Mutex Conclusion

Mutex Conclusion

- We have covered
  - Basic mutex class (std::mutex)
  - Mutex wrappers (std::lock_guard and std::unique_lock)
  - Mutexes and time-outs (std::timed_mutex, std::unique_lock)
  - Shared mutexes (std::shared_mutex and std::shared_lock)
  - Static and thread-local data
  - Deadlock
  - Deadlock avoidance (std::scoped_lock, std::lock())
  - Livelock and livelock avoidance

Locking Guidelines

- Locking impacts on other threads
  - They will have to wait longer for a resource they need
  - This affects performance
- Always hold a lock for the shortest possible time
- Avoid locking lengthy operations if possible
  - e.g. input/output

Recommendations for Reading Shared Data

- Reading
  - Lock
  - Make a copy of the shared data
  - Unlock and process the copy

Locking Guidelines for Data Structures

- Do not lock any more elements than necessary
  - e.g. Accessing a single element in a linked list
  - Do not lock the entire list
  - This will block other threads from accessing unrelated elements
- Do not make locking too fine-grained
  - Do not lock individual elements when inserting and deleting
  - Another thread may need to access a neighbouring element
  - Data race
  - Need to lock neighbouring elements in a single operation

Pros and Cons of Mutexes

- Fairly straightforward way to protect shared data
  - Prevent data races and race conditions
- Locking and unlocking are slow operations
- Low-level
  - The programmer must remember to use a mutex
  - The programmer must use the right mutex
  - The programmer must understand how different threads can modify the data
- Most real-world programs use higher-level structures
  - Mutex wrapper classes
  - Classes from the next section!

Thread Coordination

Coordination Between Workers

- Problem:
  - Worker A is writing a document
  - Worker A needs an image that Worker B is producing
  - Worker A cannot continue until Worker B has finished

Coordination Between Workers

- One solution is to introduce a manager
  - The manager coordinates the two workers

Coordination Between Workers

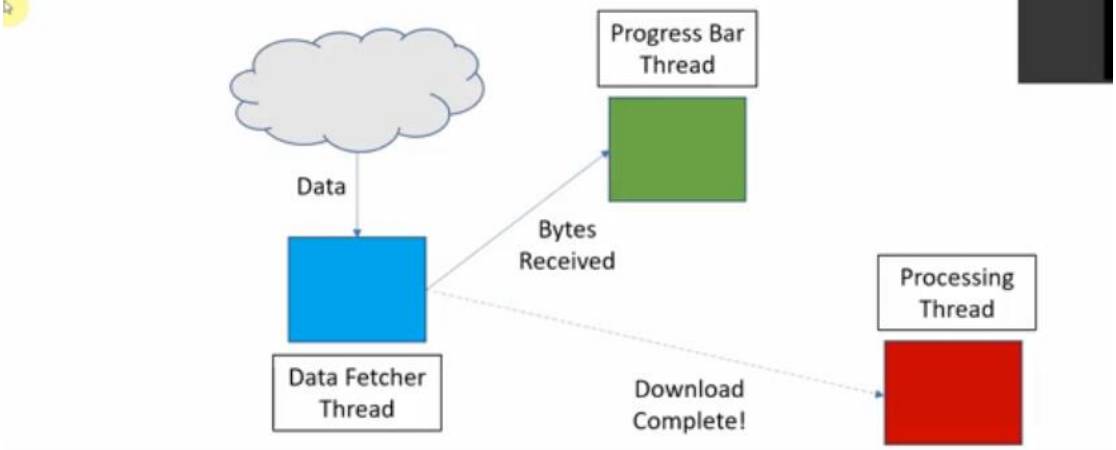
- Solution:
- Worker B tells the manager when they have finished
- The manager tells Worker A to resume work
  - B is working
  - A is waiting
  - B finishes their work
  - B tells manager
  - Manager tells A to resume
  - A resumes work

Coordination Between Threads

- Similar situations arise with threads
- e.g. A program which is performing a download
  - One thread fetches the data over the network
  - Another thread displays a progress bar
  - A third thread will process the data when the download is complete

Thread Coordination Practical

Program Performing a Download



Communication Between Threads

- The threads run concurrently
  - The data fetching thread runs continually
  - The progress bar thread waits for information
  - The processor thread waits until all the data has been received
- When the download is complete
  - The fetching thread terminates
  - The progress bar thread terminates
  - The processor thread runs

Data Sharing Between Threads

- The downloaded data is shared by all three threads
  - The data fetching thread appends to it
  - The progress bar thread calculates its size
  - The processor thread uses the data
- Potential Data Race
  - Multiple threads
  - Modification

Coordination of Threads

- We will use two bools to coordinate the threads
- "progress" flag
  - The fetching thread sets this when it has new data
  - The progress bar thread checks this flag
- "completed" flag
  - The fetching thread sets this when it finishes
  - The other two threads check this flag
- Potential Data Race
  - Multiple threads
  - Modification
- Use mutexes

Hot Loop

- We need to lock the mutex while checking a bool

```
// In progress bar task function
std::lock_guard data_lck(data_mutex);
while (!update_progress) {}
```

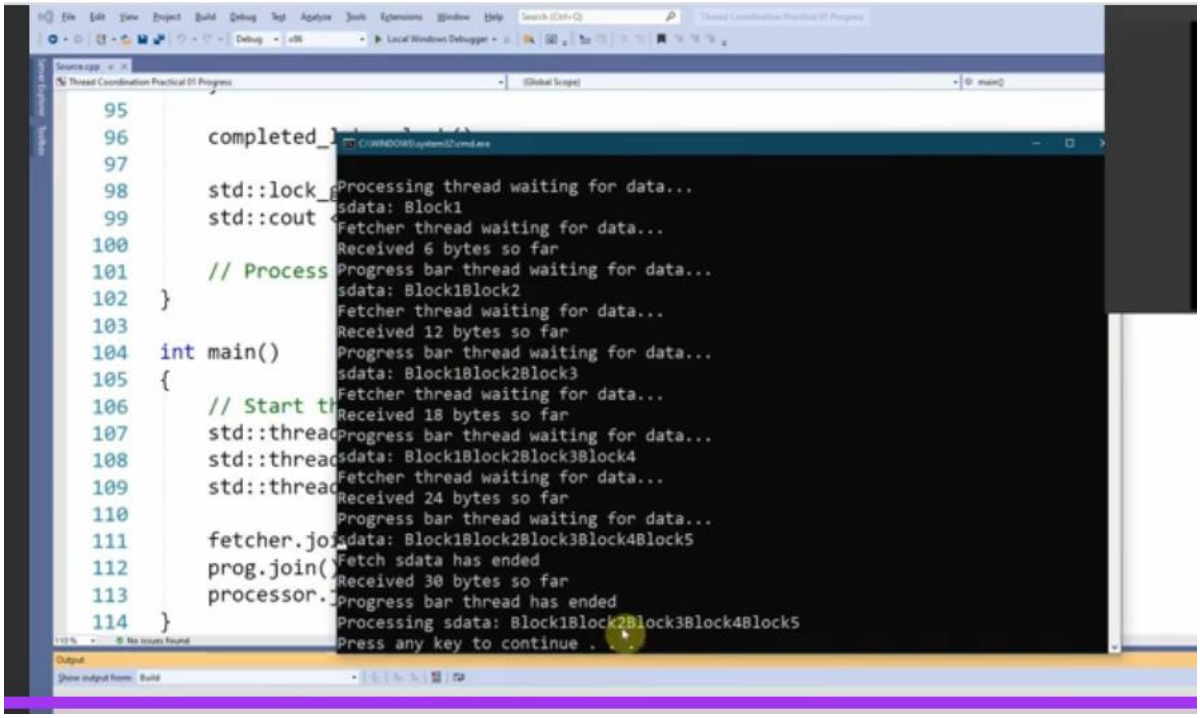
- The thread will run flat out
  - The processor core will run at 100%
  - Other threads cannot do useful work
  - Uses a lot of electricity
- The fetcher thread cannot set the flag

Hot Loop Avoidance

- To avoid this, unlock the mutex inside the loop

```
std::unique_lock<std::mutex> data_lck(data_mutex);

while (!update_progress) {
  data_lck.unlock();
  std::this_thread::sleep_for(10ms);
  data_lck.lock();
}
```

  - Sleeping allows other threads to use the core
  - The fetcher thread can set the flag



Implementation with Mutex

- This is not ideal
  - Too many loops
  - Too much explicit locking and unlocking
  - How do we choose the sleep duration?
- Better solution
  - Thread A indicates that it is waiting for something
  - Thread B does the "something"
  - Thread A is woken up and resumes

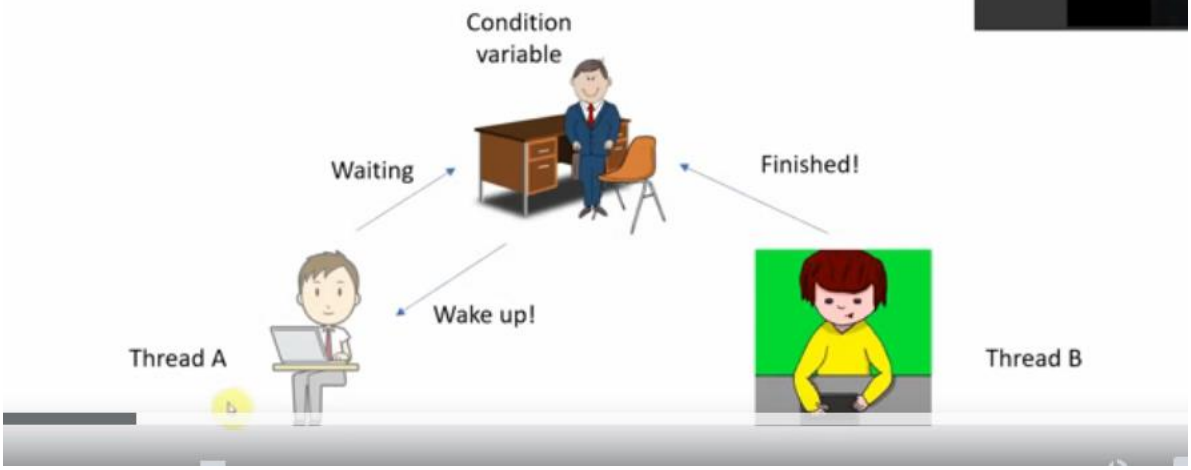
Condition Variables

Condition Variable Motivation

- Suppose we have two threads
  - "Writer" thread modifies a shared string
  - "Reader" thread uses the modified string
- The two threads need to be coordinated
- We must also avoid a data race

Coordination Between Threads

- One solution is to introduce a condition variable



Condition Variable Overview

- Scenario
  - Thread A tells the condition variable it is waiting
  - Thread B notifies the condition variable when it updates the string
  - The condition variable wakes thread A up
  - Thread A then uses the string

Condition Variable and Thread Safety

- We use a mutex to protect critical sections
- The condition variable also uses the same mutex
  - Thread coordination
  - No data race

std::condition_variable

- Defined in <condition_variable>
- wait()
  - Takes an argument of type std::unique_lock
  - It unlocks its argument and blocks the thread until a notification is received
- wait_for() and wait_until()
  - Re-lock their argument if a notification is not received in time
- notify_one()
  - Wake up one of the waiting threads
  - The scheduler decides which thread is woken up
- notify_all()
  - Wake up all the waiting threads

Condition Variable Scenario

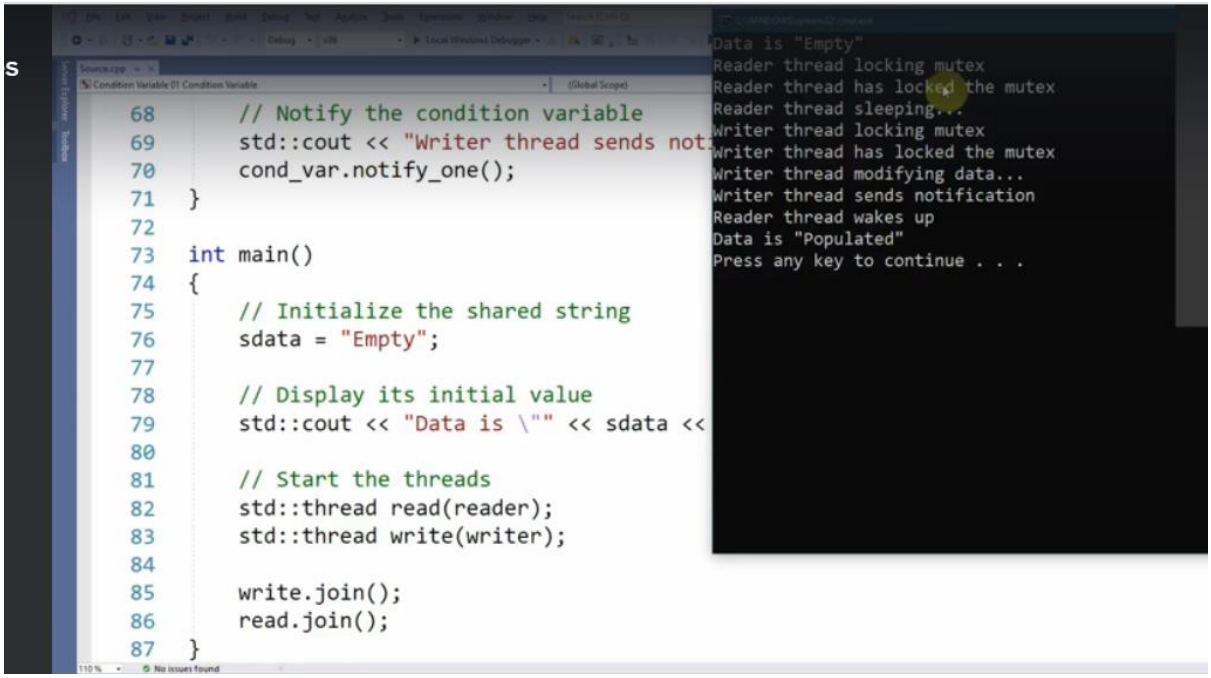
- Thread A locks the mutex
  - It calls the condition variable's wait() member function
  - The condition variable unlocks the mutex
  - The condition variable blocks this thread
- Thread B locks the mutex
  - It modifies the string and unlocks the mutex
  - It calls notify_one()
- The condition variable wakes thread A up
  - The wait() call returns with the mutex locked
  - Thread A resumes execution and uses the string

Reader thread

```
// Waiting thread
void reader()
{
   // Lock the mutex
   std::unique_lock<std::mutex> uniq_lck(mut);

   // Call wait() on the condition variable
   // Unlocks the mutex and makes this thread sleep
   cond_var.wait(uniq_lck);

   // The condition variable wakes this thread up and locks the mutex

   // Use the shared data
   // ...
}
```

Writer thread

```
// Notifying thread
void writer()
{
  {
    // Lock the mutex
    std::lock_guard<std::mutex> lck_guard(mut);

    // Modify the shared data
    sdata = "Populated";
  } // Release the lock

  // Notify the condition variable
  cond_var.notify_one();
}
```



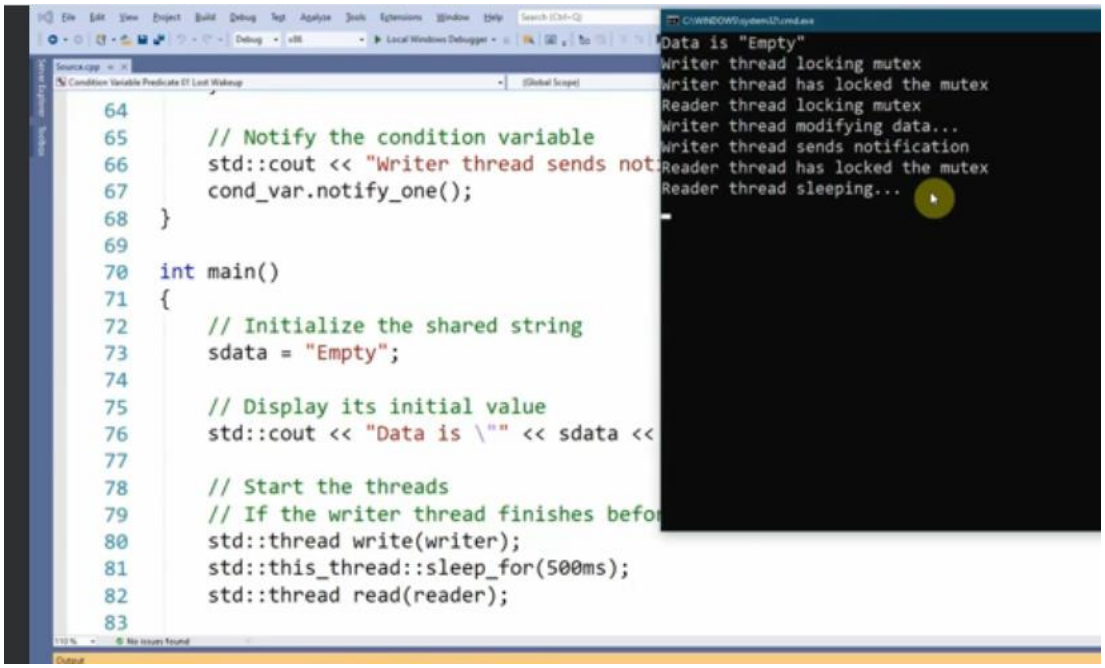
std::condition_variable_any

- std::condition_variable only works with std::mutex
  - Does not work with std::timed_mutex
- There is also std::condition_variable_any
  - Works with any mutex-like object
  - Including our own types
  - May have more overhead than std::condition_variable

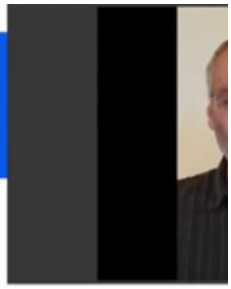
Condition Variables with Predicate

Lost Wakeup

- The example in the previous lecture has a problem
- wait() will block until the condition variable is notified
- If the writer calls notify() before the reader calls wait()
  - The condition variable is notified when there are no threads waiting
  - The reader will never be woken up
  - The reader could be blocked forever
- This is known as a "lost wakeup"



Spurious Wakeup



- Occasionally, the reader will be "spuriously" woken up
  - The reader thread has called wait()
  - The writing thread has not called notify()
  - The condition variable wakes the reader up anyway
- This is due to the way that std::condition_variable is implemented
  - Avoiding spurious wakeups adds too much overhead
- Fortunately, there is a way to solve both spurious and lost wakeups

wait() with Predicate

- wait() takes an optional second argument
  - A predicate
- Typically, the predicate checks a shared bool
  - The bool is initialized to false
  - It is set to true when the writer sends the notification
- The reader thread will call this predicate
- It will only call wait() if the predicate returns false
  - Also available with wait_for() and wait_until()

Using wait() with Predicate

- Add a shared boolean flag, initialized to false
- In the wait() call, provide a callable object that checks the flag

```
// bool flag for predicate
bool condition = false;

// Waiting thread
void reader()
{
   // Lock the mutex
   std::unique_lock<std::mutex> uniq_lck(mut);

   // Lambda predicate that checks the flag
   cond_var.wait(uniq_lck, [] {return condition;});

   // ...
}
```

Using wait() with Predicate

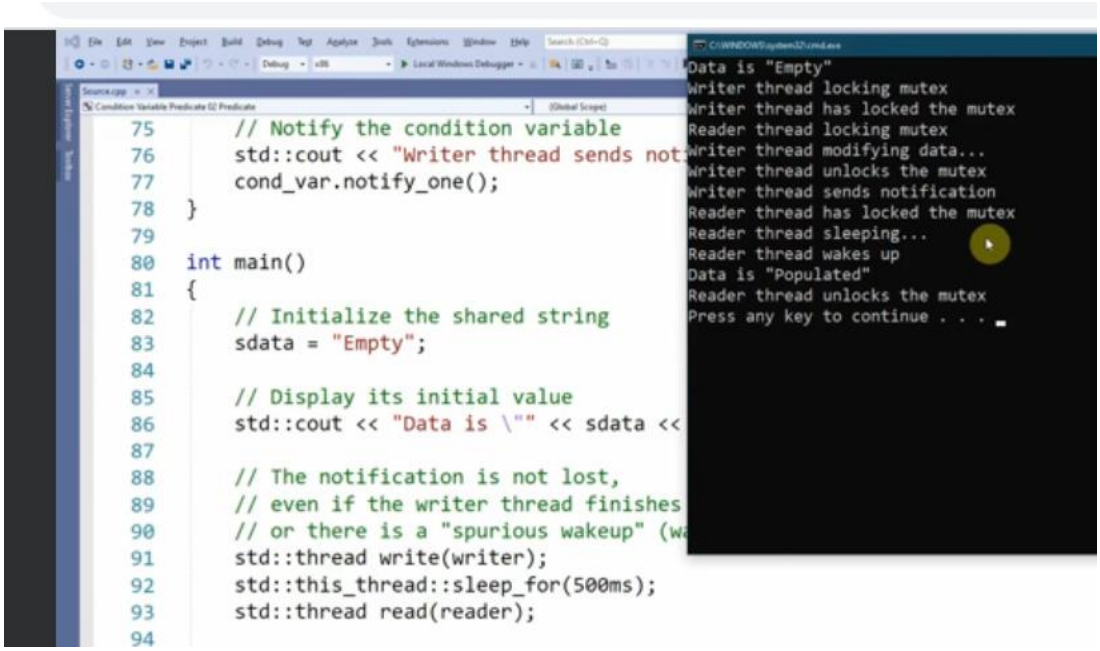
- Add a shared boolean flag, initialized to false
- In the wait() call, provide a callable object that checks the flag

```
// bool flag for predicate
bool condition = false;

// Waiting thread
void reader()
{
  // Lock the mutex
  std::unique_lock<std::mutex> uniq_lck(mut);

  while (!condition)
    cond_var.wait();

  // ...
}
```

Using wait() with Predicate

- In the writer thread, set the flag to true

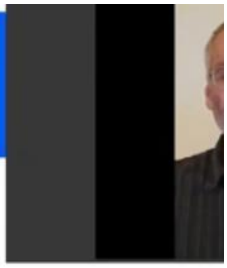
```
{
  std::lock_guard<std::mutex> lck_guard(mut);
  sdata = "Populated";

  // Set the flag
  condition = true;
}

// Notify the condition variable
cv.notify_one();
}
```



Lost Wakeup Avoidance

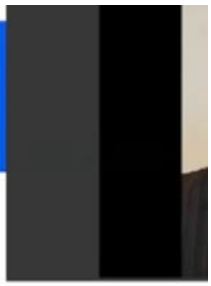
- The writer notifies the condition variable
- The reader thread locks the mutex
- The reader thread calls the predicate
- If the predicate returns true
  - Lost wakeup scenario - the writer has already sent a notification
  - The reader thread continues, with the mutex locked
- If the predicate returns false
  - Normal scenario
  - The reader thread calls wait() again

Spurious Wakeup Avoidance

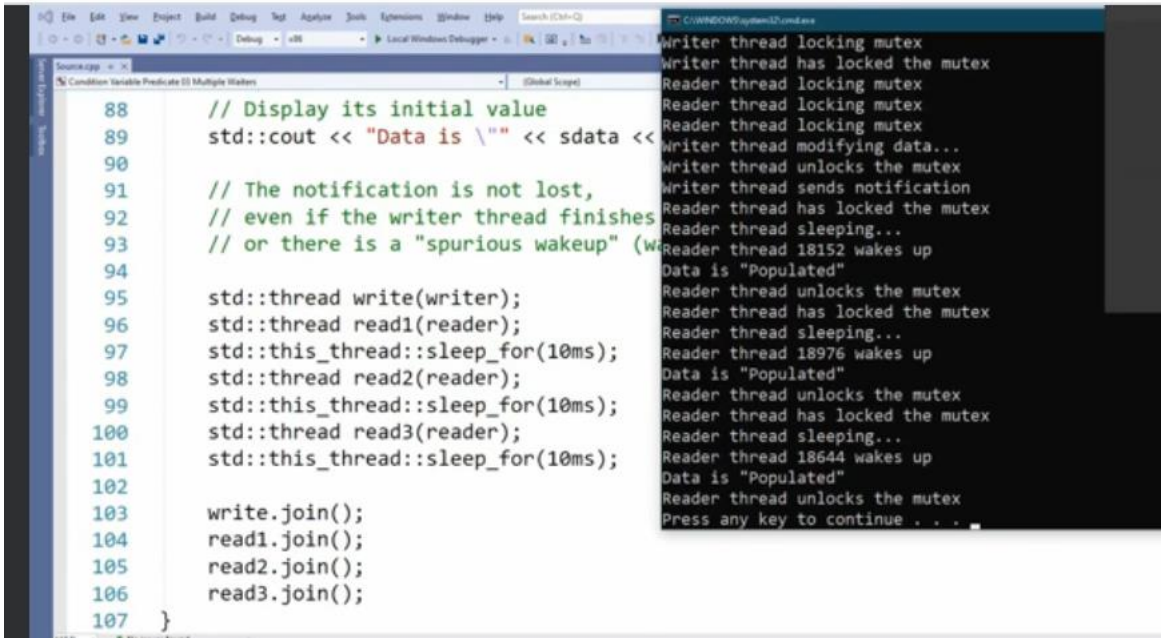


- The writer notifies the condition variable (or not)
- The reader thread locks the mutex
- The reader thread calls the predicate
- If the predicate returns true
  - Genuine wakeup - the writing thread really has sent a notification
  - The reader thread continues, with the mutex locked
- If the predicate returns false
  - Spurious wakeup scenario - the writing thread has not sent a notification
  - The reader thread calls wait() again

Multiple Threads

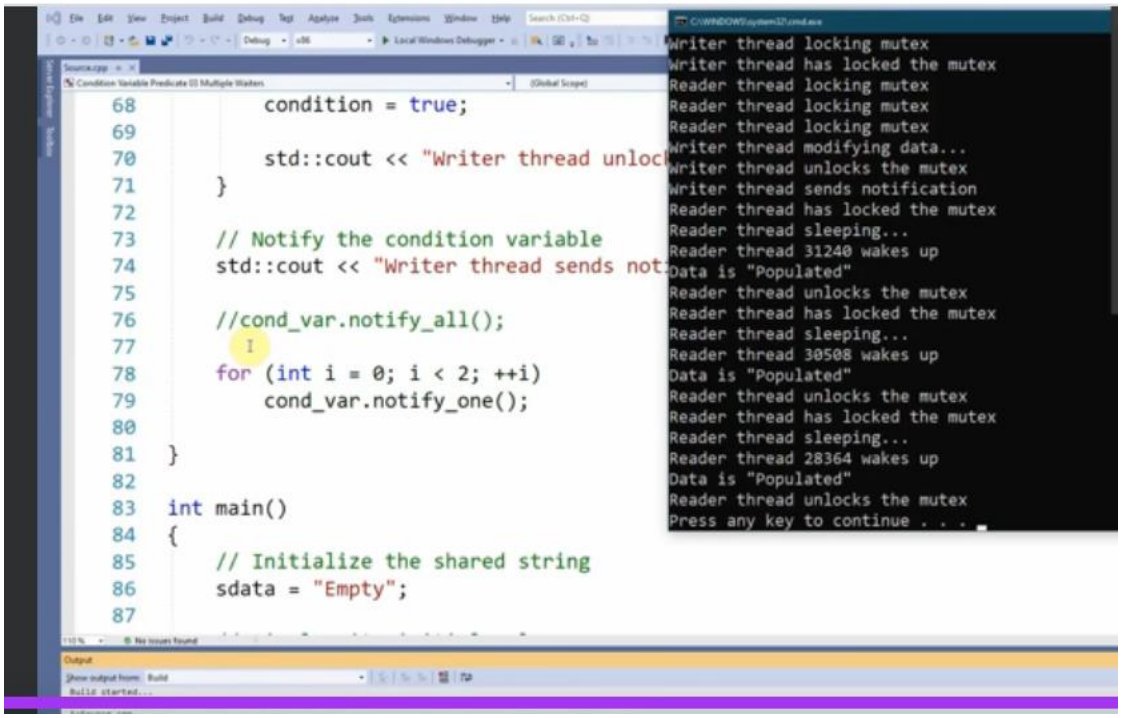


- Condition variables are particularly useful here
  - Multiple threads are waiting for the same event
- notify_all()
  - The condition variable wakes up all the threads which have called wait()
  - The threads could wake up in any order
  - All the reader threads process the data
- notify_one()
  - Only one of the threads which called wait() will be woken up
  - The other waiting threads will remain blocked
  - A different reader thread processes the data each time



```
    // Display its initial value
    std::cout << "Data is \"" << sdata <<

    // The notification is not lost,
    // even if the writer thread finishes
    // or there is a "spurious wakeup" (w

    std::thread write(writer);
    std::thread read1(reader);
    std::this_thread::sleep_for(10ms);
    std::thread read2(reader);
    std::this_thread::sleep_for(10ms);
    std::thread read3(reader);
    std::this_thread::sleep_for(10ms);

    write.join();
    read1.join();
    read2.join();
    read3.join();
}
```

```
Writer thread locking mutex
Writer thread has locked the mutex
Reader thread locking mutex
Reader thread locking mutex
Reader thread locking mutex
Writer thread modifying data...
Writer thread unlocks the mutex
Writer thread sends notification
Reader thread has locked the mutex
Reader thread sleeping...
Reader thread 18152 wakes up
Data is "Populated"
Reader thread unlocks the mutex
Reader thread has locked the mutex
Reader thread sleeping...
Reader thread 18976 wakes up
Data is "Populated"
Reader thread unlocks the mutex
Reader thread has locked the mutex
Reader thread sleeping...
Reader thread 18644 wakes up
Data is "Populated"
Reader thread unlocks the mutex
Press any key to continue . . .
```

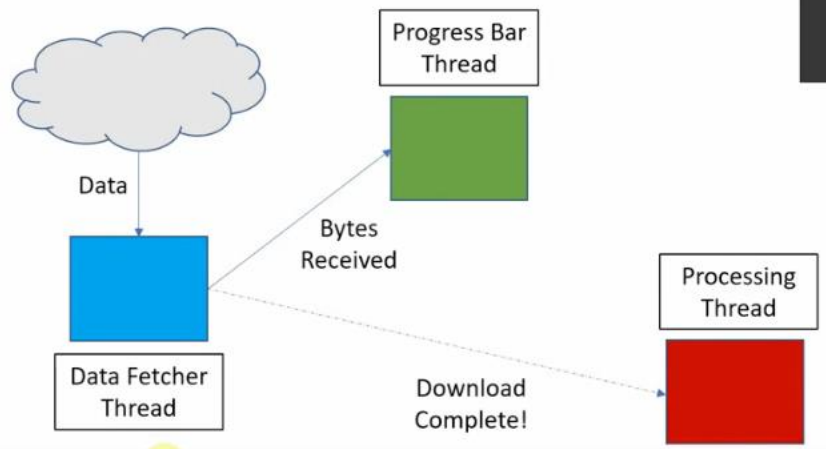


```
        condition = true;

        std::cout << "Writer thread unloc

    }

    // Notify the condition variable
    std::cout << "Writer thread sends not

    //cond_var.notify_all();

    for (int i = 0; i < 2; ++i)
        cond_var.notify_one();

}

int main()
{
    // Initialize the shared string
    sdata = "Empty";
```

```
Writer thread locking mutex
Writer thread has locked the mutex
Reader thread locking mutex
Reader thread locking mutex
Reader thread locking mutex
Writer thread modifying data...
Writer thread unlocks the mutex
Writer thread sends notification
Reader thread has locked the mutex
Reader thread sleeping...
Reader thread 31240 wakes up
Data is "Populated"
Reader thread unlocks the mutex
Reader thread has locked the mutex
Reader thread sleeping...
Reader thread 30508 wakes up
Data is "Populated"
Reader thread unlocks the mutex
Reader thread has locked the mutex
Reader thread sleeping...
Reader thread 28364 wakes up
Data is "Populated"
Reader thread unlocks the mutex
Press any key to continue . . .
```

Condition Variable Practical

Program Performing a Download

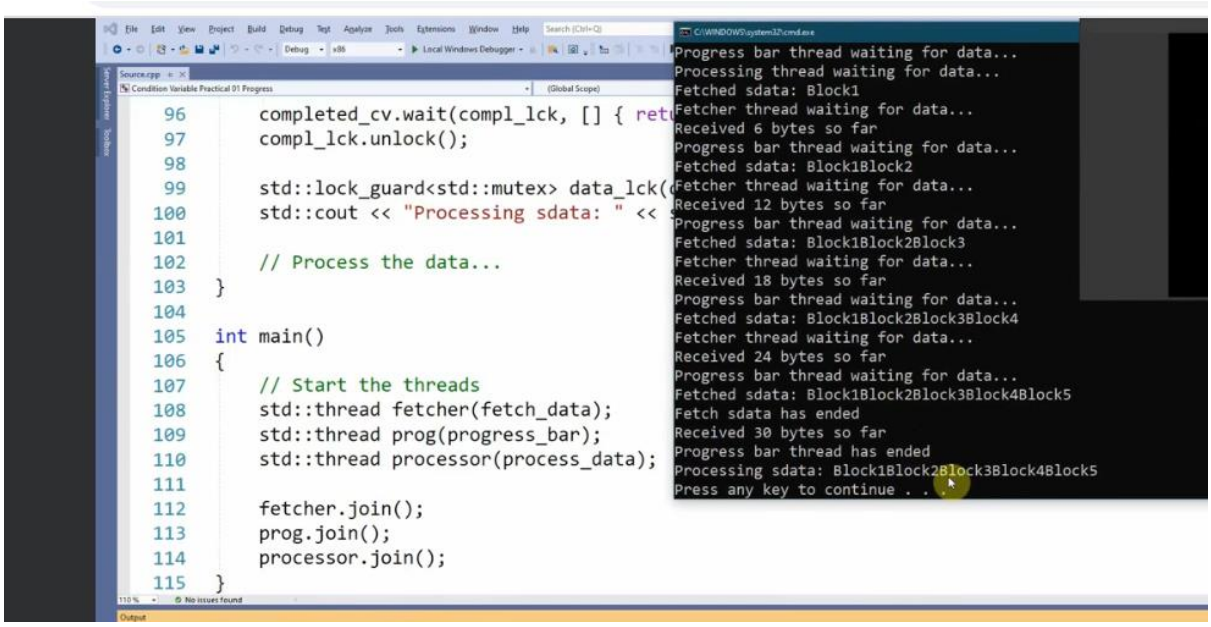


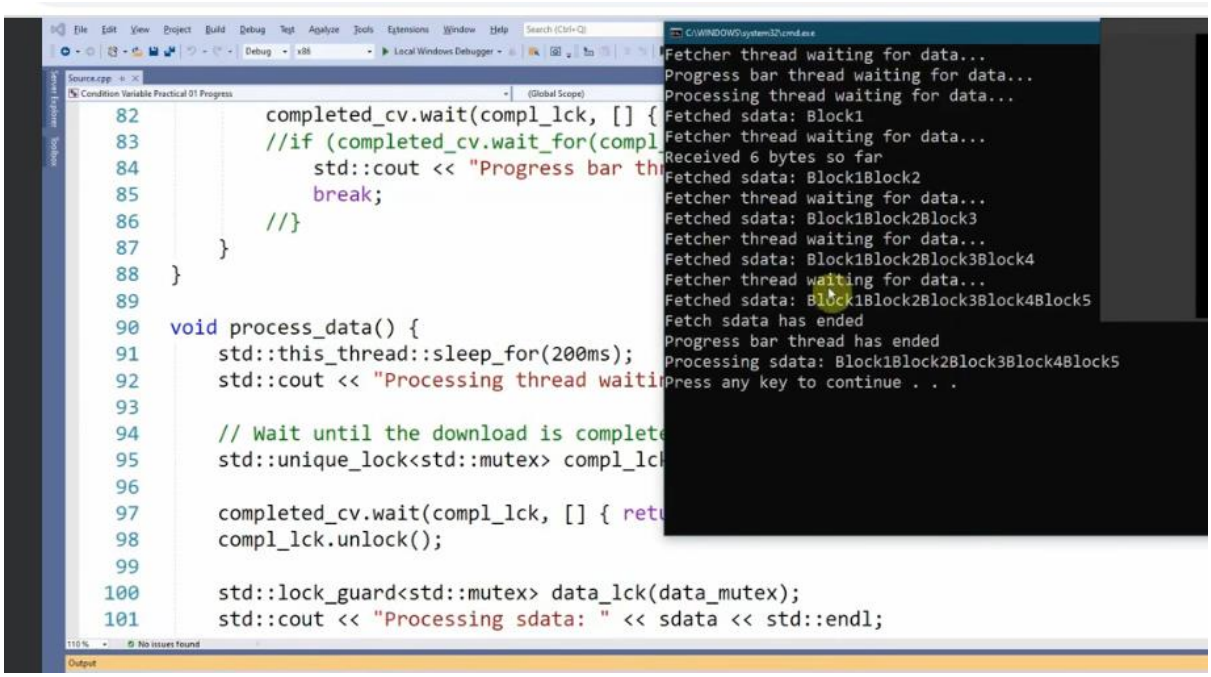
Coordination of Threads

- We will use two condition variables
- "data_cv"
  - The fetching thread notifies this when it has new data
  - The progress bar waits on it and updates itself
- "completed_cv"
  - The fetching thread notifies this when the download completes
  - The progress bar waits on it and exits
  - The processing thread waits on it and processes the data
- We use predicates with the condition variables
  - Avoid lost and spurious wake-ups

Progress Bar

- Implemented as a loop:
  - Wait on data_cv
  - Update progress
  - Wait on completed_cv
  - If the download is complete, exit
- Use blocking wait() on data_cv
- Use non-blocking wait_for() on completed_cv





Futures and Promises Overview

Transferring Data Between Threads

- std::thread does not provide a way to return a value
  - So far, we have used a shared variable
  - Access to the shared variable needs to be protected by locks
- Condition variables allow us to coordinate threads
  - A thread can signal to another thread that shared data has been modified
  - Cannot directly transfer data from one thread to another

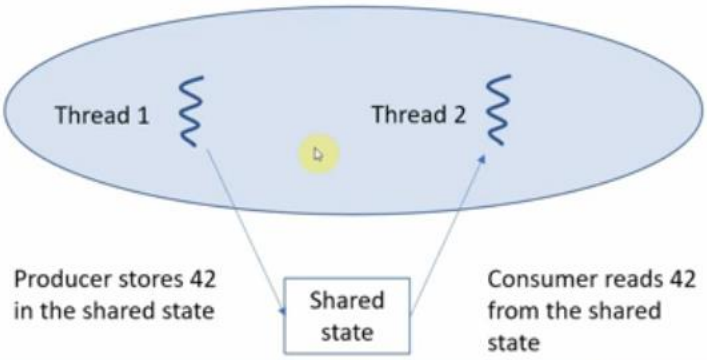
std::future and std::promise

- Classes for transferring data between threads
- Together, these set up a "shared state" between threads
- The shared state can transfer data from one thread to another
  - No shared data variables
  - No explicit locking

Producer-Consumer Model

- Futures and promises use a producer-consumer model
  - Reader/writer threads are an example of this model
- A "producer" thread will generate a result
- A "consumer" thread waits for the result
- The producer thread generates the result
- The producer thread stores the result in the shared state
- The consumer thread reads the result from the shared state

Producer-Consumer Transfer of Data



Transfer of Data using Future and Promise

- An std::promise is associated with the producer
- An std::future object is associated with the consumer
  - The consumer calls a member function of the future object
  - The function blocks until the result becomes available
- The producer thread sends the result
  - The promise object stores the result in the shared state
- The consumer thread receives the result
  - The member function reads the result from the shared state
  - The member function returns the result

Exception Handling

- Futures and promises also work with exceptions
  - The promise stores the exception in the shared state
- This exception will be rethrown in the consumer thread
  - By the future's blocking function
- The producer thread "throws" the exception to the consumer

std::future and std::promise Classes

Future and Promise

- An promise object is associated with a future object
- Together, they create a "shared state"
  - The promise object stores a result in the shared state
  - The future object gets the result from the shared state

std::future

- Represents a result that is not yet available
- One of the most important classes in C++ concurrency
  - Works with many different asynchronous objects and operations
  - Not just std::promise
- An std::future object is not usually created directly
  - Obtained from an std::promise object
  - Or returned by an asynchronous operation

std::future

- Template class defined in <future>
  - The parameter is the type of the data that will be returned
- get() member function
  - Obtains the result when ready
  - Blocks until the operation is complete
  - Fetches the result and returns it
- wait() and friends
  - Block but do not return a result
  - wait() blocks until the operation is complete
  - wait_for() and wait_until() block with a timeout

std::promise

- Template class defined in <future>
  - The parameter is the type of the result
- Constructor
  - Creates an associated std::future object
  - Sets up the shared state with it
- get_future() member function
  - Returns the associated future

```
std::promise<int> prom;
std::future fut  = prom.get_future();
```

std::promise Interface

- get_future() member function
  - Returns the std::future object associated with this promise
- set_value()
  - Sets the result to its argument
- set_exception()
  - Indicates that an exception has occurred
  - This can be stored in the shared state

Producer-Consumer Model

- Parent thread
  - Creates an std::promise object
- Producer task function
  - Takes the std::promise object as argument
  - Calls set_value()
    - Or set_exception()
- Consumer task function
  - Takes the associated std::future object as argument
  - Calls get()
    - Or wait() and friends

Futures and Promises Examples

Producer-Consumer Model

- Parent thread
  - Creates an std::promise object
- Producer task function
  - Takes the std::promise object as argument
  - Calls set_value()
    - Or set_exception()
- Consumer task function
  - Takes the associated std::future object as argument
  - Calls get()
    - Or wait() and friends

Producer Thread Example

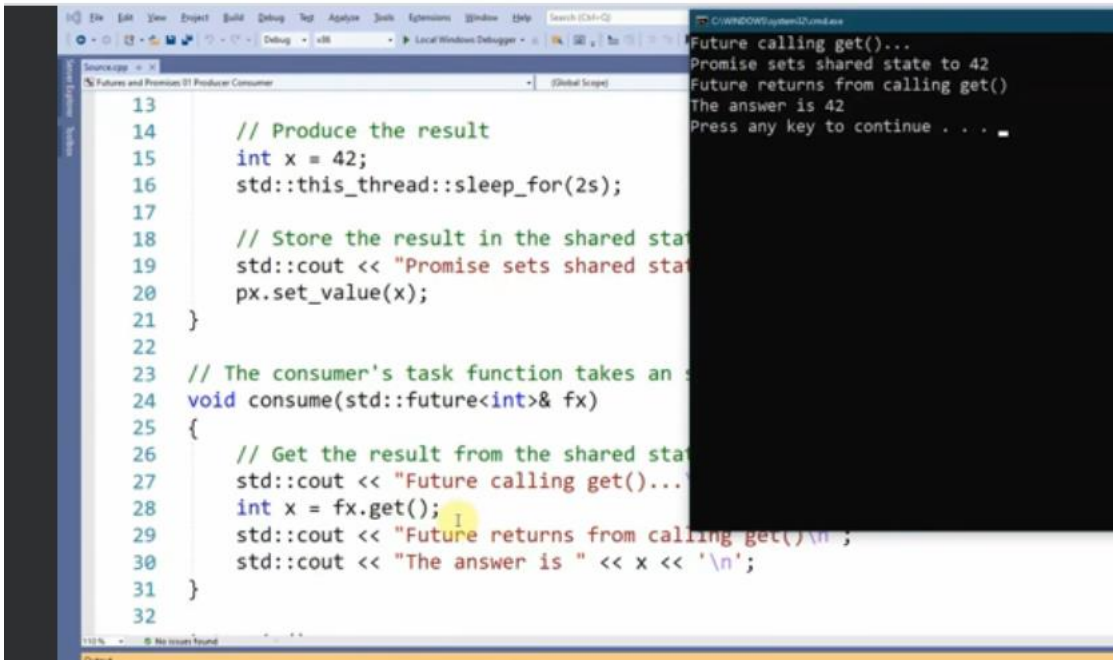
```
// The producer's task function takes an std::promise as argument
void produce(std::promise<int>& px)
{
   // Produce the result
   int x = 42;

   // Store the result in the shared state
   px.set_value(x);
}
```

Consumer Thread Example

```
// The consumer's task function takes an std::future as argument
void consume(std::future<int>& fx)
{
   // Get the result from the shared state
   int x = fx.get();
}
```

Parent Thread Example

```
// Create an std::promise object
std::promise<int> prom;

// Get the associated future
std::future<int> fut = prom.get_future();

// The producer task function takes the promise as argument
std::thread thr_producer(produce, std::ref(prom));

// The consumer task function takes the future as argument
std::thread thr_consumer(consume, std::ref(fut));
```



Producer-Consumer with Exception Handling

- In the producer thread
  - Put a try block around code that might throw
  - In the catch block, call set_exception() on the promise
  - This captures the active exception
- set_exception() takes a pointer to the exception object
  - We can use a catch-all handler
  - Pass the return value from std::current_exception()

Producer-Consumer with Exception Handling

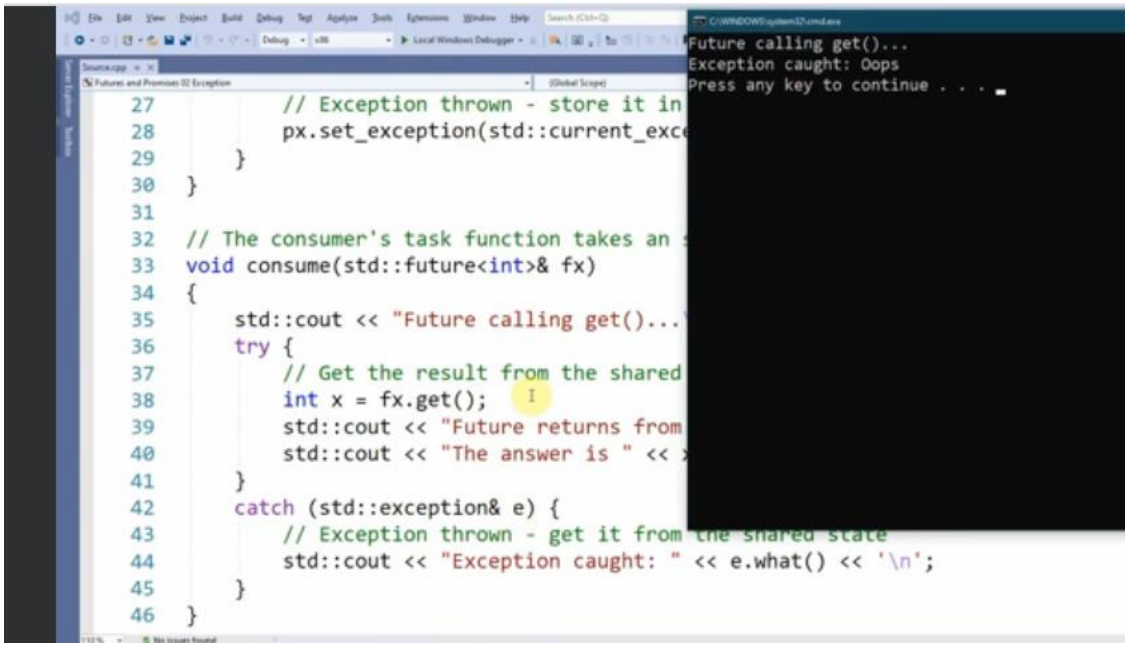
- In the consumer thread
  - Put a try block around the call to get() or wait()
  - Write a catch block to handle the exception

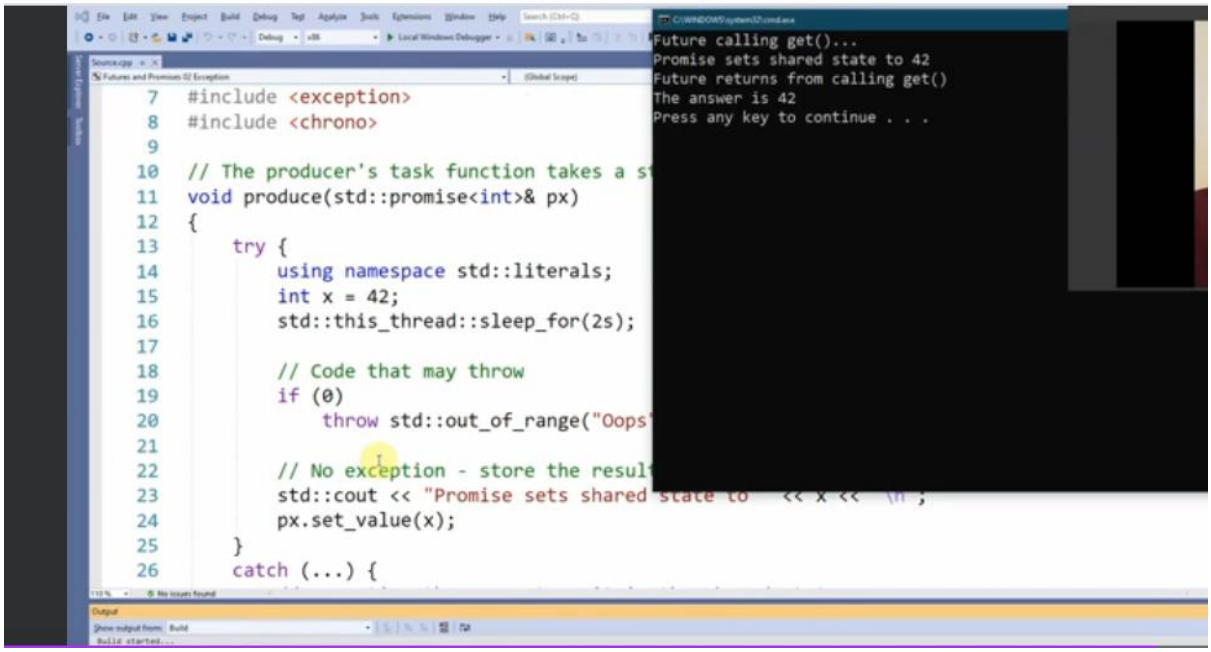
Producer with Exception Handling

```
void produce(std::promise<int>& px)
{
   try {
      // Code that may throw
      ...
      // If no exception, store the result in the shared state
      px.set_value(x);
   }
   catch (...) {
      // Exception caught - store it in the shared state
      px.set_exception(std::current_exception());
   }
}
```

Consumer with Exception Handling

```
void consume(std::future<int>& fx)
{
  try {
    // Get the result from the shared state - may throw
    int x = fx.get();
  }
  catch (...) {
    // Exception thrown - get it from the shared state
  }
}
```

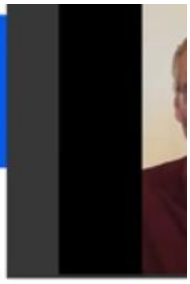




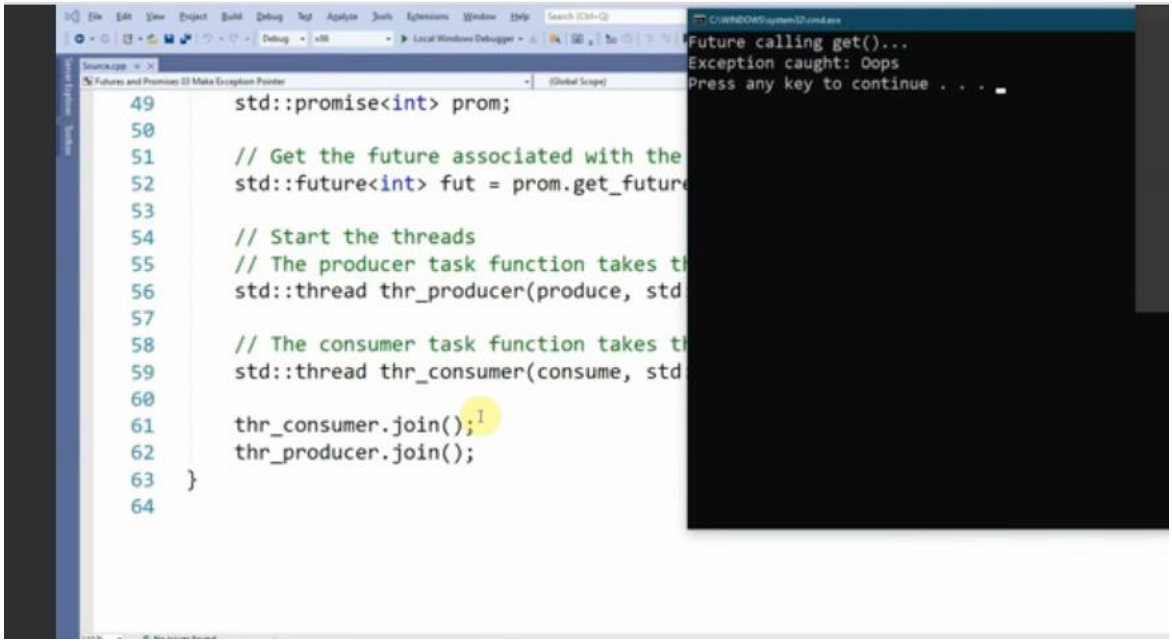
Producer with std::make_exception_ptr()

- To throw an exception ourselves, we could
  - Throw the exception inside a try block
  - Write a catch block that calls set_exception()
- C++11 has std::make_exception_ptr()
  - Takes the exception object we want to throw
  - Returns a pointer to its argument
  - Pass this pointer to set_exception()
  - Avoid "boilerplate" code
  - Better code generation

Producer with std::make_exception_ptr()



```
void produce(std::promise<int>& px)
{
  ...
  if (...) {
    px.set_exception(std::make_exception_ptr(std::out_of_range("Oops")));
    return;
  }

  // Store the result in the shared state
  px.set_value(x);
}
```



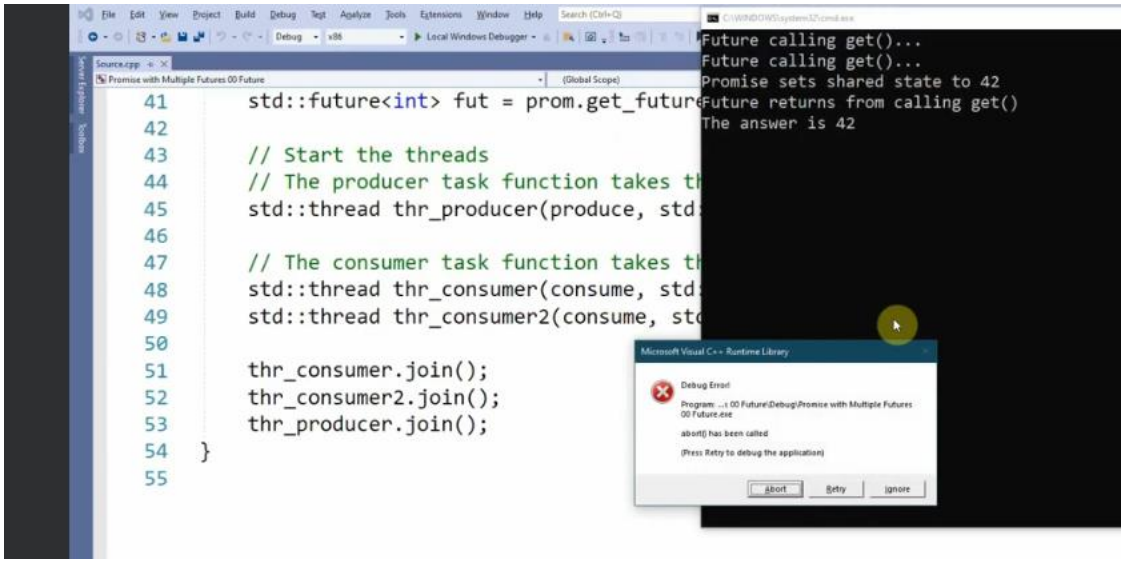
Promises and Multiple Waiting Threads

Single Producer with Multiple Consumers

- Single producer thread
  - Produces a result or an event
- Multiple consumer threads
  - Use the result
  - Or wait for the event to occur
- Used in many applications

std::future and Multiple Waiting Threads

- Designed for use with a single consumer thread
  - Assumes it has exclusive read access to the shared state
- Cannot be safely shared between threads
  - Data race
- Cannot be copied
  - Move-only class



std::shared_future

- Can be copied
  - Each thread has its own object
  - They all share the same state with the std::promise
  - Calling get() or wait() from different copies is safe

Obtaining an std::shared_future object

- Normally, we do not create a shared future directly
- We can move from an existing std::future

```
std::shared_future<int> shared_fut1 =  std::move(fut);
```

- We can call share() on the std::future

```
std::shared_future<int> shared_fut2 = fut.share();
```

- We can also obtain a shared_future directly from a promise

```
shared_future<int> shared_fut3 = prom.get_future();
```

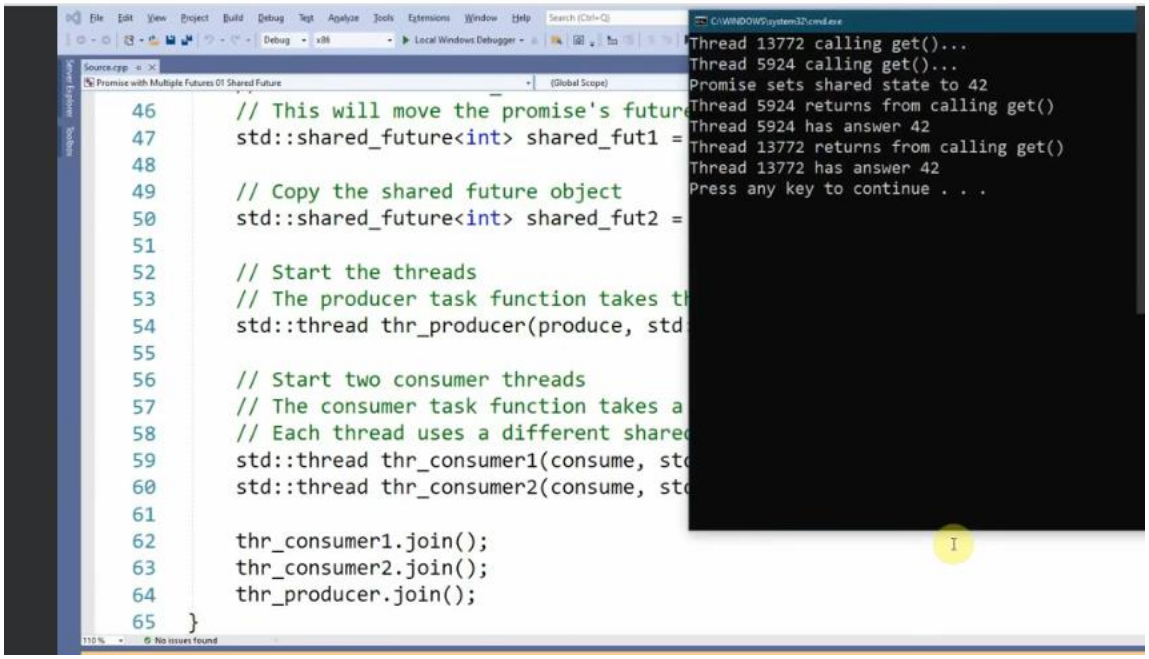
std::shared_future Example

- The producer will be the same as before
- The consumer now takes an std::shared_future

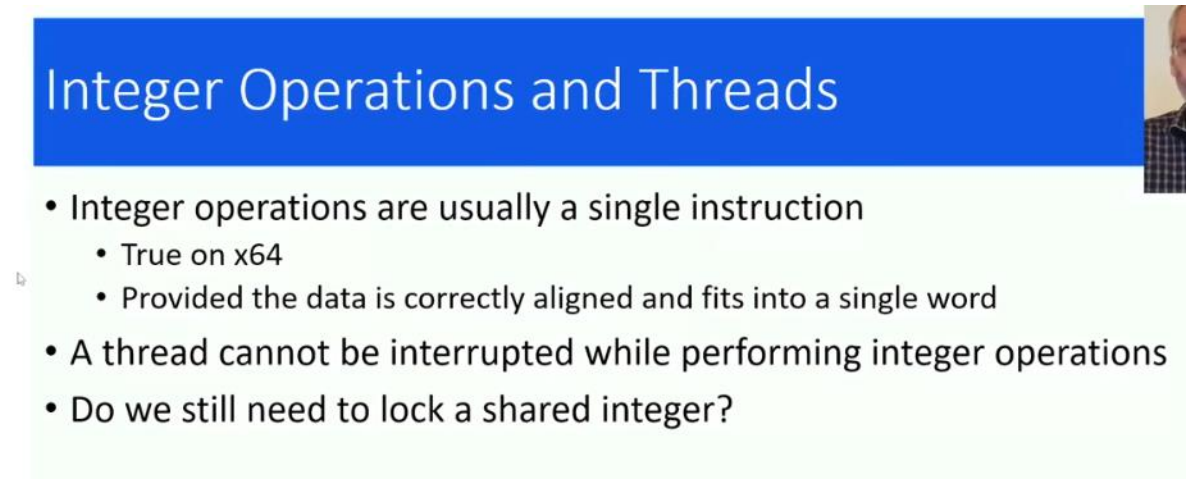
```
void consume(std::shared_future<int>& fx);
```

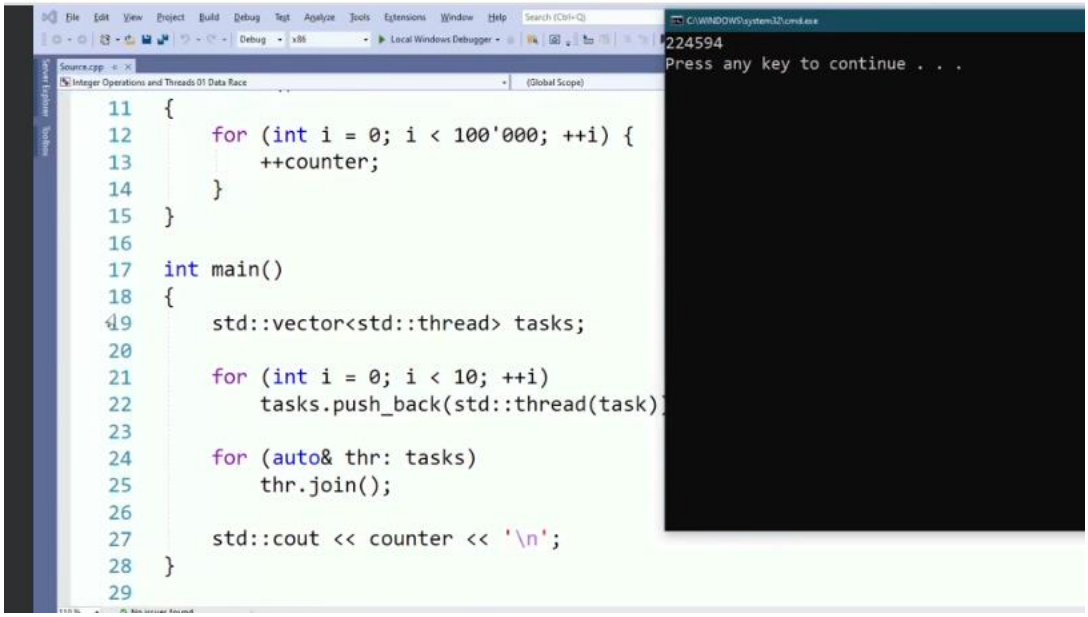
std::shared_future Example

```
// Parent thread
std::promise<int> prom;

// Move the promise's future into a shared future
std::shared_future<int> shared_fut1 = prom.get_future();

// Copy the shared future object
std::shared_future<int> shared_fut2 = shared_fut1;

// Start two consumer threads, each with its own shared future object
std::thread thr_consumer1(consume, std::ref(shared_fut1));
std::thread thr_consumer2(consume, std::ref(shared_fut2));
std::thread thr_producer(produce, std::ref(prom));
```



Integer Operations and Threads

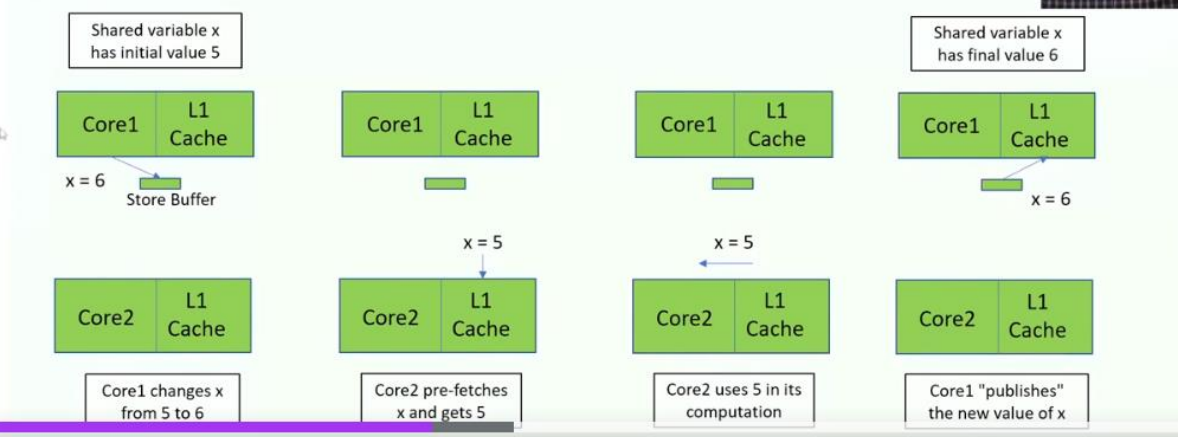




Integer Operation Discussion

- The output is a number like 258413
  - Data race
- The ++ operation is a single instruction
- However, ++count involves three operations
  - Pre-fetch the value of count
  - Increment the value in the processor core's register
  - Publish the new value of count
- The thread could use a stale value in its calculation
- The thread could publish its result after a thread which ran later

Synchronization Issues

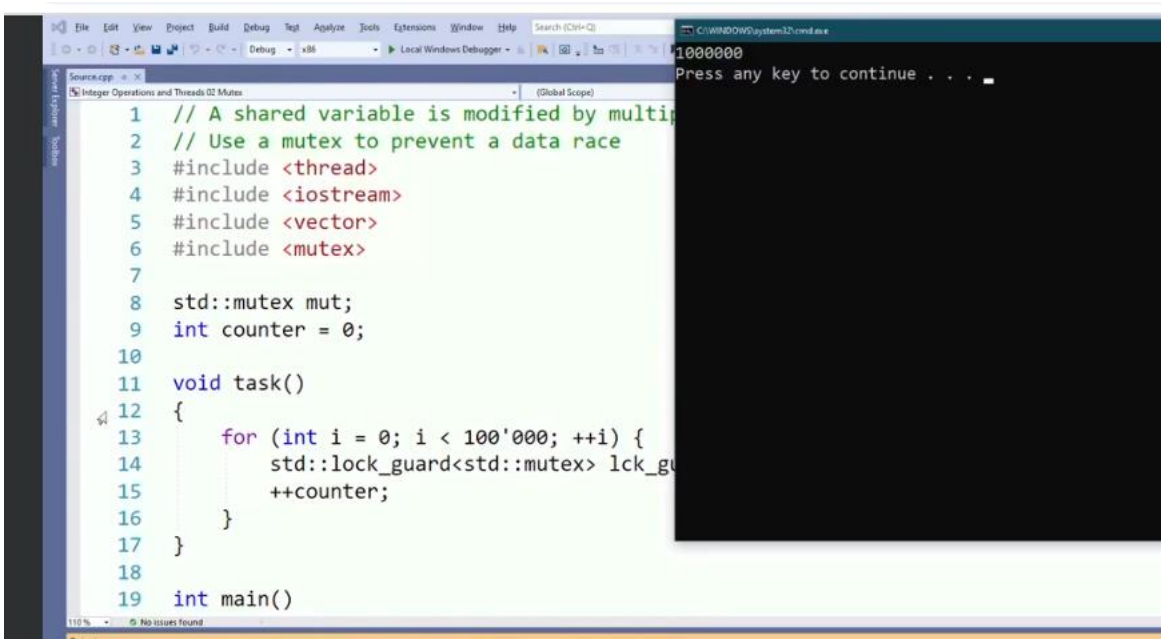


Data Race Scenario

- Threads A and B run on different processor cores
  - Thread A's core pre-fetches the value 5 for count
  - Thread B's core pre-fetches the value 5 for count
  - Thread A's core increments the value to 6
  - Thread B's core increments the value to 6
  - Thread A publishes the value 6 for count
  - Thread B publishes the value 6 for count
- count initially had the value 5
  - It was incremented twice
  - It has the final value 6, when it should be 7

Thread Synchronization

- We need to make sure that
  - Thread B uses the latest value for count
  - Thread A publishes its result immediately
- A mutex does this internally when we call lock() and unlock()
- Here, we can do this by declaring count as "atomic"



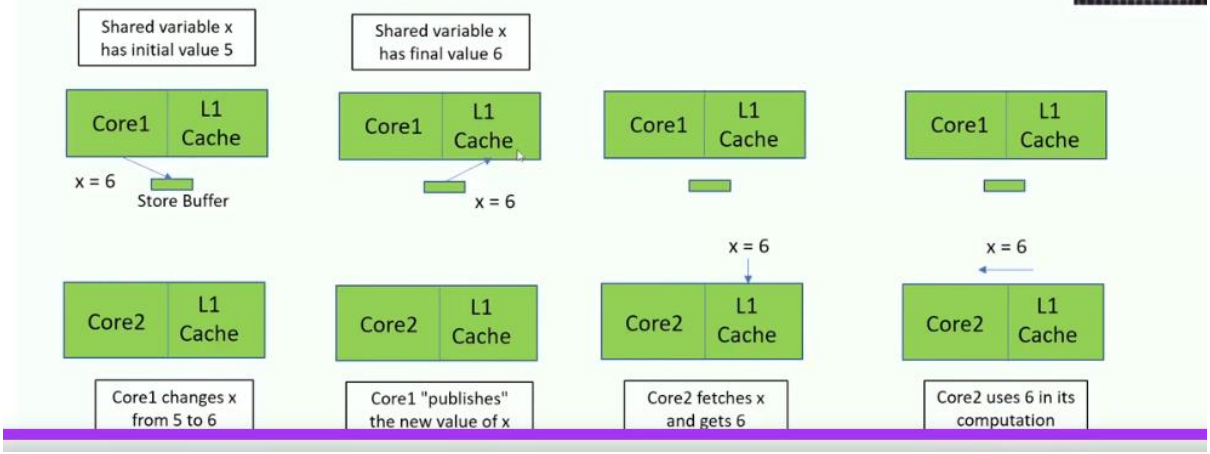
Thread Synchronization

- We need to make sure that
  - Thread B uses the latest value for count
  - Thread A publishes its result immediately
- A mutex does this internally when we call lock() and unlock()
- Here, we can do this by declaring count as "atomic"

Atomic Keyword

- The compiler will generate special instructions which
  - Disable pre-fetch for count
  - Flush the store buffer immediately after doing the increment
- This also avoids some other problems
  - Hardware optimizations which change the instruction order
  - Compiler optimizations which change the instruction order
- The result is that only one thread can access count at a time
- This prevents the data race
  - It also makes the operation take much longer

Synchronization with Atomic Variable



Scenario without Data Race

- Threads A and B run on different processor cores
  - Thread A's core fetches the value 5 for count
  - Thread A's core increments the value to 6
  - Thread A publishes the value 6
  - Thread B's core fetches the value 6 for count
  - Thread B's core increments the value to 7
  - Thread B publishes the value 7
- count initially had the value 5
  - It was incremented twice
  - It has the final value 7, which is what it should be

Atomic Types

Atomic Types

- All operations on the variable will be atomic
- C++11 defines an atomic template
  - In the <atomic> header
  - The parameter is the type of the object
  - The object must be initialized

```
// Atomic int, initialized to 0
atomic<int> x = 0;
```

Atomic Types

- The parameter must be a type which is "trivially copyable"
  - Scalar type
  - Class where all the copy and move constructors are trivial
- Normally only integer types and pointers are used
- For more complex types, locks may be silently added
  - Locking a mutex takes longer
  - To avoid this, use a pointer to the type

Using an std::atomic<T> Object

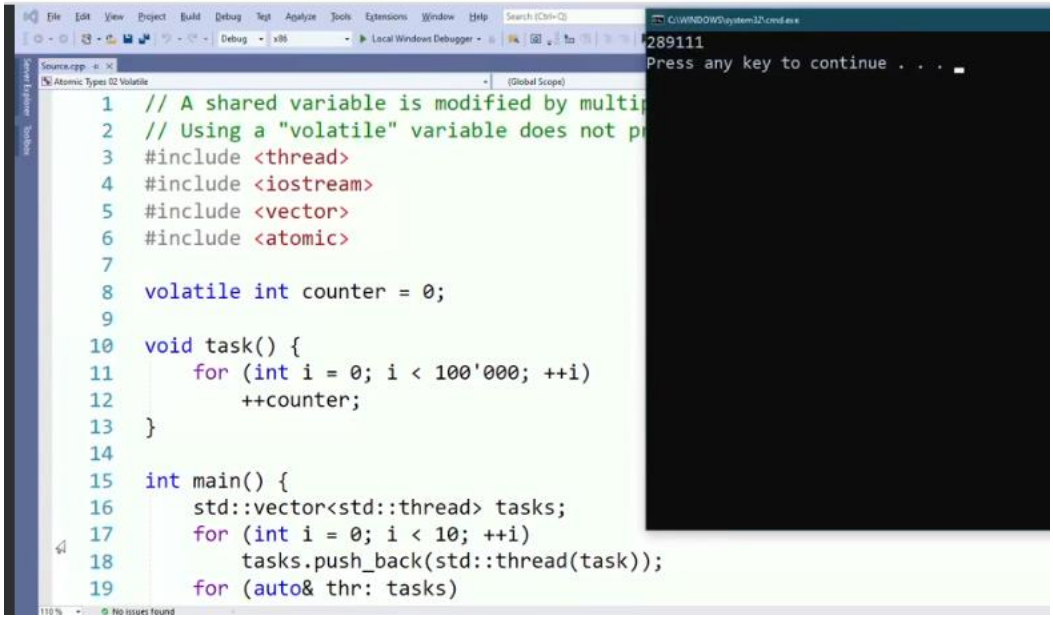
- We can assign to and from the object

```
x = 2;          // Atomic assignment to x
y = x;          // Atomic assignment from x. y can be non-atomic
```

- These are two distinct atomic operations
  - Other threads could interleave between them
- Operations such as ++ are atomic
  - Fetch old value
  - Increment
  - Store new value

Volatile Keyword

- May change without explicit modification
  - Prevents some compiler optimizations
  - Typically used when accessing hardware
- Often mistakenly used in threading
  - Some programmers expect the Java/C# behaviour
  - Has no impact on thread safety
- At one point, Visual Studio supported this in C++
  - Removed before C++11



Double-checked Locking

- One solution is to make the initialized object atomic

```
atomic<Test *> ptest = nullptr;
```

- Atomic types do not support the . or -> operators
- We must copy to a non-atomic pointer before we can use it

```
Test *ptr = ptest;
ptr->func();
```

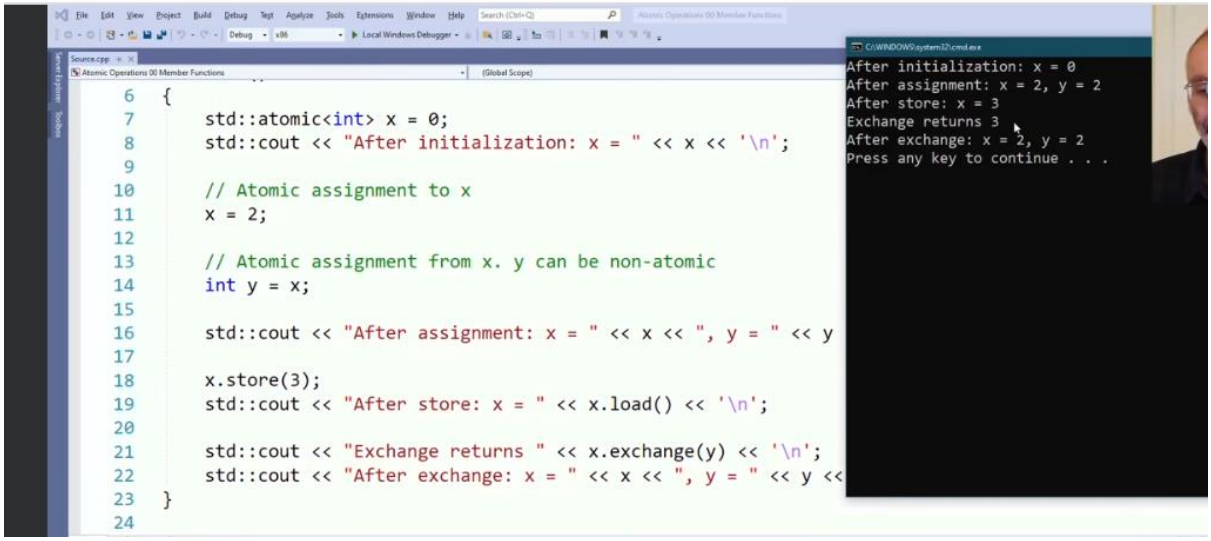
Atomic Operations

Member Functions for Atomic Types

- store()
  - Atomically replace the object's value with its argument
- load()
  - Atomically return the object's value
- operator =()
- operator T()
  - Synonyms for store() and load()
- exchange()
  - Atomically replace the object's value with its argument
  - Returns the previous value

Member Functions for Specializations

- Atomic pointers support pointer arithmetic
  - increment and decrement operators
  - fetch_add() synonym for x++
  - fetch_sub() synonym for x--
  - += and -= operators
- Integer specializations have these, plus
  - Atomic bitwise logical operations &, | and ^



std::atomic_flag

- std::atomic_flag is an atomic boolean type
  - Has less overhead than std::atomic<bool>
- Only three operations
  - clear() sets flag to false
  - test_and_set() sets flag to true and returns previous value
  - operator =()
- Must be initialized to false

```
atomic_flag lock = ATOMIC_FLAG_INIT;
```

Spin Lock

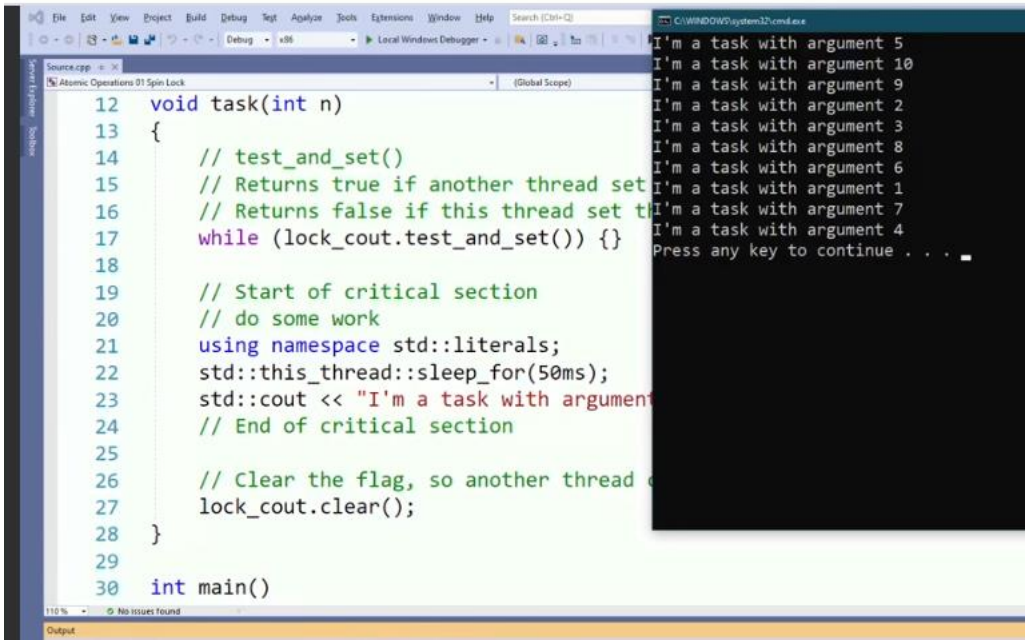
- A spin lock is essentially an infinite loop
  - It keeps "spinning" until a condition becomes true
- An alternative to locking a mutex or using a condition variable
- We can use std::atomic_flag to implement a basic spin lock
  - The loop condition is the value of the flag

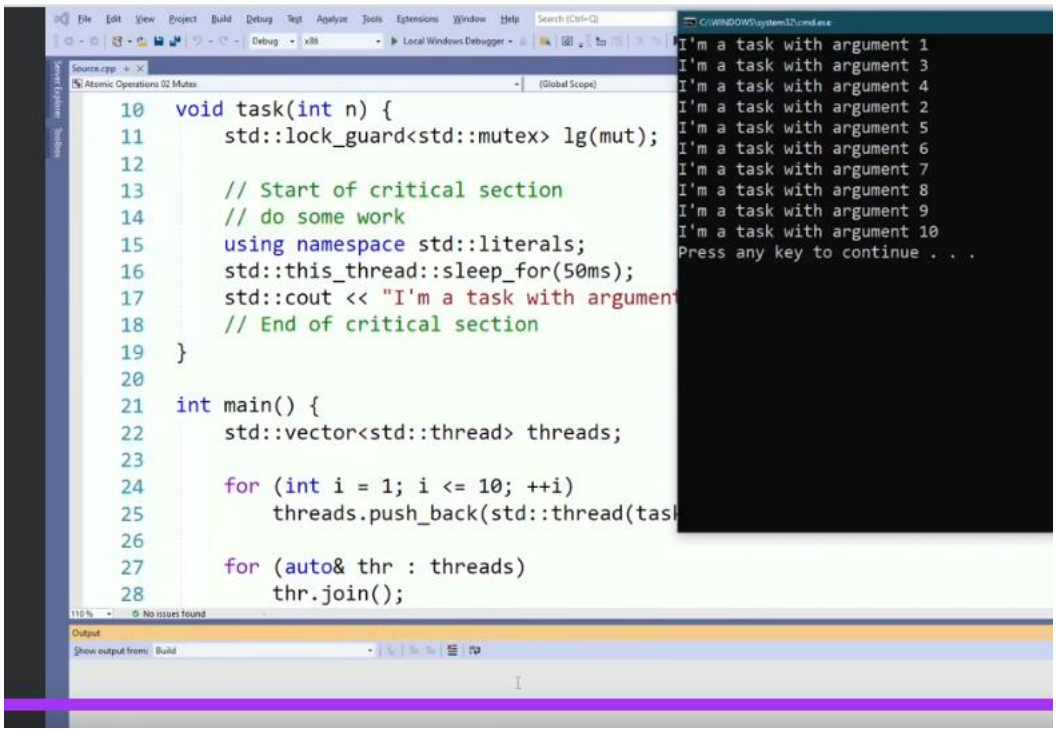
Spin Lock with std::atomic_flag

- Each thread calls test_and_set() in a loop
- If this returns true
  - Some other thread has set the flag and is in the critical section
  - Iterate again
- If it returns false
  - This thread has set the flag
  - Exit the loop and proceed into the critical section
- After the critical section, set the flag to false
  - This allows another thread to execute in the critical section

Spin lock

```
// Initialize flag to false
std::atomic_flag flag = ATOMIC_FLAG_INIT;

void task(int n)
{
  // Loop until we can set the flag
  while (flag.test_and_set()) {}

  // Critical section
  ...

  // Clear the flag
  flag.clear();
}
```





Pros and cons of spin lock

- A spinning thread remains active
  - A mutex may block the thread
- It can continue immediately when it "gets the lock"
  - With a mutex, the thread may need to be reloaded or woken up
- Processor-intensive
  - Only suitable for protecting very short critical sections
  - And/or very low contention
  - Performance can be heavily impacted if spinning threads interrupt each other
  - Usually only used in operating systems and libraries

Hybrid Mutex

- Often used to implement std::mutex
- Start with a spin lock with a time out
  - If the thread sets the flag in time, enter the critical section
  - If the thread cannot set the flag in time, use the normal mutex implementation
- This gives better performance than the conventional implementation

Lock-free Programming

Lock-free Programming

- Threads execute critical sections concurrently
  - Without data races
  - But without using the operating system's locking facilities
- Avoids or reduces some of the drawbacks to using locks
  - Race conditions caused by forgetting to lock, or using the wrong mutex
  - Lack of composability
  - Risk of deadlock
  - High overhead
  - Lack of scalability caused by coarse-grained locking
  - Code complexity and increased overhead caused by fine-grained locking

Locking vs Lock-free

- Both programming styles are used to manage shared state
  - Analogous to managing a traffic intersection
- Locks
  - Traffic lights control access
  - Stop and wait until able to proceed into critical section

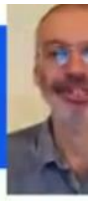
Locking vs Lock-free

- Lock-free
  - Motorway-style intersection
  - Traffic from different levels can go over the same section at the same time
  - Traffic from one level can merge with traffic from a different level without stopping
  - If not done carefully, collisions can occur!

Advantages of Lock-free Programming

- If done correctly, threads can never block each other
  - No possibility of deadlock or livelock
  - If a thread is blocked, other threads can continue to execute
  - Useful if work must be completed within a time limit
  - (e.g. real time systems)

Drawbacks of Lock-free Programming



- Very difficult to write code which is correct and efficient
- The extra complexity makes it unsuitable for many applications
  - e.g. user interface code with separation of concerns
  - May be useful in performance-critical code, such as infrastructure
- Should be used only if
  - A data structure in the program is subject to high contention
  - Which causes unacceptable performance
  - And the lock-free version brings performance up to acceptable levels

Lock-free Programming Continued

The Everyday World of Programming with Locks

- We can make some very useful assumptions
- Global state is consistent
  - Provided we only access shared data inside a locked region
  - No other threads will see our changes
  - Until the lock is released
- Logical consistency
  - When working inside a locked region, global state will not change
  - e.g. between evaluating an "if" statement and executing the body
- Code order
  - Statements will execute in the same order as in the source code
  - Or at least, they will appear to...

The Strange World of Lock-free Programming

- None of these assumptions apply to lock-free programs
  - Shared data may have different values in different threads
  - The value may change between an "if" statement and its body
  - Statements may execute in a different order from the source code

Bank Account Transfer

Transactions

- Transactional model of lock-free programming
  - "ACID"
- Atomic/All-or-nothing
  - A transaction either completes successfully ("commit")
  - Or it fails and leaves everything as it was ("rollback")
- Consistent
  - The transaction takes the database from one consistent state to another
  - As seen by other users, the database is never in an inconsistent state

Transactions

- Transactional model of lock-free programming
  - "ACID"
- Isolated
  - Two transactions can never work on the same data simultaneously
- Durable
  - Once a transaction is committed, it cannot be overwritten
  - ...until the next transaction sees the result of the commit
  - There is no possibility of "losing" an update

Transactional Memory

- Put shared data in transactional memory
- All operations on shared data will be transactional
- However, there is no standard implementation in C++

Atomic Instructions

- Atomic/All-or-nothing
  - An instruction completes successfully (we hope!)
- Consistent
  - The instruction takes the data from one consistent value to another
  - As seen by other threads, the data never has an inconsistent value
- Isolated
  - Two atomic instructions can never work on the same data simultaneously
- Durable
  - Once an instruction is completed, the data cannot be overwritten
  - ...until the next instruction sees the result
  - There is no possibility of "losing" a modification

Lock-free Programming

- To achieve this in lock-free programming
  - Use atomic instructions
- We need to think very carefully about thread interactions
  - Other threads can interleave between each statement
  - Or between expressions within statements
  - How do concurrent writers interact with each other?
  - How do concurrent writers interact with concurrent readers?

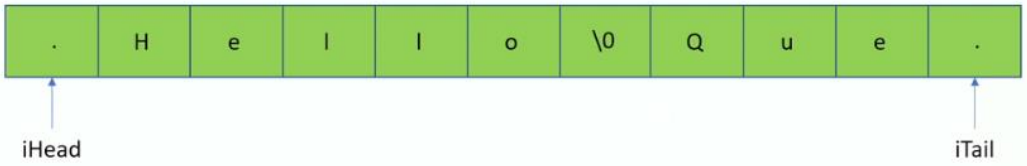
Lock-free Programming Practical

Lock-free Queue

- We will implement a simple queue
  - No internal or external locks
- The queue is only accessed by two threads
  - A Producer thread inserts elements into the queue
  - A Consumer thread removes elements from the queue
- The code is carefully designed
  - The Consumer and Producer threads never work on adjacent elements
  - The two threads always work on different parts of the queue

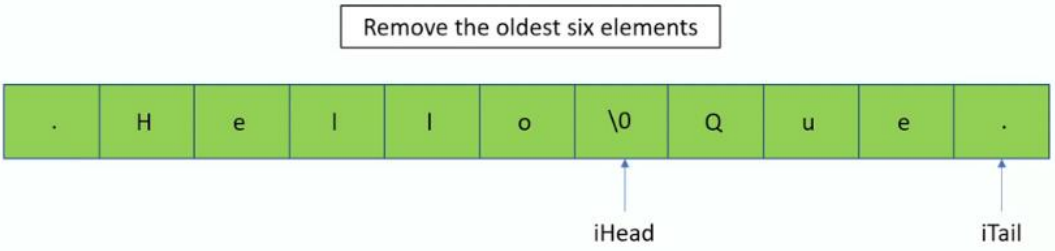
Lock-free Queue

- The queue has two iterators, iHead and iTail
  - iHead points to the element before the oldest element
  - iTail points to the element after the newest (most recently added)



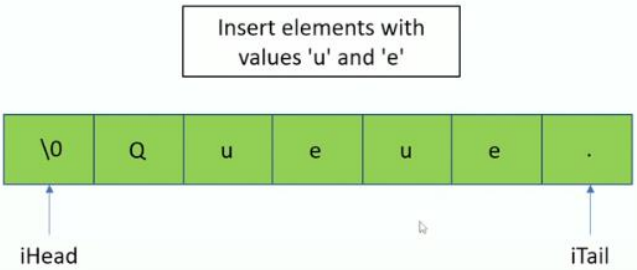
Consumer Thread

- The Consumer thread does not modify the queue
  - It "removes" an element by incrementing iHead



Producer Thread

- The Producer inserts elements
  - It increments iTail
  - It also erases any elements which the Consumer has removed



Thread Separation

- Only the Producer thread can modify the queue
  - The Producer queue inserts elements
  - The Producer queue erases elements
- The two threads never overlap
  - iHead and iTail never refer to the same element
  - The Producer thread never modifies iHead
  - The Consumer thread never accesses elements after iHead

Lock-free Queue Class

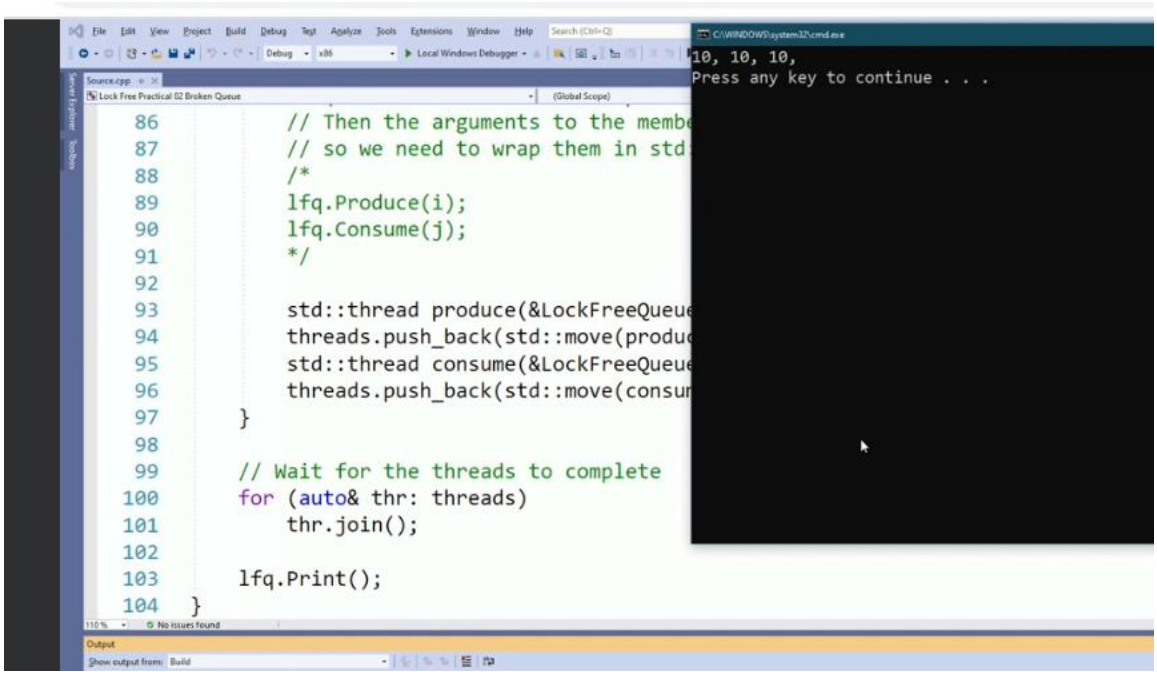
```
template <typename T>
struct LockFreeQueue
{
 private:
  std::list<T> list;
  typename std::list<T>::iterator iHead, iTail;

 public:
  LockFreeQueue() {
     list.push_back(T());          // Create a "dummy" element
     iHead = list.begin();
     iTail = list.end();
  }
```

Consumer Task Member Function

```
bool Consume(T& t)
{
   auto iFirst = iHead;              // Go to the first element
   ++ iFirst;
   if (iFirst != iTail) {            // If queue is not empty
      iHead = iFirst;                // Update iHead
      t = *iHead;                    // Fetch this first element
      return true;
   }
   return false;                     // No elements to fetch
}
```

Producer Task Member Function

```
void Produce(const T& t)
{
   list.push_back(t);                // Add the new element
   iTail = list.end();               // Update iTail
   list.erase(list.begin(), iHead);  // Erase the removed elements
}
```



Thread Safety

- The code has a data race
  - iHead and iTail can be accessed from different threads
  - At least one thread modifies them
  - The threads are not synchronized when they access these variables

Avoiding the Data Race

- iHead and iTail cannot be atomic types
  - std::list<T>::iterator is not trivially copyable
- We must use mutexes

Asynchronous Programming

Synchronous and Asynchronous Programming

- Synchronous
  - Wait for each task to complete
- Asynchronous
  - Continue without waiting for tasks to complete

Synchronous Programming

- A task starts another task
- The current task is blocked
- Must wait until the new task completes before it can continue
  - e.g. Synchronous database access
  - Do some work
  - Request data from database
  - Wait for data
  - Receive data from database
  - Continue working

Synchronous Tasks

- Normal function calls are synchronous tasks

```
data.save(filename);
// Stop and wait for func to return
// ... wait ...

// Now we can continue with the next operation
```

- We have to stop and wait for the save operation to complete
  - Even if the next operation does not depend on it
- This reduces throughput and user satisfaction
  - GUI applications appear unresponsive
  - Clients experience slow service

Asynchronous Programming

- A task starts another task
- The current task can continue
- The new task runs in the background
  - e.g. Asynchronous database access
  - Request data from database as a separate task
  - Do some more work in our task
  - Receive data from database

Asynchronous Tasks

- Start off another task

```
data.async_save(filename);
// The asynchronous task runs in the background
// We continue with the next operation
// ... do something else ...
```

- Our thread can continue its work

```
// At some point, we may need to check if the async call has completed
// Or to get its result
```

Advantages of Asynchronous Programming

- The current task can do other work
  - Provided it does not require the data
- The current task only blocks when it needs the data
  - If the data is already available, it can continue without stopping
- This maintains throughput and user satisfaction
  - GUI applications appear responsive
  - Clients experience normal service

Blocking and Multi-threading Programs

- Blocking is undesirable in threaded programs
  - Blocking reduces throughput and responsiveness of the blocked thread
  - Any threads which join with this thread will also be blocked
- Particularly in a critical section
  - Any threads which are waiting to enter the critical section are also blocked
  - Possibility of deadlock, if we are using locks
- Using asynchronous programming reduces the need to block
  - But may not avoid it completely
  - e.g. if the database fetch is not complete when the data is needed

Blocking Synchronization

- Blocking operations
- Synchronized by mutexes
  - A thread is blocked until another thread unlocks the mutex
- Or atomic operations
  - A thread is blocked until another thread completes an atomic operation

Non-blocking Synchronization

- Non-blocking operations
- Synchronized by message queues
  - A thread pushes a message onto a concurrent queue
  - Another thread takes the message off the queue and processes it
  - The first thread continues running without waiting
- The messages often consist of callable objects
  - The message is processed by invoking the callable object
- C++ does not have a standard concurrent queue
  - Available in Boost, Microsoft's PPL, Intel's TBB

Asynchronous Programming and Parallelism

- Asynchronous programming
- Can be used to perform parallel operations
  - Start new threads which all perform the same task
  - Collect the result from each thread as it completes its task
  - Combine the results into the final answer
- It can also be used in single-threaded programs
  - Using operating system features