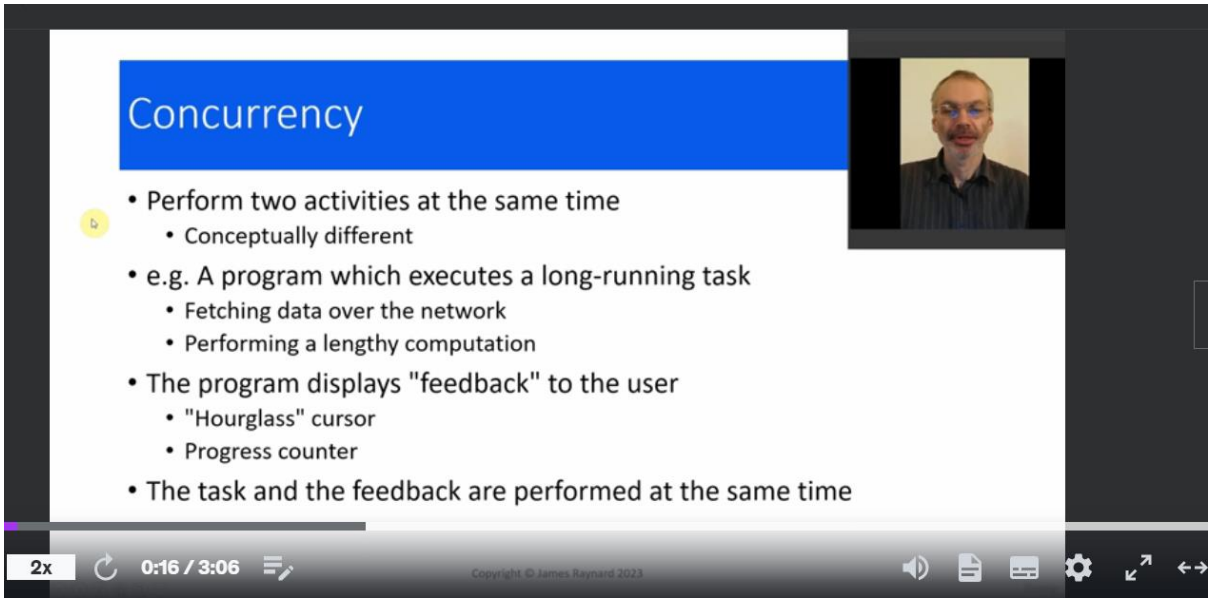
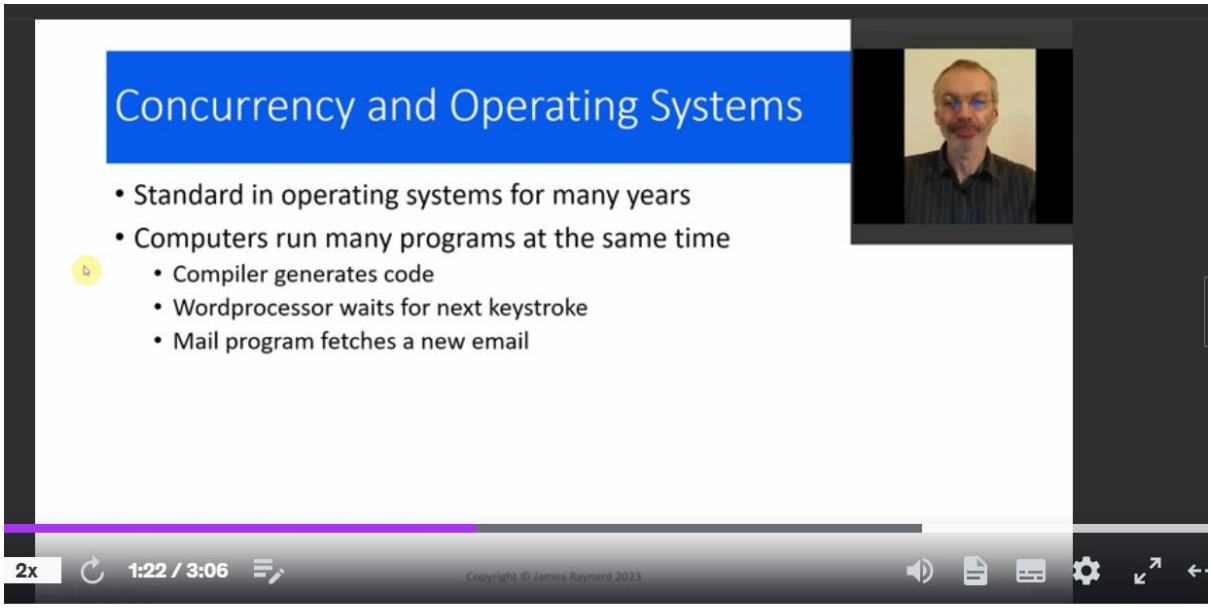
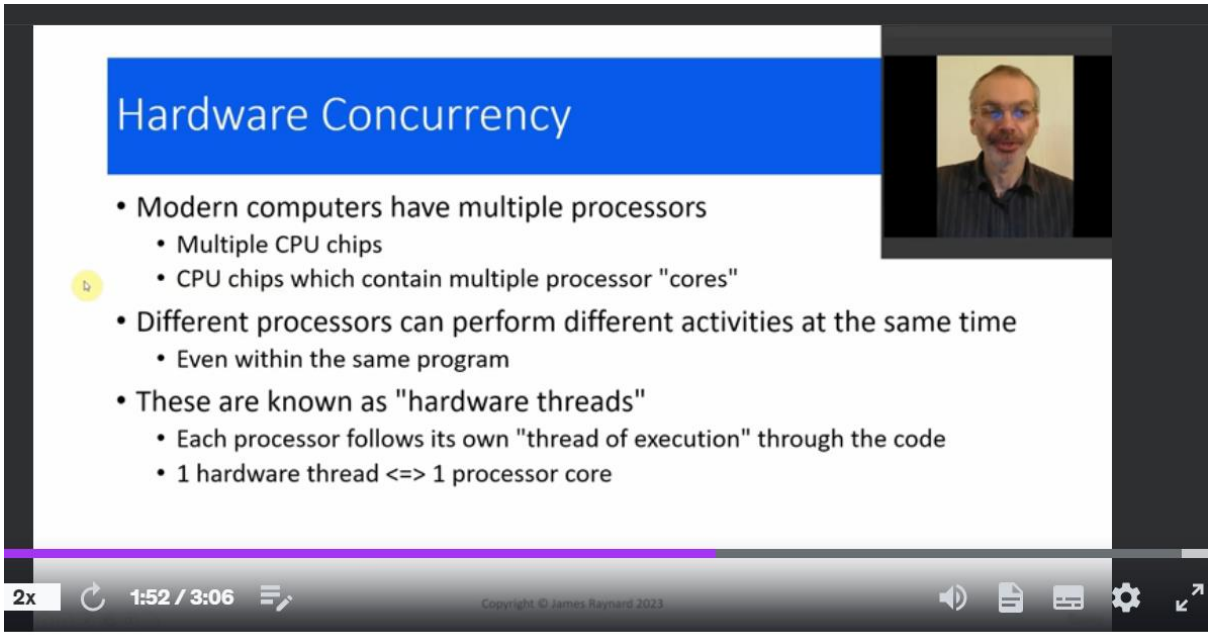
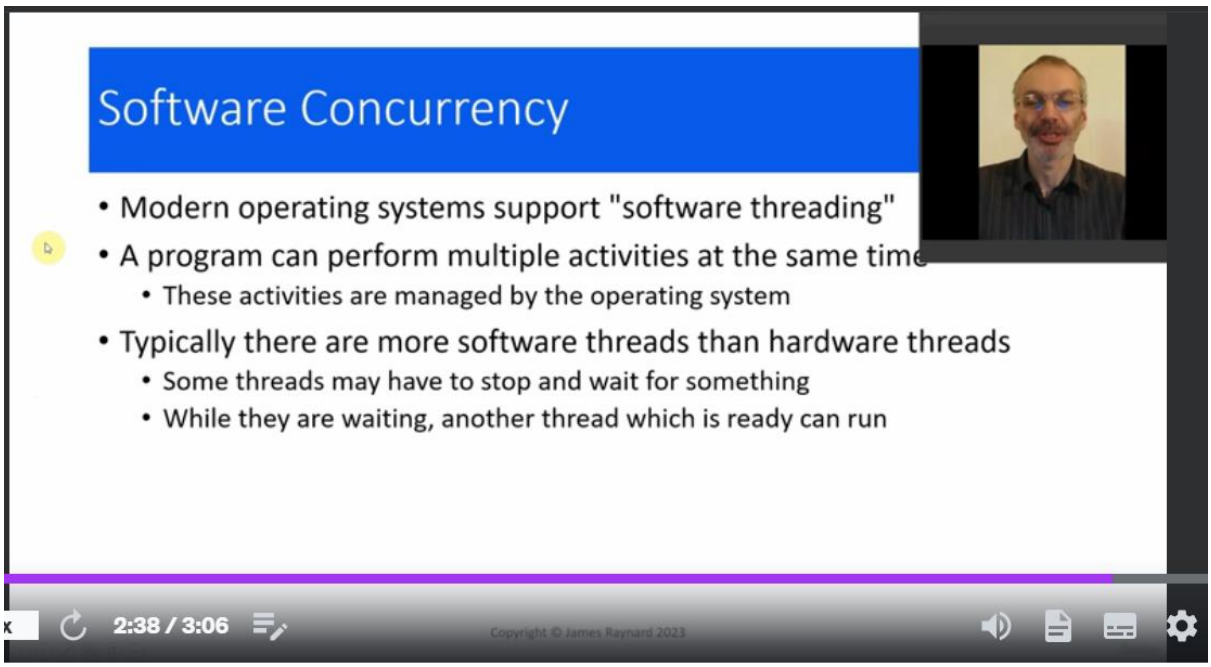
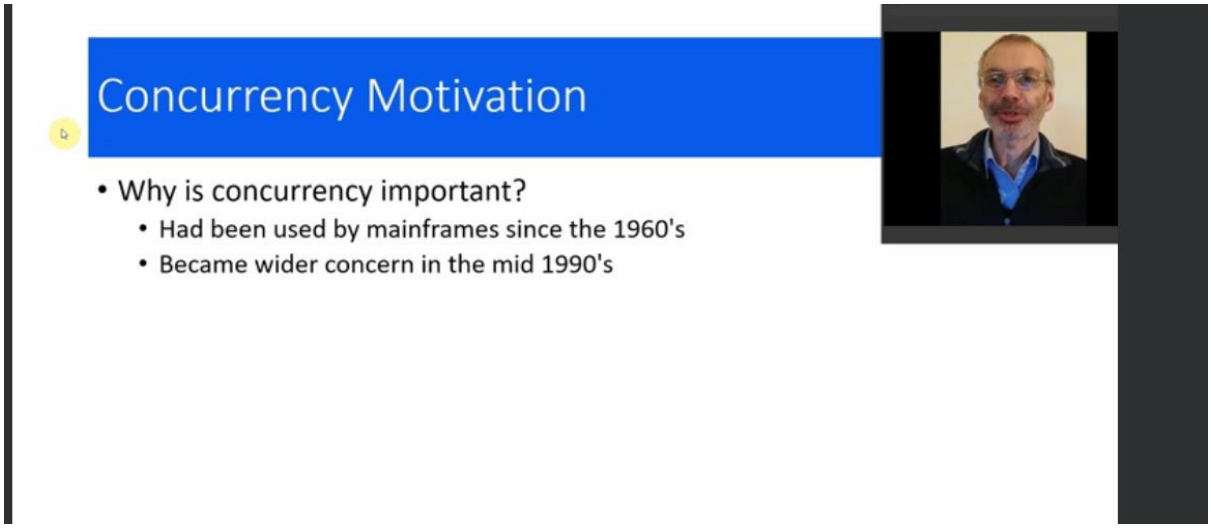
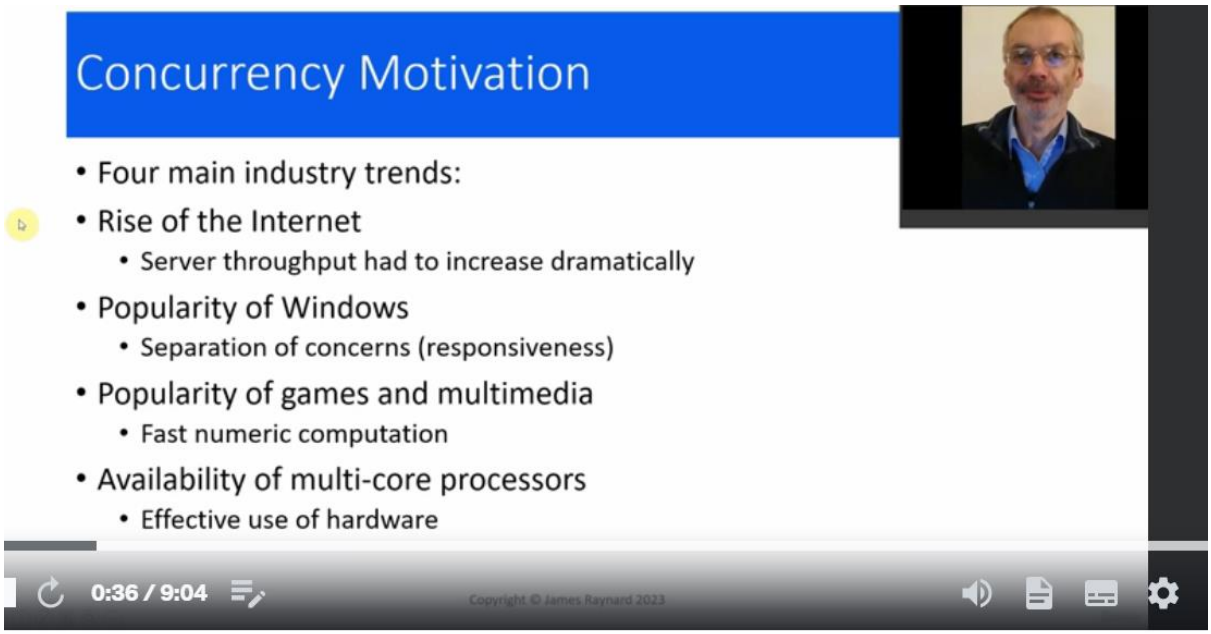

Concurrency
• Perform two activities at the same time
• Conceptually different
• e.g. A program which executes a long-running task
• Fetching data over the network
• Performing a lengthy computation
• The program displays "feedback" to the user
• "Hourglass" cursor
• Progress counter
• The task and the feedback are performed at the same time
2x
0:16 / 3:06
Copyright © James Raynard 2023

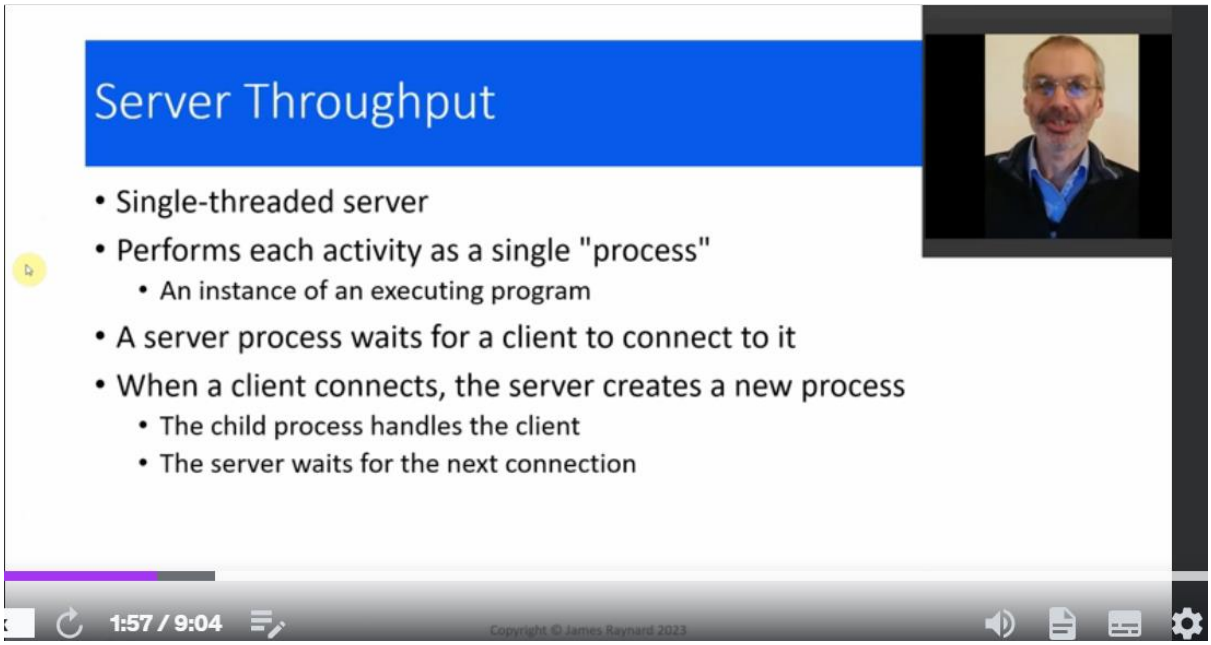

Concurrency and Operating Systems
• Standard in operating systems for many years
• Computers run many programs at the same time
• Compiler generates code
• Wordprocessor waits for next keystroke
• Mail program fetches a new email
2x
1:22 / 3:06
Copyright © James Raynard 2023

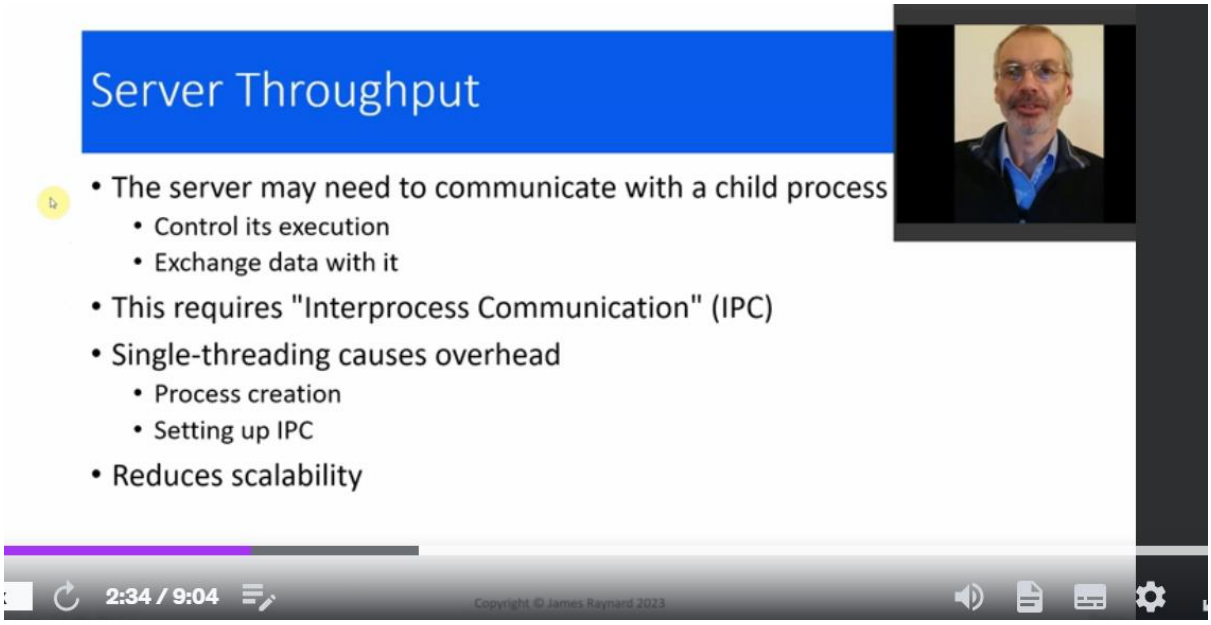

Hardware Concurrency
• Modern computers have multiple processors
• Multiple CPU chips
• CPU chips which contain multiple processor "cores"
• Different processors can perform different activities at the same time
• Even within the same program
• These are known as "hardware threads"
• Each processor follows its own "thread of execution" through the code
• 1 hardware thread <=> 1 processor core
2x
1:52 / 3:06
Copyright © James Raynard 2023

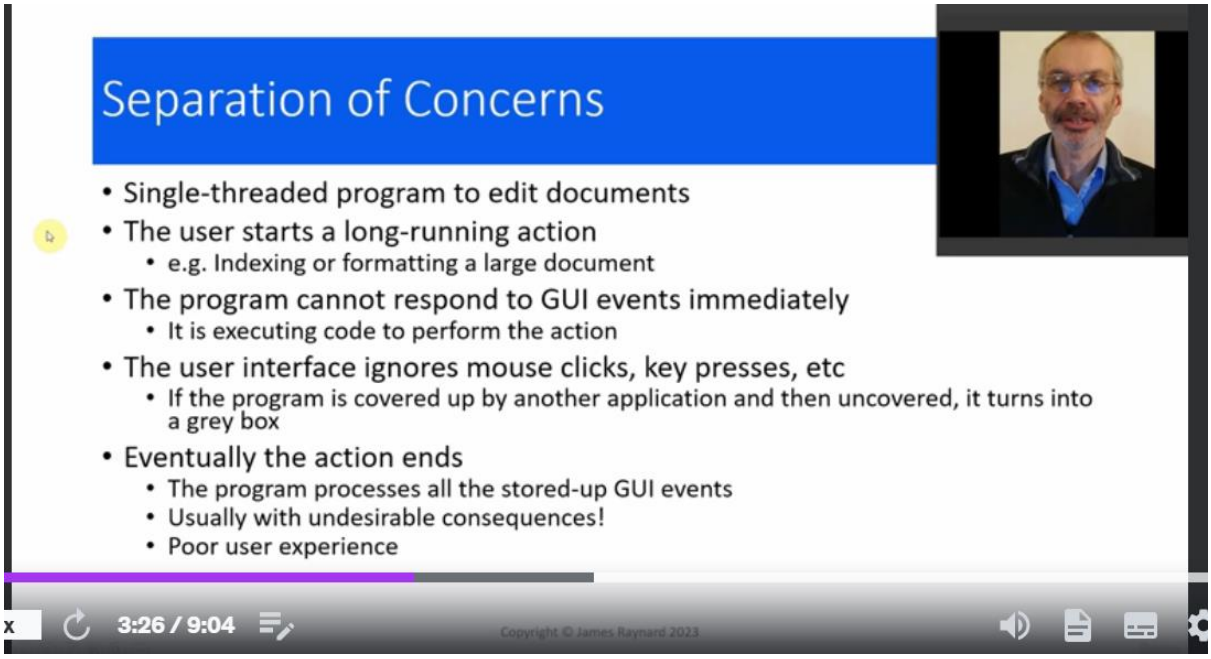

Software Concurrency
• Modern operating systems support "software threading"
• A program can perform multiple activities at the same time
• These activities are managed by the operating system
• Typically there are more software threads than hardware threads
• Some threads may have to stop and wait for something
• While they are waiting, another thread which is ready can run
2:38 / 3:06
Copyright © James Raynard 2023

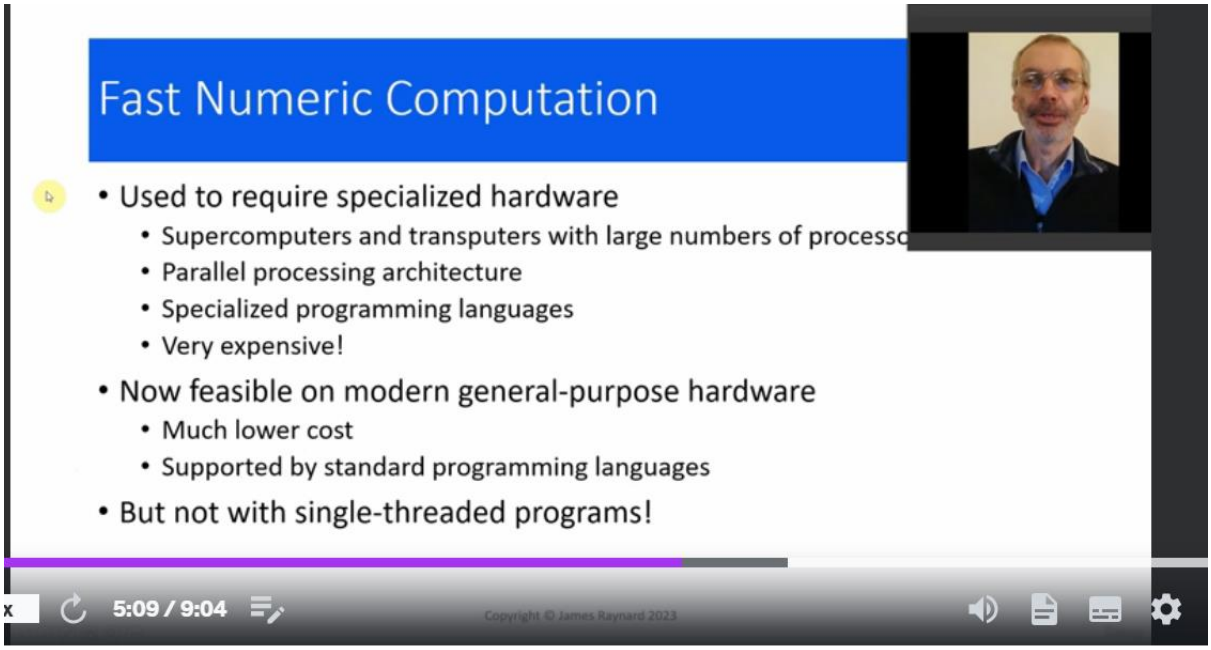

Concurrency Motivation
• Why is concurrency important?
• Had been used by mainframes since the 1960's
• Became wider concern in the mid 1990's


Concurrency Motivation
• Four main industry trends:
• Rise of the Internet
• Server throughput had to increase dramatically
• Popularity of Windows
• Separation of concerns (responsiveness)
• Popularity of games and multimedia
• Fast numeric computation
• Availability of multi-core processors
• Effective use of hardware
0:36 / 9:04
Copyright © James Raynard 2023

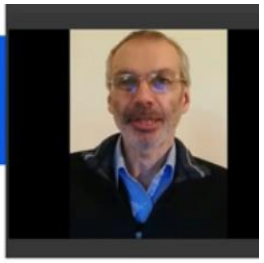

Server Throughput
• Single-threaded server
• Performs each activity as a single "process"
• An instance of an executing program
• A server process waits for a client to connect to it
• When a client connects, the server creates a new process
• The child process handles the client
• The server waits for the next connection
1:57 / 9:04
Copyright © James Raynard 2023


Server Throughput
• The server may need to communicate with a child process
• Control its execution
• Exchange data with it
• This requires "Interprocess Communication" (IPC)
• Single-threading causes overhead
• Process creation
• Setting up IPC
• Reduces scalability
2:34 / 9:04
Copyright © James Raynard 2023


Separation of Concerns
• Single-threaded program to edit documents
• The user starts a long-running action
• e.g. Indexing or formatting a large document
• The program cannot respond to GUI events immediately
• It is executing code to perform the action
• The user interface ignores mouse clicks, key presses, etc
• If the program is covered up by another application and then uncovered, it turns into a grey box
• Eventually the action ends
• The program processes all the stored-up GUI events
• Usually with undesirable consequences!
• Poor user experience
3:26 / 9:04
Copyright © James Raynard 2023

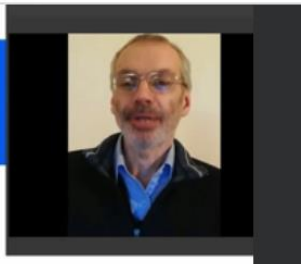

Fast Numeric Computation
• Used to require specialized hardware
• Supercomputers and transputers with large numbers of processo
• Parallel processing architecture
• Specialized programming languages
• Very expensive!
• Now feasible on modern general-purpose hardware
• Much lower cost
• Supported by standard programming languages
• But not with single-threaded programs!
5:09 / 9:04
Copyright © James Raynard 2023

Effective Use of Hardware



- Demand for faster and more powerful computers
  - Make the processor bigger
  - Raise the clock frequency
- Hardware engineers approaching physical limits
  - Speed at which electrons can move through silicon
  - Heat generated
- Hardware designers began using "cores" instead
  - Several processors on the same silicon chip
- A single-threaded program can only run on one core at a time
  - Only uses a fraction of the system's potential

Benefits of Concurrency



- Server throughput
  - The child and server run in the same process
  - No overhead from creating processes
  - Can have direct access to each other's data
- Separation of concerns
  - A program can respond to GUI events while performing a long-running task
  - Improves the user experience
- Fast numeric computation
- Effective use of hardware
  - A program can execute on different cores at the same time

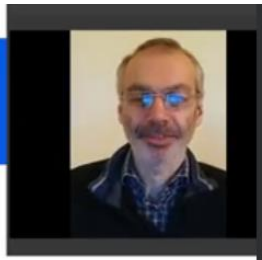
Single-threaded Concurrency

- Each activity requires a separate process
  - Has one execution path or "thread" through the program's code
- May need "interprocess communication"
  - Message queue, pipe, semaphore, shared memory, network socket, etc
- e.g. Program which displays a progress counter

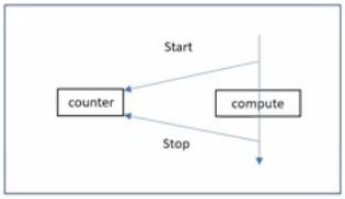
Single-threaded Concurrency

- Each process has its own private memory space
  - Cannot accidentally alter another process's data
- Processes can be run on different computers over a network
- Creating a new process can be slow on some systems
- IPC
  - Adds complication
  - Can be slow to set up
  - No direct support in C++

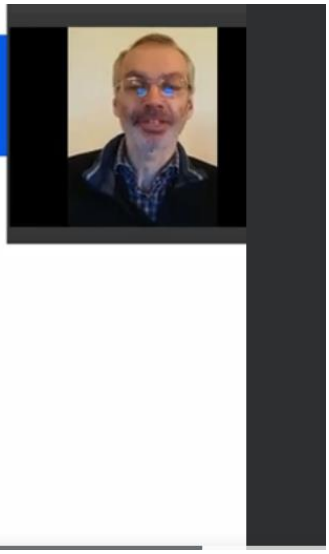
Multi-threaded Concurrency



- A single process performs all activities
- Each activity has its own execution path or "thread"
- Concurrency is achieved by starting a new thread for each activity



Threads

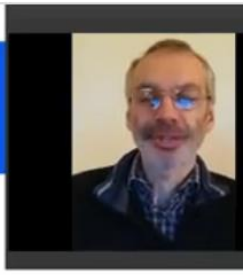


- Each thread is an independent execution environment
  - Has its own execution stack
  - Has its own processor state
- Threads are "light weight processes"
  - Less overhead to start up
  - Smaller task switching overhead
  - Easy to share data between threads

Threads

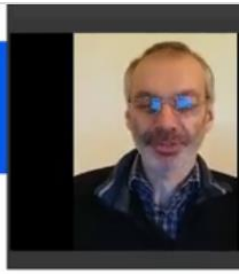
- All the threads in a process share its memory space
  - Can access global data
  - Pointers can be passed between threads
- Lack of data protection between threads
  - Can cause data inconsistency and data corruption
  - Requires careful programming

Advantages of Concurrency



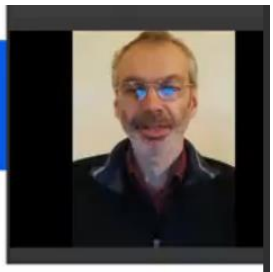
- Improves responsiveness of the program
  - The user is never left staring at a "stuck" program
- Improves throughput
  - Processing large amounts of data in parallel takes less time
- Allows separation of logically distinct operations
  - e.g. mail program starts new threads
  - Compose an email
  - New fetch messages etc
- Takes full advantage of modern hardware
  - Threads can be distributed among processor cores

Disadvantages of Concurrency



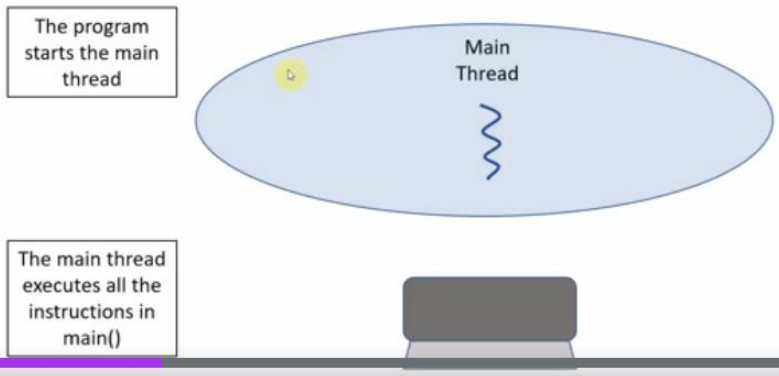
- Adds complexity to programs
  - Code is harder to write and harder to understand
  - Bugs are more likely
  - Requires careful programming
- May not result in faster programs
  - Data protection overhead
  - Thread coordination overhead
- Use only when the benefits outweigh the costs!

Single-threaded C++ Program Structure

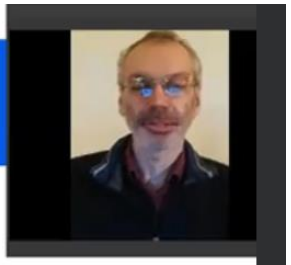


- Non-threaded C++ programs have a single thread
  - When this thread starts, it executes the code in main()
  - main() is the entry point function for the thread
- main() can call other functions
  - These run in the main thread
- When main() returns, the thread ends
  - The program ends when the main thread ends

Single-threaded Program

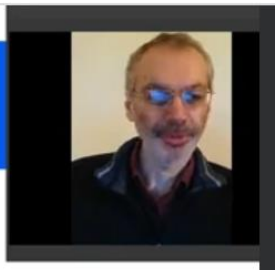


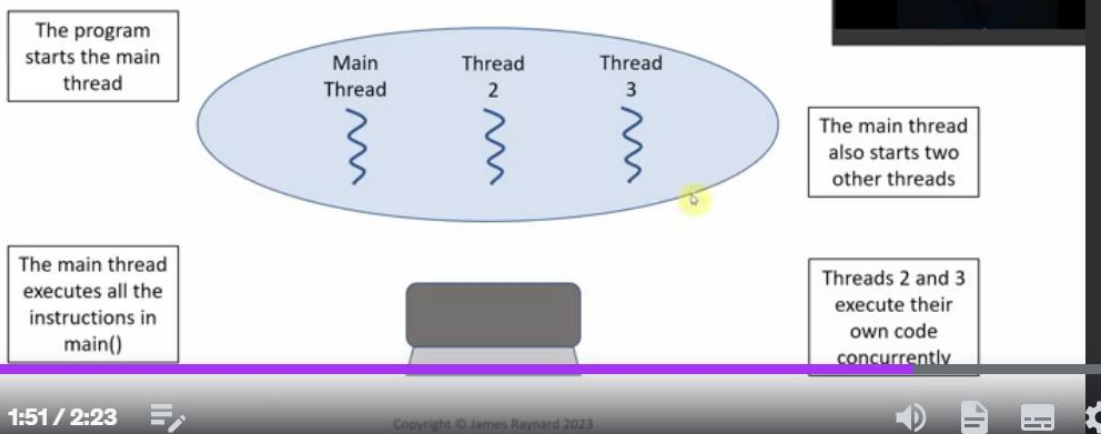
Multi-threaded C++ Program Structure



- Multi-threaded C++ programs also have a main thread
  - The main thread starts additional threads
  - These could, in turn, start further threads
- Each thread has its own entry point function
  - When the thread starts, it executes the code in this function
  - When the function returns, the thread ends
  - The main thread continues to execute its own code
  - It does not wait for the other threads, unless we explicitly tell it to

Multi-threaded Program

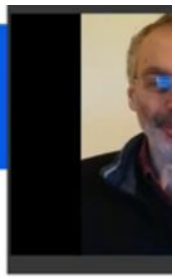




Multi-threading with C API's

- At first, no direct support for threads in C++
- Programmers could only use threads through C interfaces
  - pthreads library
    - For POSIX systems
  - Windows API
    - Provided access to threads in the "Base Services"
  - Various Unix API's
    - System calls provided access to operating system's thread support
    - Varied between providers
  - OpenMP
    - Process C-style array loops in parallel

Multi-threading with C++ API's



- Later, C++ libraries appeared
  - ACE
    - Cross-platform networking library with support for threads
  - Boost
    - Project to develop new features for C++
    - Basically a wrapper around the operating system's threading facilities
    - Provided a consistent interface
    - Used as basis for C++11's thread support
  - Poco
    - More modern and lightweight version of ACE

Multi-threading with Standard C++

- C++ added support for concurrency in 2011
- Programmers can write threaded code which is
  - Portable and efficient
  - Has well-defined semantics
  - Allows fine-grained control at low levels
- Standard Library changes
  - Features added to create and manage threads
  - All classes and functions made thread-safe
- We will only consider standard C++ features in this course

Multi-threading with C++11

- The std::thread class is the base level of concurrency
  - Rather low-level implementation
  - Maps onto a software thread
  - Managed by the operating system
  - Similar to Boost threads, but with some important differences
    - No thread cancellation
    - Different argument passing semantics
    - Different behaviour on thread destruction

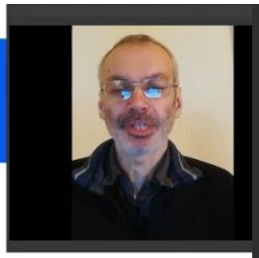
Multi-threading with later C++

- C++14
  - Read-write locks
- C++17
  - Many standard algorithms can execute in parallel
- C++20
  - Joining threads
  - Thread cancellation
  - Coroutines, semaphores, latches and barriers
- C++23
  - Improved support for coroutines

Threads and Tasks

- The words are often used interchangeably
- In this course:
- A thread is
  - A software thread
  - An object of the C++ std::thread class
- A task is
  - A higher-level abstraction
  - Some work that should be performed concurrently

Launching a Thread

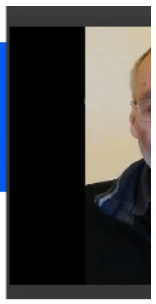


- We need to create an std::thread object
  - The class is defined in <thread>
- The constructor starts a new execution thread
  - This will execute immediately
- The parent thread will continue its own execution

Thread Entry Point

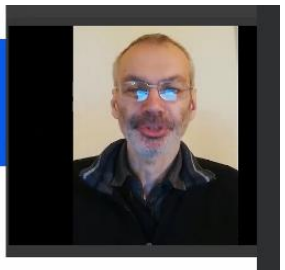
- std::thread's constructor takes a callable object
  - The thread's entry point function
- The execution thread will invoke this function
- The entry point function
  - Can be any callable object
  - Cannot be overloaded
  - Any return value is ignored

"Hello, Thread" Program



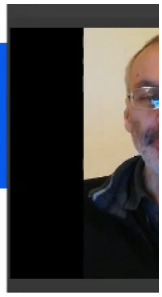
```
#include <thread>

// Callable object - thread entry point
void hello() {
   std::cout << "Hello, Thread!\n";
}

// Create an std::thread object
// Pass the entry point function to the constructor
std::thread thr(hello);
```

Thread Termination

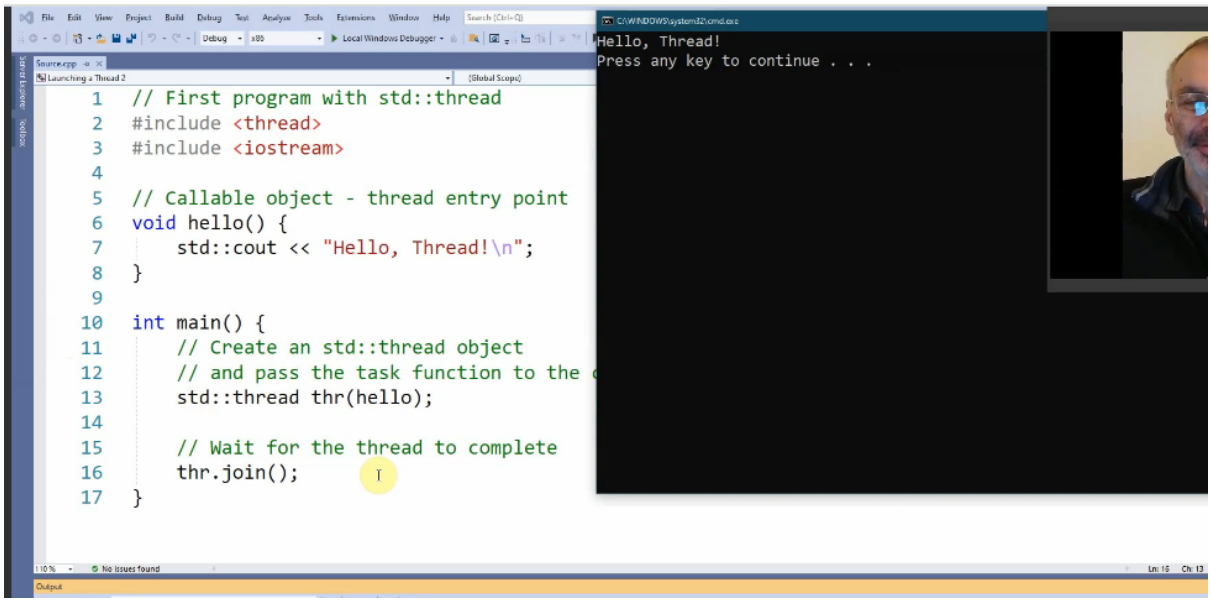


- The parent thread completes its execution
  - The std::thread object goes out of scope
  - Its destructor is called
- This can happen while the child is still running
  - "Zombie" threads?
  - By default, the destructor calls std::terminate()
  - This terminates all threads, including main()

Joining a Thread



- std::thread has a join() member function
- This is a "blocking" call
  - Does not return until the thread has completed execution
  - The parent has to stop and wait for the thread to complete
- Prevents std::thread's destructor calling std::terminate()
- This is known as "joining" the thread
  - The program can continue after the std::thread object is destroyed



"Hello, Functor" Program

- We can use any callable object
- For example, an object of a "functor" class
  - Overloads the () operator
- Pass the object to std::thread's constructor
- The object's () operator will be the entry point function

Functor as Entry Point

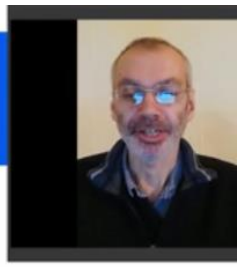
```
// Functor class with overloaded () operator
class Hello {
 public:
   void operator()() { std::cout << "Hello, Functor Thread!\n"; }
};

// Create an object of the functor class
Hello hello;

// Pass the object to std::thread's constructor
std::thread thr(hello);

// Wait for the thread to complete
thr.join();
```

Lambda Expression as Entry Point

- We can use a lambda expression

```
// Capture by reference
std::thread thr([]() {
   std::cout << "Hello, Lambda Thread!\n";
});
```

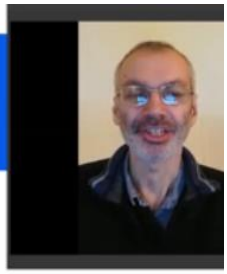
Thread Function with Arguments



- We can pass arguments to the entry point function
- We list them as extra arguments to the constructor

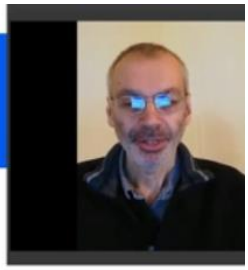
```
// Thread entry point function
void hello(std::string);

// hello() takes a string argument
std::thread thr(hello, "Hello, Thread!");
```

Thread Function with Arguments



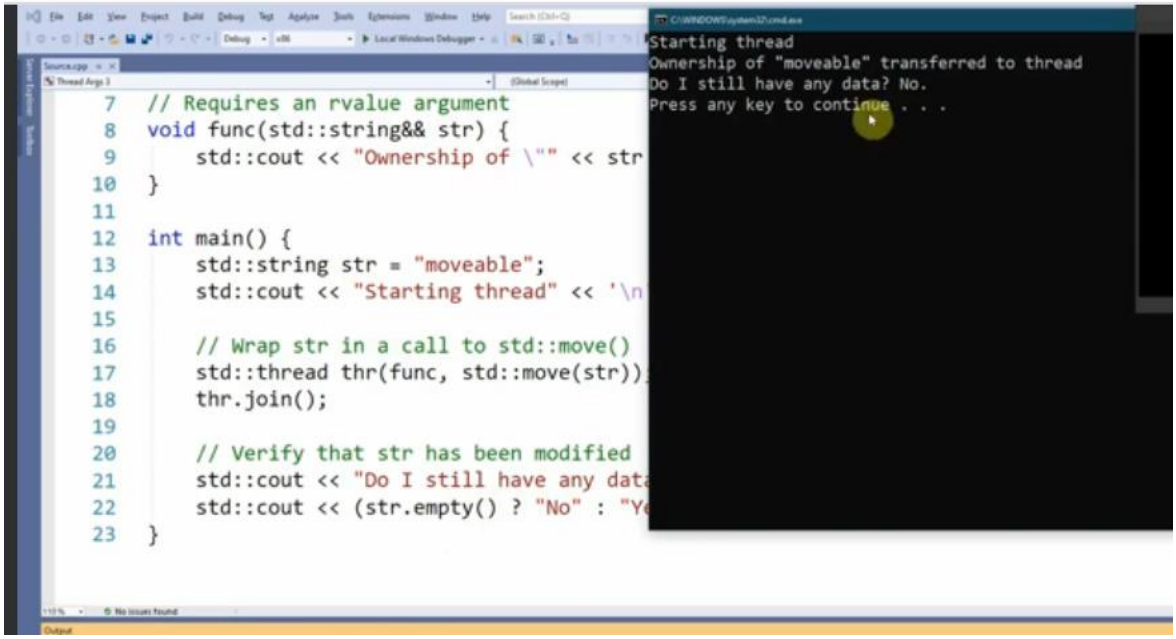
- The std::thread object owns the arguments
  - lvalue arguments are passed by value
  - rvalue arguments are passed by move

Thread function with Pass by Move



- To pass by move, we must provide an rvalue
  - The argument must have a move constructor

```
// Callable object - thread entry point
void func(std::string &&);

std::string str = "abc";

// Wrap the argument in a call to std::move()
std::thread thr(func, std::move(str));
```

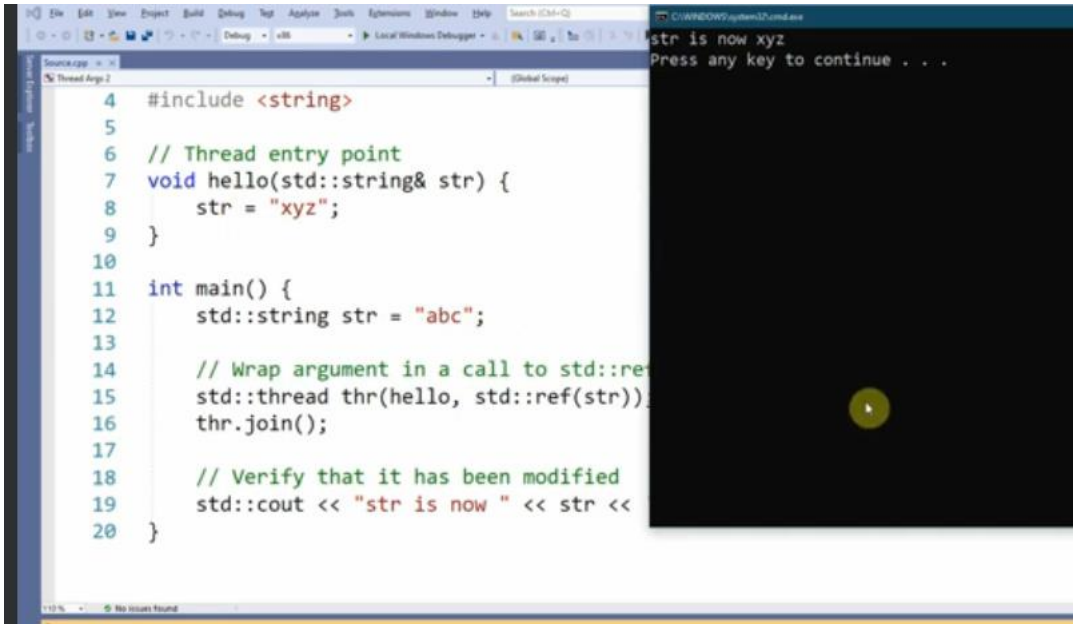


Thread Function with Reference Argument

- Use a reference wrapper
  - Wrap the argument in a call to std::ref()

```
// Callable object - thread entry point
void hello(std::string&);

std::string str = "abc";

// Wrap the argument in a call to std::ref()
std::thread thr(hello, std::ref(str));
```

  - Use std::cref() for a const reference
  - Beware of dangling references!

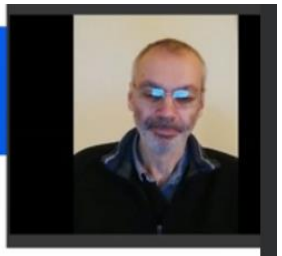


Member Function as Entry Point

- We can use a member function as the entry point
- Requires an object of the class

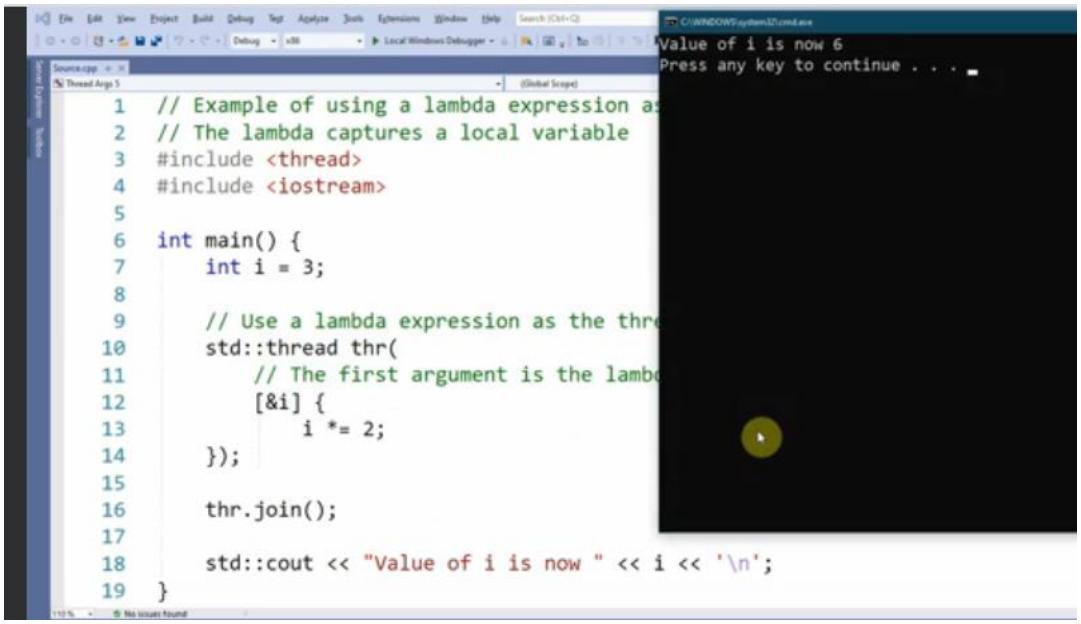
```
// Class which has the member function
class greeter {
 public:
   void hello();
};

// An object of the class
greeter greet;

// Pass a pointer to the member function and a pointer to the object
std::thread thr(&greeter::hello, &greet);
```

Lambda Expression as Entry Point



- We can use a lambda expression

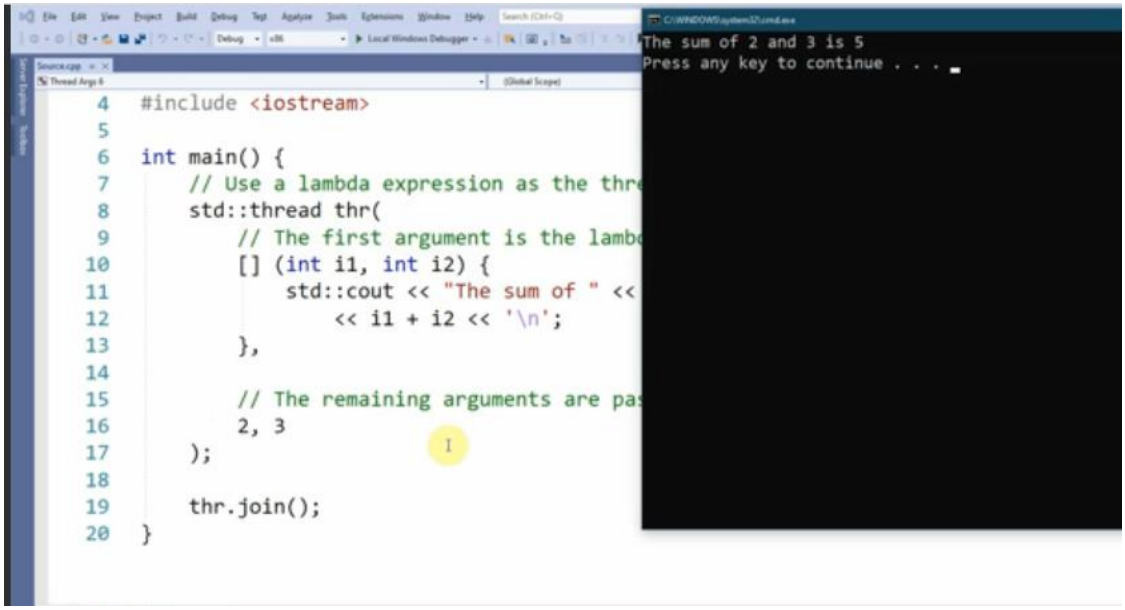
```
int i = 3;

// Capture by reference
std::thread thr([&i] {
  i *= 2;
});
```



Lambda Expression with Arguments

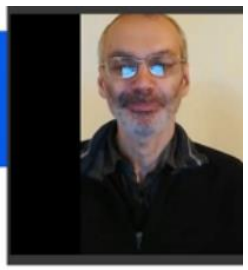
- We can pass arguments to the lambda expression

```
std::thread thr(
  // The first argument is the lambda expression
  [] (int i1, int i2) {
        std::cout << i1 + i2 << '\n';
  },

  // The remaining arguments are passed to the lambda expression
  2, 3
);
```



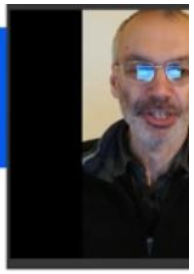
Computer with Single Processor

Single-threaded Program on Computer with Single Processor



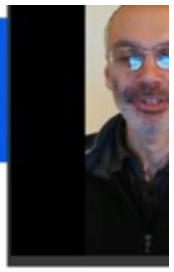
- A program runs on a single processor
  - The program's instructions are loaded from memory
  - The instructions are executed on the processor
- Registers store information needed to execute the current instruction
  - Stack pointer
  - Operand arguments
  - etc
- Data is loaded from memory into processor registers as needed
- If modified, the new value is then written back to memory

Multi-threaded Program on Computer with Single Processor



- Implemented by "time slicing"
- Each thread runs on the CPU for a short time, e.g.
  - Thread A starts, runs for a short period, then pauses
  - Thread B starts, runs for a short period, then pauses
  - Thread C starts, runs for a short period, then pauses
  - Thread B runs again from where it left off, then pauses
  - Thread C runs again from where it left off, then pauses
- This is done very quickly
- The threads appear to run concurrently

Thread Scheduler



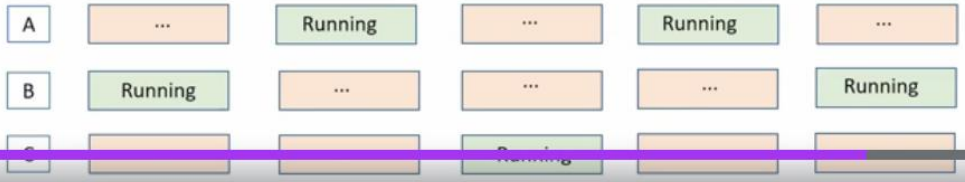
- A scheduler controls how threads execute
  - Similar to the operating system's scheduler for processes
- Pre-emptive task switching
  - A thread can run for a certain time
  - The scheduler will interrupt the thread when it has used up its time slot
  - Another thread can then run
  - The scheduler will make interrupted thread "sleep"

Thread Scheduling

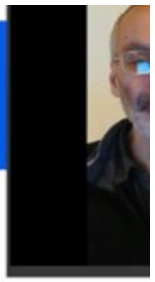
- Threads may start in any order
- A thread may be interrupted at any time
- A thread may be restarted at any time
- While a thread is sleeping, another thread may run
  - This can run the same code as the inactive thread
  - It can access data which it shares with the inactive thread

Thread Scheduling

- The execution of the threads is interleaved
  - Thread B starts and is interrupted
  - Thread A starts and is interrupted
  - Thread C starts and is interrupted
  - Thread A executes some more instructions and is interrupted
  - Thread B executes some more instructions and is interrupted



Disadvantages of Time Slicing



- Requires a "context switch"
- Save the processor state for the current thread
  - Current values of processor registers
  - Program's instruction pointer
  - etc
- Load the saved processor state for the next thread
  - Values of processor registers when stopped, etc
  - May also have to reload the thread's instructions and data
- The processor cannot execute any instructions during a context switch

Cache

- CPUs became much faster at processing instructions
- However, RAM access speed did not improve very much
- The CPU could not process instructions during data transfers
  - The CPU spent a lot of time waiting for RAM to respond
  - Memory latency became a problem
- Computers started using small amounts of "cache" memory

Computer with Single Processor and Cache

Cache Memory

- Cache memory
  - Physically close to the CPU
  - Stores copies of current and recently used data
  - Reduces the time spent communicating with main memory
- Uses "static" RAM
  - This is faster than the "dynamic" RAM used in main memory
  - More expensive
  - Uses more power
  - Usually much smaller than main memory

Cache Fetch

- The CPU wants to fetch data
- Does the cache have a copy of the data?
- Yes
  - There is a "cache hit"
  - The CPU uses that copy
- No
  - There is a "cache miss"
  - The data is fetched from main memory

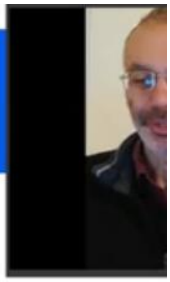
Cache Write

- The CPU wants to store data
- It writes the data to the cache
- The data in the cache is written to main memory
  - The CPU can continue working while this happens

Cache Controller

- The process is managed by a "cache controller"
- The data is transferred in fixed-size blocks
  - Known as "cache lines"
  - Typically 64 bytes on modern hardware (8 words)
- Each cache line relates to an address in main memory

Computer with Multiple Processors



- Multiple sockets
  - 2 or more processor chips in the computer
- Multiple processor cores
  - Several processors within the same silicon chip
- Hyperthreads
  - Duplicate some of the circuitry within a processor core
  - Enough to run a separate thread, with its own execution stack

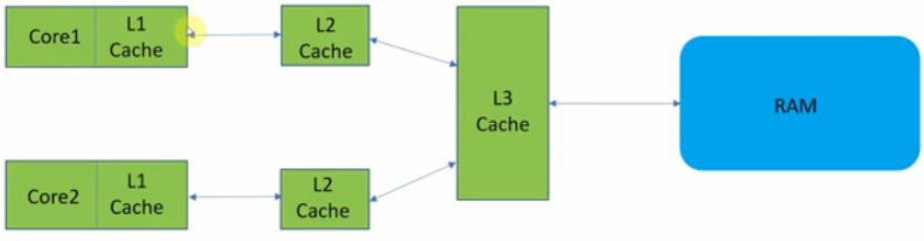
Memory

- Processor speed has increased hugely
- Memory performance has not kept up
- Designers had to find new ways to improve performance
  - Many more registers for storing data and code in the processor
  - Additional caches between processors and main memory

Multiple Levels of Cache

- Level 1 cache
  - Private to each processor core
  - As close to the core as possible
- Level 2 cache
  - Usually private to each core
- Level 3 cache
  - Shared by all the cores on the same socket

Multiple Core Architecture



Cache Controller

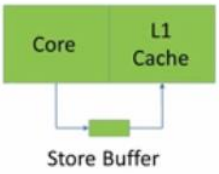
- Coordinates the caches
  - The same data should have the same value in all caches
  - The same data should have the same value on all cores
  - "Cache coherency"
- Monitors caches for data changes
  - A core modifies data in its level 1 cache
  - The data is updated in the core's level 2 cache
  - The data is updated in the level 3 cache
  - The data is updated in the the other cores's caches

Typical Cache Sizes and Latency

- Intel "Coffee Lake" architecture (2018)
  - Level 1 cache 32kB data, 32kB instructions per core
  - Level 2 cache 256kB data per core
  - Level 3 cache 2-12MB data shared by all cores
- Latency measures the time taken to retrieve data
  - Level 1 cache 4 cycles
  - Level 2 cache 12 cycles
  - Level 3 cache 26-37 cycles
  - Main memory ~300 cycles

Optimizations

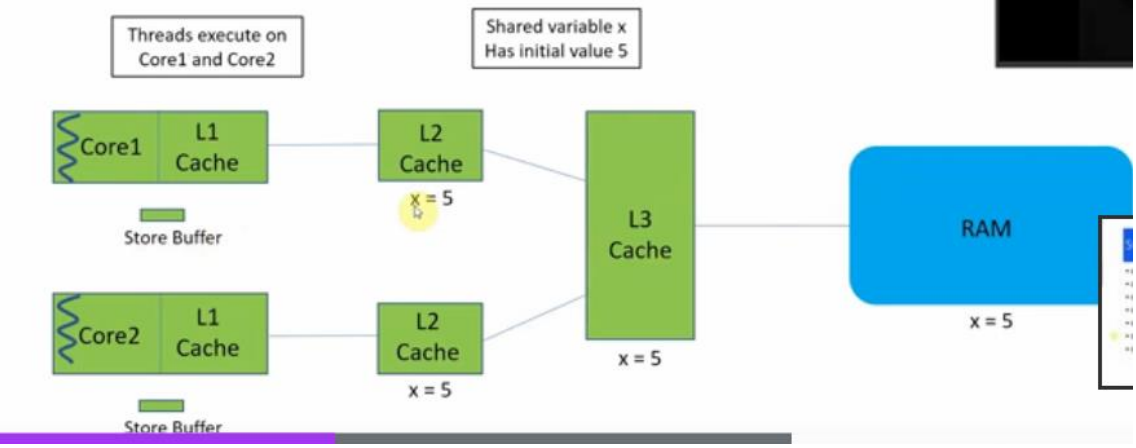
- Pre-fetcher
  - Looks at incoming instructions
  - Fetches data before it is needed
- Store buffer
  - This is between the core and the level 1 cache
  - Modified data is written to this buffer
- The core can proceed to the next instruction
  - Does not need to wait for the L1 cache
- These optimizations provide huge improvements
  - Avoid blocking the core when there is a cache miss



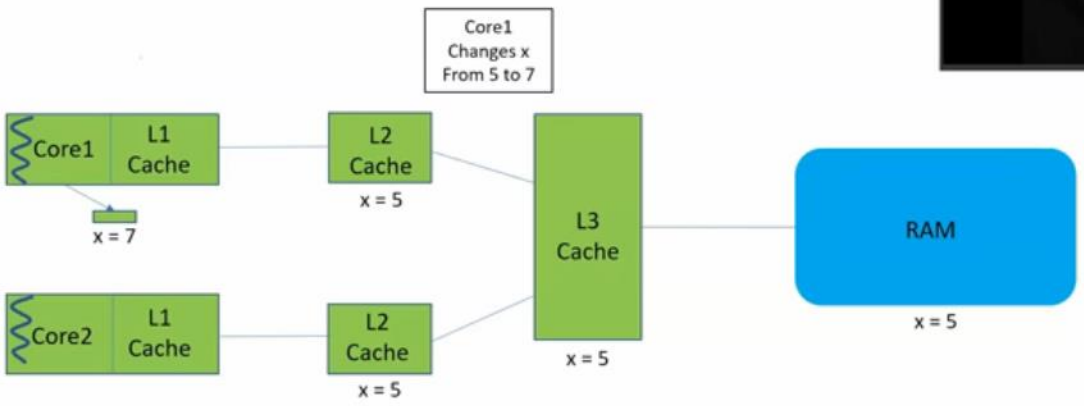
Synchronization Issues

- Different threads execute on different cores
- They may share data
- This can cause synchronization issues

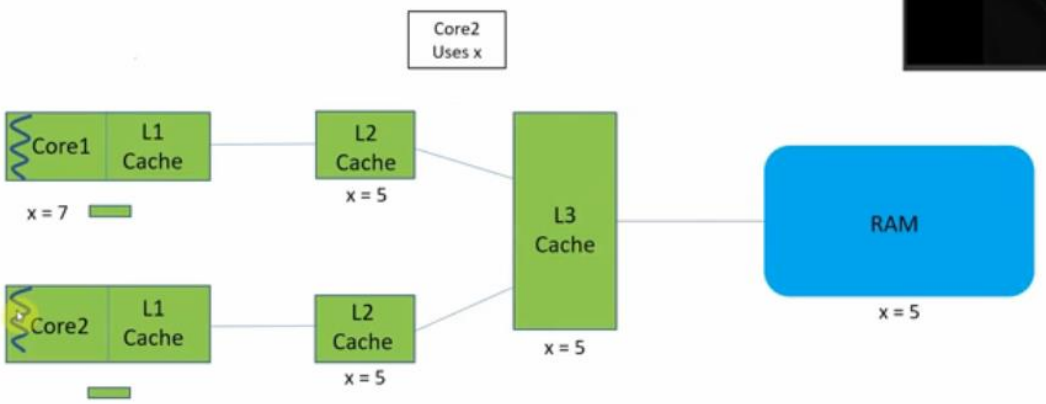
Synchronization Issues



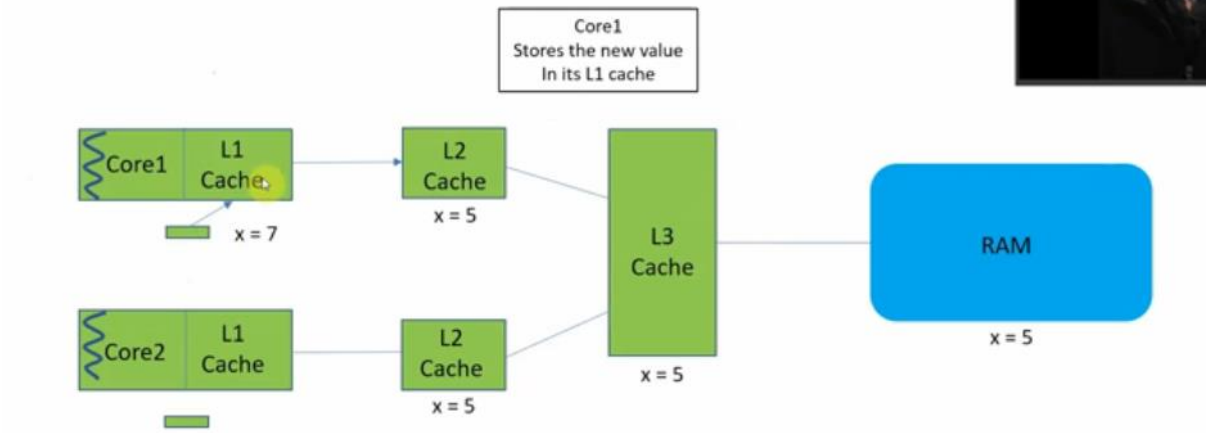
Synchronization Issues



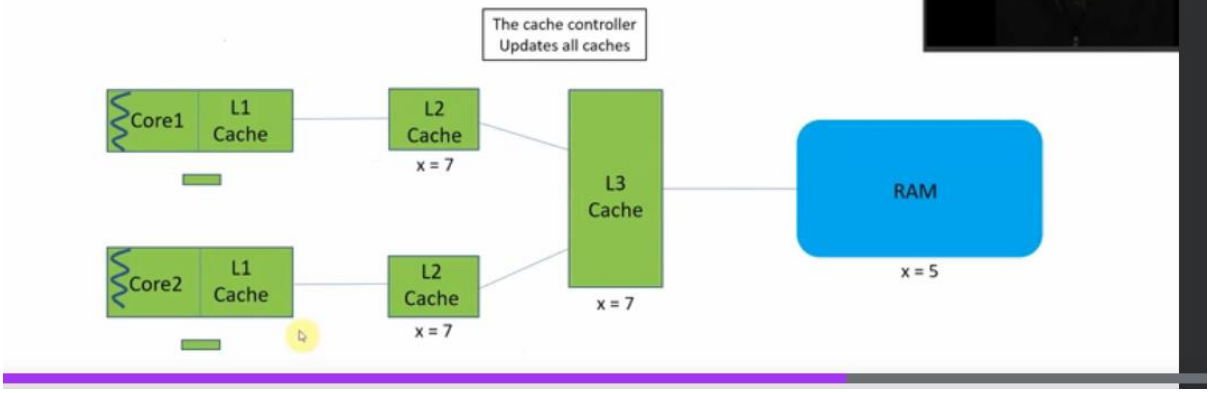
Synchronization Issues



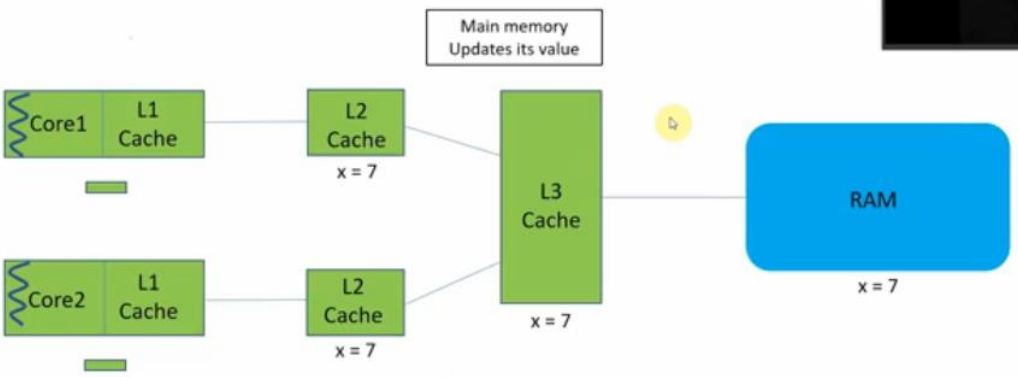
Synchronization Issues



Synchronization Issues



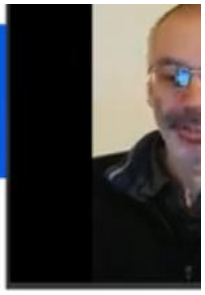
Synchronization Issues



Synchronization Issues

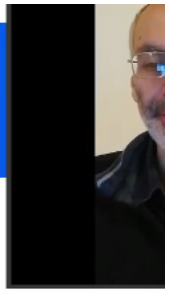
- Core 1's thread modifies the shared data
- Core 1 writes the new value to its store buffer
- Core 2's thread wants to use the shared data
- Core 2 pre-fetches the shared data, or loads it from cache
- Core 2 gets the old value
- Core 2's thread does its computation, using the old value
- Core 1's store buffer writes the new value to cache

Synchronization Issues



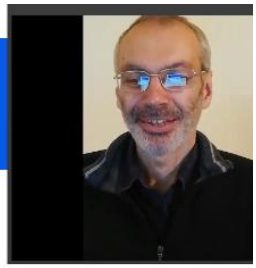
- Core 1's thread modifies the shared data
- Core 1 writes the new value to its store buffer
- Core 1's store buffer writes the new value to cache
- Core 2's thread wants to use the shared data
- Core 2 pre-fetches the shared data, or loads it from cache
- Core 2 gets the old value
- Core 2's thread does its computation, using the old value

System Thread Interface



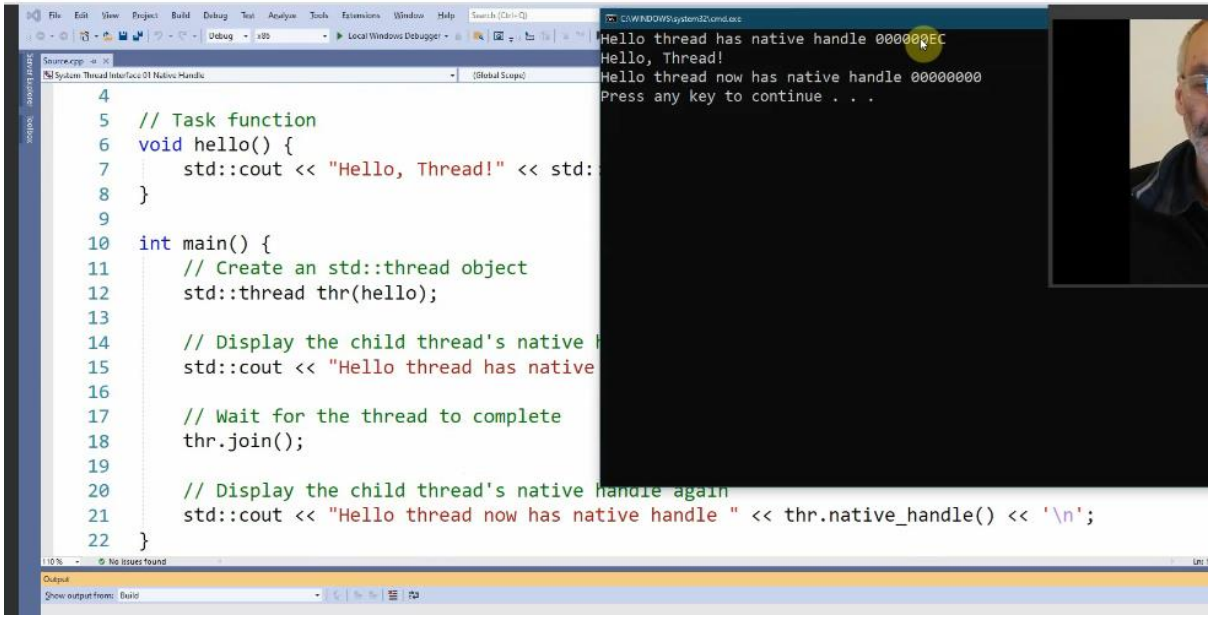
- std::thread uses the system's thread implementation
- We may need to use the thread implementation directly
- Some functionality is not available in standard C++
- Thread priority
  - Give a thread higher or lower share of processor time
- Thread affinity
  - "Pin" a thread on a specific processor core

native_handle()



- Each execution thread has a "handle"
  - Used internally by the system's thread implementation
  - Needed when making calls into the implementation's API
- Returned by the native_handle() member function

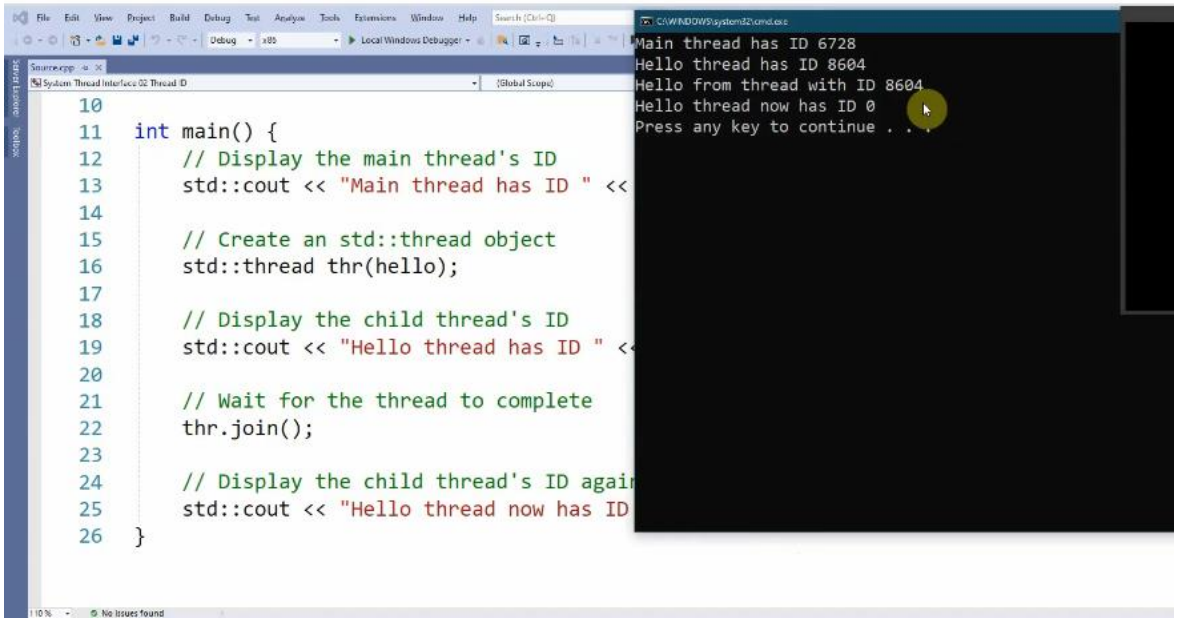
```
// Get the handle associated with an std::thread object
thr.native_handle()
```

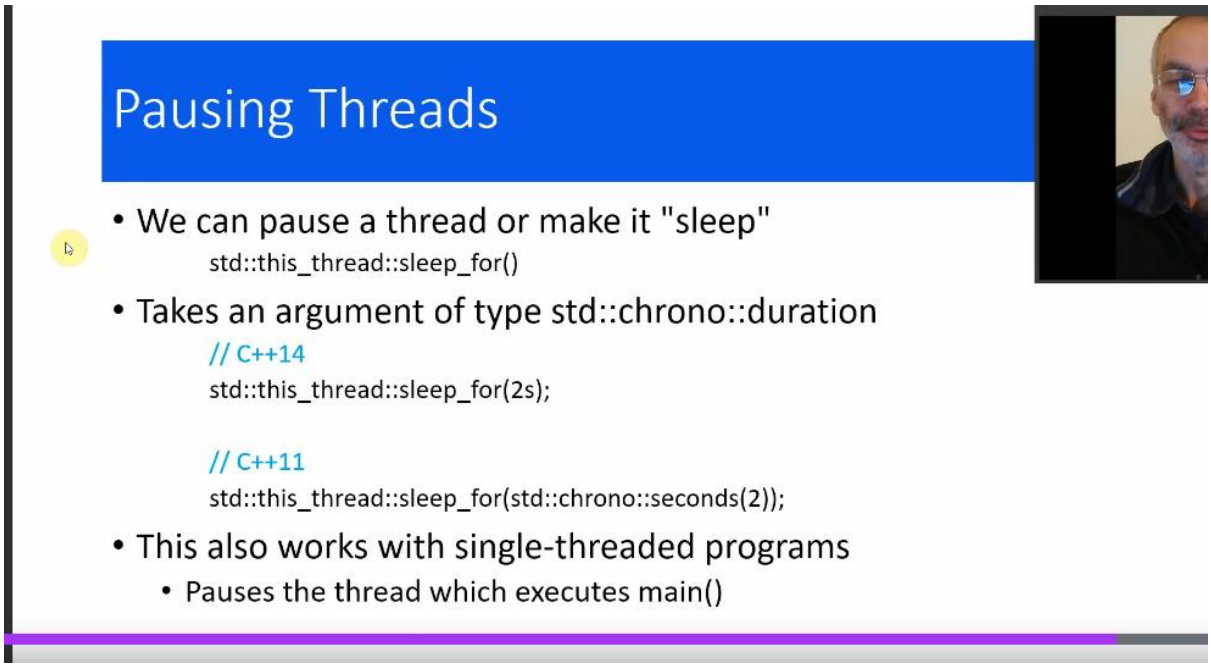


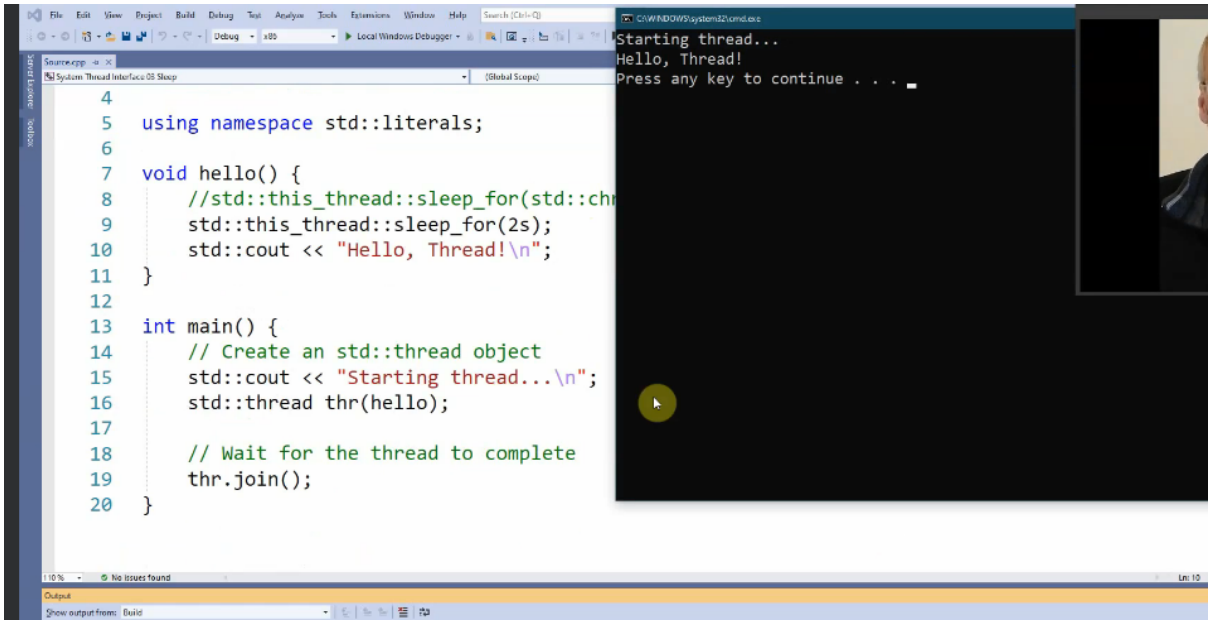
std::thread ID

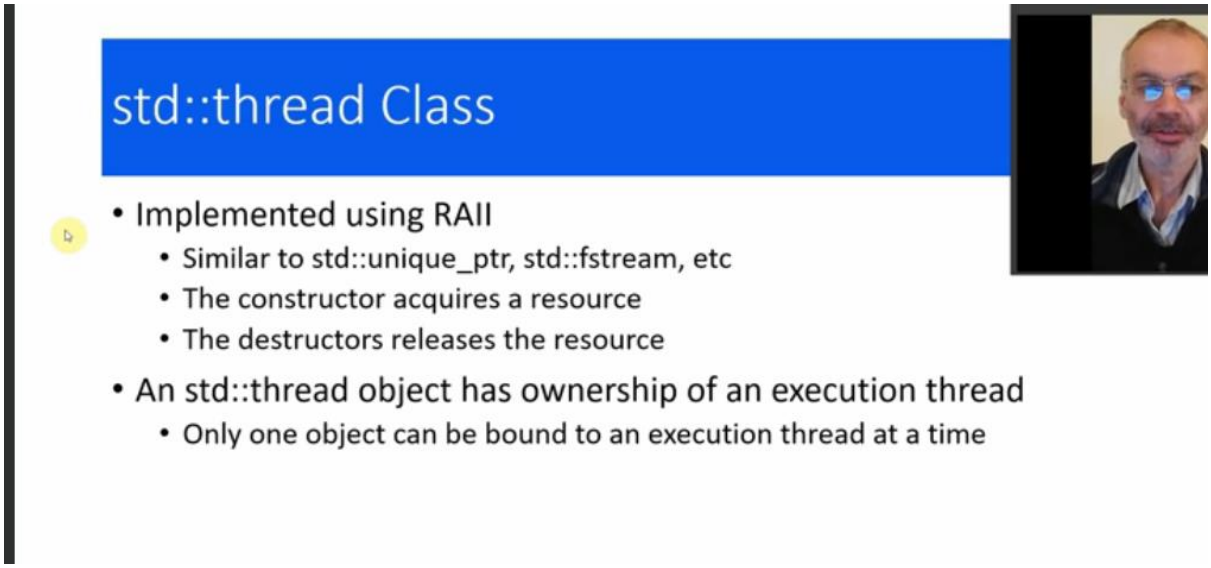
- Each execution thread has a thread identifier
  - Used internally by the system's implementation
- Guaranteed to be unique
  - If two thread identifiers are equal, the related objects are identical
  - Could be used to store std::thread objects in associative containers
  - A new thread may get the ID of an earlier thread which has completed

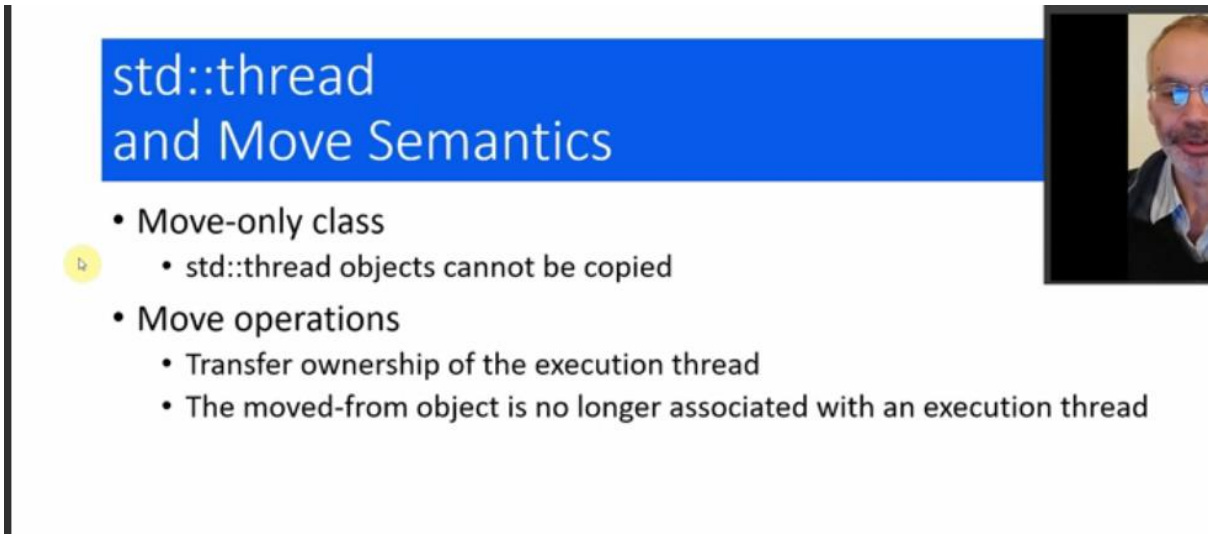
```
// Return the identifier of the current thread
std::this_thread::get_id();

// Return the identifier associated with an std::thread object
thr.get_id();
```


10
11 int main() {
12 // Display the main thread's ID
13 std::cout << "Main thread has ID " <<
14
15 // Create an std::thread object
16 std::thread thr(hello);
17
18 // Display the child thread's ID
19 std::cout << "Hello thread has ID " <<
20
21 // Wait for the thread to complete
22 thr.join();
23
24 // Display the child thread's ID agai
25 std::cout << "Hello thread now has ID
26 }
Main thread has ID 6728
Hello thread has ID 8604
Hello from thread with ID 8604
Hello thread now has ID 0
Press any key to continue . . .


Pausing Threads
• We can pause a thread or make it "sleep"
std::this_thread::sleep_for()
• Takes an argument of type std::chrono::duration
// C++14
std::this_thread::sleep_for(2s);
// C++11
std::this_thread::sleep_for(std::chrono::seconds(2));
• This also works with single-threaded programs
• Pauses the thread which executes main()


Starting thread...
Hello, Thread!
Press any key to continue . . .
using namespace std::literals;
void hello() {
//std::this_thread::sleep_for(std::ch
std::this_thread::sleep_for(2s);
std::cout << "Hello, Thread!\n";
}
int main() {
// Create an std::thread object
std::cout << "Starting thread...\n";
std::thread thr(hello);
// Wait for the thread to complete
thr.join();
}


std::thread Class
• Implemented using RAII
• Similar to std::unique_ptr, std::fstream, etc
• The constructor acquires a resource
• The destructors releases the resource
• An std::thread object has ownership of an execution thread
• Only one object can be bound to an execution thread at a time


std::thread
and Move Semantics
• Move-only class
• std::thread objects cannot be copied
• Move operations
• Transfer ownership of the execution thread
• The moved-from object is no longer associated with an execution thread

Passing a std::thread Object

- Must use pass by move

```
// Function taking a thread object as argument
void func(std::thread thr);

// Pass a named object
// Use std::move() to cast it to rvalue
std::thread thr(...);
func(std::move(thr));

// Pass a temporary object
func(std::thread(...));
```

Returning a std::thread Object

- The compiler will automatically move it for us

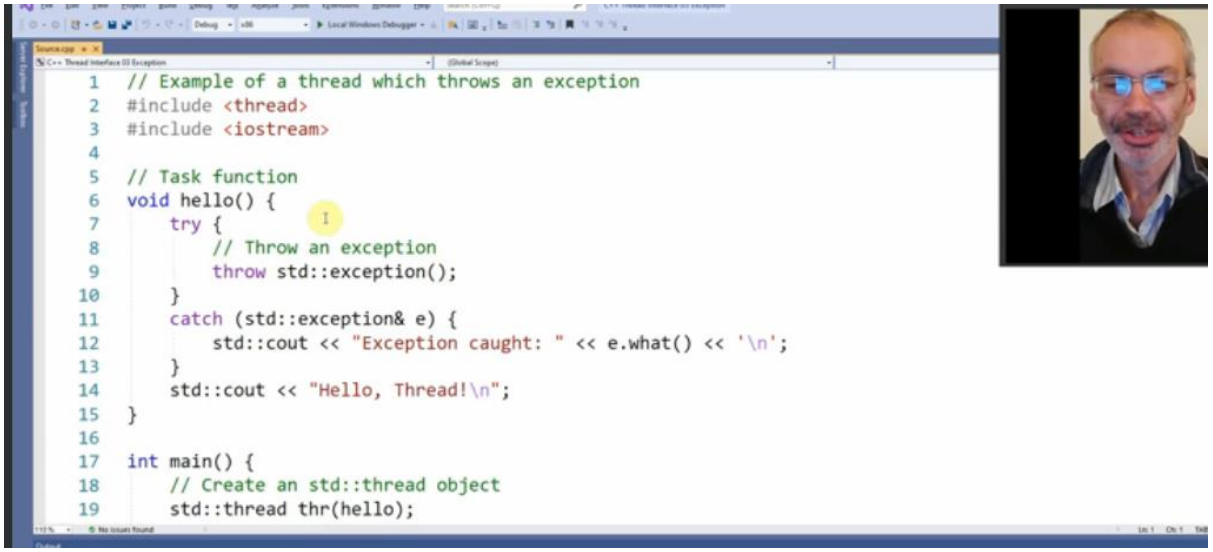
```
// Function returning a std::thread object
std::thread func() {
  // Return a local variable
  std::thread thr(...);
  return thr;

  // Return a temporary object
  return std::thread(...);
}
```

```cpp
#include <chrono>

// Task function for the thread
void hello() {
    std::cout << "Hello, Thread!\n";
}

// Function returning a std::thread object
std::thread func() {
    // Start the thread
    std::thread thr(hello);

    // Return a local variable
    return thr;

    // Return a temporary object
    // return std::thread(hello);
}
```

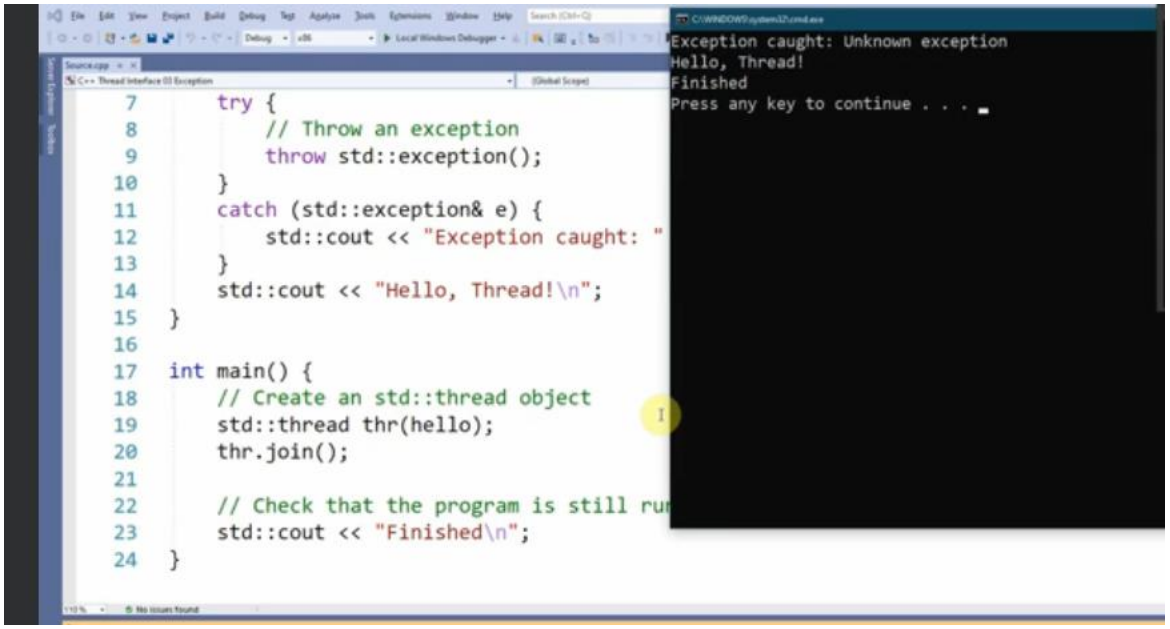
```cpp
    // Return a local variable
    return thr;

    // Return a temporary object
    // return std::thread(hello);
}

int main() {
    // Call a function which return an std::thread object
    std::thread thr = func();
    std::cout << "Received thread with ID " << thr.get_id() << '\n';

    // Our thr object now "owns" the system thread
    // It is responsible for calling join()
    thr.join();
}
```

Threads and Exceptions

- Each thread has its own execution stack
- This stack is "unwound" when the thread throws an exception
  - The destructors for all objects in scope are called
  - The program moves up the thread's stack until it finds a suitable handler
  - If no handler is found, the program is terminated
- Other threads in the program cannot catch the exception
  - Including the parent thread and the main thread
- Exceptions can only be handled in the thread where they occur
  - Use a try/catch block in the normal way



```cpp
// Example of a thread which throws an exception
#include <thread>
#include <iostream>

// Task function
void hello() {
    try {
        // Throw an exception
        throw std::exception();
    }
    catch (std::exception& e) {
        std::cout << "Exception caught: " << e.what() << '\n';
    }
    std::cout << "Hello, Thread!\n";
}

int main() {
    // Create an std::thread object
    std::thread thr(hello);
```

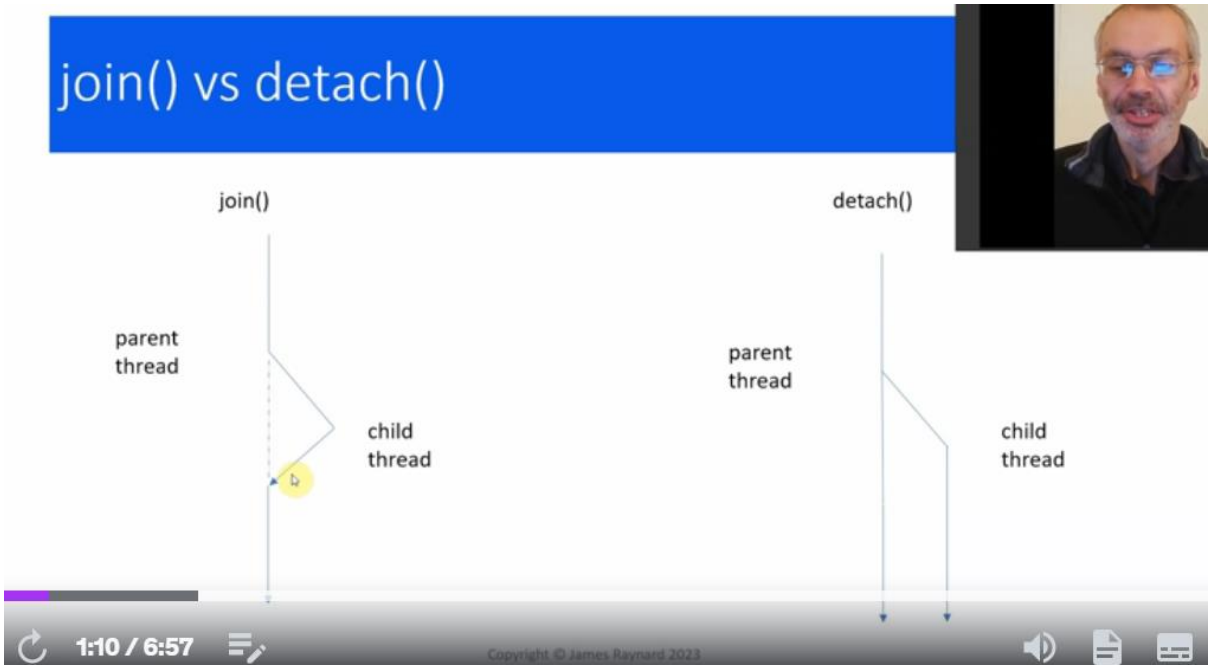


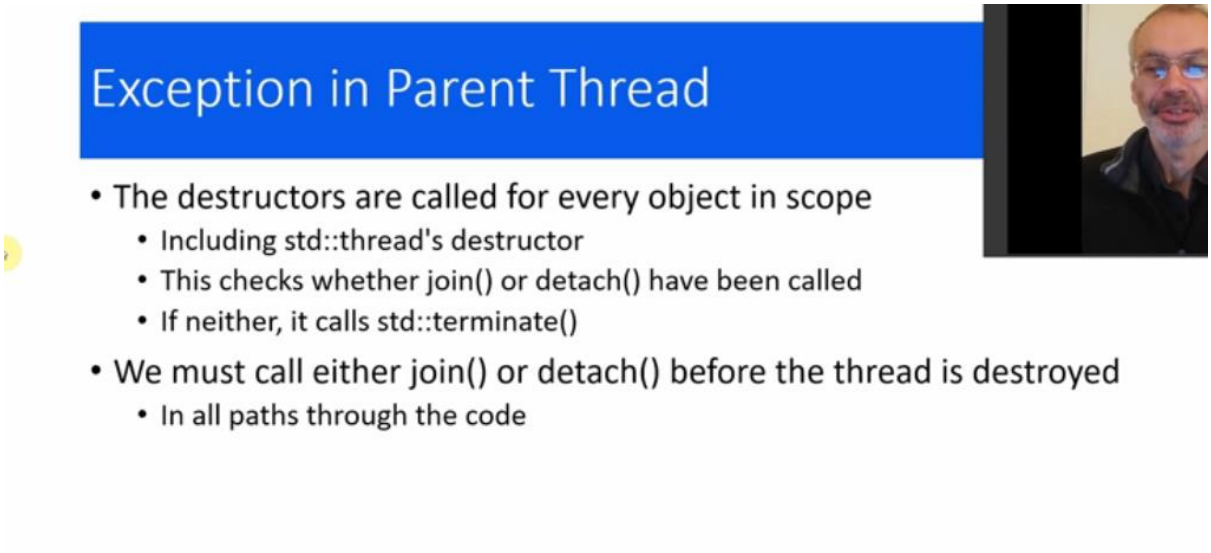
```cpp
    try {
        // Throw an exception
        throw std::exception();
    }
    catch (std::exception& e) {
        std::cout << "Exception caught: "
    }
    std::cout << "Hello, Thread!\n";
}

int main() {
    // Create an std::thread object
    std::thread thr(hello);
    thr.join();

    // Check that the program is still ru
    std::cout << "Finished\n";
}
```

```
Exception caught: Unknown exception
Hello, Thread!
Finished
Press any key to continue . . .
```

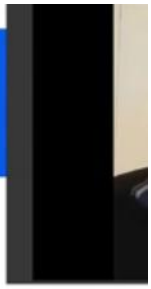
Detaching a Thread

- Instead of calling join(), we can call detach()
  - The parent thread will continue executing
  - The child thread will run until it completes
  - Or the program terminates
  - Analogous to a "daemon" process
- When an execution thread is detached
  - The std::thread object is no longer associated with it
  - The destructor will not call std::terminate()


join() vs detach()
join()
detach()
parent thread
child thread
parent thread
child thread
1:10 / 6:57
Copyright © James Raynard 2023


Exception in Parent Thread
• The destructors are called for every object in scope
• Including std::thread's destructor
• This checks whether join() or detach() have been called
• If neither, it calls std::terminate()
• We must call either join() or detach() before the thread is destroyed
• In all paths through the code

Try/catch Solution



- The obvious solution is to add a try/catch block

```
{
  std::thread thr(func);
  try {
    ...                          // Code which might throw an exception
    thr.join();                  // No exception if we got here - call join() as usual
  }
  catch (std::exception& e) {
    thr.join();                  // Call join() before thr's destructor is called
  }
}                                // Calls ~thr()
```

- This is verbose and not very elegant

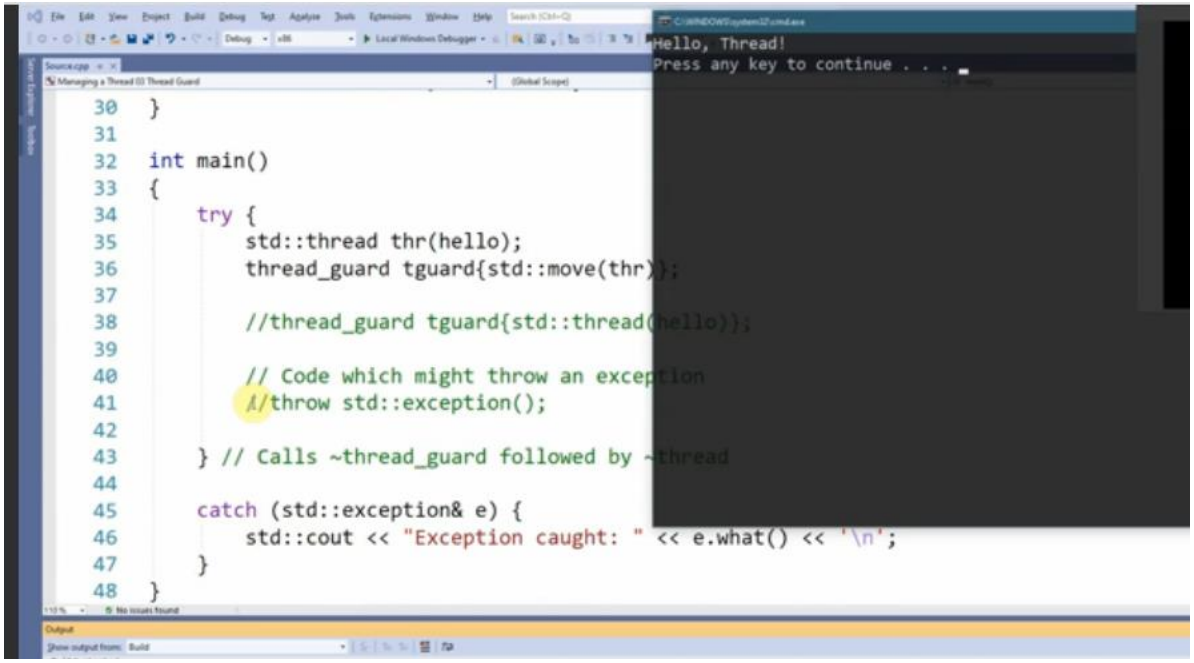
RAII Solution

- A better solution is to use the RAII idiom
  - Wrap the std::thread object inside a class
  - The class's destructor calls join() on the std::thread object
- An std::thread object can only be joined once
- The joinable() member function
  - Returns false
  - If join() or detach() have already been called
  - Or if the thread object is not associated with an execution thread
- Returns true if we need to call join()

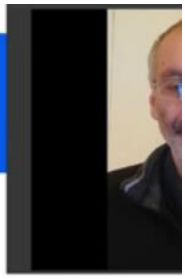
```cpp
// Uses a wrapper class for std::thread
// Ensures safe destruction when an exception is throw
#include <thread>
#include <iostream>

class thread_guard {
    std::thread thr;
public:
    // Constructor takes rvalue reference argument (std::thread is move-only)
    explicit thread_guard(std::thread&& thr): thr(std::move(thr)) {
    }

    // Destructor - join the thread if necessary
    ~thread_guard()
    {
        if (thr.joinable())
            thr.join();
    }

```

```cpp
    ~thread_guard()
    {
        if (thr.joinable())
            thr.join();
    }

    thread_guard(const thread_guard&) = delete;       // Deleted copy operators prevent copying
    thread_guard& operator=(const thread_guard&) = delete;

    // The move assignment operator is not synthesized
};
```

```cpp

// Callable object - thread entry point
void hello()
{
    std::cout << "Hello, Thread!\n";
}

int main()
{
    try {
        std::thread thr(hello);
        thread_guard tguard{std::move(thr)};

        //thread_guard tguard{std::thread(hello)};

        // Code which might throw an exception
        throw std::exception();

    } // Calls ~thread_guard followed by ~thread
```

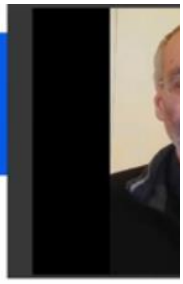


Code using RAII Class



- The destructors are called in reverse order
  - The thread_guard's destructor is called first
  - If necessary, it calls thr.join() and waits for the execution thread to finish
  - The thread_guard's std::thread member is then destroyed
  - It is not associated with an execution thread
  - Its destructor does not call std::terminate()
- This applies in normal execution
  - And when an exception is thrown

Stopping Threads



- Execution threads can be interrupted or stopped
  - Killed, cancelled, terminated
- In general, abruptly terminating a thread is not a good idea
- std::thread does not support this
  - The Williams book has an interruptible_thread class
  - The operating system can be used to stop the underlying execution thread

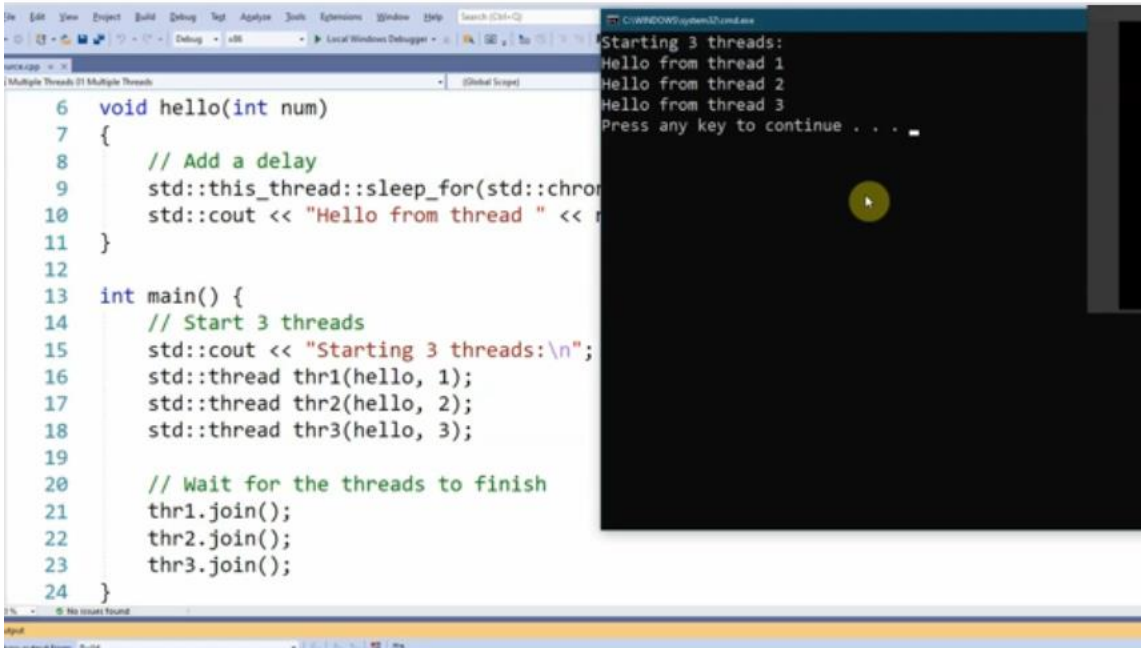
C++20 and std::jthread

- Destructor does not call std::terminate()
  - Calls join() if necessary
- Supports cooperative interruption

Starting multiple threads

- We can start multiple threads

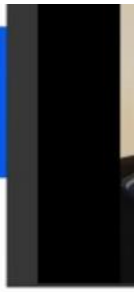
```
// Start 3 threads
std::thread thr1(hello);
std::thread thr2(hello);
std::thread thr3(hello);

// Wait for them to finish
thr1.join();
thr2.join();
thr3.join();
```



Data Sharing Between Threads

- The threads in a program share the same memory space
  - It is very easy to share data between the threads
- The only requirement is that the data is visible to the thread functions
  - Global or static variable, for global thread functions
  - Static class member, for class member thread functions
  - Local variable captured by lambda expressions (by reference)

Data Sharing Between Threads



- Threads interleave their execution
- Threads can interfere with each other's actions
- Modifying shared data can cause data corruption
  - This is the main source of bugs in concurrent programs

Data Race

- A "data race" occurs when:
  - Two or more threads access the same memory location
  - And at least one of the threads modifies it
  - Potentially conflicting accesses to the same memory location
- Only safe if the threads are synchronized
  - One thread accesses the memory location at a time
  - The other threads have to wait until it is safe for them to access it
  - In effect, the threads execute sequentially while they access it
- A data race causes undefined behaviour
  - The program is not guaranteed to behave consistently

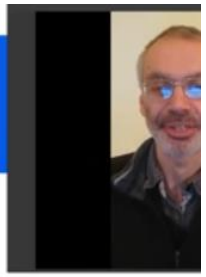
Race Condition

- The outcome is affected by timing changes
  - e.g. One client clears a database table
  - Another client inserts an entry into the same table
- A data race is a special case of a race condition
  - The outcome depends on when threads are scheduled to run

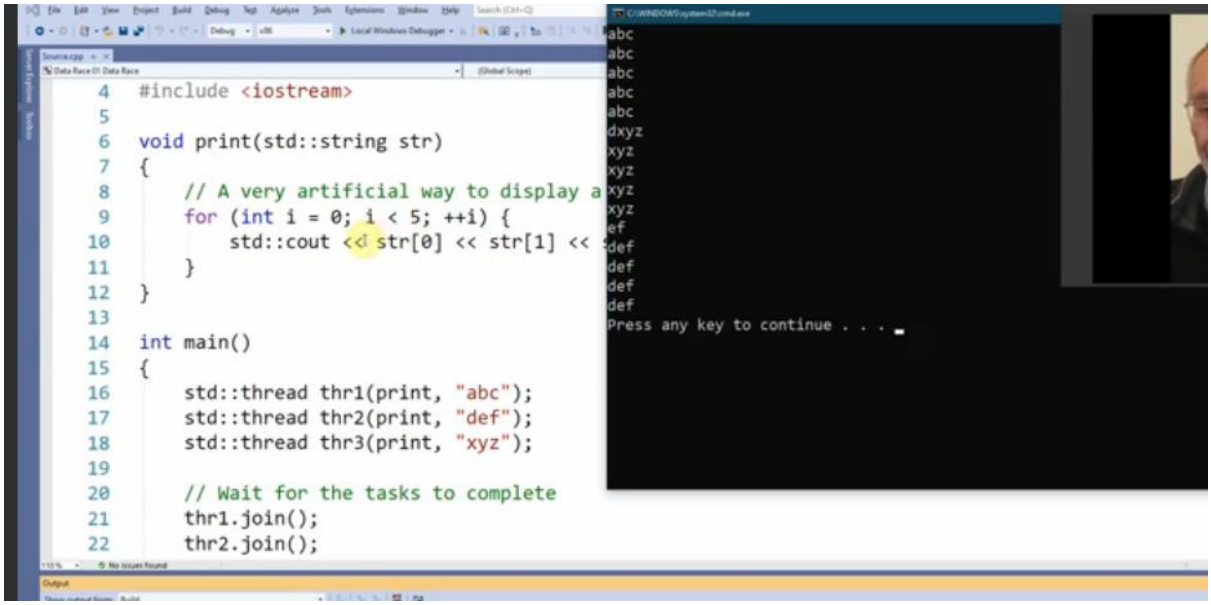
Memory Location

- In C++, a "memory location" is a scalar object:
  - A built-in variable
  - A pointer
  - An element in a container
- Or a scalar sub-object:
  - A struct or class member which is a scalar object
- Also an obscure case:
  - A series of contiguous bitfields within the same word
  - Unless they contain a zero-length bitfield (!)

Compound Objects



- C++ STL containers are memory locations
  - Multiple threads modifying the same object may conflict
  - Should be synchronized
- For our own types, we can choose the behaviour
  - Classes can provide their own synchronization
  - Easier to work with
  - Calling a sequence of member functions may be problematic
  - Usually better to implement them as memory locations



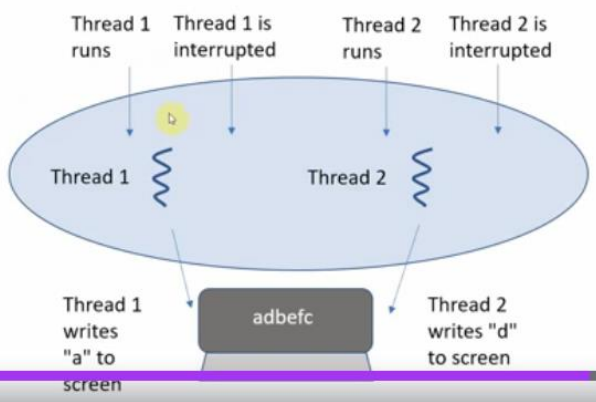
Output

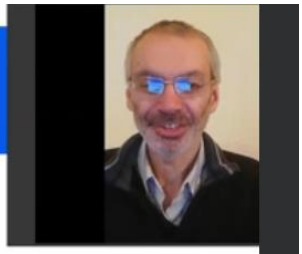
- The output will be scrambled up, like this

```
abc
def
xyz
adbefc
xyz
xyz
adbefc
```

- What has gone wrong?
  - Threads can be interrupted during output
  - Other threads can run and write their output
  - The output from different threads is interleaved

Write with Data Race





Data Race Consequences

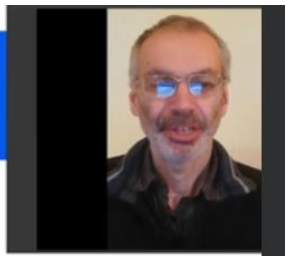
- In this program, the data race caused interleaved output
  - std::cout is a special case
  - Nothing worse than output interleaving can happen
- The consequences of a data race can be much more serious
  - Incorrect results
  - Incorrect program flow
  - "Torn" writes and reads
  - Objects which are improperly constructed or destroyed

Incorrect Results

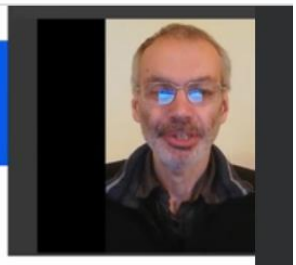
```
int x = 5;              // Shared variable x

int y = 2*x + 3;        // Calculate the value of y (1)
int z = 3*x + 2;        // Calculate the value of  z (2)
```

- Thread A evaluates y
  - Thread B interleaves and changes the value of x to 6
  - Thread A uses the new value of x to evaluate z
  - The calculated values of y and z are inconsistent
- This can even occur inside a statement

```
y = x*x + 2*x + 1;      // Another thread could change the value of x
```



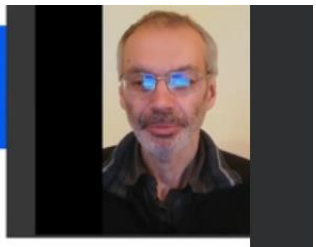
Incorrect Program Flow



```
int x = 3;

if (x > 0) {              // Test whether it is safe to use x (1)
  z = sqrt(x);            // Use x (2)
  ...
}
```

- Thread A checks that x > 0
  - Thread B interleaves and changes the value of x to a negative number
  - Thread A uses the new, negative value of x
  - A runtime error occurs which "shouldn't be possible"

Incorrect Program Flow

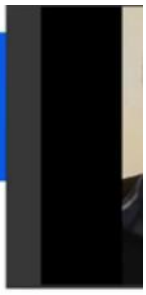


```
if (x > 0 && x < 10) {          // Check x is within bounds (1)
  switch(x) {                   // Use x (2)
    case 1:
      func1();
      break;

    case 2:
      func2();
      break;
    ...
    case 9:
      func9();
      break;
  }
}
```

```
table[1] = func1;
table[2] = func2;
...
```

```
table[1]();
```

Incorrect Program Flow



- The compiler may optimize this as a branch table
  - An array of function pointers
  - table[1] holds the address of func1
  - ...
  - table[9] holds the address of func9
  - When x == 1, the program executes the code in table[1]
- Thread A checks that x is within bounds
- Thread B interleaves and sets x to 10

  Thread A executes the code at table[10]
  - Arbitrary code!