << std::endl;
9     std::cout << "Argument passed in:" << x << std::endl;
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::cout << "Hello from the main thread!" << std::endl;
22
23     return 0;
24 }
```

Thre

```
mike:concurrency$ g++ -std=c++17 thread2.cpp -o prog -lpthread
mike:concurrency$ ./prog
Hello from our thread!
Argument passed in:100
Hello from the main thread!
```

- `#include <thread>`
 - `std::thread`
- (Aside: For those familiar, this is essentially going to do 'fork-join' parallelism)

```
3 #include <iostream>
4 #include <thread> // Include the
5
6 // Test function which we'll launch
7 void test(int x) {
8     std::cout << "Hello from our th
9     std::cout << "Argument passed i
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::cout << "Hello from the main thread!" << std::endl;
22
23     return 0;
24 }
```

Don't forget to link in the pthread library for posix users.

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

```
1 // @file thread1.cpp
2 // g++ -std=c++17 thread1.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5
6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::cout << "Hello from our thread!" << std::endl;
9     std::cout << "Argument passed in:" << x << std::endl;
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::cout << "Hello from the main thread!" << std::endl;
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++ -std=c++17 thread1.cpp -o prog -lthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5
6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::cout << "Hello from our thread!" << std::endl;
9     std::cout << "Argument passed in:" << x << std::endl;
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::cout << "Hello from the main thread!" << std::endl;
22
23     return 0;
24 }
```

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

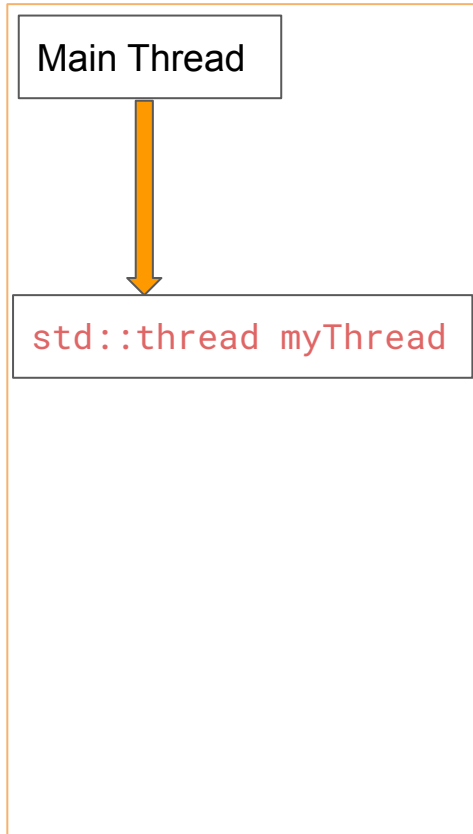
Main Thread



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

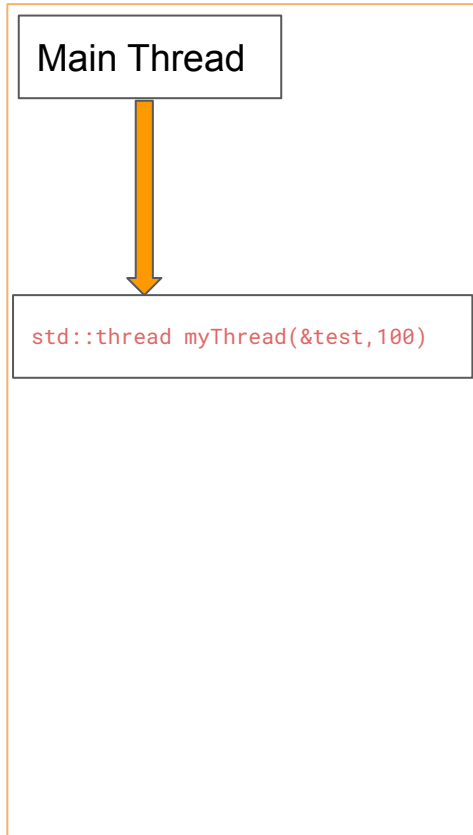
```
1 // @file thread1.cpp
2 // g++ -std=c++17 thread1.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5
6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::cout << "Hello from our thread!" << std::endl;
9     std::cout << "Argument passed in:" << x << std::endl;
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::cout << "Hello from the main thread!" << std::endl;
22
23     return 0;
24 }
```


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



```
1 // @file thread1.cpp
2 // g++ -std=c++17 thread1.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5
6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::cout << "Hello from our thread!" << std::endl;
9     std::cout << "Argument passed in:" << x << std::endl;
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::cout << "Hello from the main thread!" << std::endl;
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()`

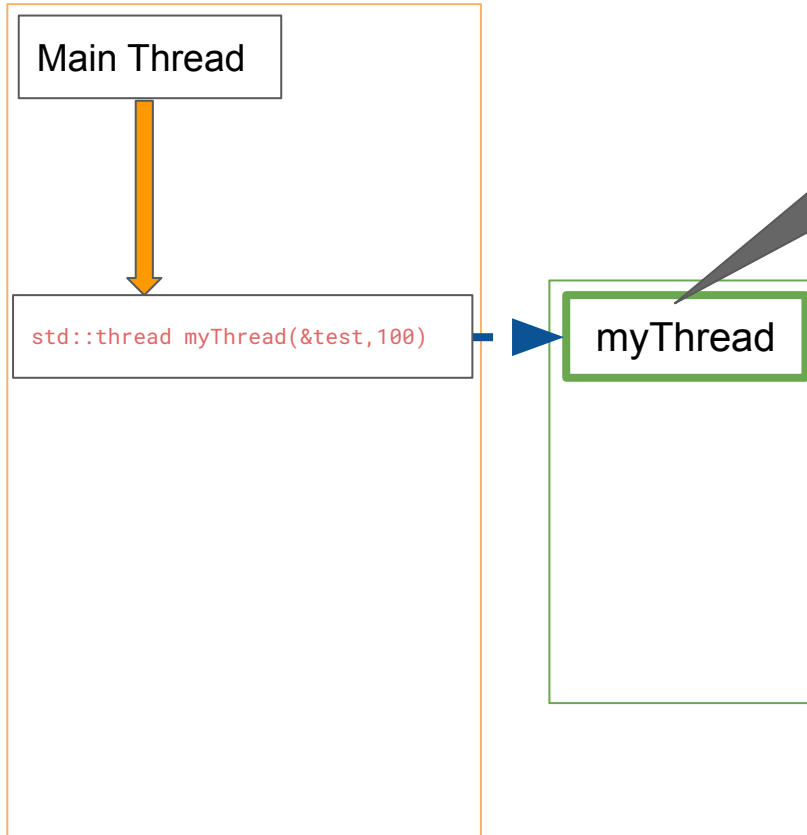
The thread will start executing immediately on construction

(Remember, threads shares code and the heap)

```
1 // @file thread1.cpp
2 // g++ -std=c++17 thread1.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5
6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::cout << "Hello from our thread!" << std::endl;
9     std::cout << "Argument passed in:" << x << std::endl;
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::cout << "Hello from the main thread!" << std::endl;
22
23     return 0;
24 }
```

Visual execution of "Hello Thread" (6/13)

So now we have two "threads" executing

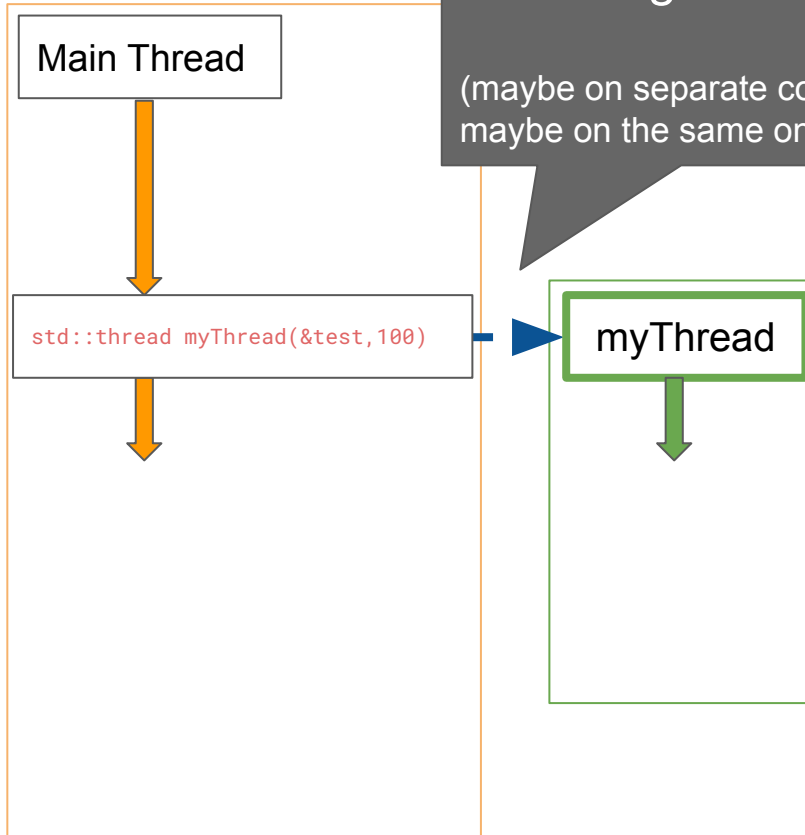


```
prog -lpthread
4 #include <thread> // Include the thread library
5
6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::cout << "Hello from our thread!" << std::endl;
9     std::cout << "Argument passed in:" << x << std::endl;
10 }
11
12 int main() {
13     // Create 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::cout << "Hello from the main thread!" << std::endl;
22
23     return 0;
24 }
```

Visual execution of "Hello Thread" (7/13)

Both threads are
executing concurrently!

(maybe on separate cores, or
maybe on the same one)

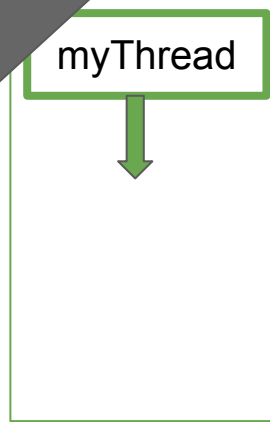
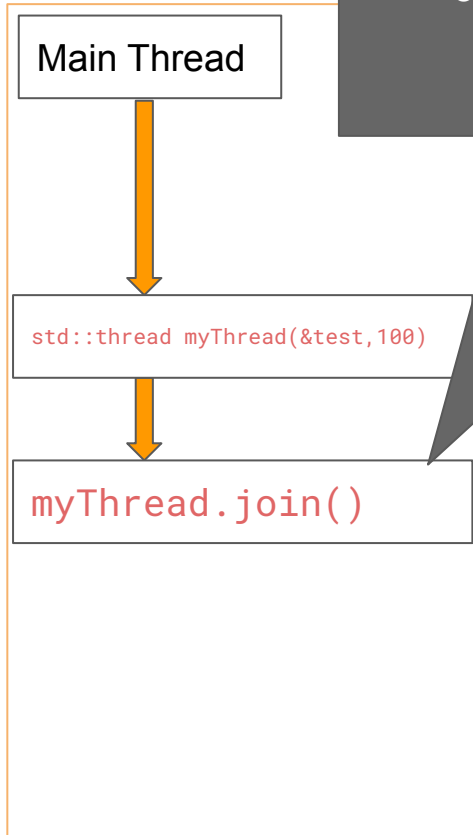


```
thread1.cpp
g++ -std=c++17 thread1.cpp -o prog -lpthread
#include <iostream>
#include <thread> // Include the thread library

6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::cout << "Hello from our thread!" << std::endl;
9     std::cout << "Argument passed in:" << x << std::endl;
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::cout << "Hello from the main thread!" << std::endl;
22
23     return 0;
24 }
```

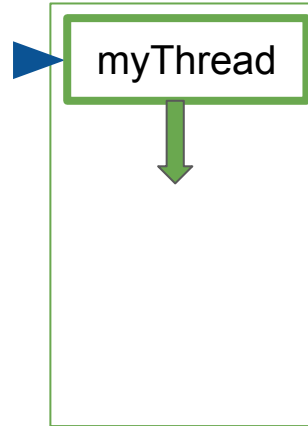
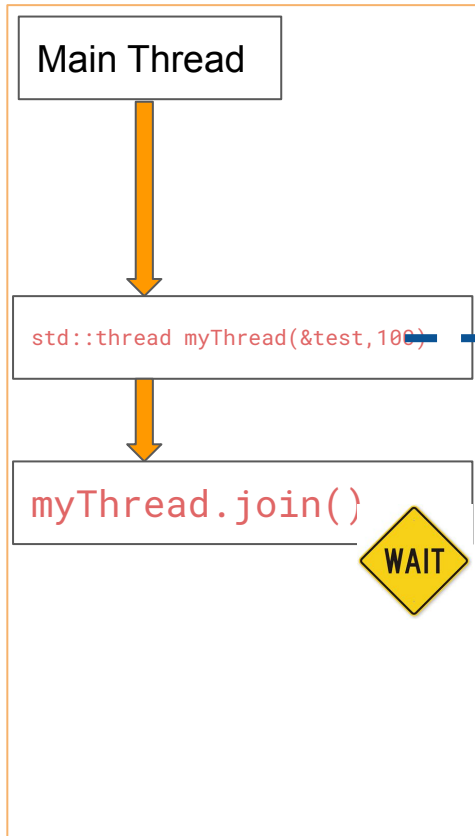
Visual execution

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



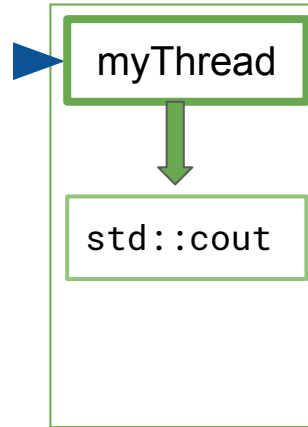
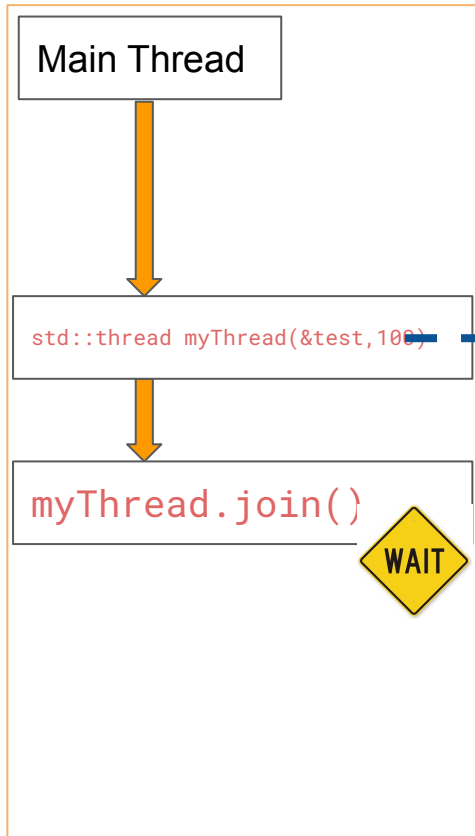
```
5 // ... g -lpthread
6 // ... thread library
7 void test(int x) {
8     std::cout << "Hello from our thread!" << std::endl;
9     std::cout << "Argument passed in:" << x << std::endl;
10 }
11
12 int main() {
13     // Create a new thread and pass one parameter
14     std::thread myThread(&test, 100);
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::cout << "Hello from the main thread!" << std::endl;
22
23     return 0;
24 }
```

Visual execution of "Hello Thread" (9/13)



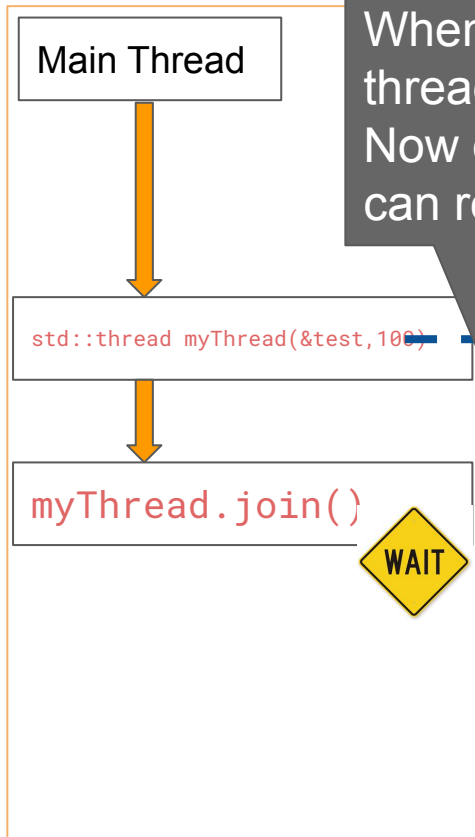
```
1 // @file thread1.cpp
2 // g++ -std=c++17 thread1.cpp -o prog -lthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5
6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::cout << "Hello from our thread!" << std::endl;
9     std::cout << "Argument passed in:" << x << std::endl;
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::cout << "Hello from the main thread!" << std::endl;
22
23     return 0;
24 }
```

Visual execution of "Hello Thread" (10/13)

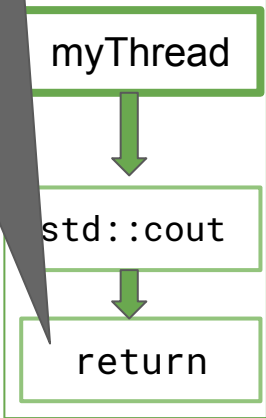


```
1 // @file thread1.cpp
2 // g++ -std=c++17 thread1.cpp -o prog -lthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5
6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::cout << "Hello from our thread!" << std::endl;
9     std::cout << "Argument passed in:" << x << std::endl;
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::cout << "Hello from the main thread!" << std::endl;
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++ -std=c++17 thread1.cpp -o prog -lthread
#include <iostream>
#include <thread> // Include the thread library

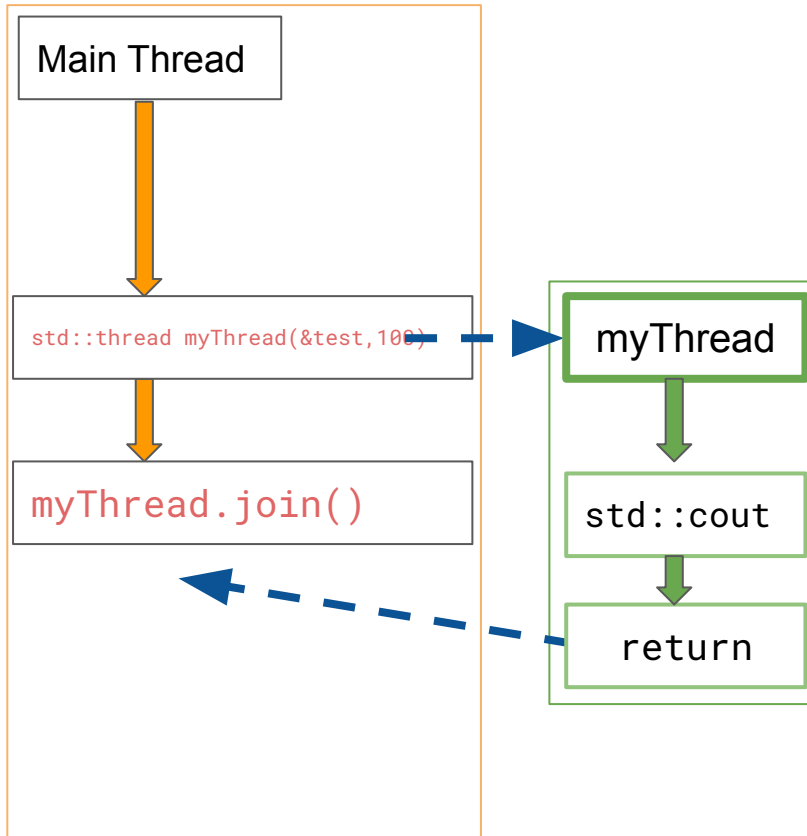
// Test function which we'll launch threads from
void test(int x) {
    std::cout << "Hello from our thread!" << std::endl;
    std::cout << "Argument passed in:" << x << std::endl;
}

int main() {
    // Create a new thread and pass one parameter
    std::thread myThread(&test, 100);
    // Join with the main thread, which is the same as
    // saying "hey, main thread--wait until myThread
    // finishes before executing further."
    myThread.join();

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

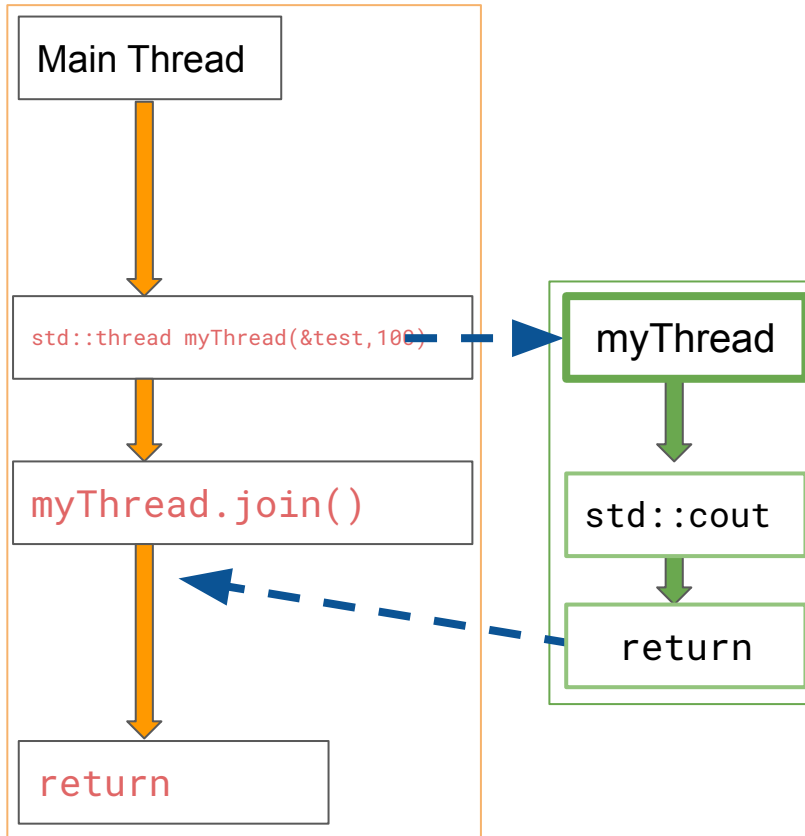
    return 0;
}
```


Visual execution of "Hello Thread" (12/13)



```
1 // @file thread1.cpp
2 // g++ -std=c++17 thread1.cpp -o prog -lthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5
6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::cout << "Hello from our thread!" << std::endl;
9     std::cout << "Argument passed in:" << x << std::endl;
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::cout << "Hello from the main thread!" << std::endl;
22
23     return 0;
24 }
```

Visual execution of "Hello Thread" (13/13)



```
1 // @file thread1.cpp
2 // g++ -std=c++17 thread1.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5
6 // Test function which we'll launch threads from
7 void test(int x) {
8     std::cout << "Hello from our thread!" << std::endl;
9     std::cout << "Argument passed in:" << x << std::endl;
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::cout << "Hello from the main thread!" << std::endl;
22
23     return 0;
24 }
```

Same example as before -- but with a lambda!

- Same example as before, but instead of a function, I have a lambda with 1 parameter (and no return type)
 - `std::thread` takes a [callable](#) as the parameter--so lambdas, functions, etc. are all fine!

```
1 // @file thread2.cpp
2 // g++ -std=c++17 thread2.cpp -o prog -lpthread
3 #include <iostream>
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::cout << "Hello from our thread!" << std::endl;
11         std::cout << "Argument passed in:" << x << std::endl;
12     };
13
14     // Create a new thread with our lambda this time
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::cout << "Hello from the main thread!" << std::endl;
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!

```
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

```
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

The result seems a little strange...anyone see the problem?

- So here we create each of our threads and join

```
mike:concurrency$ g++ -std=c++17 thread3.cpp prog -lpth
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 we

- So here we create each of our threads and join

- 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

```
mike:concurrency$ g++ -std=c++17 thread3.cpp -o prog -lpth
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 is a lambda function
10 auto lambda = [&x](int i) {
11     std::cout << "Thread " << 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 want

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

- So here we create each of our threads and join

```
mike:concurrency$ g++ -std=c++17 thread3_fix.cpp -o prog -lpthread
mike:concurrency$
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!
```

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'll use a lambda function
10    auto lambda = [ ] {
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 }
```

C++ 20 - std::jthread

- std::jthread launches a thread and joins the thread on destruction
 - This may be more useful (especially for beginners) as we don't forget to join!
 - If you need more control on when to join, then prefer std::thread and join explicitly
 - (Note: This code does the right thing--threads are immediately launched and not sequentially waited upon)

```
1 // @file thread4.cpp
2 // g++-10 -std=c++20 thread4.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    // Note: We now have a jthread
16    //      No joins in the program
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::cout << "Hello from the main thread!" << std::endl;
25
26    return 0;
27 }
```

- Now that we have the basics of threads, I want to focus on a few more use cases of threads.
- I offer another 'slower' walkthrough of the previous concepts here if you'd like to revisit any.
 - I also focus more on pitfalls of deadlock and locking strategies

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.)

main thread

main

Foo()

Bar() 1

Baz()

thread 1

func()

func2()

func3()

func4()

Execute

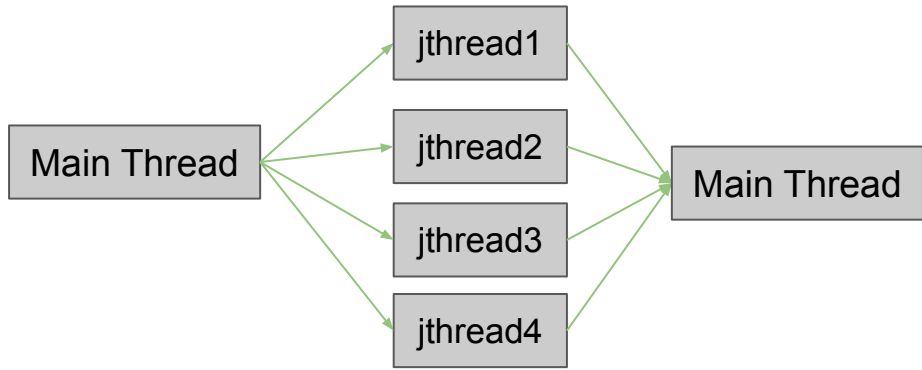
Execute

38

Mike Shah

Back to Basics: Concurrency

[Back to Basics: Concurrency - Mike Shah - CppCon 2021](#)



Teams of threads

Data Parallelism

Thread Team (1/9)

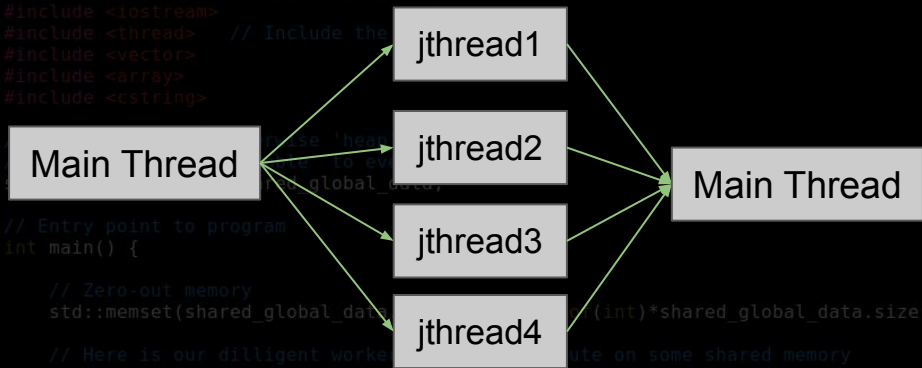
- So now that we have the idea of a 'jthread' let's do a more interesting problem
 - Let's spawn multiple threads that work on some 'shared data' to solve a problem
 - We'll increment some values in shared memory to start.

```
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 (2/9)

- So now that we have the idea of a 'jthread' let's do a more interesting problem
 - Let's spawn multiple threads that work on some 'shared data' to solve a problem
 - We'll increment some values in shared memory to start.

```
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 'new' to create shared global data,
11 // and 'shared_ptr' to share it.
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 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 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 }
```

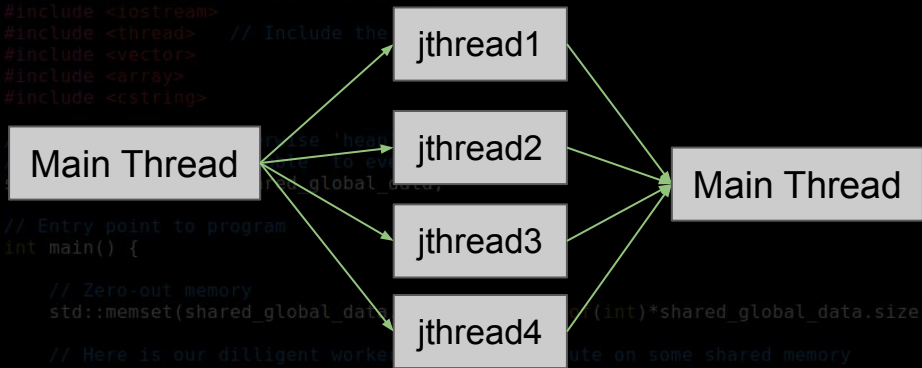


From the main thread we'll 'spawn' 4 threads in a loop -- push them into a vector (like previous) and have them work on a separate block of memory

Thread Team (3/9)

- So now that we have the idea of a 'jthread' let's do a more interesting problem
 - Let's spawn multiple threads that work on some 'shared data' to solve a problem
 - We'll increment some values in shared memory to start.

```
1 #include <file team.cpp>
2 // g++ -std=c++23 team.cpp -o prog -pthread
3 #include <iostream>
4 #include <thread> // Include the
5 #include <vector>
6 #include <array>
7 #include <string>
8
9
10 // Use 'new' to create shared global data,
11 // and 'delete' to free it.
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 threads that 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 simulat
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::jthrea
38         }
39     }
40     std::cout << "threads.size: " << thre
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_dat
46         std::cout << shared_global_data[i
47     }
48     std::cout << std::endl;
49     return 0;
50 }
```



Below is an example of 'shared memory'

Shared memory (i.e. a big array)

Thread Team (5/9)

- Here is the resulting code

```
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

```
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
17     std::memset(shared_global_data.data(), 0, sizeof(int)*shared_global_data.size());
18
19     // Here we initialize a chunk of shared memory
```

```
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 we initialize a chunk of shared memory
```

Here we initialize a chunk of shared memory

```
46         std::cout << shared_global_data[i] << " ";
47     }
48     std::cout << std::endl;
49     return 0;
50 }
```

T

```
1 // gfile team.cpp
2 // g++ -std=c++23 team.cpp -o prog -lpthread
3 #include <iostream>
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 };
```

```
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 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 (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)

```
mike@system76-pc:~/Talks/2024/french_cpp_user_group_frug$ g++ -g -W
all -std=c++23 team.cpp -o prog -lpthread
mike@system76-pc:~/Talks/2024/french_cpp_user_group_frug$ time ./pr
og
threads.size: 20
Job completed -- in main thread and printing results
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
real    0m0.006s
user    0m0.004s
sys     0m0.004s
```

```
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 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 // g++ -std=c++23 team50000.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 < 50'000; 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 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@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 ./prog
```

```
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 // g++ -std=c++23 team50000.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 < 50'000; 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 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 -W31
all -std=c++23 team50000.cpp -o prog -lpthread
mike@system76-pc:~/Talks/2024/french_cpp_user_group_frug$ time ./pr34
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 // g++ -std=c++23 team50000.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 < 50'000; 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 Round 2 (4/5)

- **Question to Audience:**
 - What is the issue? (Hint highlighted)
 - Answer: Perhaps too many threads created on stack at once
 - 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 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 // g++ -std=c++23 team50000.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 < 50'000; 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 Round 2 (5/5)

- **Live GDB Session:**

- `-gdb-set mi-async [on]`
 - Then load executable: file `./prog`
 - Then
- `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 // g++ -std=c++23 team50000.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 < 50'000; 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 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_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 i=0; i < 50000; 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     }
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 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?

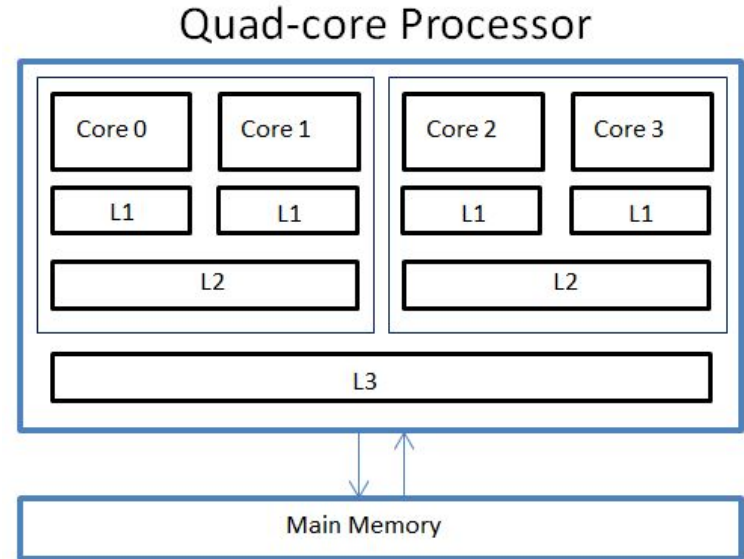
- Searching: `nl /etc/systemd/system.conf`
 - I'm allowed to have 15% of my maximum allowable threads allocated to a process on Ubuntu 22.04
 - (This seems reasonable -- I could for instance launch 25,000 threads no problem -- probably way too many though!)
- Probably not a good idea to launch this many 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

Sequential Execution is Sometimes Better

False Sharing

(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.
- 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 dilligent 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::jthread> 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::jthread(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 dilligent 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::jthread> 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::jthread(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
```

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.
- We’ll need to however think about some way to otherwise ‘keep our thread alive’
 - Recall that threads just start executing otherwise when they are invoked.

Thread Pools

- A first attempt points to some sort of 'struct' like on the right -- the example works the same
- Maybe we can just wrap the code and use 'move semantics'
 - This works -- and we get similar performance when compared to our first data-parallel working example
 - **But** we've not yet solving our problem of thread creation -- but we are getting closer, and getting some encapsulation.
- **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-----
48 auto AdditionWorker= [](size_t index, size_t jobSize){
49     for(size_t i = index*jobSize; i < (index+1) * jobSize; i++){
50         shared_global_data[i] += 1;
51     }
52 };
53
54 auto threadPool = ThreadPool<4>(AdditionWorker);
55
56 threadPool.executeAll(50000,64);
```

Condition Variables

Introducing Condition Variables

- Condition variables
 - Allows us to keep threads alive (without having to respawn new threads, which is expensive)
 - Then we can dispatch work to worker threads periodically in order to do work on a subset of data.
- This can be used as a ‘signaling pattern’
- Condition variables
 - Work with a ‘shared memory’ variable (e.g. use a boolean as a flag)
 - Typically that shared memory is protected by a mutex
 - You must use shared memory with the mutex
 - mutex is automatically acquired by the worker.

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:
- some form of synchronization (a mutex)
- a `condition_variable`
- a ‘variable’ (i.e. `ready`)

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 conditional 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 awaits' to acquire the lock
 - The 'wait' portion otherwise is where we awaken when we are notified by the producer.
 - We won't get here until we otherwise acquire the lock anyway -- so that remains the blocking operation

```
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::cout << "Consuming new value from shared_queue: " << shared_queue.front() << std::endl;
53         shared_queue.pop();
54     }
55 }
```

```
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;
```

Troubleshooting and Debugging

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

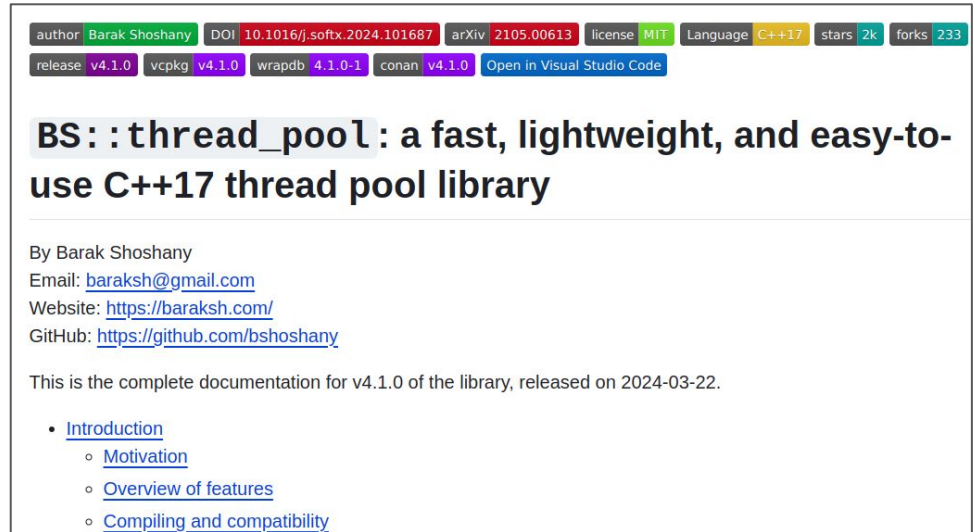
- Run and create a recording from Undo -- if you prefer instead of gdb
 - /home/mike/Downloads/undo-7.2.1/**live-record** ./prog
 - Then use 'rr' or 'udb' to replay
 - /home/mike/Downloads/undo-7.2.1/udb
prog-3008963-2024-05-14T10-37-40.324.undo
 - Try 'start' 'layout src' and then using 'n' to step 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>

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

- We went from a data parallel problem to a '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.

Thread Pool Implementation

- A complete implementation for a thread pool can be found here
 - I found it is a good example of combining these ideas of ‘submitting a task’ to a pool, and reusing threads
 - Note: This takes us into another corner of the language with promises, futures, and `packaged_tasks`.



The screenshot shows the top section of a GitHub repository page for 'BS::thread_pool'. At the top, there are several metadata tags: 'author Barak Shoshany', 'DOI 10.1016/j.softx.2024.101687', 'arXiv 2105.00613', 'license MIT', 'Language C++17', 'stars 2k', and 'forks 233'. Below these are more tags: 'release v4.1.0', 'vcpkg v4.1.0', 'wrapdb 4.1.0-1', 'conan v4.1.0', and a button 'Open in Visual Studio Code'. The main heading reads 'BS::thread_pool: a fast, lightweight, and easy-to-use C++17 thread pool library'. Below the heading, it says 'By Barak Shoshany' and provides contact information: 'Email: baraksh@gmail.com', 'Website: https://baraksh.com/', and 'GitHub: https://github.com/bshoshany'. A note states 'This is the complete documentation for v4.1.0 of the library, released on 2024-03-22.' A table of contents is visible with links for 'Introduction', 'Motivation', 'Overview of features', and 'Compiling and compatibility'.

<https://github.com/bshoshany/thread-pool>

Some High Level Takeaways

- Revisiting the gaming example -- we may have something that is visualized like this
 - 'Updates' happen in highly parallel fashion
 - Updates can be arbitrary, so need some sort of 'task' or 'thread' pool
 - Sync points (purple bars) are condition variables -- or otherwise other primitives (e.g. barrier)
- Thread errors take some thought
 - Think about the problem for a bit
 - Utilize tools like live-recorder to replay the output of program.
 - Concurrent programs are non-deterministic, and hard to reproduce!



More Resources for Going Further

Operating Systems: Three Easy Pieces

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

Concurrency

25 *Dialogue*

26 Concurrency and Threads code

27 Thread API code

28 Locks code

29 Locked Data Structures

30 Condition Variables code

31 Semaphores code

32 Concurrency Bugs

33 Event-based Concurrency

34 Summary

More Thread Patterns/Ideas

- <https://greenteapress.com/wp/semaphores/>

The Little Book of Semaphores

by Allen B. Downey

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

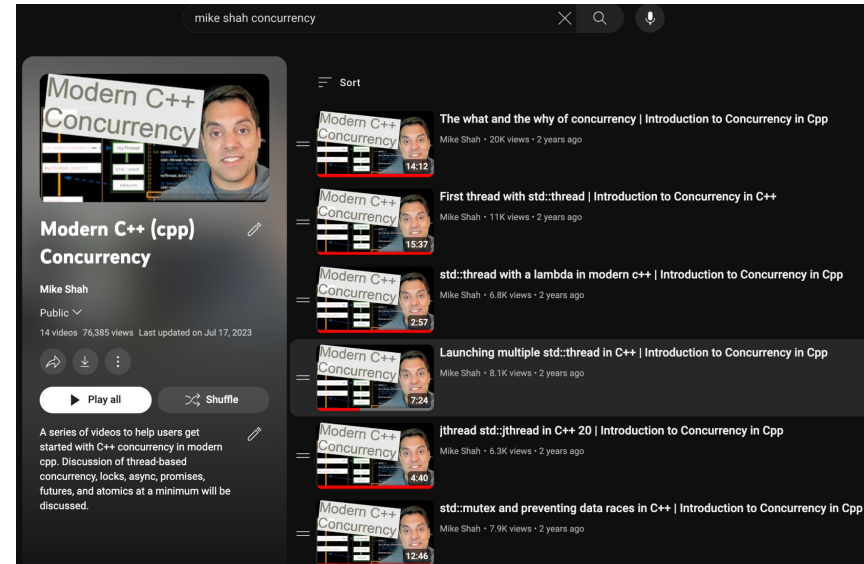
The Little Book of Semaphores is a free (in both senses of the word) textbook that introduces the principles of synchronization for concurrent programming.

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.

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

- More C++ Software Design Videos:

- <https://www.youtube.com/playlist?list=PLvv0ScY6vfd9wBflF0f6ynlDQuaeKYzyc>

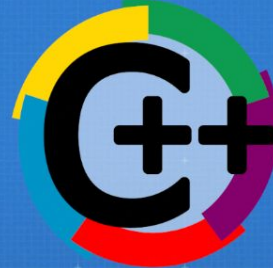
	Design Patterns - Command Pattern Explanation and Implementation in C++ Mike Shah • 12K views • 2 years ago
	Design Patterns - Singleton Pattern Explanation and Implementation in C++ Mike Shah • 4.4K views • 2 years ago
	Design Patterns - Factory Method Pattern Explanation and Implementation in C++ Mike Shah • 5.6K views • 2 years ago
	Design Patterns - Factory Method Pattern Adding More Power to Count Allocated Objects in C++ Mike Shah • 1.7K views • 2 years ago
	Design Patterns - The Extensible Factory Pattern in C++ Register Objects at Runtime Mike Shah • 2K views • 2 years ago
	Design Patterns - Iterator Pattern Explanation and usage with STL in C++ Mike Shah • 1.6K views • 2 years ago
	The Observer Design Pattern in C++ - Part 1 of n - A simple implementation Mike Shah • 3.8K views • 11 months ago
	The Observer Design Pattern in C++ - Part 2 of n - Extensibility and Abstraction Mike Shah • 1.7K views • 11 months ago

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>

Merci beaucoup C++ FRench User Group
pour le invitation!



Fundamentals of **Concurrency** **Threads, Pools, and Patterns**

-- in C++
with Mike Shah

Social: [@MichaelShah](#)
Web: [mshah.io](#)
Courses: [courses.mshah.io](#)

 **YouTube**
www.youtube.com/c/MikeShah
<http://tinyurl.com/mike-talks>

19:00 - 21:00 Tue, May 14, 2024

~60 minutes | Introductory/Advanced
Audience

Thank you!