Back to Basics: Concurrency

MIKE SHAH

Please do not redistribute slides without prior permission.

Back to Basics: Concurrency

Mike Shah, Ph.D.
@MichaelShah | mshah.io | www.youtube.com/c/MikeShah

4:45 pm MDT, Mon. October 25

60 minutes | Introductory Audience

Abstract

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

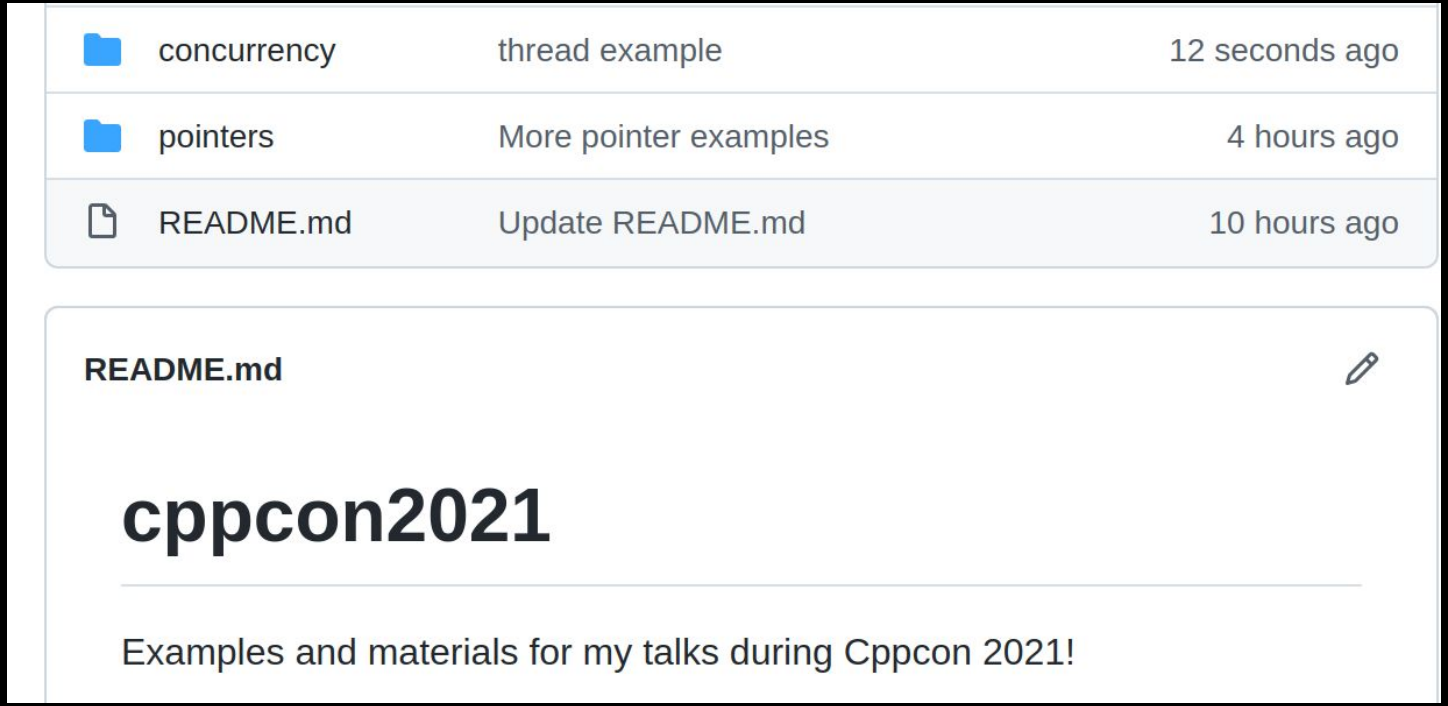
You have spent your hard earned money on a multi-core machine. But what does that mean for you as a programmer or for the consumers of your software who also spent their hard earned money on a multi-core machine? Well the deal is, you only get an increase in performance, if you know how to take advantage of your hardware. Perhaps you have also heard something about the free lunch being over for programmers?

In this talk we provide a gentle introduction to concurrency with the modern C++ std::thread library. We will introduce topics with pragmatic code examples introducing the ideas of threads and locks, and showing how these programming primitives enable concurrency. In addition, we will show the common pitfalls to be aware of with concurrency: data races, starvation, and deadlock (the most extreme form of starvation!). But don't worry--I will show you how to fix these concurrency bugs!

The session will wrap up with discussion and examples of other concurrency primitives and how to use them to write concurrent programs for common patterns(different types of locks, conditional variables, promises/futures). Attendees will leave this session being able to use threads, locks, and start thinking about architecting multithreaded software. All materials and code samples will also be posted online.

Code for the talk

Available here: https://github.com/MikeShah/cppcon2021



Who Am I?

by Mike Shah

- Assistant Teaching Professor at Northeastern University in Boston, Massachusetts.
  - I teach courses in computer systems, computer graphics, and game engine development.
  - My research in program analysis is related to performance building static/dynamic analysis and software visualization tools.
- I do consulting and technical training on modern C++, Concurrency, OpenGL, and Vulkan projects
  - (Usually graphics or games related)
- I like teaching, guitar, running, weight training, and anything in computer science under the domain of computer graphics, visualization, concurrency, and parallelism.
- Contact information and more on: www.mshah.io

What you are going to learn today (1/2)

- (So you know what to pay attention to)
  - **The foundations of Concurrency (From a beginners standpoint)**
- Pragmatically
  - You are someone who wants to learn about concurrency, and wants to learn about some of the fundamental building blocks to write for instance a multi-threaded program
  - We are going to focus on `std::thread` for creating concurrent programs.
    - We will review briefly other methods of writing concurrent programs (e.g., `std::async`)
  - I'll focus on the foundations, and my hope is this will help you also choose the right higher level primitives to write more scalable concurrent software.

Concurrency is a massive topic--so let's start from the beginning

What you are going to learn today (2/2)

- (So you know what to pay attention to)
  - **The foundations of Concurrency (From a beginners standpoint)**
- Pragmatica
  - You are s
    the funda
  - We are g
    - We
  - I'll focus
    level prin

Talk contains two core components

1. Motivation and the 'why' to concurrency
2. Several toy examples to learn

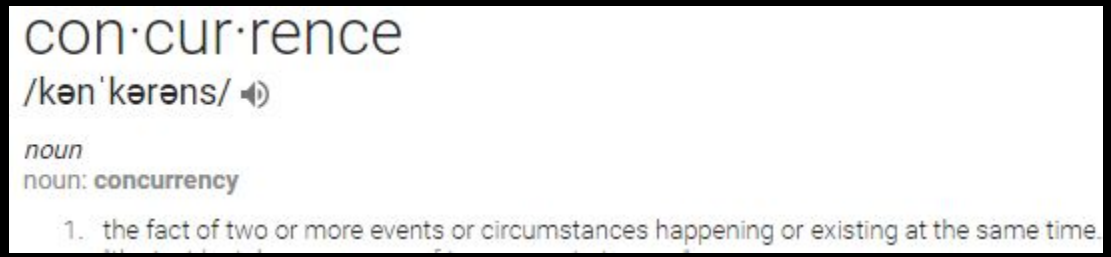
I hope this is a talk that you can revisit in the future and do a deep dive with the resources at the end

arn about some of
gram
s.
(e.g., `std::async`)
e the right higher

Concurrency is a massive topic--so let's start from the beginning

What is Concurrency?

And the possible motivation to work concurrently



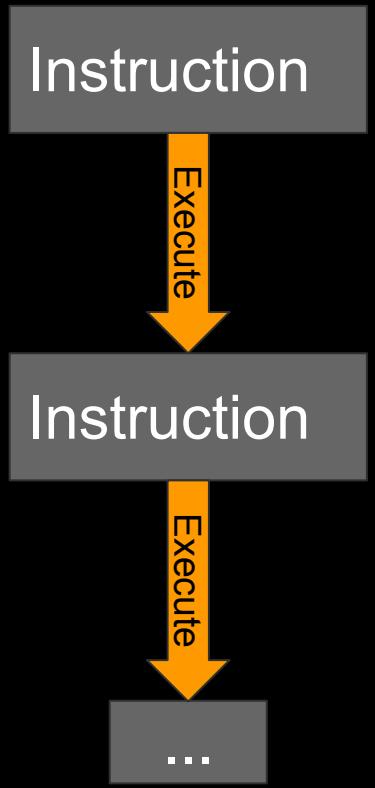
Human Psychology

- The reality is that we have to multitask:
  - We balance work, family, to-do lists, 8 half-finished hobby C++ programming projects, etc.
- Our brains (i.e., human cpus) however tend to perform best when we can focus on one task at a time
  - (e.g., I am unable to code and listen to music with any lyrics...unfortunately)

Thinking about all the *potential sequences* of events is difficult for humans:
https://www.psychologicalscience.org/news/why-humans-are-bad-at-multitasking.html

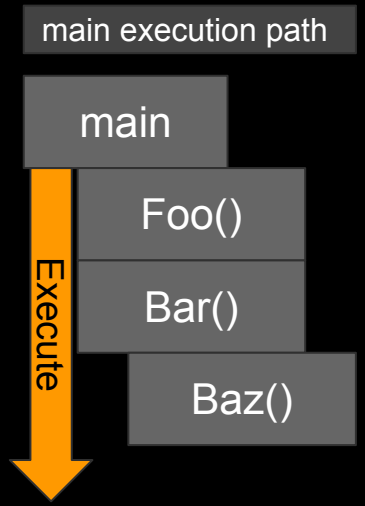
This somewhat reflects how we write software (1/3)

- Typically we have one main, sequential thread of execution
- One CPU core executes code sequentially
  - i.e. One instruction after the other.



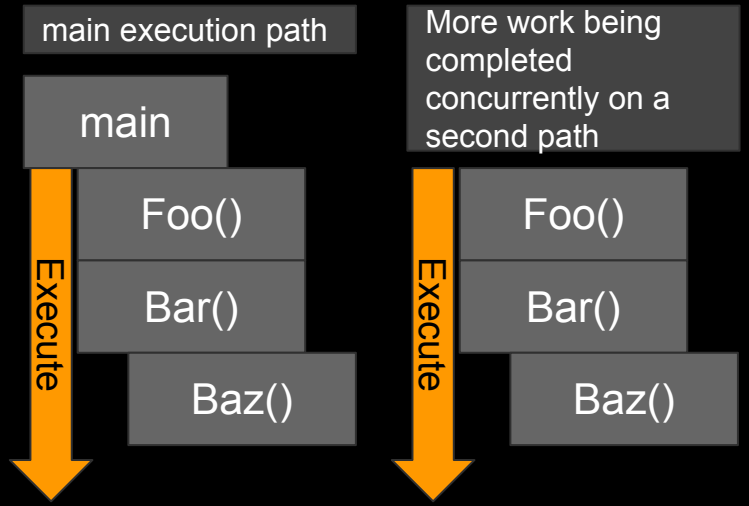
This somewhat reflects how we write software (2/3)

- Typically we have one main, sequential thread of execution
- One CPU core executes code sequentially
  - i.e. One instruction after the other.
  - We can abstract our visualization, and just show the call stack.
    - (One function calling the other, with the indentation indicating the call stack)



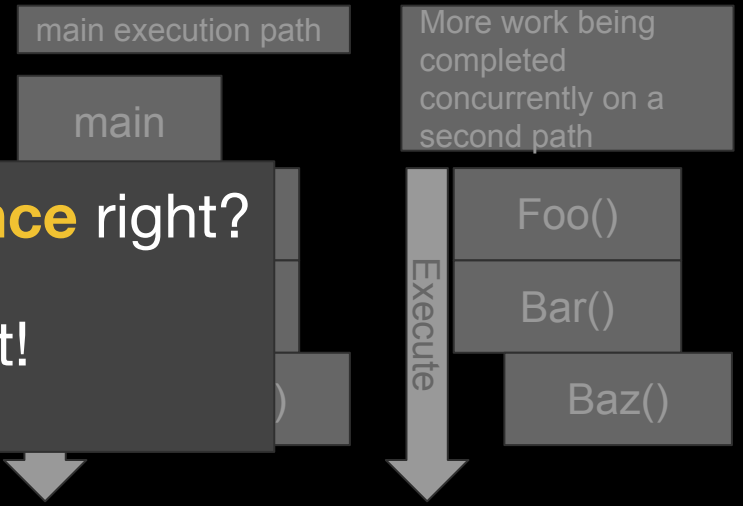
This somewhat reflects how we write software (3/3)

- Typically we have one main, sequential thread of execution
- One CPU core executes code serially
  - i.e. One instruction after the other.
  - We can abstract our visualization, and just show the call stack.
    - (One function calling the other, with the indentation indicating the call stack)
  - But! What if we could have a second path in our code executing.



This somewhat reflects how we write software (3/3)

- Typically we have one main, sequential thread of execution
- One CPU core e
  - i.e. One instruc
  - We can abstrac the call stack.
    - (One func indentation indicating the call stack)
  - But! What if we could have a second path in our code executing.

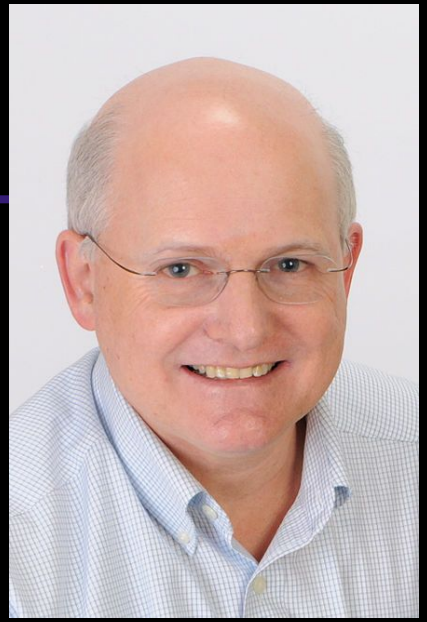


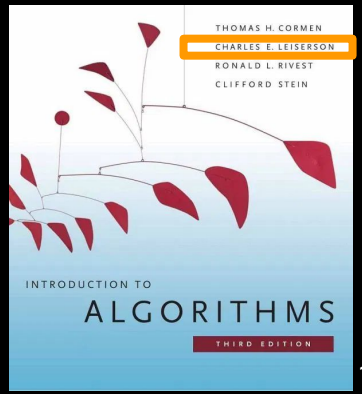
Twice the **performance** right?

Sounds great!

Performance is the currency of computing.

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

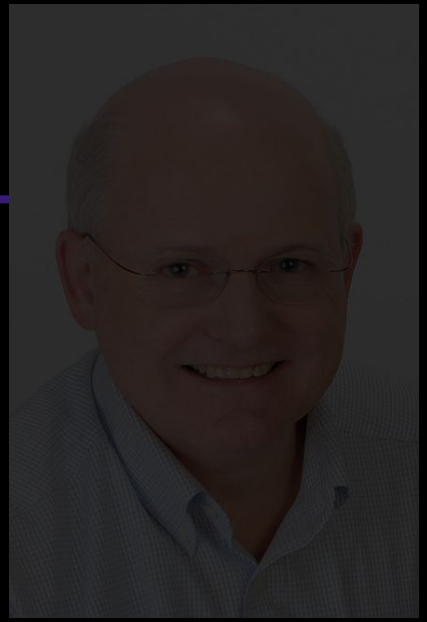


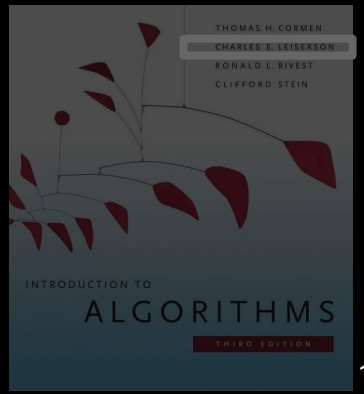


So how can we get more performance? What are our tools?

Back to our original question--*what is concurrency* ?

"**Performance is the currency of computing.** You can often "buy" needed properties [of software] with performance" - Charles Leiserson





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

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

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

Parallelism vs Concurrency (programming context) (2/4)

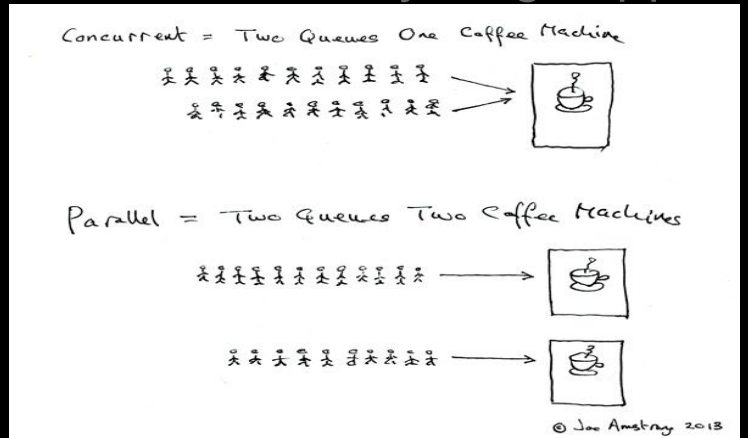
Concurrency is often used interchangeably with parallelism--so let's separate those two terms.

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

Parallelism vs Concurrency (programming context) (3/4)

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

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

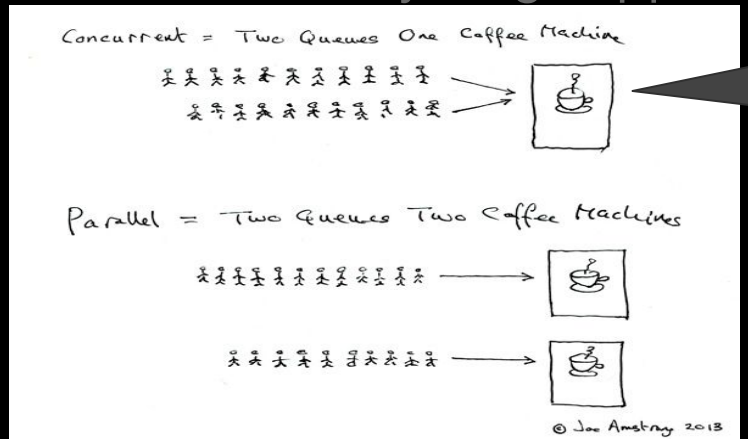


"The world is concurrent." - Joe Armstrong

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

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

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



Concurrency and parallelism (implemented correctly) should yield better performance.

"The world is concurrent." - Joe Armstrong

The Necessity of Concurrency*

- In general, concurrency (like parallelism) is used because it is necessary for a system to function.
  - 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)
- Again, concurrency is largely motivated by increased performance
  - *The potential* for more tasks to happen at once can thus increases performance (typically if we have multiple cores on our machine)

*The original title of this talk :)

Concurrency – as Heard in Music

- Ideally the instruments do not all play at once
- Ideally, at least some of the instruments are playing as well.
  - (as opposed to sitting silently)
- There is some notion of *synchronization* here.

Good Concurrency = Good Conversation (1/3)

I don’t know who these people are??

Good Concurrency = Good Conversation (2/3)

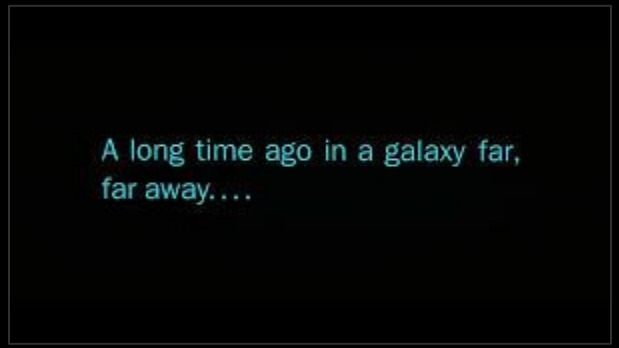
I don’t know who these people are??

Good Concurrency = Good Conversation (3/3)

I don’t know who these people are??

Does our hardware support Concurrency?

Brief Architecture History

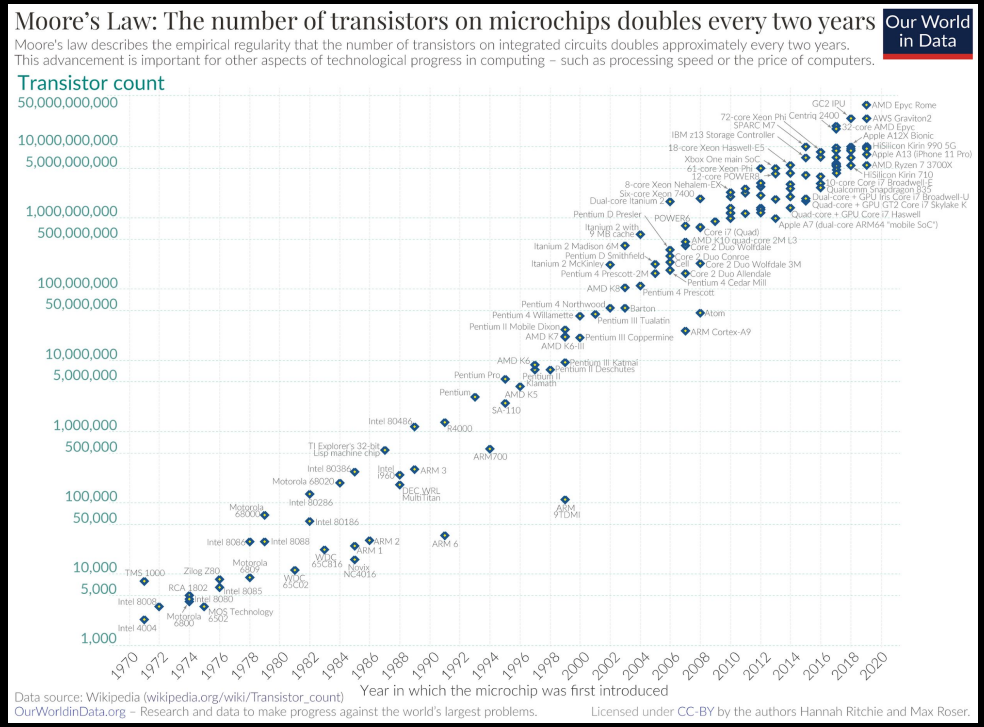


Moore's Law (1/2)

"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 on a single core
  - And largely this held true!

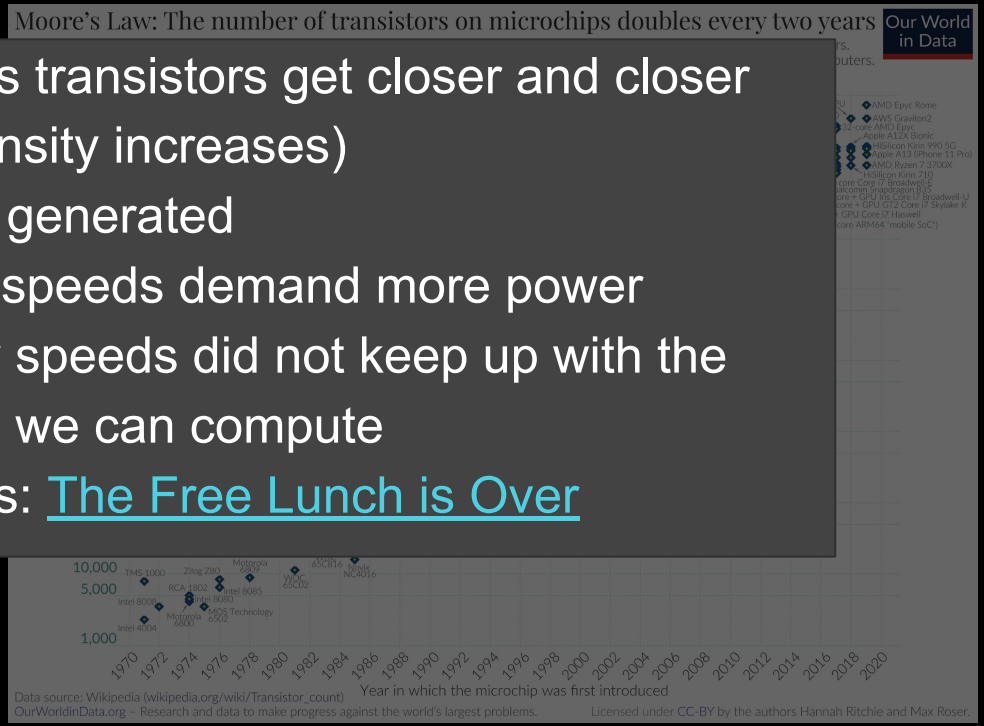


Moore's Law (2/2)

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

--Gordon Moore, Intel co-founder

- Around 1965 Gordon Moore predicted transistor double e on a sing
  - And la
    - 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
    - See Herb Sutter's: The Free Lunch is Over

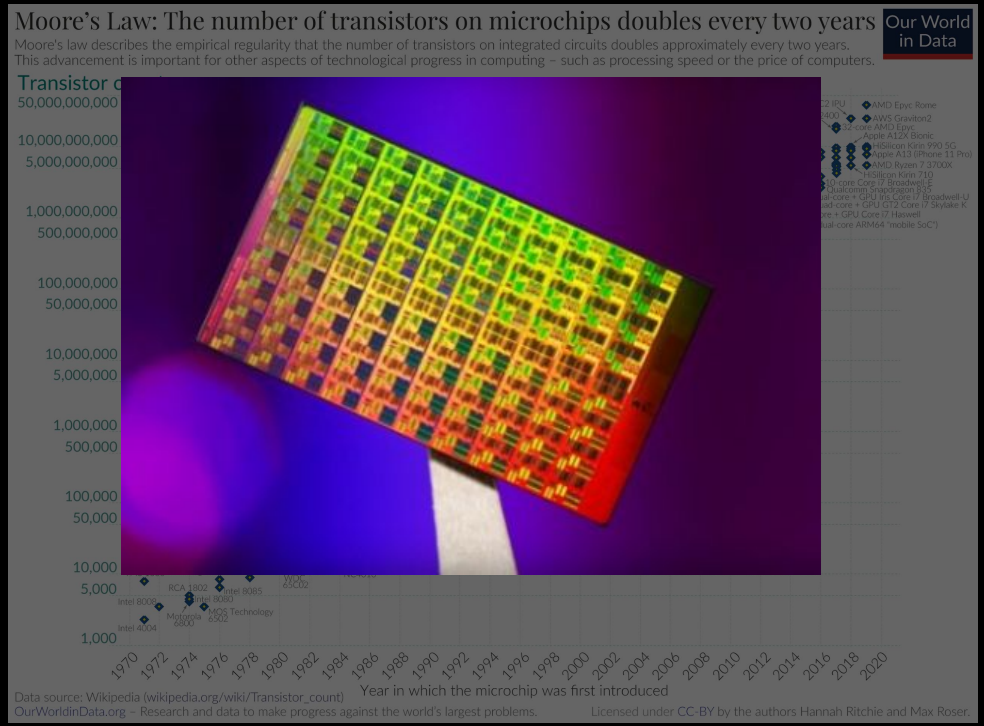


Dennard Scaling (1/3)

"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)



http://www-cs-faculty.stanford.edu/~eroberts/cs181/projects/2010-11/TechnologicalSingularity/pageviewa478.html?file=forfeasibility.html

Dennard Scaling (2/3)

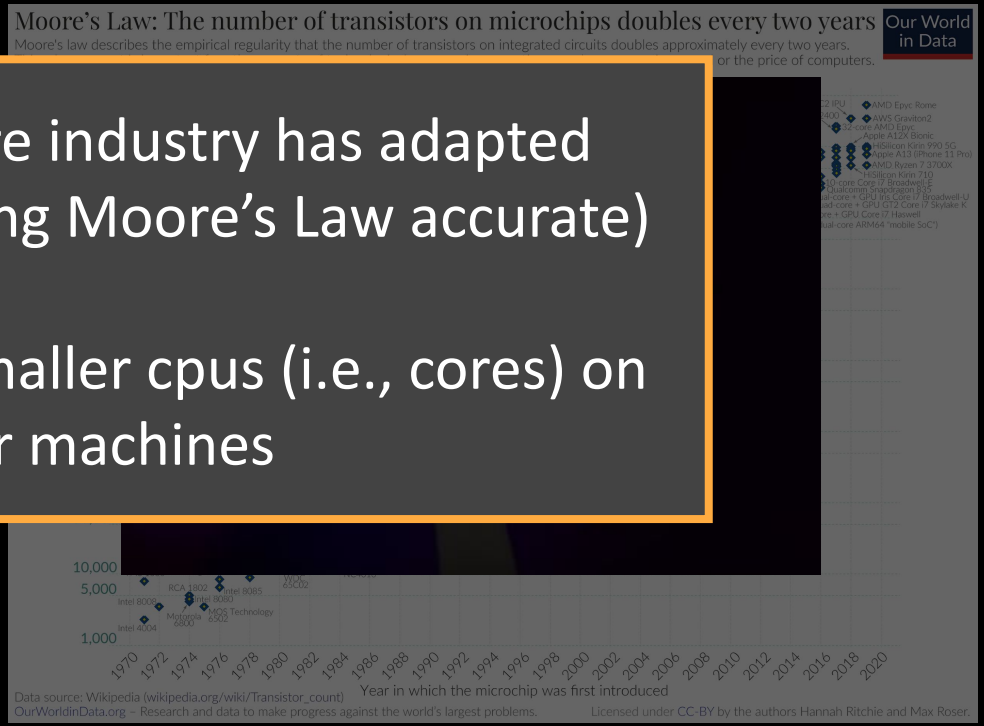
"The number of transistors incorporated in a chip will approximately double every 24 months."

--Gordon Moore, Intel co-founder

- Physically (
  transistors
  tightly toge
- Heat becom
- Energy cons
  - (i.e. Den

So the hardware industry has adapted (effectively keeping Moore's Law accurate)

We have more smaller cpus (i.e., cores) on our machines



http://www-cs-faculty.stanford.edu/~eroberts/cs181/projects/2010-11/TechnologicalSingularity/pageviewa478.html?file=forfeasibility.html

Dennard Scaling (3/3)

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

--Gordon Moore, Intel co-founder

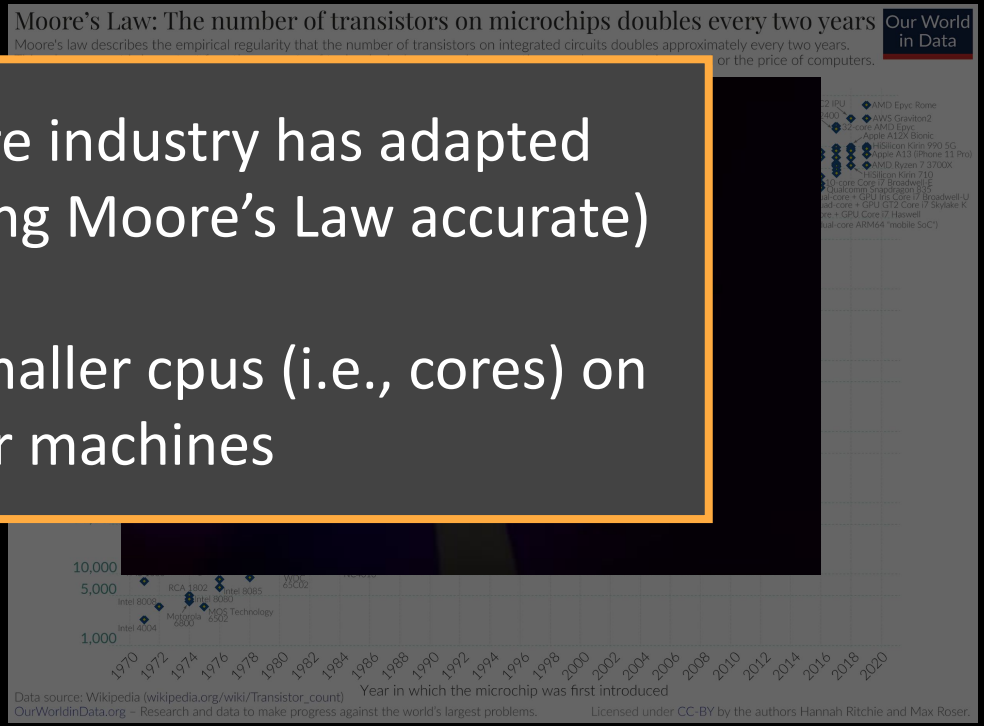
- Physically (c
  transistors a
  tightly toge
- Heat becom
- Energy cons

So the hardware industry has adapted (effectively keeping Moore's Law accurate)

We have more smaller cpus (i.e., cores) on our machines

So--does our hardware support concurrency?

Yes! Each core can be working on a separate task. (Note: A single core can also handle concurrency)



projects/2010-11/TechnologicalSingularity/pageviewa478.html?file=forfeasibility.html

(Quick Examples) X-Box One

- Yes, the hardware has multiple CPUs
- Source: http://www.videogames101.net/videogame.htm

Quick Specs

- CPU: AMD 1.75GHz 8 Core CPU
- GPU: AMD APU with 5GB dedicated VRAM
- Memory: 8GB DDR3 RAM
- Ports: 3 USB 3.0 ports, 1 Ethernet port, and 1 HDMI Output port
- Additional Storage: Upgradable internal hard drive storage and removable hard drive support.

(Quick Examples) X-Box

- X-Box Series X
  - Also 8 cores
- [Source: https://www.xbox.com/en-US/consoles/xbox-series-x#specs](https://www.xbox.com/en-US/consoles/xbox-series-x#specs)

TECH SPECS

PROCESSOR

CPU. 8X Cores @ 3.8 GHz (3.66 GHz w/SMT) Custom Zen 2 CPU

GPU. 12 TFLOPS, 52 CUs @1.825 GHz Custom RDNA 2 GPU

SOC Die Size. 360.45 mm

Process. 7nm Enhanced

MEMORY & STORAGE

Memory. 16GB GDDR6 w/320 bit-wide bus

Memory Bandwidth. 10GB @ 560 GB/s, 6GB @ 336 GB/s.

Internal Storage. 1TB Custom NVME SSD

I/O Throughput. 2.4 GB/s (Raw), 4.8 GB/s (Compressed, with custom hardware decompression block)

Expandable Storage. Support for 1TB Seagate Expansion Card for Xbox Series X|S matches internal storage exactly (sold separately). Support for USB 3.1 external HDD (sold separately).

VIDEO CAPABILITIES

Gaming Resolution. True 4K

High Dynamic Range. Up to 8K HDR

Optical Drive. 4K UHD Blu-Ray

Performance Target. Up to 120 FPS

HDMI Features. Auto Low Latency Mode. HDMI Variable Refresh Rate. AMD FreeSync.

(Quick Examples) PS5

- Yes, also utilizing multiple cores
- https://en.wikipedia.org/wiki/PlayStation_5

CPU	Custom 8-core AMD Zen 2, variable frequency, up to 3.5 GHz
Memory	16 GB GDDR6 SDRAM 512 MB DDR4 RAM (for background tasks)[1]
Storage	Custom 825 GB SSD
Removable storage	Internal (user upgradeable) NVMe M.2 SSD, or external USB-based HDD
Display	Video output formats HDMI: 720p, 1080i, 1080p, 4K UHD, 8K UHD
Graphics	Custom AMD RDNA 2, 36 CUs @ variable frequency up to 2.23 GHz

More cores = The general trend in architecture.

- The idea of increasing cores to has existed for a long time looking back at early supercomputers
  - 1964 - Seymour Cray pioneers pipelining
  - 1972 - CDC 8600 Multiprocessor
  - 1975 - Cray-1 pioneered vector processing (i.e. SIMD)

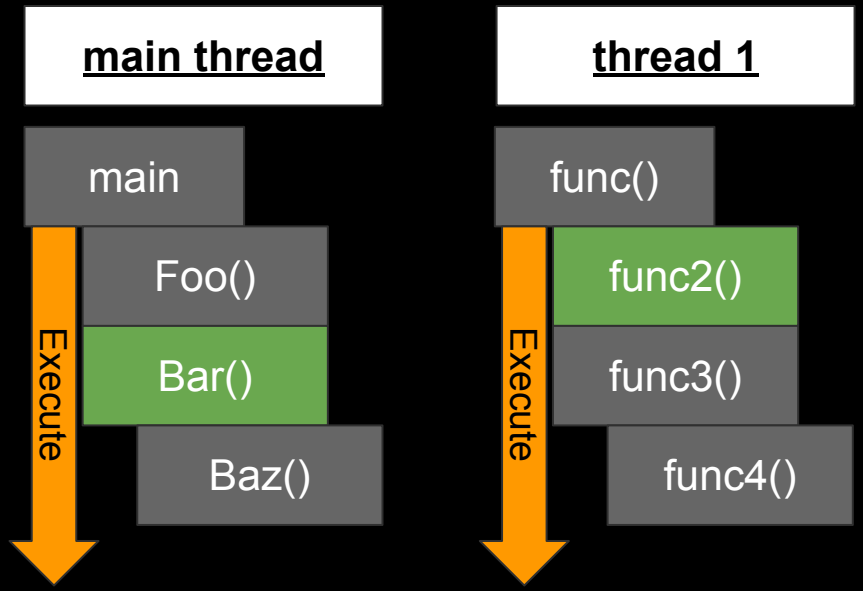
Now for code...(almost)

Let’s see our first mechanism for concurrent programming

Reference to American movie Jerry Maguire in which actor Tom Cruise yells “Show me the money” loudly into the phone to win a deal.

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.)



Threads

Often defined as a “lightweight process”

High Level View of Thread

- 1 Process (i.e. your application) can have many threads:
  - Each thread shares the same code, data, and kernel context
  - A thread has its own thread id (TID)
  - A thread has its own logical control flow
  - A thread has its own stack for local variables

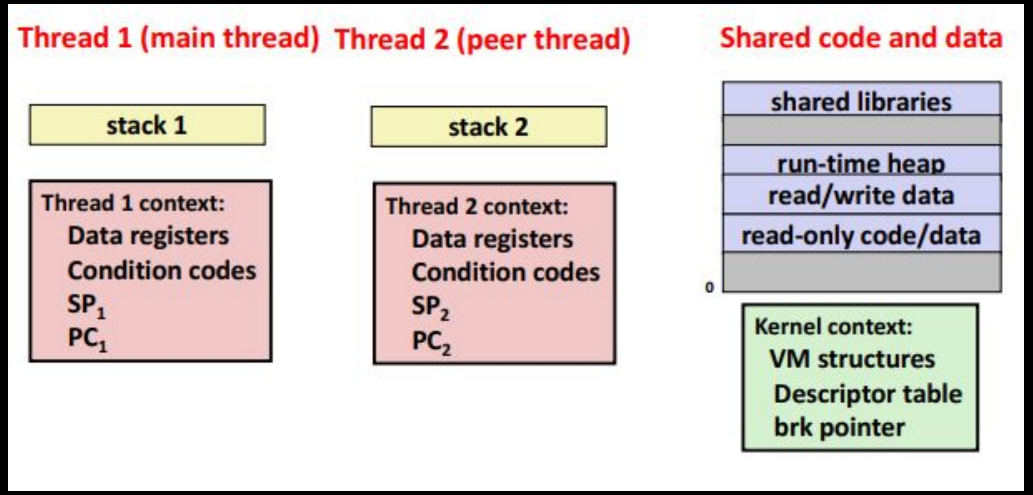


Figure from: Computer Systems a Programmer's Perspective 3rd Edition

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

Threads in Modern C++

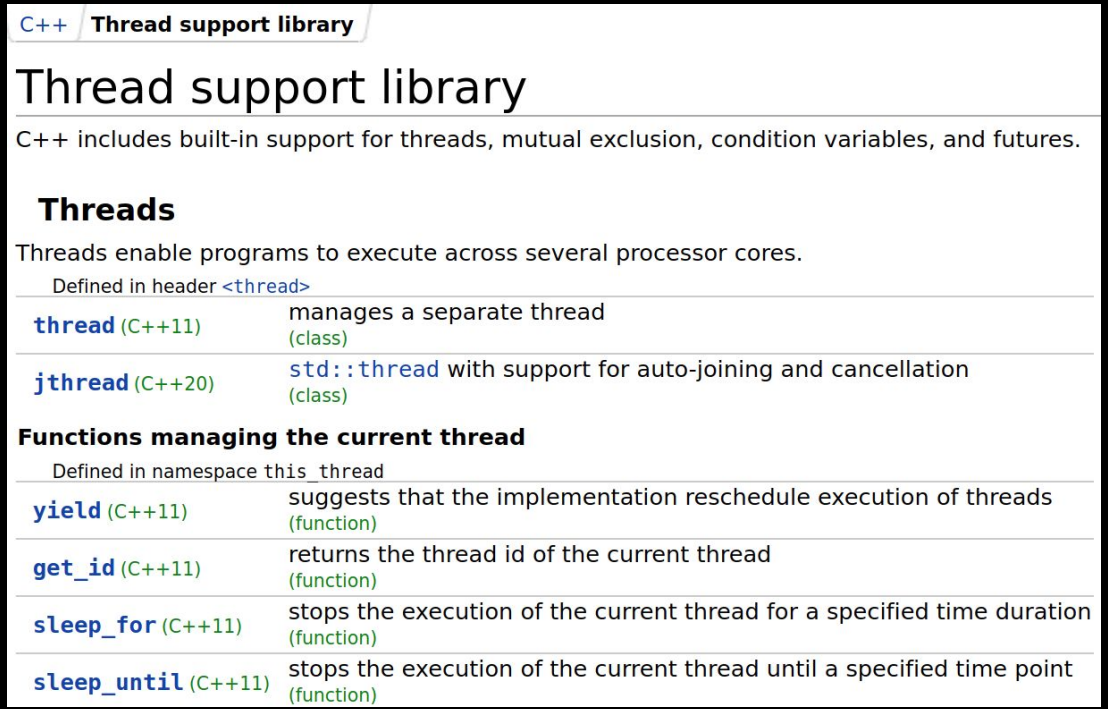
The `std::thread`

Thread Libraries

- Before C++11/14/17/20, 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.

#include <thread> | http://en.cppreference.com/w/cpp/thread

- With Modern C++
  - Here's what you get!
  - Let's start with 'thread'



C++ | Thread support library

Thread support library

C++ includes built-in support for threads, mutual exclusion, condition variables, and futures.

Threads

Threads enable programs to execute across several processor cores.

Defined in header <thread>

thread (C++11)	manages a separate thread (class)
jthread (C++20)	std::thread with support for auto-joining and cancellation (class)

Functions managing the current thread

Defined in namespace this_thread

yield (C++11)	suggests that the implementation reschedule execution of threads (function)
get_id (C++11)	returns the thread id of the current thread (function)
sleep_for (C++11)	stops the execution of the current thread for a specified time duration (function)
sleep_until (C++11)	stops the execution of the current thread until a specified time point (function)

Thread Example - Launching a thread (1/2)

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

```cpp
// @file thread1.cpp
// g++ -std=c++17 thread1.cpp -o prog -lpthread
#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;
}
```

Thr

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

```
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!
```

```
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)

```cpp
// @file thread1.cpp
// g++ -std=c++17 thread1.cpp -o prog -lpthread
#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” (2/13)

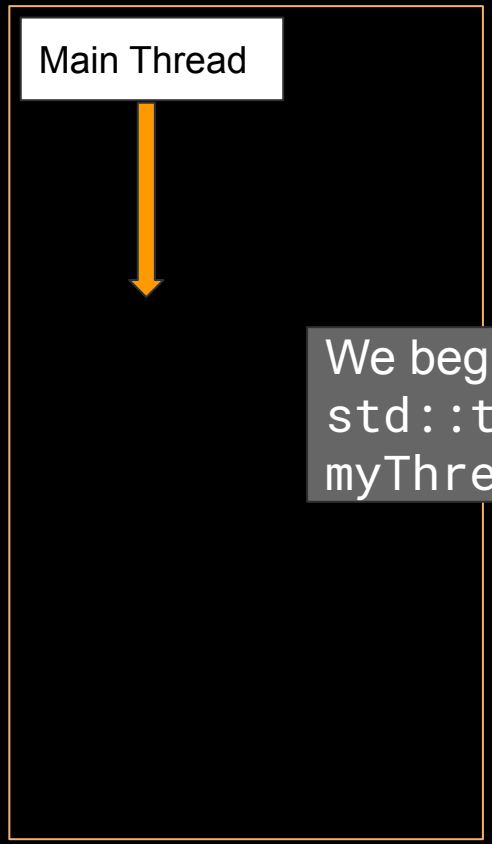
Main Thread

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

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

```cpp
// @file thread1.cpp
// g++ -std=c++17 thread1.cpp -o prog -lpthread
#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;
}
```

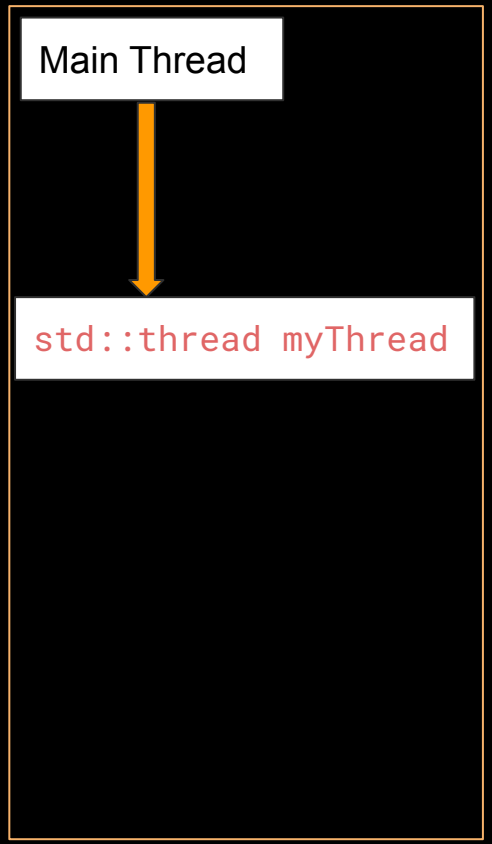
Visual execution of “Hello Thread” (3/13)



We begin constructing `std::thread myThread`

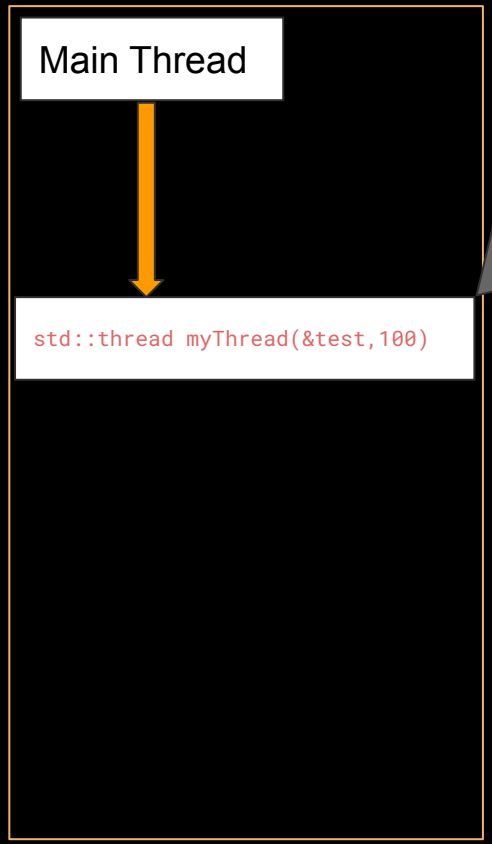
```cpp
// @file thread1.cpp
// g++ -std=c++17 thread1.cpp -o prog -lpthread
#include <iostream>
#include <thread>   // Include the thread library

// 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” (4/13)




```cpp
// @file thread1.cpp
// g++ -std=c++17 thread1.cpp -o prog -lpthread
#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” (5/13)



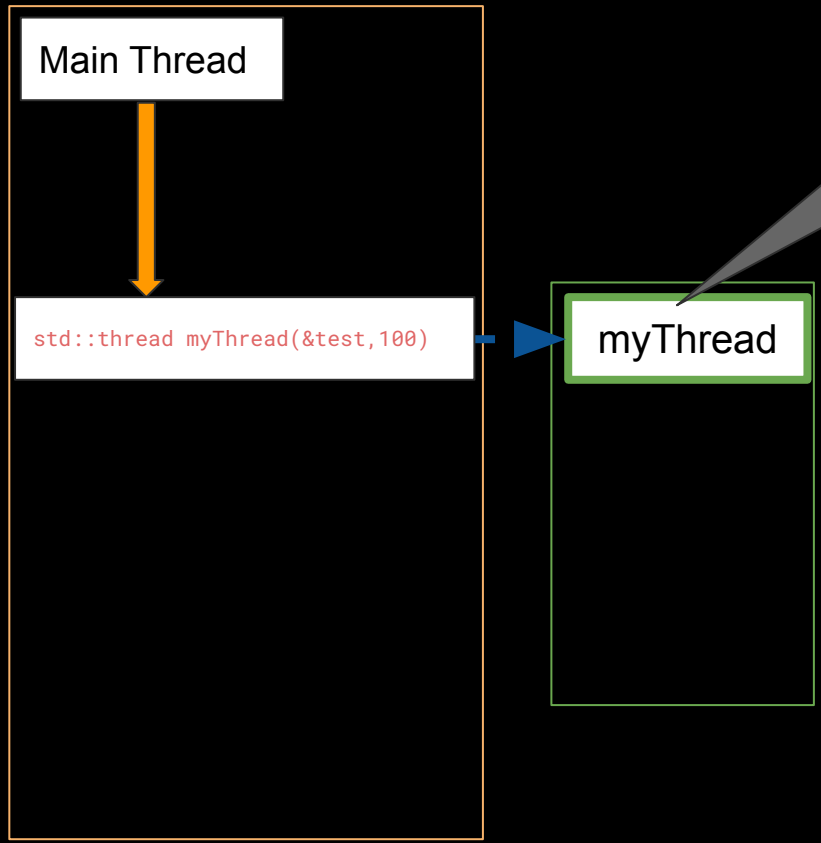
Our new thread will begin executing it’s logical control flow from the ‘test’ function. *separately* from main()

The thread will start executing immediately on construction

(Remember, threads shares code and the heap)

```cpp
// @file thread1.cpp
// g++ -std=c++17 thread1.cpp -o prog -lpthread
#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;
}
```

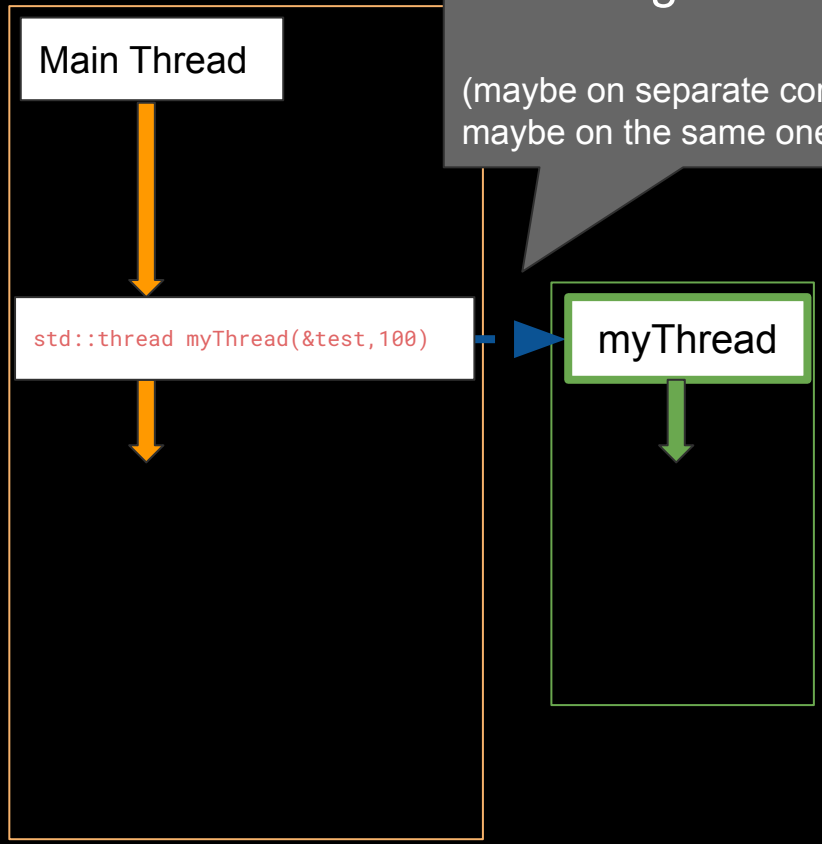
Visual execution of “Hello Thread” (2/10)



So now we have two “threads” executing

```
prog -lpthread

4 #                <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  }
```

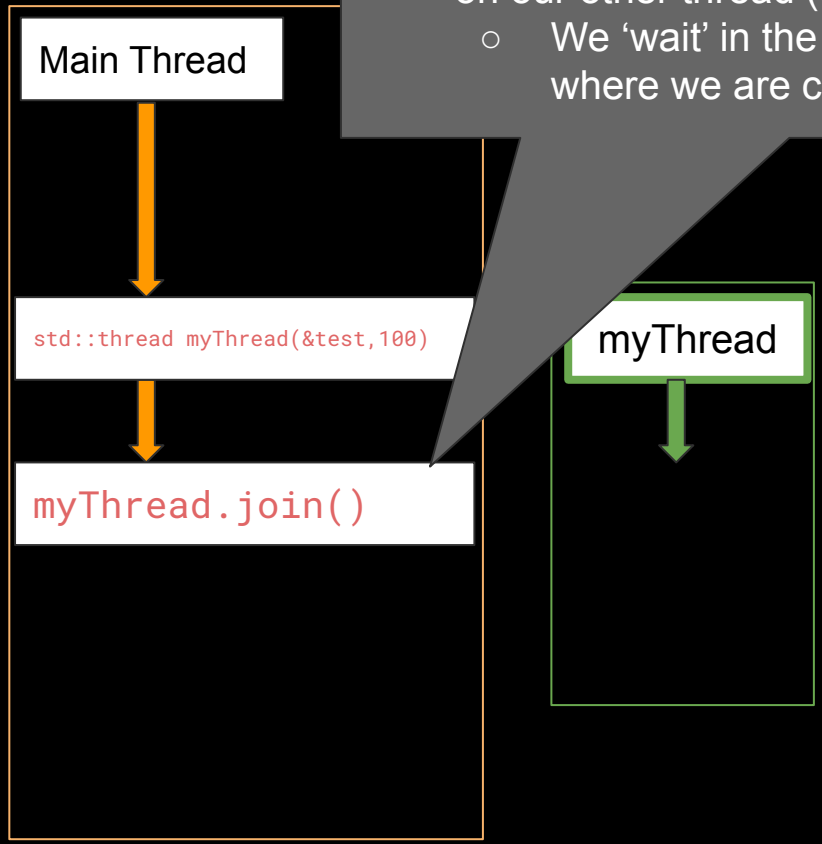
Visual execut ad" (7/13)




```
thread1.cpp
td=c++17 thread1.cpp -o prog -lpthread
<iostream>
<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;
}
```

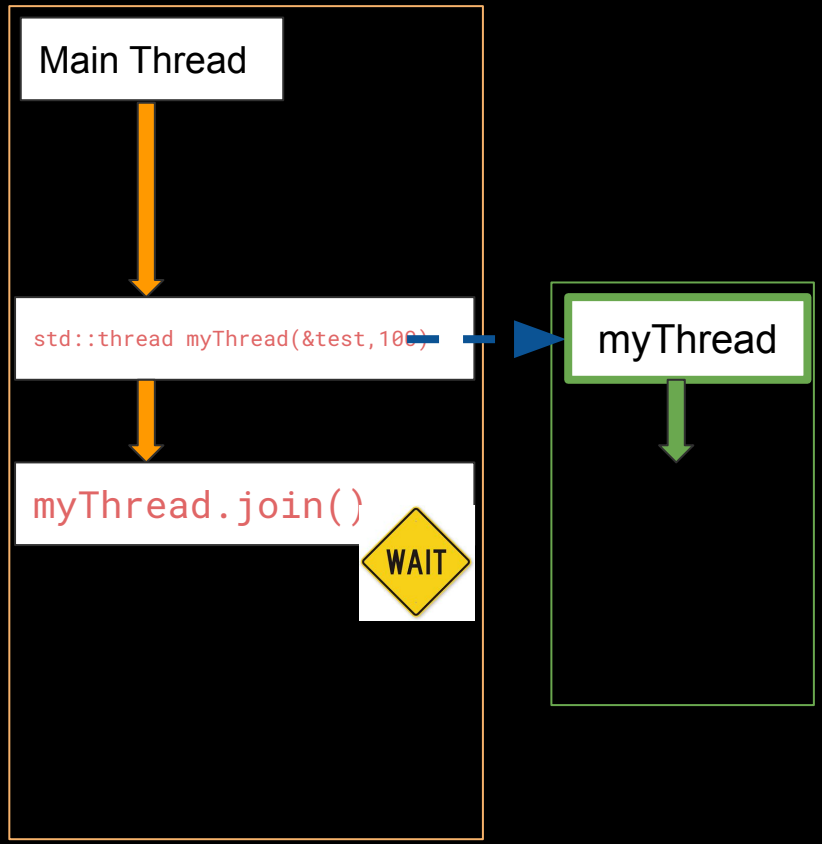
Visual exe

- 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



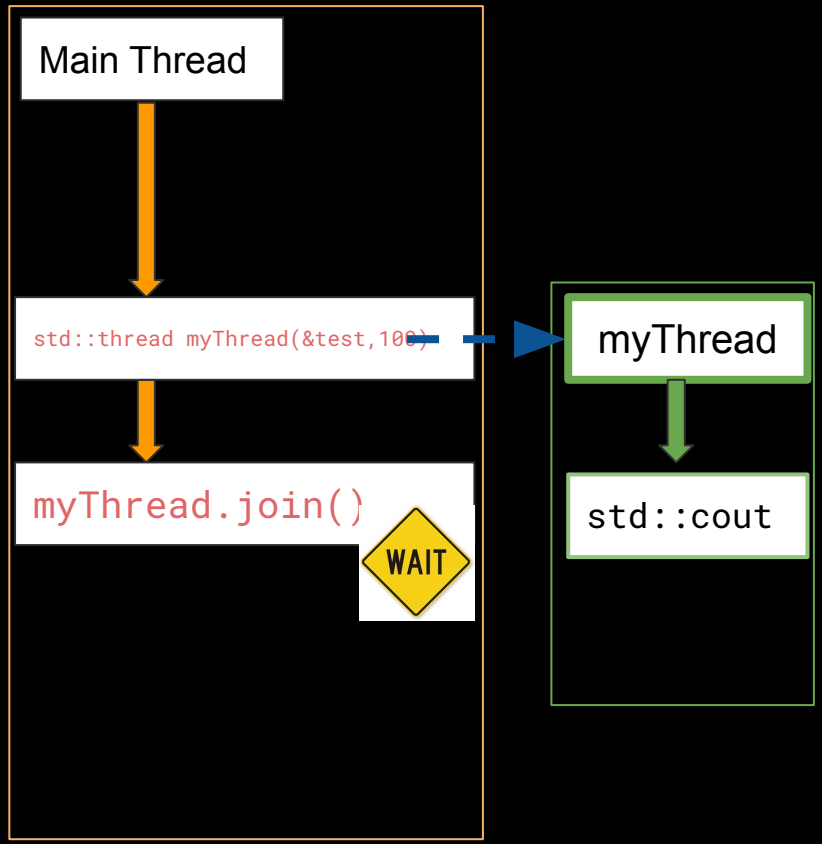

```
g -lpthread

hread 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” (9/13)



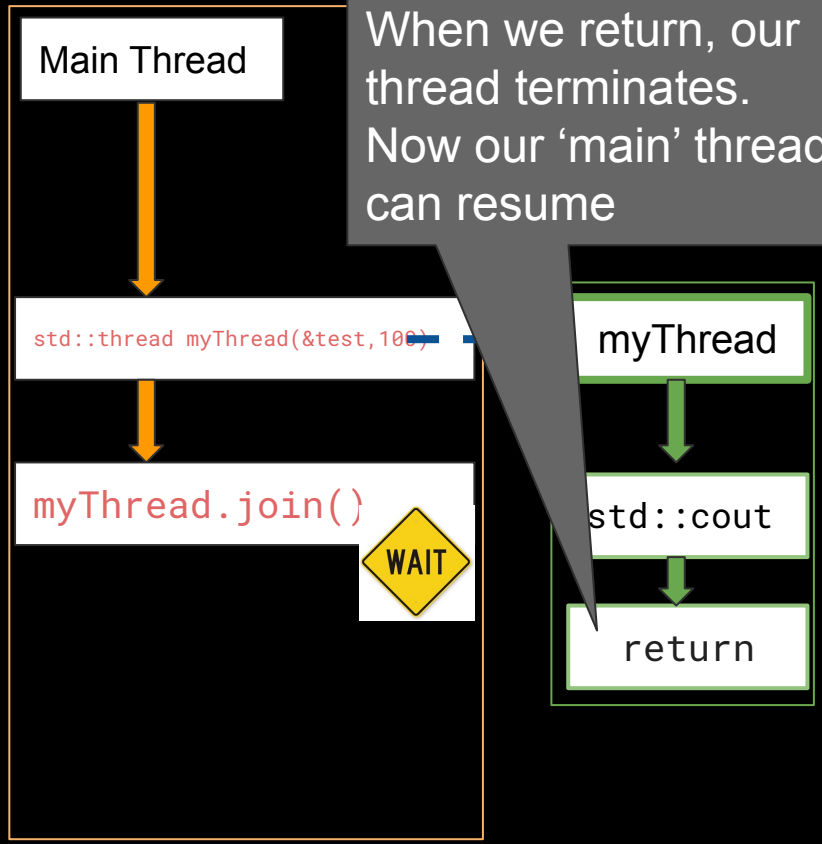

```cpp
// @file thread1.cpp
// g++ -std=c++17 thread1.cpp -o prog -lpthread
#include <iostream>
#include <thread>   // Include the thread library

// 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” (10/13)



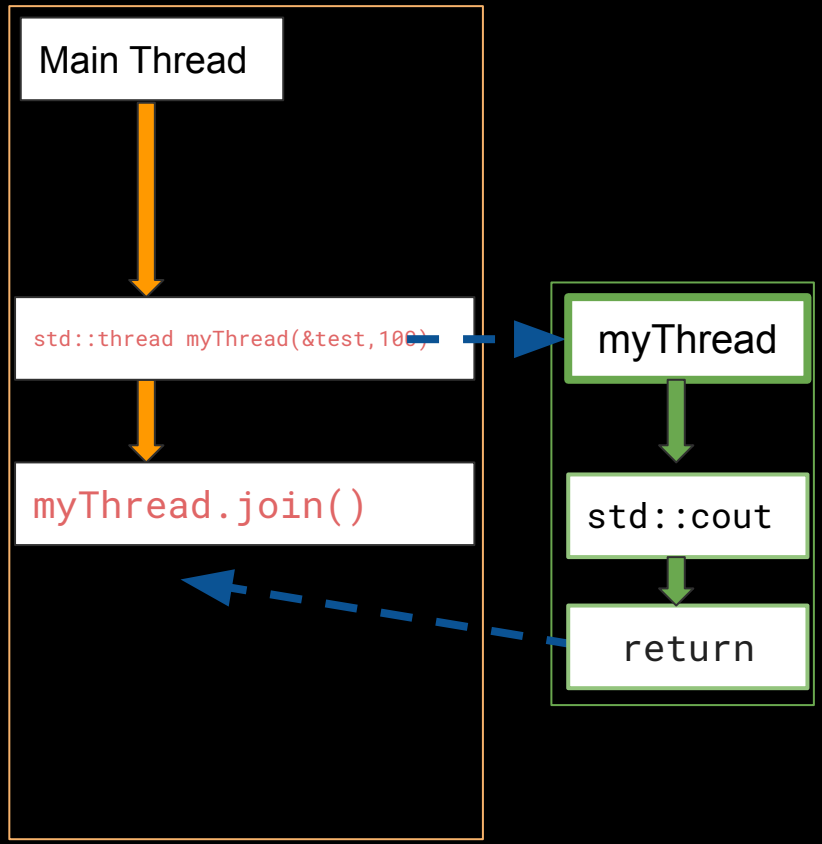

```cpp
// @file thread1.cpp
// g++ -std=c++17 thread1.cpp -o prog -lpthread
#include <iostream>
#include <thread>   // Include the thread library

// 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” (11/13)



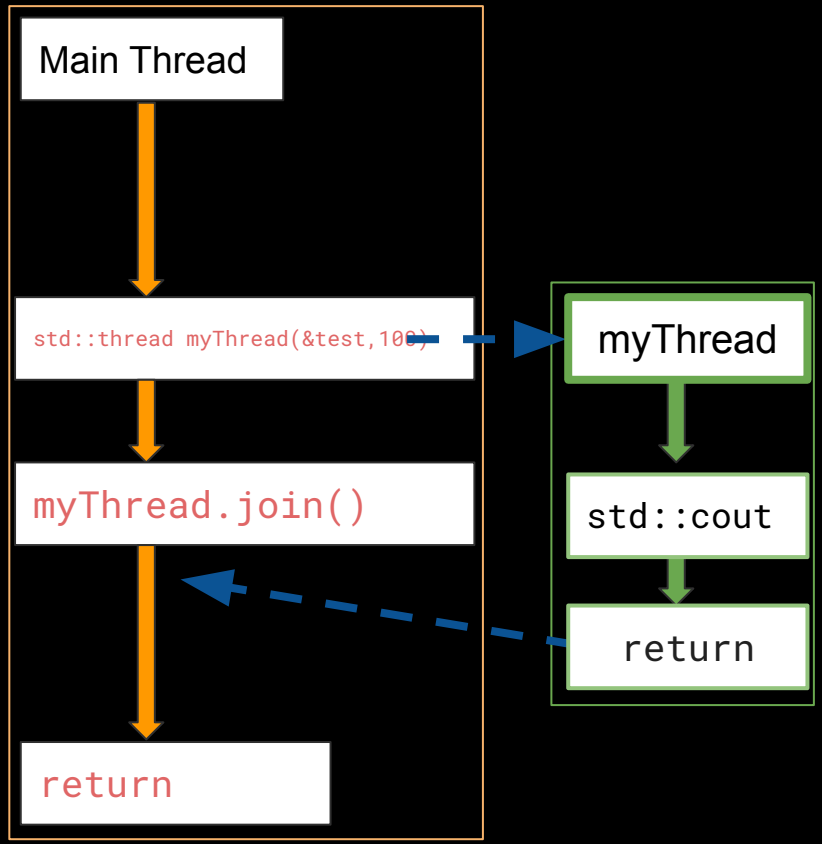

```cpp
// @file thread1.cpp
// g++ -std=c++17 thread1.cpp -o prog -lpthread
#include <iostream>
#include <thread>   // Include the thread library

// 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)




```cpp
// @file thread1.cpp
// g++ -std=c++17 thread1.cpp -o prog -lpthread
#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” (13/13)




```cpp
// @file thread1.cpp
// g++ -std=c++17 thread1.cpp -o prog -lpthread
#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;
}
```

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!

```cpp
// @file thread2.cpp
// g++ -std=c++17 thread2.cpp -o prog -lpthread
#include <iostream>
#include <thread>   // Include the thread library

int main() {

    // This time create a lambda function
    auto lambda = [](int x){
        std::cout << "Hello from our thread!" << std::endl;
        std::cout << "Argument passed in:"   << x << std::endl;
    };

    // Create a new thread with our lambda this time
    std::thread myThread(lambda,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;
}
```

Now how about if we wanted 10 threads

- 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!

```cpp
// @file thread3.cpp
// g++ -std=c++17 thread3.cpp -o prog -lpthread
#include <iostream>
#include <thread>   // Include the thread library
#include <vector>

int main() {

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

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

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

    return 0;
}
```

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

- So here we create each of our threads and join them

```cpp
// @file thread3.cpp
// g++ -std=c++17 thread3.cpp -o prog -lpthread
#include <iostream>
#include <thread>   // Include the thread library
#include <vector>

int main() {

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

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

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

    return 0;
}
```

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

- So here we create each of our threads and join them

```
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!
```

```cpp
// @file thread3.cpp
// g++ -std=c++17 thread3.cpp -o prog -lpthread
#include <iostream>
#include <thread>   // Include the thread library
#include <vector>

int main() {

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

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

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

    return 0;
}
```

Now how about if we

- So here we create each of our threads and join th

```
mike:concurrency$ g++ -std=c++17 thread3          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!
```

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

```
 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 them

```
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!
```

By joining our threads immediately after launching our code, we've effectively made our program sequential (i.e. no performance gain)

```
 8
 9      // 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 1

- So here we create each of our threads and join them

```
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!
```

```
1  // @file thread
2  // g++ -std=c++
3  #include <iostr
4  #include <thread>    // In                library
5  #include <vector>
6
7  int main() {
8
9      // This time                mbda function
10     auto lambda              x){
11         std::c               hread.get_id:" << std::this_thread::get_id() << std::endl;
12         std::                "Argument passed in:"    << x << std::endl;
13     };
14
15     std:                 tor<std::thread> threads;
16     //                   eate a collection of threads
17     f                (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 }
```

Here's the fix and new output

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

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 codes does the right thing--threads are immediately launched and not sequentially waited upon)

```cpp
// @file thread4.cpp
// g++-10 -std=c++20 thread4.cpp -o prog -lpthread
#include <iostream>
#include <thread>   // Include the thread library
#include <vector>

int main() {

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

    // Note: We now have a jthread
    //       No joins in the program
    std::vector<std::jthread> threads;
    // Create a collection of threads
    for(int i=0; i < 10; i++){
        threads.push_back(std::jthread(lambda,i));
    }

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

    return 0;
}
```

What about threads working together?

How do threads work with data?

1000 threads working together (1/4)

- Assume you have some shared task:
  - Perhaps 1000 threads counting the sizes of files in a directory to find the total bytes of the files in a directory
    - i.e., a parallel wc tool
  - Each of those threads then needs to write to some ‘shared value’ to sum the total.
- On the example on the right, let’s simplify and just have 1000 threads work together to increment a value

```cpp
// @file thread5.cpp
// g++ -std=c++17 thread5.cpp -o prog -lpthread
#include <iostream>
#include <thread>   // Include the thread library
#include <vector>

// Some shared value
// A little ugly that it's a static global  but it's a toy example.
static int shared_value = 0;

void increment_shared_value(){
    shared_value = shared_value + 1;
}

int main() {
    std::vector<std::thread> threads;
    // Create a collection of threads
    for(int i=0; i < 1000; i++){
        threads.push_back(std::thread(increment_shared_value));
    }
    // Join our threads
    for(int i=0; i < 1000; i++){
        threads[i].join();
    }
    // Retrieve our result
    std::cout << "Result = " << shared_value << std::endl;

    return 0;
}
```

1000 threads working together (2/4)

- Assume you have some shared task:
  - Perhaps 1
    sizes of fil
    total bytes
    - i.e.,
  - Each of th
    write to so
    total.
- On the example on the right, let's simplify and just have 1000 threads work together to increment a value

Here we launch and join our 1000 threads

```cpp
// @file thread5.cpp
// g++ -std=c++17 thread5.cpp -o prog -lpthread
#include <iostream>
#include <thread>   // Include the thread library
#include <vector>

// Some shared value
// A little ugly that it's a static global  but it's a toy example.
static int shared_value = 0;

void increment_shared_value(){
    shared_value = shared_value + 1;
}

int main() {
    std::vector<std::thread> threads;
    // Create a collection of threads
    for(int i=0; i < 1000; i++){
        threads.push_back(std::thread(increment_shared_value));
    }
    // Join our threads
    for(int i=0; i < 1000; i++){
        threads[i].join();
    }
    // Retrieve our result
    std::cout << "Result = " << shared_value << std::endl;

    return 0;
}
```

1000 threads working together (3/4)

- Assume you have some shared task:
  - Each thread calls ‘`increment_shared_value`’ and adds 1 to the shared_value
  - write to some “shared value” to sum the total.
- On the example on the right, let’s simplify and just have 1000 threads work together to increment a value

```cpp
// @file thread5.cpp
// g++ -std=c++17 thread5.cpp -o prog -lpthread
#include <iostream>
#include <thread>   // Include the thread library
#include <vector>

// Some shared value
// A little ugly that it's a static global  but it's a toy example.
static int shared_value = 0;

void increment_shared_value(){
    shared_value = shared_value + 1;
}

int main() {
    std::vector<std::thread> threads;
    // Create a collection of threads
    for(int i=0; i < 1000; i++){
        threads.push_back(std::thread(increment_shared_value));
    }
    // Join our threads
    for(int i=0; i < 1000; i++){
        threads[i].join();
    }
    // Retrieve our result
    std::cout << "Result = " << shared_value << std::endl;

    return 0;
}
```

1000 threads working together (4/4)

- Ass
  tasl
  - i.e., a parallel wc tool
  - Each of those
    write to some
    total.
- On the example
  simplify and jus
  work together t

And...here's the result.
It's not deterministic! (We probably want the correct answer more often than 2/6 tries...)

Why?

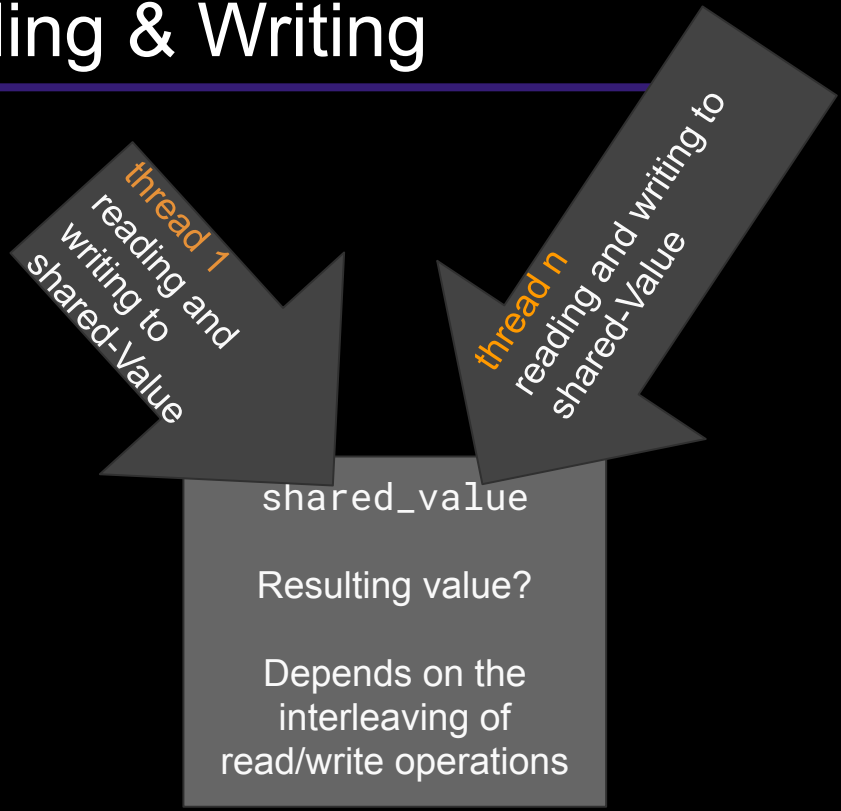
```
mike:concurrency$ g++-10 -std=c++17 thread5.cpp -o prog -lpthread
mike:concurrency$ ./prog
Result = 998
mike:concurrency$ ./prog
Result = 1000
mike:concurrency$ ./prog
Result = 1000
mike:concurrency$ ./prog
Result = 997
mike:concurrency$ ./prog
Result = 998
mike:concurrency$ ./prog
Result = 996
```

Pitfalls of Concurrent Programming (Specifically with threads)

Data Races, Starvation, and Deadlock

Problem with Threads -- Reading & Writing

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



Solving a Data Race

- Fixing Data Races
  - We can use a ‘lock’ to protect data, so that only one thread at a time can thus access the memory.
  - In C++, we have these available, and they are known as ‘mutexes’
- A mutex, allows ‘mutual exclusion’ to a block of code.
  - Thus, the operation is ‘atomic’ in the sense that only 1 operation can happen while the lock is held.
  - *Analogy:*
    - *Think about having exactly 1 key to your home, and you always carry the key with you.*
    - *Only the person who has the key can access the house.*
    - *When the person enters, they lock the door*
    - *When the person leaves, they can pass on the key to someone else to enter, who will also lock the door when they enter.*

std::mutex (1/2)

- Four new lines of code added
  - The mutex library
  - A global lock
  - A lock and unlock call on our global lock

```cpp
// @file thread6.cpp
// g++ -std=c++17 thread6.cpp -o prog -lpthread
#include <iostream>
#include <thread>   // Include the thread library
#include <vector>
# include <mutex>  // New library

// Some shared value
static int shared_value = 0;
std::mutex gLock; // A global lock

void increment_shared_value(){
    gLock.lock();
        shared_value = shared_value + 1;
    gLock.unlock();
}

int main() {
    std::vector<std::thread> threads;
    // Create a collection of threads
    for(int i=0; i < 1000; i++){
        threads.push_back(std::thread(increment_shared_value));
    }
    // Join our threads
    for(int i=0; i < 1000; i++){
        threads[i].join();
    }
    // Retrieve our result
    std::cout << "Result = " << shared_value << std::endl;

    return 0;
}
```

std::mutex (2/2)

- Four new lines of code added
  - The mutex library
  - A global lock
  - A lock and unlock call on our global lock
- Results look good to me!

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

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

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

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

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

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

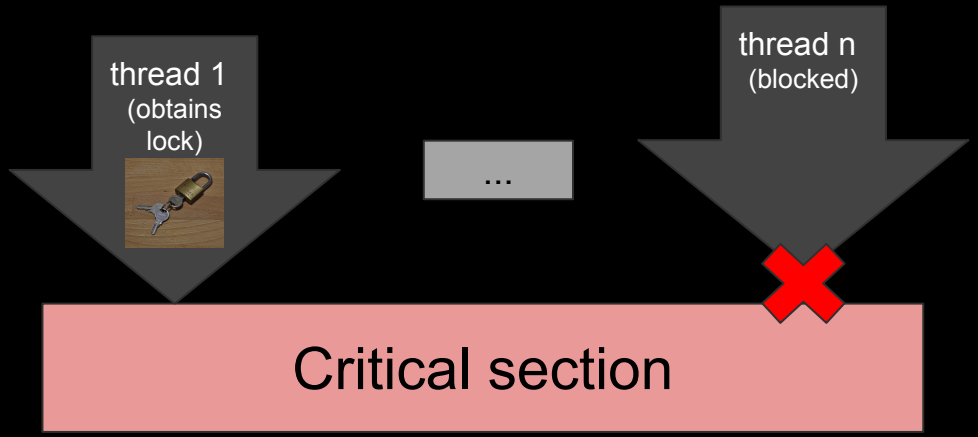
- So what our lock is doing is providing access to only one thread at a time (mutually exclusive access).
  - This region is called the ‘critical section’ that is protected by the lock.
  - Critical because we only want one thread at a time to enter and modify the shared state in the program.

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

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

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

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



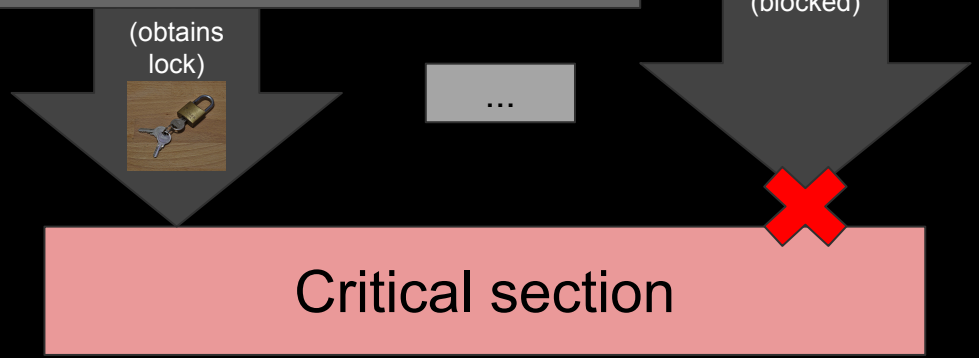
std::mutex and mutual exclusion (4/4)

- So wha
  providi
  thread
  exclusi
  - Thi
    sec
    lock.
  - Critical because we only want one thread at a time to enter and modify the shared state in the program.

What happens if the lock is never returned?

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

```
e(){
red_value + 1;
```



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

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

```cpp
// @file thread7_deadlock.cpp
// g++ -std=c++17 thread7_deadlock.cpp -o prog -lpthread
#include <iostream>
#include <thread>   // Include the thread library
#include <vector>
#include <mutex>  // New library

// Some shared value
static int shared_value = 0;
std::mutex gLock; // A global lock

void increment_shared_value(){
    gLock.lock();
        shared_value = shared_value + 1;
//    gLock.unlock(); // Oops, never return lock
}
```

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

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

```cpp
// @file thread7_deadlock.cpp
// g++ -std=c++17 thread7_deadlock.cpp -o prog -lpthread
#include <iostream>
#include <thread>   // Include the thread library
#include <vector>
#include <mutex>  // New library

// Some shared value
static int shared_value = 0;
std::mutex gLock; // A global lock

void increment_shared_value(){
    gLock.lock();
        shared_value = shared_value + 1;
//    gLock.unlock(); // Oops, never return lock
}
```

Careful with std::mutex (1/2)

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

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

Careful with std::mutex (2/2)

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

```cpp
// @file thread7_deadlock.cpp
// g++ -std=c++17 thread7_deadlock.cpp -o prog -lpthread
#include <iostream>
#include <thread>   // Include the thread library
#include <vector>
#include <mutex>  // New library

// Some shared value
static int shared_value = 0;
std::mutex gLock; // A global lock

void increment_shared_value(){
    gLock.lock();
        shared_value = shared_value + 1;
    gLock.unlock(); // Fixed right?
}
```

```cpp
void increment_shared_value(){
    gLock.lock();
        try{
            shared_value = shared_value + 1;
            throw "Dangerous exception abort";
        }catch(...){
            std::cout << "handle exception by returning from thread\n";
            return;
        }
    gLock.unlock(); // Oops, never return lock
}
```

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

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

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

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

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

```cpp
// @file thread9.cpp
// g++ -std=c++17 thread9.cpp -o prog -lpthread
#include <iostream>
#include <thread>   // Include the thread library
#include <vector>
#include <mutex>  // New library

// Some shared value
static int shared_value = 0;
std::mutex gLock; // A global lock

void increment_shared_value(){
    // lock_guard follows RAII principles and will
    // release lock after leaving scope.
    // This includes if an exception is thrown.
    std::lock_guard<std::mutex> lockGuard(gLock);
        try{
            shared_value = shared_value + 1;
            throw "Dangerous exception abort";
        }catch(...){
            std::cout << "handle exception by returning from thread\n";
            return;
        }
}
```

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

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

- We instead of a lockGuard that can ‘wrap’ an individual `std::mutex`
  - The destructor of lock_guard will take care of releasing the lock
- std::lock_guard is a good example of

```cpp
// @file thread9.cpp
// g++ -std=c++17 thread9.cpp -o prog -lpthread
#include <iostream>
#include <thread>   // Include the thread library
#include <vector>
#include <mutex>  // New library

// Some shared value
static int shared_value = 0;
std::mutex gLock; // A global lock

void increment_shared_value(){
    // lock_guard follows RAII principles and will
    // release lock after leaving scope.
    // This includes if an exception is thrown.
    std::lock_guard<std::mutex> lockGuard(gLock);
        try{
            shared_value = shared_value + 1;
            throw "Dangerous exception abort";
```

std::lock_guard is only 3 member functions

Member functions

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

```
hread\n";
handle exception by returning from thread
Result = 1000
```

`std::scoped_lock` - Other mechanisms

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

Member functions

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

std::atomic (1/3)

- For the problem we were trying to solve (incrementing a `shared_value` with multiple threads), we could actually use atomics.
  - atomics

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

// Some shared value
static std::atomic<int> shared_value=0;

void increment_shared_value(){
    shared_value ++;
}

int main() {
    std::vector<std::thread> threads;
    // Create a collection of threads
    for(int i=0; i < 1000; i++){
        threads.push_back(std::thread(increment_shared_value));
    }
    // Join our threads using a ranged-based loop
    // (Our intent is to iterate through all threads in container)
    for(auto& th: threads){
        threads.join();
    }
    // Retrieve our result
    std::cout << "Result = " << shared_value << std::endl;

    return 0;
}
```

std::atomic (2/3)

- For the problem we were trying to solve (incrementing a `shared_value` with multiple threads), we could actually use atomics.
  - atomics

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

std::atomic

Defined in header <atomic>

```
template< class T >
struct atomic;                                  (1)  (since C++11)

template< class U >
struct atomic<U*>;                              (2)  (since C++11)
```

Defined in header <memory>

```
template< class U >
struct atomic<std::shared_ptr<U>>;              (3)  (since C++20)

template< class U >
struct atomic<std::weak_ptr<U>>;                (4)  (since C++20)
```

Defined in header <stdatomic.h>

```
#define _Atomic(T) /* see below */              (5)  (since C++23)
```

Each instantiation and full specialization of the `std::atomic` template defines an atomic type. If one thread writes to an atomic object while another thread reads from it, the behavior is well-defined (see memory model for details on data races).

In addition, accesses to atomic objects may establish inter-thread synchronization and order non-atomic memory accesses as specified by `std::memory_order`.

`std::atomic` is neither copyable nor movable.

Works on primitive types (or any type that is trivially copyable)

atomic variables ensure any read or write synchronizes

```cpp
    return 0;
}
```

std::atomic (3/3)

- Here's the example, with minimal changes
  - No need for std::mutex on shared_value
  - If two or more threads are writing (or one reading and one writing) to the object, the operations are well-defined.
    - (Note: I would probably rename variable with a suffix of _atomic for readability)
    - Accidentally putting a lock, (or excessive locking) is bad for performance

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

// Some shared value
static std::atomic<int> shared_value=0;

void increment_shared_value(){
    shared_value ++;
}

int main() {
    std::vector<std::thread> threads;
    // Create a collection of threads
    for(int i=0; i < 1000; i++){
        threads.push_back(std::thread(increment_shared_value));
    }
    // Join our threads using a ranged-based loop
    // (Our intent is to iterate through all threads in container)
    for(auto& th: threads){
        threads.join();
    }
    // Retrieve our result
    std::cout << "Result = " << shared_value << std::endl;

    return 0;
}
```

(Brief Aside) static in C++11 (and beyond)

- static local variable are guaranteed by the C++11 standard to only be initialized once (See also std::call_once)
  - Intuitively this makes sense, because the static variable in some way needs to be initialized exactly 1 time
  - Takeaway: No lock needed

Static local variables

Variables declared at block scope with the specifier `static` or `thread_local` (since C++11) have static or thread (since C++11) storage duration but are initialized the first time control passes through their declaration (unless their initialization is zero- or constant-initialization, which can be performed before the block is first entered). On all further calls, the declaration is skipped.

If the initialization throws an exception, the variable is not considered to be initialized, and initialization will be attempted again the next time control passes through the declaration.

If the initialization recursively enters the block in which the variable is being initialized, the behavior is undefined.

If multiple threads attempt to initialize the same static local variable concurrently, the initialization occurs exactly once (similar behavior can be obtained for arbitrary functions with `std::call_once`). (since C++11)

Note: usual implementations of this feature use variants of the double-checked locking pattern, which reduces runtime overhead for already-initialized local statics to a single non-atomic boolean comparison.

The destructor for a block-scope static variable is called at program exit, but only if the initialization took place successfully.

Function-local static objects in all definitions of the same inline function (which may be implicitly inline) all refer to the same object defined in one translation unit, as long as the function has external linkage.

https://en.cppreference.com/w/cpp/language/storage_duration#Static_local_variables

Condition Variables

A way to signal an event between 2 or more threads

Condition Variables (1/4)

- Perhaps a *stranger* thing when working with threads
  - Idea:
    - Given 2 threads (a worker and reporter), we want one thread to wait on the result of the other
    - Condition_variable allows us to in a sense, take one thread off the queue, until it is notified that it should start working
      - (See example on right)

```cpp
// @file cv.cpp
// g++ -std=c++17 cv.cpp -o prog -lpthread
#include <iostream>
#include <thread>   // Include the thread library
#include <vector>
#include <chrono>
#include <mutex>
#include <condition_variable> // New library

std::mutex gLock; // A global lock
std::condition_variable gConditionVariable;

int main() {
    std::cout << "main() starts" << std::endl;

    int result =0; // The result we're tring to compute
    bool notified = false;
    // This thread reports the result (consumes the result)
    // Could call this a 'consumer' or 'reporter' thread
    std::thread reporter([&]{
        std::unique_lock<std::mutex> lock(gLock);
        // wait here until notified
        if(!notified){ // We need some variable to wait on
            gConditionVariable.wait(lock);
        }
        std::cout << "\tReporter Result = " << result << std::endl;
    });

    // This thread does the work
    std::thread worker([&]{
        std::unique_lock<std::mutex> lock(gLock);
        // Do our work
        result = 42;
        // Update our variable
        notified = true;
        // artificial pause to show that reporter will indeed wait
        std::this_thread::sleep_for(std::chrono::seconds(2));
        // Output a message
        std::cout << "\twork complete" << std::endl;
        // Notify one thread about a change
        // Could also notify_all
        gConditionVariable.notify_one();
    });


    // Don't forget to join!
    reporter.join();
    worker.join();
    std::cout << "main end " << std::endl;
    return 0;
}
```

Condition Variables (2/4)

- Perhaps a *stranger* thing when working with threads
  - Idea:
    - Given 2 threads (a worker and reporter), we want one thread to wait on the result of the other
    - Condition_variable allows us to in a sense, take one thread off the queue, until it is notified that it should start working

Here's our reporter thread

reporter will wait on a 'variable' to be updated (notified) Then we'll be unblocked (gConditionVariable.wait) once we have been notified we can proceed

```cpp
// @file cv.cpp
// g++ -std=c++17 cv.cpp -o prog -lpthread
#include <iostream>
#include <thread>   // Include the thread library
#include <vector>
#include <chrono>
#include <mutex>
#include <condition_variable> // New library

std::mutex gLock; // A global lock
std::condition_variable gConditionVariable;

int main() {
    std::cout << "main() starts" << std::endl;

    int result =0; // The result we're tring to compute
    bool notified = false;
    // This thread reports the result (consumes the result)
    // Could call this a 'consumer' or 'reporter' thread
    std::thread reporter([&]{
        std::unique_lock<std::mutex> lock(gLock);
        // wait here until notified
        if(!notified){ // We need some variable to wait on
            gConditionVariable.wait(lock);
        }
        std::cout << "\tReporter Result = " << result << std::endl;
    });

    // This thread does the work
    std::thread worker([&]{
        std::unique_lock<std::mutex> lock(gLock);
        // Do our work
        result = 42;
        // Update our variable
        notified = true;
        // artificial pause to show that reporter will indeed wait
        std::this_thread::sleep_for(std::chrono::seconds(2));
        // Output a message
        std::cout << "\twork complete" << std::endl;
        // Notify one thread about a change
        // Could also notify_all
        gConditionVariable.notify_one();
    });


    // Don't forget to join!
    reporter.join();
    worker.join();
    std::cout << "main end " << std::endl;
    return 0;
}
```

Condition Variables (3/4)

- Perhaps a *stranger* thing when working with threads
  - Idea:
    - Given 2 threads (a worker and reporter), we want one thread to wait on the result of the other
    - Condition_variable allows us to in a sense, take one thread off the queue, until it is notified that it should start working

Here's our worker thread

We update our result (i.e., do our work), and then change notified.
At the end, we notify_one() to signal to our reporter that the reporter thread no longer needs to wait

```cpp
// @file cv.cpp
// g++ -std=c++17 cv.cpp -o prog -lpthread
#include <iostream>
#include <thread>   // Include the thread library
#include <vector>
#include <chrono>
#include <mutex>
#include <condition_variable> // New library

std::mutex gLock; // A global lock
std::condition_variable gConditionVariable;

int main() {
    std::cout << "main() starts" << std::endl;

    int result =0; // The result we're tring to compute
    bool notified = false;
    // This thread reports the result (consumes the result)
    // Could call this a 'consumer' or 'reporter' thread
    std::thread reporter([&]{
        std::unique_lock<std::mutex> lock(gLock);
        // wait here until notified
        if(!notified){ // We need some variable to wait on
            gConditionVariable.wait(lock);
        }
        std::cout << "\tReporter Result = " << result << std::endl;
    });

    // This thread does the work
    std::thread worker([&]{
        std::unique_lock<std::mutex> lock(gLock);
        // Do our work
        result = 42;
        // Update our variable
        notified = true;
        // artificial pause to show that reporter will indeed wait
        std::this_thread::sleep_for(std::chrono::seconds(2));
        // Output a message
        std::cout << "\twork complete" << std::endl;
        // Notify one thread about a change
        // Could also notify_all
        gConditionVariable.notify_one();
    });


    // Don't forget to join!
    reporter.join();
    worker.join();
    std::cout << "main end " << std::endl;
    return 0;
}
```

Con

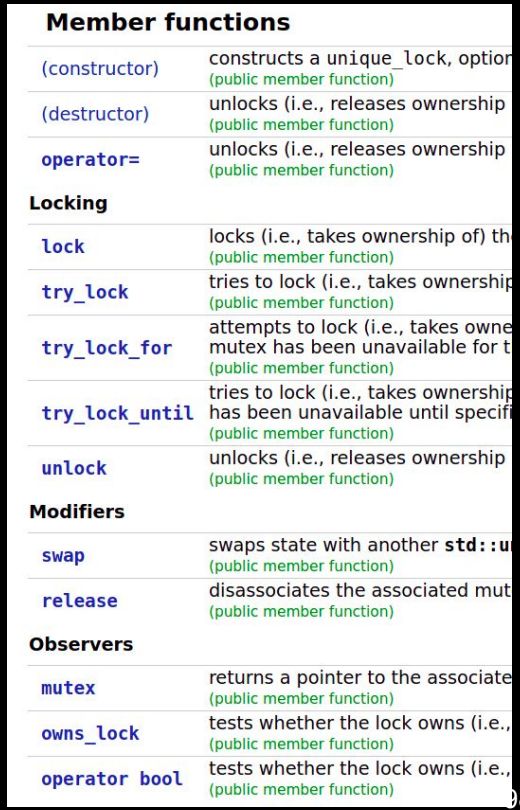
- P
with threads
  - Idea:
    - Given 2 threads (a worker and reporter), we want one thread to wait on the result of the other
    - Condition_variable allows us to in a sense, take one thread off the queue, until it is notified that it should start working
      - (See example on right)

```
mike:concurrency$ g++-10 -std=c++17 cv.cpp -o prog -lpthread
mike:concurrency$ ./prog
main() starts
        work complete
        Reporter Result = 42
main end
```

```
1  // @file cv.cpp
                                                                    brary
                                                                    l;
                                                                    to compute
17     bool notified = false;
18     // This thread reports the result (consumes the result)
19     // Could call this a 'consumer' or 'reporter' thread
20     std::thread reporter([&]{
21         std::unique_lock<std::mutex> lock(gLock);
22         // wait here until notified
23         if(!notified){ // We need some variable to wait on
24             gConditionVariable.wait(lock);
25         }
26         std::cout << "\tReporter Result = " << result << std::endl;
27     });
28
29     // This thread does the work
30     std::thread worker([&]{
31         std::unique_lock<std::mutex> lock(gLock);
32         // Do our work
33         result = 42;
34         // Update our variable
35         notified = true;
36         // artificial pause to show that reporter will indeed wait
37         std::this_thread::sleep_for(std::chrono::seconds(2));
38         // Output a message
39         std::cout << "\twork complete" << std::endl;
40         // Notify one thread about a change
41         // Could also notify_all
42         gConditionVariable.notify_one();
43     });
44
45
46     // Don't forget to join!
47     reporter.join();
48     worker.join();
49     std::cout << "main end " << std::endl;
50     return 0;
51 }
```

`std::unique_lock` - Other mechanisms

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

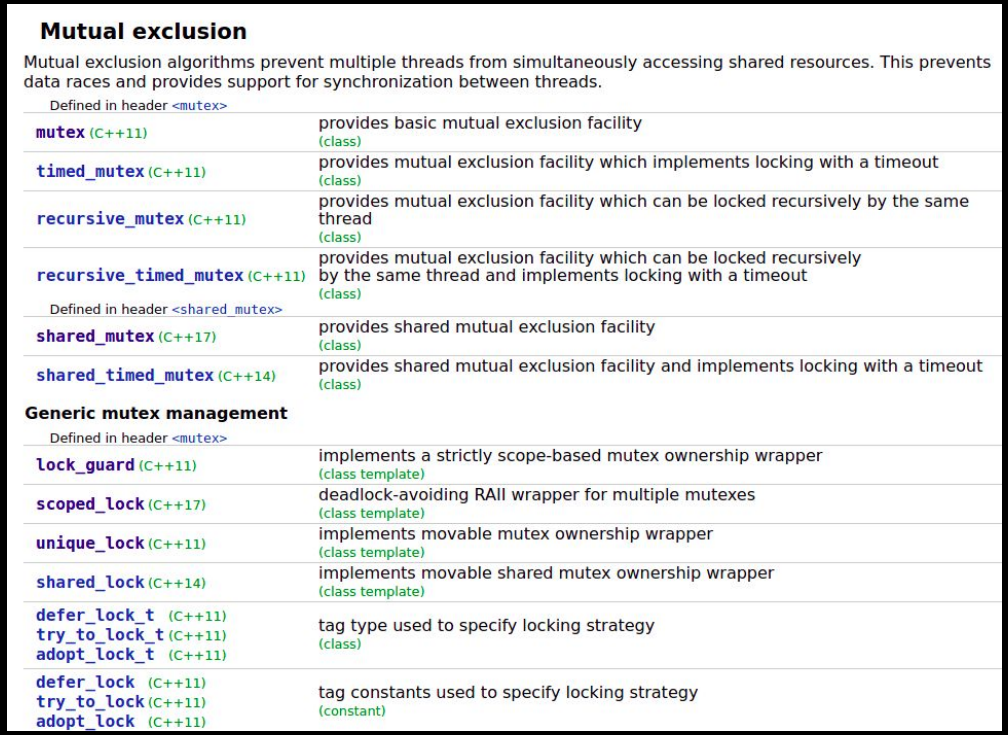


Member functions

(constructor)	constructs a unique_lock, optior (public member function)
(destructor)	unlocks (i.e., releases ownership (public member function)
operator=	unlocks (i.e., releases ownership (public member function)
Locking	
lock	locks (i.e., takes ownership of) th (public member function)
try_lock	tries to lock (i.e., takes ownership (public member function)
try_lock_for	attempts to lock (i.e., takes owne mutex has been unavailable for t (public member function)
try_lock_until	tries to lock (i.e., takes ownership has been unavailable until specifi (public member function)
unlock	unlocks (i.e., releases ownership (public member function)
Modifiers	
swap	swaps state with another std::u (public member function)
release	disassociates the associated mut (public member function)
Observers	
mutex	returns a pointer to the associate (public member function)
owns_lock	tests whether the lock owns (i.e., (public member function)
operator bool	tests whether the lock owns (i.e., (public member function)

There exist several other primitives you can find here

- https://en.cppreference.com/w/cpp/thread#Mutual_exclusion



Mutual exclusion

Mutual exclusion algorithms prevent multiple threads from simultaneously accessing shared resources. This prevents data races and provides support for synchronization between threads.

Defined in header <mutex>

`mutex` (C++11)	provides basic mutual exclusion facility (class)
`timed_mutex` (C++11)	provides mutual exclusion facility which implements locking with a timeout (class)
`recursive_mutex` (C++11)	provides mutual exclusion facility which can be locked recursively by the same thread (class)
`recursive_timed_mutex` (C++11)	provides mutual exclusion facility which can be locked recursively by the same thread and implements locking with a timeout (class)

Defined in header <shared_mutex>

`shared_mutex` (C++17)	provides shared mutual exclusion facility (class)
`shared_timed_mutex` (C++14)	provides shared mutual exclusion facility and implements locking with a timeout (class)

Generic mutex management

Defined in header <mutex>

`lock_guard` (C++11)	implements a strictly scope-based mutex ownership wrapper (class template)
`scoped_lock` (C++17)	deadlock-avoiding RAII wrapper for multiple mutexes (class template)
`unique_lock` (C++11)	implements movable mutex ownership wrapper (class template)
`shared_lock` (C++14)	implements movable shared mutex ownership wrapper (class template)
`defer_lock_t` (C++11) `try_to_lock_t` (C++11) `adopt_lock_t` (C++11)	tag type used to specify locking strategy (class)
`defer_lock` (C++11) `try_to_lock` (C++11) `adopt_lock` (C++11)	tag constants used to specify locking strategy (constant)

Quick Recap

(Of the C++ we have learned)

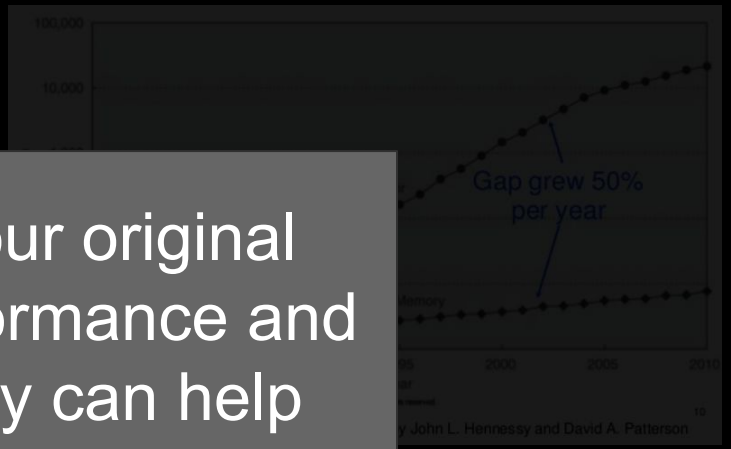
The C++ What we have so far

- std::thread used with ‘join’
- std::jthread (C++20 feature)
- std::mutex
  - lock and unlock (to create a critical section that is atomically executed)
- std::lock_guard (and also a note on scoped_lock and unique_lock)
  - Which helps us from forgetting to ‘unlock’ a lock (Uses RAII to release lock)
- condition_variable
  - A way to ‘signal’ between multiple threads when work is done
- std::atomic<T>
  - Useful for primitive types we want to synchronize
- And 2 caveats to worry about with concurrency
  - data races (non-deterministic behavior with reads and write order)
  - deadlock (lack of progress)

Our original motivation was about performance (1/2)

- An interesting reality is that many applications we write are I/O bound
  - That mea
    operation
  - The figur
    processo
    accessin
    magnitud
    - Thus, we often cannot fetch data at the rate that we process it.

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



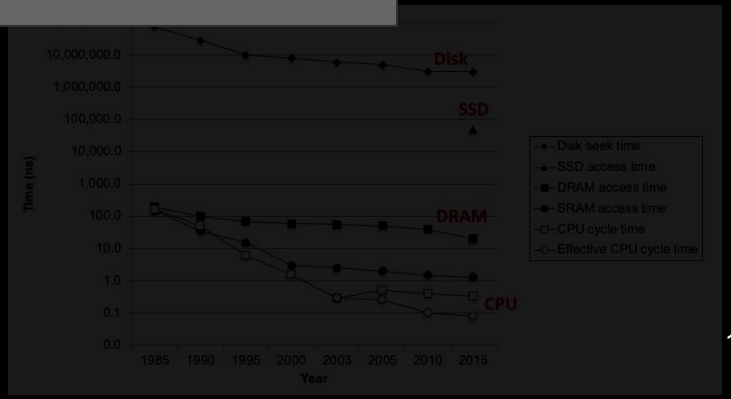
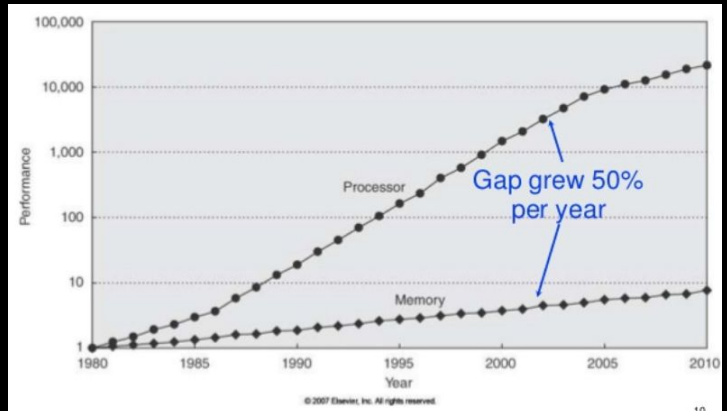


Image source: https://www.dice.se/wp-content/uploads/2014/12/Introduction_to_Data-Oriented_Design.pdf

Our original motivation was about performance (2/2)

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



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

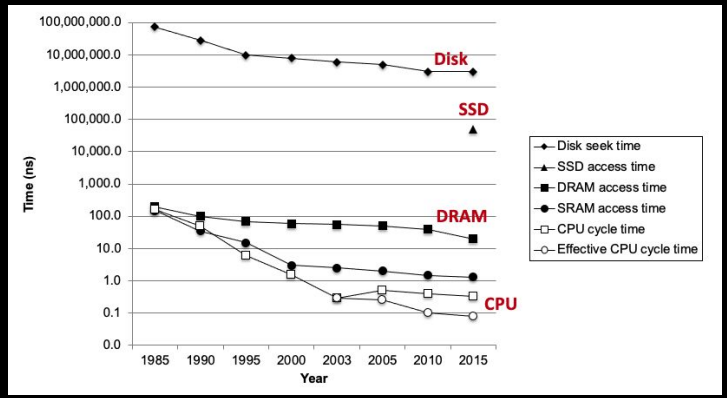


Image source: https://www.dice.se/wp-content/uploads/2014/12/Introduction_to_Data-Oriented_Design.pdf

Asynchronous Programming

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

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

std::async example

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

```cpp
// @file thread10.cpp
// g++ -std=c++17 thread10.cpp -o prog -lpthread
#include <iostream>
#include <future>

int square(int x){
    return x*x;
}

int main() {

    // asyncFunction is a 'future'
    // type = std::future<int>
    auto asyncFunction = std::async(&square,12);
    // .... some time passes
    int result = asyncFunction.get(); // We are blocked here if
                                      // our value has not been
                                      // computed. Otherwise, the
                                      // value from get() (which
                                      // is wrapped in a promise
                                      // is returned.
    std::cout << "The async thread has returned! " << result
              << std:: endl;

    return 0;
}

                                                          1,1
```

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

A Concrete Example for std::async

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

Async I/O Simulation (1/5)

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

```cpp
// @file async.cpp
// g++ -std=c++17 async.cpp -o prog -lpthread
#include <iostream>
#include <future>
#include <thread>
#include <chrono>

// Toy example - This will be our 'background thread'
//               that executes asynchronously
bool bufferedFileLoading(){
    size_t bytesLoaded= 0;
    while(bytesLoaded < 20000){
        std::cout << "Loading File..." << std::endl;
        std::this_thread::sleep_for(std::chrono::milliseconds(250));
        bytesLoaded+= 1000;
    }
    return true;
}

int main() {
    // Launch thread asynchronusly, and this will execute in background
    std::future<bool> backgroundThread = std::async(std::launch::async,
                                                    bufferedFileLoading);
    // Store status of our future
    std::future_status status;
    // Meanwhile, we have our main thread of execution
    while(true){
        std::cout << "Main thread running" << std::endl;
        // artificial pause
        std::this_thread::sleep_for(std::chrono::milliseconds(50));
        // Check if our
        status = backgroundThread.wait_for(std::chrono::milliseconds(1));
        // If our data is ready--do something
        // we'll just terminate for now
        if(status == std::future_status::ready){
            std::cout << "data ready..." << std::endl;
            break;
        }
    }

    return 0;
}
```

As

-

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

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

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

```cpp
// @file async.cpp
// g++ -std=c++17 async.cpp -o prog -lpthread
#include <iostream>
#include <future>
#include <thread>
#include <chrono>

// Toy example - This will be our 'background thread'
//               that executes asynchronously
bool bufferedFileLoading(){
    size_t bytesLoaded= 0;
    while(bytesLoaded < 20000){
        std::cout << "Loading File..." << std::endl;
        std::this_thread::sleep_for(std::chrono::milliseconds(250));
        bytesLoaded+= 1000;
    }
    return true;
}

int main() {
    // Launch thread asynchronously, and this will execute in background
    std::future<bool> backgroundThread = std::async(std::launch::async,
                                                    bufferedFileLoading);
    // Store status of our future
    std::future_status status;
    // Meanwhile, we have our main thread of execution
    while(true){
        std::cout << "Main thread running" << std::endl;
        // artificial pause
        std::this_thread::sleep_for(std::chrono::milliseconds(50));
        // Check if our
        status = backgroundThread.wait_for(std::chrono::milliseconds(1));
        // If our data is ready--do something
        // we'll just terminate for now
        if(status == std::future_status::ready){
            std::cout << "data ready..." << std::endl;
            break;
        }
    }

    return 0;
}
```

Here we are reading in ‘bytes’ from a file.

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

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

```cpp
// @file async.cpp
// g++ -std=c++17 async.cpp -o prog -lpthread
#include <iostream>
#include <future>
#include <thread>
#include <chrono>

// Toy example - This will be our 'background thread'
//               that executes asynchronously
bool bufferedFileLoading(){
    size_t bytesLoaded= 0;
    while(bytesLoaded < 20000){
        std::cout << "Loading File..." << std::endl;
        std::this_thread::sleep_for(std::chrono::milliseconds(250));
        bytesLoaded+= 1000;
    }
    return true;
}

int main() {
    // Launch thread asynchronusly, and this will execute in background
    std::future<bool> backgroundThread = std::async(std::launch::async,
                                                    bufferedFileLoading);
    // Store status of our future
    std::future_status status;
    // Meanwhile, we have our main thread of execution
    while(true){
        std::cout << "Main thread running" << std::endl;
        // artificial pause
        std::this_thread::sleep_for(std::chrono::milliseconds(50));
        // Check if our
        status = backgroundThread.wait_for(std::chrono::milliseconds(1));
        // If our data is ready--do something
        // we'll just terminate for now
        if(status == std::future_status::ready){
            std::cout << "data ready..." << std::endl;
            break;
        }
    }

    return 0;
}
```

Async I/O Simulation (4/5)

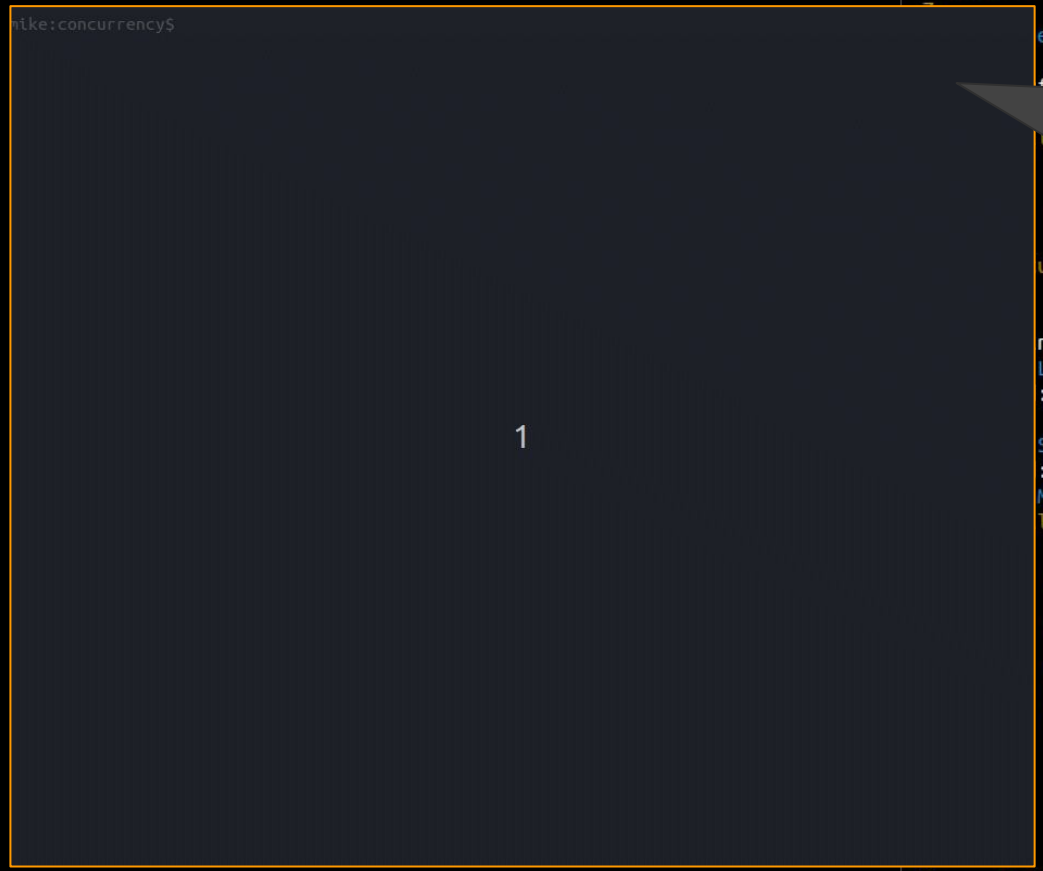
- “mocked” version of using an async thread to load data
  - We spawn a ‘background thread’ asynchronously using std::async

leeps

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

```cpp
// @file async.cpp
// g++ -std=c++17 async.cpp -o prog -lpthread
#include <iostream>
#include <future>
#include <thread>
#include <chrono>

// Toy example - This will be our 'background thread'
//               that executes asynchronously
bool bufferedFileLoading(){
    size_t bytesLoaded= 0;
    while(bytesLoaded < 20000){
        std::cout << "Loading File..." << std::endl;
        std::this_thread::sleep_for(std::chrono::milliseconds(250));
        bytesLoaded+= 1000;
    }
    return true;
}

int main() {
    // Launch thread asynchronously, and this will execute in background
    std::future<bool> backgroundThread = std::async(std::launch::async,
                                                    bufferedFileLoading);
    // Store status of our future
    std::future_status status;
    // Meanwhile, we have our main thread of execution
    while(true){
        std::cout << "Main thread running" << std::endl;
        // artificial pause
        std::this_thread::sleep_for(std::chrono::milliseconds(50));
        // Check if our
        status = backgroundThread.wait_for(std::chrono::milliseconds(1));
        // If our data is ready--do something
        // we'll just terminate for now
        if(status == std::future_status::ready){
            std::cout << "data ready..." << std::endl;
            break;
        }
    }

    return 0;
}
```

Async I/O Simulation (5/5)



Here's what our execution looks like

(Again--code available from github)

```
1 // @file async.cpp
2 // g++ -std=c++
3 #include <iostr
4 #include <futur
5 #include <threa
6 #include <chron

example

fferedFi

 sto.
 std::t
 bytesLo

urn true;


n() {
Launch thread asynchronusly, and this will execute in background
::future<bool> backgroundThread = std::async(std::launch::async,
                                             bufferedFileLoading);
Store status of our future
::future_status status;
Meanwhile, we have our main thread of execution
le(true){
 std::cout << "Main thread running" << std::endl;
 // artificial pause
 std::this_thread::sleep_for(std::chrono::milliseconds(50));
 // Check if our
 status = backgroundThread.wait_for(std::chrono::milliseconds(1));
 // If our data is ready--do something
 // we'll just terminate for now
 if(status == std::future_status::ready){
     std::cout << "data ready..." << std::endl;
     break;
 }


41     return 0;
42 }
```

Other Odds and Ends

Coming to the finale--a rapid fire section of topics not covered

Thread Sanitizer

- Since we have talked quite a bit about threads in the first portion--thread sanitizer is a useful tool for detecting concurrency violations (i.e., data races).
  - See an example with clang: https://clang.llvm.org/docs/ThreadSanitizer.html

Performance

- I haven’t touched on this--but launching 10000 threads does not make your application 10000 times faster.
  - There is some overhead with threads (creation of the threads, context switching, managing the threads, etc.)
  - Likely you’ll want to have some fixed number of threads (perhaps as a function of how many CPU cores you have).
    - Folks will create something called a ‘thread pool’ to structure their concurrent programs.

C++20 Coroutines [reference]

- Coroutines are another way to ‘collaboratively’ write code.
  - They’re new enough in C++ I personally need to learn/think more about them.
  - Coroutines to my understanding allowed you to yield and save state (which could be done asynchronously) and give you more control of your program (i.e., lazy-computation).
  - Nice explanation of coroutines:
    https://stackoverflow.com/questions/553704/what-is-a-coroutine
    https://en.cppreference.com/w/cpp/language/coroutines

C++ Memory Model [reference]

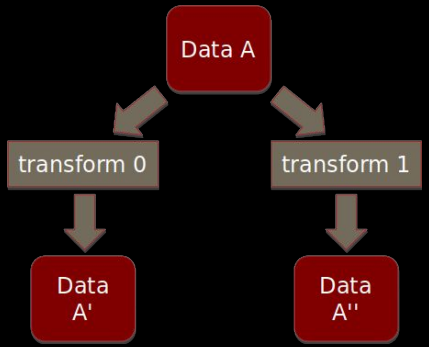
- The memory model in C++ is what has enabled std::thread and other concurrent programming techniques
  - This defines the order of ‘reads’ and ‘writes’ to memory
  - Having this order defined, helps stabilize the language, and ensure that whoever implements <thread> is able to do so safely.
- For a beginner, it’s worth running the examples provided here a few times, then perhaps viewing the memory model as you move further.
  - (For more advanced users, please take a look)

Common Concurrency Patterns

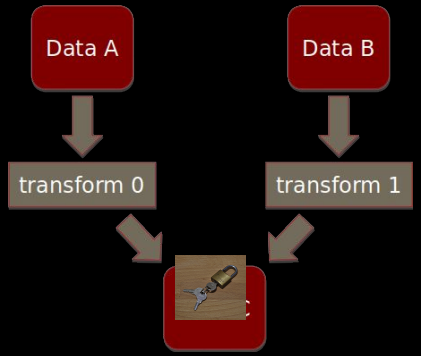
You will know enough to implement these (They take a little thought!)

Reader/Writers Problem

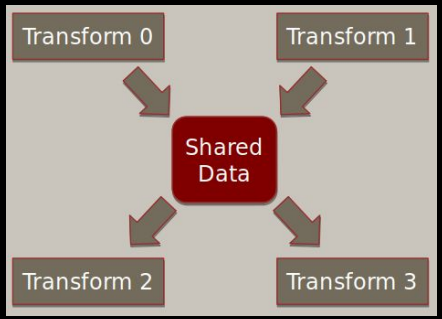
- Consider this pattern where you have to either read to a location, write to a location, or read and write to a location
  - We have std::shared_mutex which allows multiple threads to read a value
  - Utilizing std::shared_mutex with std::unique_lock can help solve the readers/writers problem



Here multiple reader threads are ‘reading’ Data A and performing some transformation.



Here multiple writer threads are generating and combining their result into Data ‘C’.



Here multiple writer threads are creating shared data, and multiple readers than take that data as input.

Other popular concurrency patterns

- Producer Consumer pattern
  - Pattern in which ‘producer’ threads generate data
    - (Usually adding to some queue)
  - Consumer threads then consume any data from the queue, and process it.
  - https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem
- Barrier
  - Allow a fixed number of threads to enter a critical section
  - (C++20) std::counting_semaphore
    - Allow multiple threads to enter
    - Related ideas
      - (C++20) std::latch
      - (C++20) std::barrier

Brief Notes on Parallelism

Parallelism being the simultaneous execution of threads (often with less thought to think about synchronization (i.e., locks) for correctness*)

*Still need to think about synchronization points in a sense for performance

Some parallelism for free (implicit parallelism)

- No programming required--hardware does the work

Reference to American show *Survivor* in which contestants receive no prize for losing a challenge

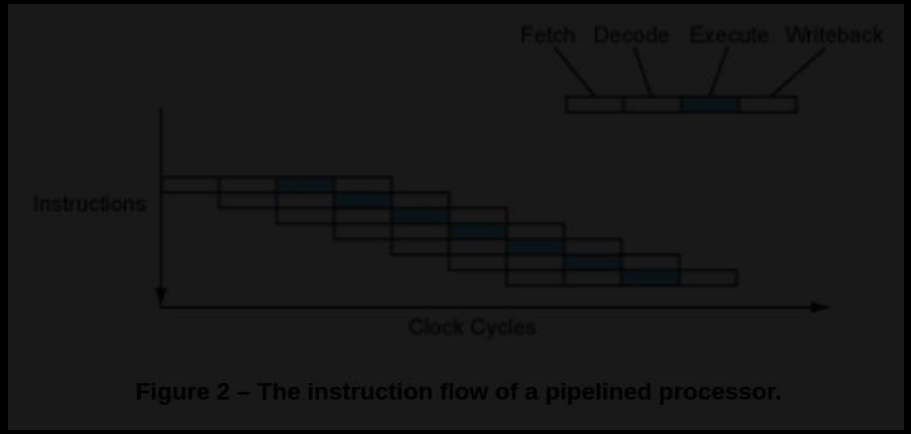


Figure 2 – The instruction flow of a pipelined processor.

Some parallelism for free (implicit parallelism)

- No programming required--hardware does the work
  - CPU Pipelining is an example of *parallelism* we typically get for free
    - (i.e. implicit parallelism)
  - Potential compiler optimizations to automatically vectorize code.

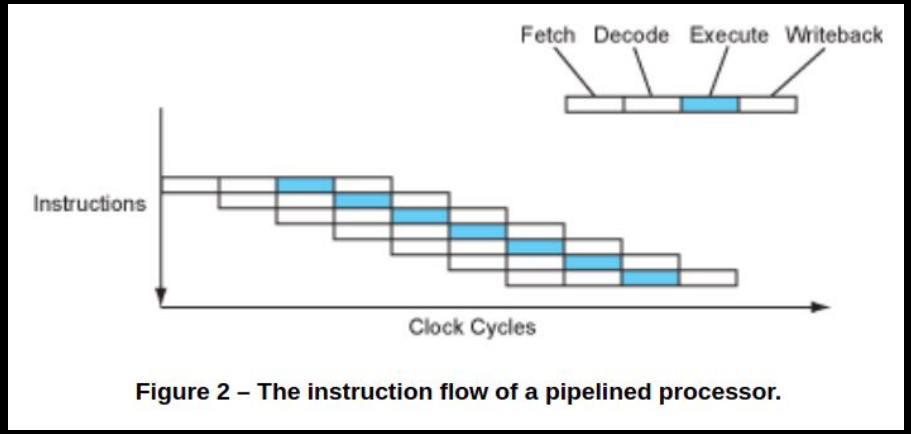
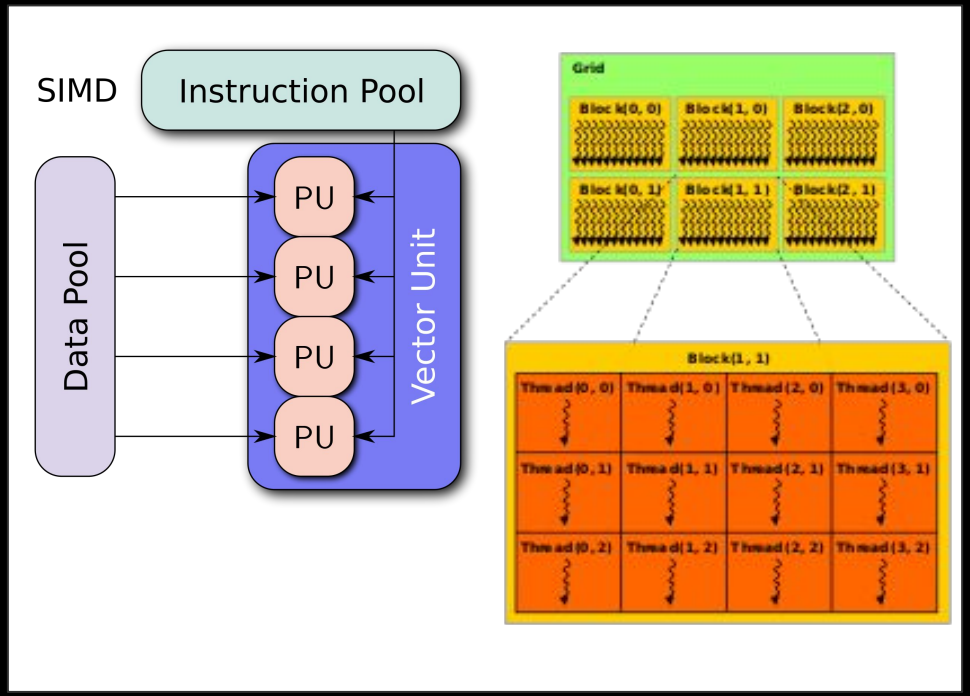


Figure 2 – The instruction flow of a pipelined processor.

https://www.lighterra.com/papers/modernmicroprocessors/

Some parallelism not for free (explicit parallelism)

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



Examples of Parallelism

- Pretty much everything in graphics
  - https://software.intel.com/en-us/articles/performance-methods-and-practices-of-directx-11-multithreaded-rendering
- Pretty much everywhere in machine learning...
- Pretty much everywhere you need to increase performance! :)

C++ std::par and std::par_unseq

- CppCon 2016: Bryce Adelstein Lelbach “The C++17 Parallel Algorithms Library and Beyond"
  - Implementation notes
- CppCon 2017: Dietmar Kühl “C++17 Parallel Algorithms”
  - Usage examples

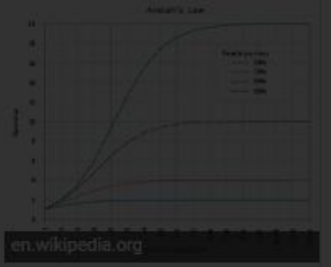
Conclusion and Further Resources

And some analogies (for teaching) and to wrap up what we've learned

Why Parallel?

- 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

I like to include Amdahl's law somewhere so we can even think if we have the opportunity to make part of our program concurrent or parallel

s = speedup of task that benefits from improved resources

p = portion of execution time benefiting from improved speedup

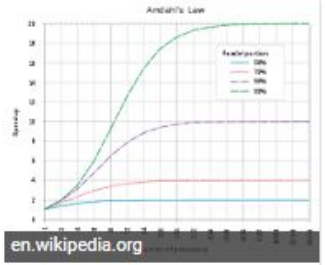
https://en.wikipedia.org/wiki/Amdahl%27s_law

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

Why Parallel?

- 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

Concurrency Caveat #1 | Data Race

- When two (or more) processes contending for one shared resource.
  - One parking spot 2 cars want to acquire
    - (Do both acquire, do neither, does only 1? Depends on the driver I suppose!)

Data race is not always as obvious...(1/4)

- Imagine you check your fridge and find there is no milk
  - So you run to the store

Googling the “Too much milk” problem has an interesting lineage -- https://blog.regehr.org/archives/1145
(Not certain the origin, but it’s used for decades)

Data race is not always as obvious...(2/4)

- Imagine you check your fridge and find there is no milk
  - So you run to the store
- Then moments later your roommate checks the fridge and finds it is empty
  - So they run to the store

Googling the “Too much milk” problem has an interesting lineage -- https://blog.regehr.org/archives/1145
(Not certain the origin, but it’s used for decades)

Data race is not always as obvious...(3/4)

- Imagine you check your fridge and find there is no milk
  - So you run to the store
- Then moments later your roommate checks the fridge and finds it is empty
  - So they run to the store
- Roommate # 3 comes and notices the same
  - ....

Googling the “Too much milk” problem has an interesting lineage -- https://blog.regehr.org/archives/1145
(Not certain the origin, but it’s used for decades)

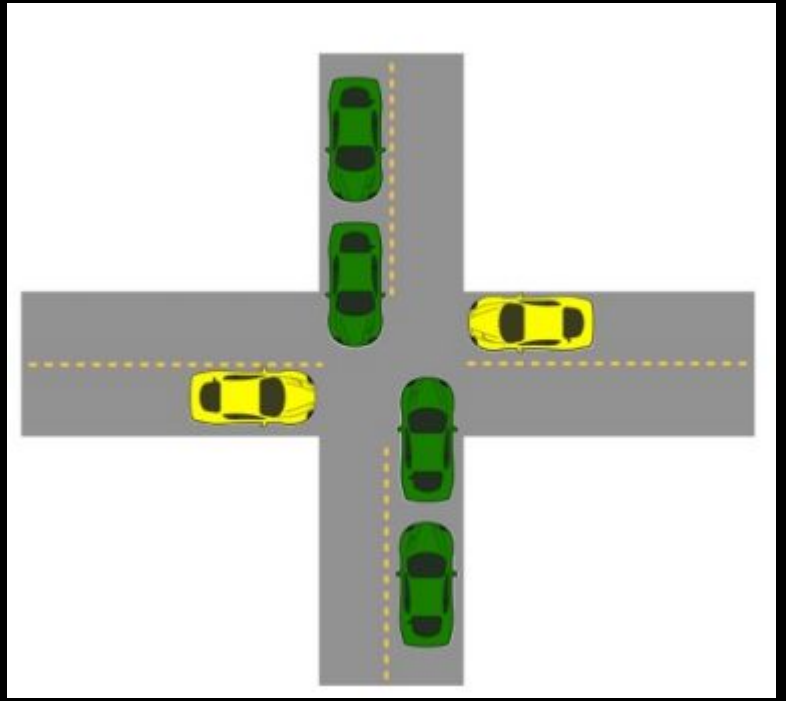
Data race is not always as obvious...(4/4)

- You get the idea when you then find out you have 3 times as much milk as your house needs when everyone returns.
  - Then you can talk to your students about using locking mechanisms for safety
  - After they learn the foundations, then provide further guidance for lock_guard, etc.

Googling the “Too much milk” problem has an interesting lineage -- https://blog.regehr.org/archives/1145
(Not certain the origin, but it’s used for decades)

Concurrency Caveat #2 | Starvation

- Imagine a constant stream of green cars
  - Progress is always made by the green cars
- The yellow cars can never make progress to get across the street.
  - They are resource starved of a shared resource (again, they cannot cross the intersection)



Concurrency Caveat #3 | Deadlock

- Grid lock in a traffic jam
  - Each car prevents others from going through a shared resource (the intersection).
- (One car needs mutually exclusive access to a portion of the intersection in order to move forward)
- (deadlock is an extreme form of starvation)

Concurrent Programming takes some extra care

1. Races: Outcome depends on the arbitrary scheduling decisions elsewhere in the system
    - e.g. Who gets the last seat on the airplane.
2. Starvation/Fairness: External events and/or scheduling decisions can prevent sub-task progress
    - e.g. Someone jumping in front of you in line
3. Deadlock: Improper resource allocation prevents forward progress
    - e.g. traffic gridlock

- But regardless, concurrent programming is important and necessary!

Concurrent Programming takes some extra care

1. Races: Outcome depends on the arbitrary scheduling decisions elsewhere in the system
2. S… …revent su…
3. D…

The examples shown today are good ‘toy’ examples to learn from.

Consider what layers of abstraction you can add to make them safer if your domain allows (i.e., avoiding managing lots of little global std::mutex.)

- But regardless, concurrent programming is important and necessary!

Further Resources

- Code from this talk: https://github.com/MikeShah/cppcon2021
- Back to Basics: Concurrency - Arthur O'Dwyer - CppCon 2020
  - Great resource diving further into concurrency primitives
- https://www.modernescpp.com/
  - Rainer Grimm (Several nice articles on website, and also several cppcon topics on beginner and advanced concurrency topics)
- CppCon 2017: Fedor Pikus “C++ atomics, from basic to advanced. What do they really do?”
  - Full look at atomics
- Little Book of Semaphores https://greenteapress.com/wp/semaphores/
  - Useful puzzles to reason about common concurrency patterns (reader/writer, producer/consumer, etc.)
- The ‘comet book’ https://pages.cs.wisc.edu/~remzi/OSTEP/
  - (Free/donation based) Operating Systems book Insight into how foundational constructs work

Closing--Why do we need concurrency and parallelism?

- Again--the main motivator is performance
  - Doing multiple things at once *things* at once is faster than doing one thing at a time.
  - Sometimes however, it makes our programs cleaner (std::async) and otherwise is a logical way to solve a problem.
- Sometimes concurrency is necessary to model how things truly work in the real world
  - e.g. An orchestra operates concurrently
  - e.g. A traffic light that controls an intersection
  - e.g. pragmatically it may just help us structure our problem and break work into smaller chunks

Back to Basics: Concurrency

Mike Shah, Ph.D.
@MichaelShah | mshah.io | www.youtube.com/c/MikeShah

Thank you Cppcon attendees, reviewers, chairs!

Thank you!