Ch. 1 - Hello, world of concurrency in C++!

- C++11 standardized threading and concurrency in without reliance on external libraries

1.1 What is concurrency?

- two or more separate activities happening at the same time

Concurrency in computer systems

- single system performing multiple independent activities in parallel, rather than sequentially, or one after the other
- single core can only perform one task at a time but can switch between many tasks very rapidly
- task switching
- hardware concurrency - multiple processors or multiple cores within a processor (or both), computers that are capable of running more than one task in parallel
- in order to interleave task execution the system has to perform a context switch every time it changes from one task to another which takes time
- to perform a context switch the OS has to
    - save the CPU state and instruction pointer for the currently running task
    - determine which task to switch to
    - reload the CPU state for the task being switched to
    - potentially load the memory for the instructions and data for the new task into cache
        - can prevent the CPU from executing any instructions causing further delay
- some processors can run more than one thread on a single processor
- the number of hardware threads is the important factor to consider to measure how many independent tasks the hardware can genuinely run concurrently

Approaches to concurrency

- communication is difficult
- two main ways to achieve concurrency in programs
    - multiple processes
    - multiple threads

Concurrency with multiple processes

- divide the application into multiple, separate, single-threaded processes that are run at the same time
  - separate processes pass messages to each other through all the normal interprocess communication channels
    - signals
    - sockets
    - files
    - pipes
    - etc.
- communication between processes can be
  - complicated to set up
  - slow
- difficult to share data
- more downsides
  - process startup takes time
  - OS must devote internal resources to managing processes
- updsides
  - added protection operating systems typically provide between processes
  - higher-level communication mechanisms mean that it can be easier to write safe concurrent code with processes rather than threads
  - can run the separate processes on distinct machines connected over a network

Concurrency with multiple threads

- like lightweight processes
  - each thread runs independently of the others
  - each thread may run a different sequence of instructions
  - all threads in a process share the same address space
  - most of the data can be accessed directly from all threads
    - global variables remain global
    - pointers or references to objects or data can be passed around among threads
- possible to share memory among processes
  - complicated to set up
  - hard to manage because memory addresses of the same data aren't necessarily the same in different processes
- shared address space and lack of protection of data between threads makes the overhead associated much smaller

- flexibility of shared memory has downsides
  - view of data seen by each thread is consistent whenever accessed if data is accessed by multiple threads
- threads are most common place to turn for concurrency (but often the most complicated when considering other options)

1.2 Why use concurrency?

- separation of concerns
- performance

Using concurrency for separation of concerns

- separate distinct areas of functionality

Using concurrency for performance

- task parallelism - divide a single task into parts and run each in parallel (reducing total runtime)
- data parallelism - each unit of execution performs same operation on different parts of the data
- embarassingly parallel
- naturally parallel
- conveniently concurrent
- can use concurrency for performance by solving bigger problems (application of data parallelism at scale)

When not to use concurrency

- when benefit is not worth the cost
- plan and do preliminary tests and benchmark
- threads are a limited resource

1.4 Concurrency and multithreading in C++

History of multithreading in C++

Concurrency support in the new standard

Efficiency in the C++ Thread Library

- abstraction penalty

Platform-specific facilities

1.4 Getting started

Hello, Concurrent World

```
#include <iostream>
#include <thread>

void func()
{
    // do something
}

int main()
{
    std::thread t{func};
    t.join();
    ...
}
```

- each thread has to have an initial function for thread to begin execution
- main thread continues execution and join is used to ensure it does not exit until the created thread has launched and run

1.5 Summary

Ch. 2 - Managing threads

2.1 Basic thread management

Launching a thread

- started by constructing a std::thread object that specifies the task to run on that thread
- works with any callable type
- the supplied function object is copied into the storage belonging to the newly created thread of execution and invoked from there
- avoid C++'s most vexing parse in thread construction
    - use uniform initialization syntax
- can prefer lambdas
- when thread is started need to explicitly decide whether to
    - wait for it to finish (by joining)

- leave it to run on its own (by detaching it)
- if not done before std::thread object is destroyed then program will be terminated
  - std::thread destructor calls std::terminate()
- only have to do this before the std::thread object is destroyed
- if program doesn't wait for thread to finish needs to ensure that the data accessed by the thread is valid until the thread has finished with it
- Listing 2.1 shows example of dangling reference
- can handle by making the thread function self-contained and copy the data into the thread rather than share the data
- be wary of objects containing pointers or references
- bad idea to create a thread within a function that has access to local variables in that function unless the thread is guaranteed to finish before the function exits

Waiting for a thread to complete

- use join for straightforward waiting for thread to finish
- join is brute force
  - can use condition variables
  - can use futures
- join cleans up storage
- only call join once for a given thread
- use joinable to check if a thread can join

Waiting in exceptional circumstances

- must call join or detach before thread is destroyed
- can call detach immediately in most cases
- must choose location of join carefully
- if an exception is thrown in a thread, a call to join may be skipped
  - use try/catch blocks
  - use a thread_guard and RAII to call join in destructor
    - mark copy/move as delete

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

Running threads in the background

- calling detach() on a std::thread object leaves the thread to run in the background with no direct means of communicating with it
    - ownership and control are passed over to the C++ runtime library which reclaims resources when thread exits
- often called daemon threads
- either for
    - long running actions
    - when there's another mechanism for identifying when the thread has completed
    - when the thread is used for a "fire and forget" task
    - multiple document user interactions

2.2 Passing arguments to a thread function

- passing arguments to the callable object or function
    - pass additional arguments to the std::thread constructor
- by default the arguments are copied into internal storage
    - can be accessed by the newly created thread of execution even if the corresponding parameter in the function is expecting a reference
- when passing arguments to threads that require conversion (constructors, etc, e.g. const char * to std::string) be aware that calling function may exit before argument has been constructed in new thread so may result in undefined behavior
- can also get situations where an object is copied into thread but what was desired was a reference
    - thread constructor is unaware of the type of argument that is supposed to be passed to callable and blindly copies in each argument
    - use std::ref(arg) to pass to thread constructor for references
- can pass member function pointers but must pass an object as the first argument (similar to std::bind)

```
std::thread t{&Class::member, &instance);
```

- may also run into move-only arguments
    - std::unique_ptr
    - etc.

2.3 Transferring ownership of a thread

- threads are movable but not copyable
    - requires std::move to reassign

- can return thread from a function without moving (moving is managed by compiler)
- requires moving into functions
- Listing 2.6 - scoped_thread builds on thread_guard but moves the passed thread into the class
- can use the move support to create containers of threads
  - rather than creating separate variables for threads and joining with them directly, they can be treated as a group

2.4 Choosing the number of threads at runtime

- std::thread::hardware_concurrency
- limit creating too many threads when an algorithm does not require it
- oversubscription - creating more threads than the hardware can support
  - context switching reduces performance

```
auto results = std::vector<T>(num_threads);
auto threads = std::vector<std::thread>(num_threads - 1);

auto block_start = first;
for (auto i = 0; i < (num_threads - 1); ++i) {
    auto block_end = block_start;
    std::advance(block_end, block_size);
    threads[i] = std::thread(accumulate_block<Iterator, T>(), block_start, block_e
    block_start = block_end;
}
```

- must pass in a reference to return a result because cannot return a value from a thread
- alternative ways of returning results from threads are addressed through the use of futures

2.5 Identifying threads

- std::thread::id
- .get_id()
- if no associated thread, returns a default constructed thread::id which indicates no thread
- can also call std::this_thread::get_id()
- writing to output is implementation dependent
- (hashable)

Ch. 3 - Sharing data between threads

3.1 Problems with sharing data between threads

- due to consequences of modifying data
- if all shared data is read-only, there's no problem, because the data read by one thread is unaffected by whether or not another thread is reading the same data
- invariants - assumptions that are required to always be true about a particular data structure (or other conditions)

Race conditions

- any situation where the outcome depends on the relative ordering of execution of operations on two or more threads
    - the threads race to perform their respective operations
    - usually used to mean the problematic type
    - data races lead to undefined behavior
- generally occur when an operation depends on modification of two or more distinct pieces of data
    - if output instructions are sequential, testing may not show the issues until a large load is placed on the system
        - the number of times the operation is performed increases
        - the chance of the problematic execution sequence occurring also increases
    - debuggers affect the timing of the program and race conditions can disappear entirely because they are usually timing sensitive

Avoiding problematic race conditions

- simplest option is to wrap data structure with a protection mechanism, to ensure that only the thread actually performing a modification can see the intermediate states where the invariants are broken
- lock-free programming - modify the design of data structure and its invariants so that modifications are done as a series of indivisible changes, each of which preserves the invariants
- software transactional memory (STM) - can handle the updates to the data structure as a transaction
    - required series of data modifications and reads is stored in a transaction log and then committed in a single step

- if the commit can't proceed because the data structure has been modified by another thread, the transaction is restarted

Protecting shared data with mutexes

- mutual exclusion - mechanism to mark data structures so that if any thread is running one of them any other thread that tried to access the data structure has to wait until the first thread is finished
    - makes it impossible for a thread to see a broken invariant except when it was the thread doing the modification
    - lock the mutex associated with the data
    - unlock the mutex when finished accessing the data structure
- risks deadlock

Using mutexes in C++

- std::mutex and .lock and .unlock
- member locking functions are not recommended
- std::lock_guard template uses RAII to lock on construction and unlock on destruction
- best to group lock guard and mutex together in a class rather than access as global variables
    - any code that has access to protected data (via pointer or reference) encapsulated in such a class can access (and potentially modify) the protected data without locking the mutex
    - do not do this, use careful interface design

Structuring code for protecting shared data

- also important to check that member functions don't pass such pointers or references in to functions they call that aren't under application programmer's control
    - those functions might store the pointer or reference in a place where it can later be used without the protection of the mutex
    - particularly dangerous in this regard are functions that are supplied at runtime via a function argument or other means
- general guideline
    - don't pass pointers and references to protected data outside the scope of the lock
        - by returning them from a function
        - storing them in externally visible memory
        - passing them as arguments to user-supplied functions

Spotting race conditions inherent in interfaces

- can still get race conditions with proper simple protection
    - consider hand-over-hand locking required for linked list type structures

- race conditions can still occur in lock free implementations
- design interfaces with care
  - which operations are absolutely essential?
  - which operations might run concurrently?
  - what happens when concurrent operations are interleaved?
- pop and return value from stack has specific issues

Option 1: Pass in a reference

- pass a reference to a variable to receive the popped value as an argument in the call to pop ()
  - requires construction of stack's value type prior to call
  - may not be possible for certain types
  - requires that the stored type is assignable

Option 2: Require a no-throw copy constructor or move constructor

- only have an exception safety problem with a value-returning pop() if the return by value can throw an exception
  - many types have copy constructors that don't throw exceptions
  - more types will have a move constructor that doesn't throw exceptions due to r-value reference support (even if their copy constructor does)
- can restrict use of thread-safe stack to those types that can safely be returned by value without throwing an exception
  - not ideal
- can detect at compile time the existence of a copy or move constructor that doesn't throw an exception using the std::is_nothrow_copy_constructible and std::is_nothrow_move_constructible type traits
- limits the number of types that this will work with
  - more user-defined types with copy constructors that can throw and don't have move constructors than there are types with copy and/or move constructors that can't throw

Option 3: Return a pointer to the popped item

- return a pointer to the popped item rather than return the item by value
  - pointers can be freely copied without throwing an exception
    - avoided Cargill's exception problem
- disadvantage is that returning a pointer requires a means of managing the memory allocated to the object
  - overhead of such memory management can exceed the cost of just returning the type by value for simple types such as integers
- try to use std::shared_ptr for any interface that uses this option
  - avoids memory leaks
  - library is in full control of the memory allocation scheme

- can be important for optimization purposes
  - requiring that each object in the stack be allocated separately with new would impose quite an overhead compared to the original non-thread-safe version

Option 4: Provide both option 1 and either option 2 or 3

- flexibility is good in generic code
  - with option 2 or 3 relatively easy to provide option 1 as well

Example definition of a thread-safe stack

- Listing 3.4
  - maximum safety with minimal interface
  - deleted assignment operator
- Listing 3.5
  - copy constructor locks the mutex in the source object and then copies the internal stack
    - does the copy in the constructor body rather than the member initializer list
      - ensures that the mutex is held across the copy
  - check for empty before trying to pop value in pop
  - in shared_ptr pop, allocate the return value before modifying the stack

```
std::shared_ptr<T> pop()
void pop(T& value)
```

- problematic race conditions in interfaces arise because of locking at too small a granularity
  - protection doesn't cover the entirety of the desired operation
- problems with mutexes can also arise from locking at too large a granularity
  - extremem single global mutex that protects all shared data
- can eliminate any performance benefits of concurrency in a system where there's a significant amount of shared data
- first versions of the Linux kernel that were designed to handle multiprocessor systems used a single global kernel lock

Deadlock: the problem and a solution

- threads contested over locks on mutexes
  - each of a pair of threads needs to lock both of a pair of mutexes to perform some operation
  - each thread has one mutex and is waiting for the other
  - neither thread can proceed, because each is waiting for the other to release its mutex
- problem with having to lock two or more mutexes in order to perform an operation
- always lock the two mutexes in the same order
  - will never deadlock mutex A is always locked before mutex B

- sometimes straightforward - the mutexes are serving different purposes
- compex example - the mutexes are each protecting a separate instance of the same class
- e.g. operation that exchanges data between two instances of the same class
  - to ensure that the data is exchanged correctly, without being affected by concurrent modifications, mutexes on both instances must be locked
  - if a fixed order is chosen this can backfire
    - if two threads try to exchange data between the same two instances with the parameters swapped - have a deadlock
- C++ Standard Library solution - std::lock
  - function that can lock two or more mutexes at once without risk of deadlock
- Listing 3.6
  - arguments are checked to ensure they are different instances
    - attempting to acquire a lock on a std::mutex when it is already held is undefined behavior
      - mutex that does permit multiple locks by the same thread is provided in the form of std::recursive_mutex
  - call to std::lock() locks the two mutexes
  - two std::lock_guard instances are constructed - one for each mutex
    - std::adopt_lock parameter is supplied in addition to the mutex to indicate to the std::lock_guard objects that the mutexes are already locked and they should just adopt the ownership of the existing lock on the mutex rather than attempt to lock the mutex in the constructor
    - ensures that the mutexes are correctly unlocked on function exit in the general case where the protected operation might throw an exception
    - also allows for a simple return
    - NOTE: locking either lhs.m or rhs.m inside the call to std::lock can throw an exception
      - exception is propagated out of std::lock
      - if std::lock has successfully acquired a lock on one mutex and an exception is thrown when it tries to acquire a lock on the other mutex, first lock is released automatically
        - std::lock provides all-or-nothing semantics with regard to locking the supplied mutexes
- std::lock can help avoid deadlock in cases where need to acquire two or more locks together
  - doesn't help if they're acquired separately
  - have to rely on discipline as developers to avoid deadlock
- deadlocks are one of the nastiest problems to encounter in multithreaded code and are often unpredictable

Further guidelines for avoiding deadlock

- deadlock can occur while waiting for any shared resource
- can create deadlock with two threads and no locks just by having each thread call join() on the std::thread object for the other
    - neither thread can make progress because it's waiting for the other to finish
    - simple cycle can occur anywhere that a thread can wait for another thread to perform some action if the other thread can simultaneously be waiting for the first thread
    - isn't limited to two threads
        - a cycle of three or more threads will still cause deadlock
- summary - don't wait for another thread if there's a chance it's already waiting on current thread

Avoid nested locks

- don't acquire a lock if it is already held
    - impossible to get a deadlock from the lock usage alone because each thread only ever holds a single lock
- can still get deadlock from other things (like the threads waiting for each other), but mutex locks are probably the most common cause of deadlock
- acquire multiple locks as a single action with std::lock in order to acquire them without deadlock

Avoid calling user-supplied code while holding a lock

- no idea what user-supplied code could do
    - could acquire a lock
        - violates the guideline on avoiding nested locks and could get deadlock
- sometimes unavoidable
    - for generic code every operation on the parameter type or types is user-supplied code

Acquire locks in a fixed order

- if must acquire two or more locks and can't acquire them as a single operation with std::lock
    - acquire them in the same order in every thread
    - key is to define the order in a way that's consistent between threads
    - previous stack definition - mutex is internal to each stack instance but the operations on the data items stored in a stack require calling user-supplied code
        - can add the constraint that none of the operations on the data items stored in the stack should perform any operation on the stack itself
        - requires user to handle but uncommon for the data stored in a container to access that container
            - obvious when it happens

- in some cases can lock the mutexes simultaneously
  - not always possible
- for linked list one possibility for protecting the list is to have a mutex per node
  - threads must acquire a lock on every node they're interested in to access list
  - threads must acquire the lock on three nodes to delete an item
    - node being deleted and the nodes on either side
      - all being modified in some way
    - to traverse the list a thread must keep hold of the lock on the current node while it acquires the lock on the next one in the sequence
      - ensures that the next pointer isn't modified while transitioning
    - once the lock on the next node has been acquired, the lock on the first can be released
  - hand-over-hand locking style allows multiple threads to access the list
    - requires each to access a different node
  - nodes must always be locked in the same order to prevent deadlock
    - if two threads tried to traverse the list in reverse order using hand-over-hand locking, they could deadlock with each other in the middle of the list
    - if nodes A and B are adjacent in the list, the thread going one way will try to hold the lock on node A and try to acquire the lock on node B
    - a thread going the other way would be holding the lock on node B and trying to acquire the lock on node A
      - could cause deadlock
  - when deleting node B that lies between nodes A and C, if that thread acquires the lock on B before the locks on A and C, it has the potential to deadlock with a thread traversing the list
    - thread would try to lock either A or C first (depending on the direction of traversal) but would then find that it couldn't obtain a lock on B because the thread doing the deleting was holding the lock on B and trying to acquire the locks on A and C
  - define an order of traversal
    - thread must always lock A before B and B before C
    - would eliminate the possibility of deadlock at the expense of disallowing reverse traversal
  - similar conventions can often be established for other data structures

Use a lock hierarchy

- particular case of defining lock ordering
- divide application into layers and identify all the mutexes that may be locked in any given layer
  - when code tries to lock a mutex, it isn't permitted to lock that mutex if it already holds a lock from a lower layer

- can check this at runtime by assigning layer numbers to each mutex and keeping a record of which mutexes are locked by each thread
- Listing 3.7

Extending these guidelines beyond locks

- deadlock doesn't just occur with locks
  - can occur with any synchronization construct that can lead to a wait cycle
- avoid acquiring nested locks if possible
- bad idea to wait for a thread while holding a lock
  - thread might need to acquire the lock in order to proceed
- if waiting for a thread to finish
  - worth identifying a thread hierarchy
    - thread waits only for threads lower down the hierarchy
  - ensure that threads are joined in the same function that started them
- std::lock() and std:: lock_guard cover most of the cases of simple locking
  - provides the std::unique_lock template
    - class template parameterized on the mutex type
    - provides RAII-style lock management but with a bit more flexibility

Flexible locking with std::unique_lock

- std::unique_lock provides more flexibility by relaxing the invariants
  - an instance doesn't always own the mutex that it's associated with
  - can pass std::adopt_lock as a second argument to the constructor to have the lock object manage the lock on a mutex
  - can pass std::defer_lock as the second argument to indicate that the mutex should remain unlocked on construction
  - lock can then be acquired later by calling lock() on the std::unique_lock object (not the mutex) or by passing the std::unique_lock object itself to std::lock()
  - using std::unique_lock and std::defer_lock rather than std::lock_guard and std::adopt_lock
    - std::unique_lock takes more space and is a fraction slower to use than std::lock_guard
    - flexibility of allowing a std::unique_lock instance not to own the mutex comes at the price that information has to be stored, and it has to be updated
- Listing 3.9
  - std::unique_lock objects could be passed to std::lock() because std::unique_lock provides lock(), try_lock(), and unlock()
  - forward to the member functions of the same name on the underlying mutex to do the actual work and just update a flag inside the std::unique_lock instance to indicate whether the mutex is currently owned by that instance

- flag is necessary in order to ensure that unlock() is called correctly in the destructor
- destructor must call unlock() if the instance does own the mutex and shouldn't if it doesn't
- flag can be queried by calling the owns_lock() member function
- size of a std::unique_lock object is larger than that of a std::lock_guard object
    - slight performance penalty when using std::unique_lock over std::lock_guard because the flag has to be updated or checked, as appropriate. If
- prefer if std::lock_guard is sufficient for needs
- cases where std::unique_lock is a better fit
    - deferred locking
    - case is where the ownership of the lock needs to be transferred from one scope to another

Transferring mutex ownership between scopes

- ownership of a mutex can be transferred between instances by moving the instances around because std::unique_lock instances don't have to own their associated mutexes
    - transfer is automatic in some cases
        - returning an instance from a function
    - have to do it explicitly by calling std::move() in some cases
        - depends on whether the source is an lvalue or an rvalue
    - transfer is automatic if the source is an rvalue
    - transfer must be done explicitly for an lvalue in order to avoid accidentally transferring ownership away from a variable
    - std::unique_lock is movable but not copyable
- can allow a function to lock a mutex and transfer ownership of that lock to the caller, so the caller can then perform additional actions under the protection of the same lock

```
std::unique_lock<std::mutex> get_lock()
{
    extern std::mutex some_mutex;
    std::unique_lock<std::mutex> lk(some_mutex);
    prepare_data();
    return lk;
}

void process_data()
{
    std::unique_lock<std::mutex> lk(get_lock());
    do_something();
}
```

- lk can be returned directly without a call to std:move() because it is an automatic variable declared within the function
    - compiler handles calling move constructor

- process_data() can then transfer ownership directly into its own std::unique_lock instance
- do_something() can rely on the data being correctly prepared without another thread altering the data
- pattern is used where the mutex to be locked is dependent on the current state of the program or on an argument
    - e.g. where the lock isn't returned directly but is a data member of a gateway class used to ensure correctly locked access to some protected data
    - all access to the data is through this gateway class
        - to access data obtain an instance of the gateway class by calling a function such as get_lock()
        - then access the data through member functions of the gateway object
        - destroy the gateway object, which releases the lock and allows other threads to access the protected data
        - gateway object can be movable (so that it can be returned from a function)
            - lock object data member also needs to be movable
- std::unique_lock also allows instances to relinquish their locks before they're destroyed
    - use the unlock() member function
        - std::unique_lock supports the same basic set of member functions for locking and unlocking as a mutex does
        - can be used with generic functions such as std::lock
        - ability to release a lock before the std::unique_lock instance is destroyed means
            - can optionally release it in a specific code branch if it's apparent that the lock is no longer required
    - holding a lock for longer than required can cause performance hit

Locking at an appropriate granularity

- lock granularity is a vague term to describe the amount of data protected by a single lock

    - fine-grained lock protects a small amount of data
    - coarse-grained lock protects a large amount of data

- important to choose a sufficiently coarse lock granularity to ensure the required data is protected

- important to ensure that a lock is held only for the operations that actually require it

- if any thread holds lock for longer than necessary when multiple threads are waiting for the same resource it will increase the total time spent waiting

    - lock a mutex only while actually accessing the shared data
    - try to do any processing of the data outside the lock
    - don't do any really time-consuming activities like file I/O while holding a lock

- file I/O is typically hundreds (if not thousands) of times slower than reading or writing the same volume of data from memory
  - unless the lock is really intended to protect access to the file, performing I/O while holding the lock will delay other threads unnecessarily
- std::unique_lock works well in this situation
  - can call unlock() when the code no longer needs access to the shared data and then call lock() again if access is required later in the code

```
void get_and_process_data()
{
    std::unique_lock<std::mutex> my_lock(the_mutex);
    some_class data_to_process=get_next_data_chunk();
    my_lock.unlock();
    result_type result=process(data_to_process);
    my_lock.lock();
    write_result(data_to_process,result);
}
```

- manually unlock it and then lock it again afterward
- if there is one mutex protecting an entire data structure
  - likely to be more contention for the lock
  - less potential for reducing the time that the lock is held
- locking at an appropriate granularity isn't only about the amount of data locked
  - also concerns how long a lock is held and what operations are performed while the lock is held
  - a lock should be held for only the minimum possible time needed to perform the required operations
  - time-consuming operations such as acquiring another lock or waiting for I/O to complete shouldn't be done while holding a lock unless absolutely necessary
- Listing 3.10
- function retrieves the value while protecting it with a lock
  - comparison operator then compares the retrieved values . Note, however, that as well as
  - reduces the locking periods so that only one lock is held at a time
    - eliminates the possibility of deadlock
  - changed the semantics of the operation compared to holding both locks together
    - means that the value of lhs.some_detail at one point in time is equal to the value of rhs.some_detail at another point in time if operator returns true
    - two values could have been changed in any way in between the two reads
      - values could have been swapped
    - equality comparison might return true to indicate that the values were equal even though there was never an instant in time when the values were actually equal

- be careful when making such changes that the semantics of an operation are not changed in a problematic fashion
  - exposed to race conditions when not holding the required locks for the entire duration of an operation
- may be appropriate to use an alternative mechanism when there isn't an appropriate level of granularity because not all accesses to the data structure require the same level of protection

3.3 Alternative facilities for protecting shared data

- mutexes are the most general mechanism
- alternatives for protecting shared data
- when the shared data only needs protection from concurrent access while it's being initialized (no explicit synchronization required afterwards)
- e.g. because the data is read-only once created, and so there are no possible synchronization issues
- e.g. because the necessary protection is performed implicitly as part of the operations on the data
- locking a mutex after the data has been initialized just to protect the initialization is unnecessary
- C++ Standard provides a mechanism purely for protecting shared data during initialization

Protecting shared data during initialization

- for shared resource that's expensive to construct
  - e.g. a database connection
  - e.g. large memory allocation
- lazy initialization is common in single-threaded code
  - each operation that requires the resource checks to see if it has been initialized and then initializes it before use if not

```
std::shared_ptr<some_resource> resource_ptr;
void foo()
{
    if(!resource_ptr) {
        resource_ptr.reset(new some_resource);
    }
```

```
        resource_ptr->do_something();
}
```

- only part that needs protection while converting to multithreaded code is the initialization if the shared resource itself is safe for concurrent access
    - naive approach can cause unnecessary serialization of threads using the resource
        - each thread must wait on the mutex in order to check whether the resource has already been initialized
- Listing 3.11
- thread-safe lazy initialization using a mutex is common and leads to problematic and unnecessary serialization
    - one attempt at improvement is infamous Double-Checked Locking pattern
        - pointer is first read without acquiring the lock
        - lock is acquired only if the pointer is NULL
        - pointer is then checked again once the lock has been acquired (double-checked part) in case another thread has done the initialization between the first check and this thread acquiring the lock

```
void undefined_behaviour_with_double_checked_locking()
{
    if(!resource_ptr) {
        std::lock_guard<std::mutex> lk(resource_mutex);
        if(!resource_ptr) {
            resource_ptr.reset(new some_resource);
        }
    }
    resource_ptr->do_something();
}
```

- pattern has the potential for nasty race conditions
    - read outside the lock isn't synchronized with the write done by another thread inside the lock
    - creates a race condition for the pointer and the object pointed to
        - even if a thread sees the pointer written by another thread, it might not see the newly created instance of some_resource, resulting in the call to do_something() operating on incorrect values
    - an example of the type of race condition defined as a data race by the C++ Standard and thus specified as undefined behavior
    - C++ Standard Library provides std::once_flag and std::call_once to handle this situation
        - every thread can just use std::call_once rather than locking a mutex and explicitly checking the pointer
            - knows pointer will have been initialized by some thread in a properly synchronized fashion by the time std::call_once returns
        - typically lower overhead for use of std::call_once than using a mutex explicitly
            - especially when the initialization has already been done
```

- should be preferred when appropriate
- thread-safe lazy initialization rewritten to use std::call_once
    - initialization is done by calling a function, but it
    - could have been done with an instance of a class with a function call operator
    - std::call_once works with any function or callable object like most of the functions in the standard library that take functions or predicates as arguments
    - initialization is called exactly once
    - both the std::once_flag and data being initialized are namespace-scope objects
    - std::call_once() can easily be used for lazy initialization of class members

```cpp
std::shared_ptr<some_resource> resource_ptr;
std::once_flag resource_flag;
void init_resource()
{
    resource_ptr.reset(new some_resource);
}
void foo() {
    std::call_once(resource_flag,init_resource);
    resource_ptr->do_something();
}
```

- Listing 3.12
    - initialization is done either by the first call to send_data() or by the first call to receive_data()
    - use of the member function open_connection() to initialize the data requires that the this pointer be passed in
    - done by passing an additional argument to std::call_once() as for other functions in the Standard Library that accept callable objects such as the constructor for std::thread and std::bind()
    - NOTE: like std::mutex, std::once_flag instances can't be copied or moved
        - have to explicitly define these special member functions if they are required to be used as class members
- local variable declared with static leads to potential race condition
    - initialization of such a variable is defined to occur the first time control passes through its declaration
    - potential for a race condition to define first for multiple threads calling the function
    - pre-C++11 compilers
        - multiple threads may believe they're first and try to initialize the variable
        - threads may try to use it after initialization has started on another thread but before it's finished
    - solved for C++11
        - initialization is defined to happen on exactly one thread, and no other threads will proceed until that initialization is complete

- race condition is just over which thread gets to do the initialization rather than anything more problematic
- following can be used as an alternative to std::call_once for those cases where a single global instance is required
- initialization is guaranteed to be thread-safe, multiple threads can call get_instance safely

```
class my_class;
my_class& get_my_class_instance()
{
    static my_class instance;
    return instance;
}
```

- more general scenario is rarely updated data structure
  - protecting data only for initialization is special case
  - data structure is read-only for most of the time
    - can be read by multiple threads concurrently
  - data structure may need updating occasionally

Protecting rarely updated data structures

- for certain data structures updates are rare
  - if data structure is to be accessed from multiple threads it will need to be appropriately protected during updates to ensure that none of the threads reading the cache see a broken data structure
- such an update requires that the thread doing the update have exclusive access to the data structure until it has completed the operation
  - in the absence of a special-purpose data structure that exactly fits the desired usage and is specially designed for concurrent updates and reads
  - when update is complete the data structure is again safe for multiple threads to access concurrently
  - using a std::mutex to protect the data structure is overly pessimistic
    - will eliminate possible concurrency in reading the data structure when it isn't undergoing modification
  - need a reader-writer mutex
    - allows for exclusive access by a single "writer" thread or shared
    - allows for concurrent access by multiple "reader" threads
- C++ Standard Library doesn't provide such a mutex out of the box
  - one was proposed to the Standards Committee
- implementation provided by the Boost library based on proposal
  - doesn't solve all related problems

- performance is dependent on the number of processors involved and the relative workloads of the reader and updater threads
- important to profile the performance of the code on the target system to ensure that there's actually a benefit to the additional complexity
- boost::shared_mutex
  - std::lock_guardboost::shared_mutex and std::unique_lockboost::shared_mutex can be used for the locking for update operations
    - ensure exclusive access
  - threads that don't need to update the data structure can instead use boost::shared_lockboost::shared_mutex to obtain shared access
    - used same as std::unique_lock
      - multiple threads may have a shared lock on the same boost::shared_mutex at the same time
      - only constraint is that if any thread has a shared lock, a thread that tries to acquire an exclusive lock will block until all other threads have relinquished their locks
      - if any thread has an exclusive lock, no other thread may acquire a shared or exclusive lock until the first thread has relinquished its lock
- Listing 3.13 (std::map for DNS cache with boost::shared_mutex)
  - find_entry() uses an instance of boost::shared_lock<> to protect it for shared, read-only access
    - multiple threads can call find_entry() simultaneously without problems
  - update_or_add_entry() uses an instance of std::lock_guard<> to provide exclusive access while the table is updated
    - other threads prevented from doing updates in a call update_or_add_entry()
    - threads that call find_entry() are blocked as well

Recursive Locking

- error for a thread to try to lock a mutex it already owns with std::mutex
  - attempting to do so will result in undefined behavior
- some cases need a thread to reacquire the same mutex several times without having first released it
  - C++ Standard Library provides std::recursive_mutex
  - works like std::mutex
    - can acquire multiple locks on a single instance from the same thread
    - must release all locks before the mutex can be locked by another thread
      - lock() is called three times, must also call unlock() three times
  - correct use of std::lock_guardstd::recursive_mutex and std::unique_lockstd::recursive_mutex will handle this
- probably need to change design instead if it seems like a recursive mutex is needed in most cases

- common use of recursive mutexes is where a class is designed to be accessible from multiple threads concurrently
    - uses a mutex to protect member data
    - each public member function locks the mutex, does the work, and then unlocks the mutex
    - sometimes necessary for one public member function to call another as part of its operation
        - second member function will also try to lock the mutex leading to undefined behavior
    - can change the mutex to a recursive mutex
        - allows the mutex lock in the second member function to succeed and the function to proceed
    - such usage is not recommended
        - can lead to sloppy thinking and bad design
    - class invariants are typically broken while the lock is held
        - second member function needs to work even when called with the invariants broken
    - usually better to extract a new private member function that's called from both member functions which does not lock the mutex
        - assumes lock is already locked
    - can then carefully about the circumstances under which new function will be called and the state of the data under those circumstances

3.4 Summary

- problematic race conditions can be disastrous when sharing data between threads
    - use std::mutex and careful interface design to avoid them
- mutexes don't solve all problems
    - deadlock
- C++ Standard Library provides std::lock() to help solve mutex problems
- further techniques for avoiding deadlock
- transferring lock ownership and issues surrounding choosing the appropriate granularity for locking
- alternative data-protection facilities provided for specific scenarios
    - std::call_once()
    - boost::shared_mutex
- not covered: waiting for input from other threads
    - thread-safe stack just throws an exception if the stack is empty
        - if one thread wanted to wait for another thread to push a value on the stack it would have to repeatedly try to pop a value, retrying if an exception gets thrown

- consumes valuable processing time in performing the check without actually making any progress
- constant checking might hamper progress by preventing the other threads in the system from running
- need a way for a thread to wait for another thread to complete a task without consuming CPU time in the process

Ch. 4 - Synchronizing concurrent operations

4.1 Waiting for an event or other condition

- options if one thread is waiting for a second thread to complete a task

1. could just keep checking a flag in shared data (protected by a mutex) and have the second thread set the flag when it completes the task
   - wasteful
     - thread consumes valuable processing time repeatedly checking the flag
     - when the mutex is locked by the waiting thread, it can't be locked by any other thread
   - works against the thread doing the waiting
     - limit the resources available to the thread being waited for and even prevent it from setting the flag when finished
   - waiting thread is consuming resources that could be used by other threads in the system and may end up waiting longer than necessary
2. have the waiting thread sleep for small periods between the checks using the std::this_thread::sleep_for() function

```
bool flag;
std::mutex m;
void wait_for_flag()
{
    std::unique_lock<std::mutex> lk(m);
    while(!flag)
    {
        lk.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        lk.lock();
```

```
      }
  }
```

- function unlocks the mutex before the sleep and locks it again afterward so another thread gets a chance to acquire it and set the flag
  - thread doesn't waste processing time while it's sleeping
    - difficult to get the sleep period right
      - too short a sleep in between checks and the thread still wastes processing time checking
      - too long a sleep and the thread will keep on sleeping even when the task it's waiting for is complete introducing a delay
  - rare that this oversleeping will have a direct impact on the operation of the program
    - could mean dropped frames in a fast-paced game or overrunning a time slice in a real-time application

1. preferred option is to use the facilities from the C++ Standard Library to wait for the event itself
   - most basic mechanism for waiting for an event to be triggered by another thread (such as the presence of additional work in the pipeline mentioned previously) is the condition variable
   - condition variable is associated with some event or other condition
     - one or more threads can wait for that condition to be satisfied
     - when some thread has determined that the condition is satisfied it can then notify one or more of the threads waiting on the condition variable in order to wake them up and allow them to continue processing

Waiting for a condition with condition variables

- Standard C++ Library provides two implementations of a condition variable
  - std::condition_variable
  - std::condition_variable_any
- declared in the <condition_variable> header
- both need to work with a mutex in order to provide appropriate synchronization
  - std::condition_variable is limited to working with std::mutex
  - std::condition_variable_any can work with anything that meets some minimal criteria for being mutex-like
    - more general
    - potential for additional costs in terms of size, performance, or operating system resources
- prefer std::condition_variable

- Listing 4.1
    - queue that's used to pass the data between the two threads
        - when the data is ready, the thread preparing the data locks the mutex protecting the queue using a std::lock_guard and pushes the data onto the queue
        - calls the notify_one() member function on the std::condition_variable instance to notify the waiting thread (if there is one)
    - also have the processing thread that first locks the mutex using a std::unique_lock
    - then calls wait() on the std::condition_variable, passing in the lock object and a lambda function that expresses the condition being waited for
        - simple lambda function `[]{ return !data_queue.empty(); }` checks to see if the data_queue is not empty() i.e. there's some data in the queue ready for processing
    - wait() then checks the condition (by calling the supplied lambda function) and returns if it's satisfied (the lambda function returned true)
    - if the condition isn't satisfied (the lambda function returned false), wait() unlocks the mutex and puts the thread in a blocked or waiting state
    - when the condition variable is notified by a call to notify_one() from the data-preparation thread
        - thread wakes from sleep (unblocks)
        - reacquires the lock on the mutex
        - checks the condition again
        - returns from wait() with the mutex still locked if the condition has been satisfied
        - otherwise the thread unlocks the mutex and resumes waiting
            - why std::unique_lock is needed rather than the std::lock_guard
                - waiting thread must unlock the mutex while it's waiting and lock it again afterward (std::lock_guard can't handle this)
                - if the mutex remained locked while the thread was sleeping, the data-preparation thread wouldn't be able to lock the mutex to add an item to the queue, and the waiting thread would never be able to see its condition satisfied
    - simple lambda function for the wait which checks to see if the queue is not empty
        - any function or callable object could be passed
        - a condition variable may check the supplied condition any number of times during a call to wait()
            - always does so with the mutex locked and will return immediately if (and only if) the function provided to test the condition returns true
            - called a spurious wake when the waiting thread reacquires the mutex and checks the condition if it isn't in direct response to a notification from another thread

- isn't advisable to use a function with side effects for the condition check because number and frequency of spurious wakes are by definition indeterminate
  - side effects may occur multiple times
- flexibility to unlock a std::unique_lock can also be used once the data is available to process but before processing it
  - processing data can potentially be a time-consuming operation
  - bad idea to hold a lock on a mutex for longer than necessary
- common to use a queue to transfer data between threads
  - synchronization can be limited to the queue itself
    - greatly reduces the possible number of synchronization problems and race conditions

Building a thread-safe queue with condition variables

- look at std::queue<> container adaptor in C++ Standard Library for inspiration to design a generic queue
- Listing 4.2
- three groups of operations (ignoring the construction, assignment and swap operations)
  - queries to state of the whole queue (empty() and size())
  - queries the elements of the queue (front() and back())
  - modifications to queue (push(), pop() and emplace())
- same issues regarding race conditions inherent in the interface as for the stack
  - combine front() and pop() into a single function call
- when using a queue to pass data between threads, the receiving thread often needs to wait for the data
  - two variants on pop()
    - try_pop() - tries to pop the value from the queue but always returns immediately (with an indication of failure) even if there wasn't a value to retrieve
    - wait_and_pop() - wait until there's a value to retrieve
- Listing 4.3
- limit the constructors and eliminated assignment to simplify the code
  - two versions of both try_pop() and wait_for_pop()
    a. stores the retrieved value in the referenced variable, so it can use the return value for status
      - returns true if it retrieved a value and false otherwise
    b. returns the retrieved value directly
      - returned pointer can be set to NULL if there's no value to retrieve
- can extract the code for push() and wait_and_pop() from 4.1
- Listing 4.4
  - mutex and condition variable are now contained within the threadsafe_queue instance
    - separate variables are no longer required

- no external synchronization is required for the call to push()
- wait_and_pop() takes care of the condition variable wait
- other overload of wait_and_pop() is now trivial to write
- remaining functions can be copied almost verbatim from listing 3.5
- Listing 4.5
  - parameter to the copy constructor is a const reference
    - other threads
      - may have non-const references to the object
      - may call mutating member functions
  - empty() is a const member function
  - still need to lock the mutex
  - object must be marked mutable so it can be locked in empty() and in the copy constructor since locking a mutex is a mutating operation
- condition variables are also useful where there's more than one thread waiting for the same event
  - same structure as listing 4.1 can be used
    - if the threads are being used to divide the workload
    - only one thread should respond to a notification
  - just run multiple instances of the data—processing thread
  - when new data is ready, the call to notify_one() will trigger one of the threads currently executing wait() to check its condition and thus return from wait()
  - no guarantee which thread will be notified or even if there's a thread waiting to be notified
    - all the processing threads might be still processing data
  - several threads might be waiting for the same event and all of them need to respond
    - can happen when shared data is being initialized, and the processing threads can all use the same data but need to wait for it to be initialized
    - can happen when the threads need to wait for an update to shared data, such as a periodic reinitialization
  - thread preparing the data can call the notify_all() member function on the condition variable rather than notify_one() in these cases
    - causes all the threads currently executing wait() to check the condition they're waiting for
  - condition variable might not be the best choice of synchronization mechanisms if the waiting thread is going to wait only once
    - when the condition is true it will never wait on this condition variable again
    - especially true if the condition being waited for is the availability of a particular piece of data
  - future might be more appropriate

4.2 Waiting for one-off events with futures

- C++ Standard Library models one-off event with a future
  - if a thread needs to wait for a specific one-off event it obtains a future representing this event
  - thread can then periodically wait on the future for short periods of time to see if the event has occurred while performing some other task in between checks
  - thread can alternatively do another task until it needs the event to have happened before it can proceed and then just wait for the future to become ready
- future may have data associated with it
- future can't be reset once an event has happened (and future has become ready)
- two futures in the C++ Standard Library
  - implemented as class templates declared in the library header
  - unique futures (std::future<>)
  - shared futures (std::shared_future<>)
  - modeled after std::unique_ptr and std::shared_ptr
- instance of std::future is the one and only instance that refers to its associated event
- multiple instances of std::shared_future may refer to the same event
  - all the instances will become ready at the same time, and they may all access any data associated with the event
- associated data is the reason these are templates
  - template parameter is the type of the associated data
- std:future, std::shared_future template specializations should be used when there's no associated data
- future objects themselves don't provide synchronized accesses even though futures are used to communicate between threads
- multiple threads must protect access via a mutex or other synchronization mechanism if they need to access a single future object
  - multiple threads may each access their own copy of a std::shared_future<> without further synchronization
    - even if they all refer to the same asynchronous result
- most basic one-off events is the result of a calculation that has been run in the background
  - std::thread doesn't provide an easy means of returning a value from such a task

Returning values from background tasks

- for long-running calculation that will eventually yield a useful result but don't currently need the value

    - could start a new thread to perform the calculation
        - have to take care of transferring the result back
            - std::thread doesn't provide a direct mechanism for doing this
    - use std::async function template (in header)

- use std::async to start an asynchronous task for which result is not needed right away

    - returns a std::future object
        - will eventually hold the return value of the function
    - call get() on the future when the value is needed
        - thread blocks until the future is ready and then returns the value

- Listing 4.6

- std::async permits passing additional arguments

    - second argument should provide the object on which to apply the member function if the first argument is a pointer to a member function
        - directly
        - via a pointer
        - wrapped in std::ref
    - remaining arguments are passed as arguments to the member function
    - copies are created by moving the originals if the arguments are rvalues
        - allows the use of move-only types as both the function object and the arguments

- Listing 4.7

- up to the implementation whether

    - std::async starts a new thread
    - the task runs synchronously when the future is waited for

- can specify which option to use with an additional parameter to std::async before the function to call

    - parameter is of the type std::launch
    - std::launch::deferred
        - indicates that the function call is to be deferred until either wait() or get() is called on the future
        - NOTE: means function may never actually run

- std::launch::async
  - indicates that the function must be run on its own thread
- std::launch::deferred | std::launch::async
  - indicates that the implementation may choose
  - default

```
auto f6 = std::async(std::launch::async,Y(),1.2); // runs in a new thread
auto f7 = std::async(std::launch::deferred,baz,std::ref(x)); // runs in wait() or
auto f8 = std::async(std::launch::deferred | std::launch::async, baz, std::ref(x))
auto f9 = std::async(baz,std::ref(x)); // implementation chooses
f7.wait(); // invokes the deferred function
```

- std::async makes it easy to divide algorithms into tasks that can be run concurrently
  - not the only way to associate a std::future with a task
    - can be done by wrapping the task in an instance of the std::packaged_task<> class template
    - can be done by writing code to explicitly set the values using the std::promise<> class template
- std::packaged_task is a higher-level abstraction than std::promise

Associating a task with a future

- std::packaged_task<> ties a future to a function or callable object

  - when invoked, it calls the associated function or callable object and makes the future ready
    - return value stored as the associated data
  - can be used
    - as a building block for thread pools
    - to run each task on its own thread
    - to run all tasks sequentially on a particular background thread . If a large operation can be divided into

- each task in a large operation composed of self-contained sub-tasks can be wrapped in a std::packaged_task<> instance

  - instance can be passed to the task scheduler or thread pool

- template parameter for the std::packaged_task<> class template is a function signature

  - void() for a function taking no parameters with no return value
  - int(std::string&,double*) for a function that takes a non-const reference to a std::string and a pointer to a double and returns an int

- must pass in a function or callable object that can accept the specified parameters and that returns a type convertible to the specified return type when constructing an instance of std::packaged_task

    - types don't have to match exactly
        - can construct a std:: packaged_task<double(double)> from a function that takes an int and returns a float
            - types are implicitly convertible

- return type of the specified function signature identifies the type of the std::future<> returned from the get_future() member function

    - argument list of the function signature is used to specify the signature of the packaged task's function call operator

- Listing 4.8

- std::packaged_task object is a callable object

    - can be
        - wrapped in a std::function object
        - passed to a std::thread as the thread function
        - passed to another function that requires a callable object
        - even invoked directly

- when invoked as a function object

    - arguments supplied to the function call operator are passed on to the contained function
    - return value is stored as the asynchronous result in the std::future obtained from get_future()

- i.e. can wrap a task in a std::packaged_task and retrieve the future before passing the std::packaged_task object elsewhere to be invoked later

    - can wait for the future to become ready when the result is needed

Passing tasks between threads

- std::packaged_task provides a way of sending messages between threads
- Listing 4.9
- loop until a message has been received
    - repeatedly poll for messages to handle (events or tasks on a queue)
    - loop again if no tasks on the queue
    - otherwise
        - take task from queue
        - release the lock on the queue

- run task
    - the future associated with the task will then be made ready when the task completes
- to post a task on the queue
    - create a new packaged task from the supplied function
    - obtain the future from that task by calling get_future()
    - put the task on the list before the future is returned to the caller
- the code that posted the original message can
    - wait for the future if it needs to know that the task has been completed
    - can discard the future if it doesn't need to know

Making (std::) promises

- bad to use separate threads for operations that are going to increase to a very large number (e.g. handling connections)
    - large numbers of threads consequently consume large numbers of operating system resources and potentially cause a lot of context switching
    - operating system may run out of resources for running new threads before its capacity is exhausted (in extreme case)
    - case for a thread pool
        - results in other parts of the application waiting either for data to be successfully sent or received
- std::promise provides a means of setting a value (of type T), which can later be read through an associated std::future object
    - std::promise/std::future pair
        - waiting thread can block on the future
        - thread providing the data could use the promise half of the pairing to set the associated value and make the future ready
- can obtain the std::future object associated with a given std::promise by calling the get_future() member function
    - future becomes ready and can be used to retrieve the stored value when the value of the promise is set (using the set_value() member function)
    - an exception is stored instead if std::promise is destroyed without setting a value
- Listing 4.10
- need to handle exceptions
    - code could just report an error with an exception if performing the operation in the thread that needed the result
        - unnecessarily restrictive to require that everything go well to use a std::packaged_task or a std::promise

Saving an exception for the future

```
double square_root(double x)
{
    if(x < 0) {
        throw std::out_of_range("x<0");
    }
    return sqrt(x);
}
```

- async call

```
std::future<double> f=std::async(square_root,-1);
double y = f.get();
```

- if the function call invoked as part of std::async throws an exception
    - exception is stored in the future in place of a stored value
    - future becomes ready
    - call to get() rethrows that stored exception
- NOTE: standard leaves it unspecified whether it is the original exception object that's rethrown or a copy
- same happens if function is wrapped in a std::packaged_task
    - if the wrapped function throws an exception when the task is invoked
        - exception is stored in the future in place of the result
        - call to get() rethrows that stored exception
- std::promise provides the same facility with an explicit function call
    - call the set_exception() member function rather than set_value() to store an exception rather than a value
    - typically used in a catch block for an thrown exception

```
extern std::promise<double> some_promise;
try {
    some_promise.set_value(calculate_value());
} catch(...) {
    some_promise.set_exception(std::current_exception());
}
```

- uses std::current_exception() to retrieve the thrown exception
    - alternative would be to use std::copy_exception() to store a new exception directly without throwing

```
some_promise.set_exception(std::copy_exception(std::logic_error("foo ")));
```

- much cleaner than using a try/catch block if the type of the exception is known
    - prefer this (simplifies and compiler can optimize)
- another way to store an exception in a future is to destroy the std::promise or std::packaged_task associated with the future without calling either of the set functions on the promise or invoking the packaged task
    - destructor of the std::promise or std::packaged_task will store a std::future_error exception with an error code of std::future_errc::broken_promise in the associated state if the future isn't already ready
    - make a promise to provide a value or exception by creating a future
    - promise is borken by destroying the source of that value or exception without providing one
        - waiting threads could potentially wait forever if the compiler didn't store anything in the future in this case
- std::future has limitations
    - only one thread can wait for the result
- use std::shared_future instead to wait for the same event from more than one thread

Waiting from multiple threads

- calls to the member functions of a particular std::future instance are not synchronized with each other even though std::future handles all the synchronization necessary to transfer data from one thread to another
    - accessing a single std::future object from multiple threads without additional synchronization results in a data race and undefined behavior
    - std::future models unique ownership of the asynchronous result
    - one-shot nature of get() makes such concurrent access pointless
        - only one thread can retrieve the value
- std::shared_future instances are copyable
    - can have multiple objects referring to the same associated state
    - std::future is only moveable
        - ownership can be transferred between instances
        - only one instance refers to a particular asynchronous result at a time
- member functions of std::shared_future on an individual object are still unsynchronized
    - protect accesses with a lock to avoid data races when accessing a single object from multiple threads
- preferred way to use std::shared_future is to take a copy of the object
    - each thread can then access its own copy
    - accesses to shared asynchronous state from multiple threads are safe if each thread accesses that state through its own std::shared_future object

- potential use of std::shared_future
  - implementing parallel execution of large matrix or set of cells
    - each cell has a single final value
    - final value may be used for final values of other cells
    - formulas for calculating the results of the dependent cells can use a std::shared_future to reference the first cell
  - tasks that can proceed to completion will do so if all the formulas for the individual cells are then executed in parallel
    - those that depend on others will block until their dependencies are ready
    - will allow the system to make maximum use of the available hardware concurrency
- std::shared_future instances that reference some asynchronous state are constructed from instances of std::future that reference that state
  - std::future objects don't share ownership of the asynchronous state with any other object
  - ownership must be transferred into the std::shared_future using std::move
    - leaves std::future in an empty state

```
std::promise<int> p;
std::future<int> f(p.get_future());
assert(f.valid());
std::shared_future<int> sf(std::move(f));
assert(!f.valid());
assert(sf.valid());
```

- future f is initially valid because it refers to the asynchronous state of the promise
  - f is no longer valid after transferring the state to sf
    - sf is d
- transfer of ownership is implicit for rvalues
  - can construct a std::shared_future directly from the return value of the get_future() member function of a std::promise object

```
std::promise<std::string> p;
std::shared_future<std::string> sf(p.get_future()); // implicit transfer of owners
```

- transfer of ownership is implicit
  - std::shared_future<> is constructed from an rvalue of type std::futurestd::string
  - std::future also has an additional feature to facilitate the use of std::shared_future with the new facility for automatically deducing the type of a variable from its initializer
  - std::future has a share() member function that creates a new std::shared_future and transfers ownership to it directly

```
std::promise< std::map< SomeIndexType, SomeDataType, SomeComparator, SomeAllocator
auto sf=p.get_future().share();
```

- type of sf is deduced to be `std::shared_future<std::map<SomeIndexType, SomeDataType, SomeComparator, SomeAllocator>::iterator>`
    - only need to change the type of the promise if the comparator or allocator is changed
        - type of the future is automatically updated to match
- may want to limit the amount of time of waiting for an event
    - may have a hard time limit
    - may have other useful work that the thread can be doing if the event isn't going to happen soon

4.3 Waiting with a time limit

- blocking calls will block indefinitely
    - suspend the thread until event occurs
- may want to limit time to wait
    - to send heartbeats
    - respond to user
    - to abort if requested/necessary
- two types of timeouts
    - duration-based timeout
        - wait for a specific amount of time
    - absolute timeout
        - wait until a specific point in time
- most of the waiting functions provide variants that handle both forms of timeouts
    - duration-based timeouts have a _for suffix
    - absolute timeouts have a _until suffix
- e.g. std::condition_variable
    - two overloads of wait_for()
    - two overloads of wait_until()
    - one overload just waits until signaled, or the timeout expires, or a spurious wakeup occurs
    - other will check a supplied predicate when woken and will return only when the supplied predicate is true or the timeout expires

Clocks

- clock is a source of time information in C++ Standard
    - time now
    - type of the value used to represent the times obtained from the clock

- tick period of the clock
- whether or not the clock ticks at a uniform rate and is thus considered to be a steady clock
- current time of a clock can be obtained by calling the static member function now()
  - e.g. std::chrono::system_clock::now()
- type of the time points for a particular clock is specified by the time_point member typedef
  - return type of some_clock::now() is some_clock::time_point
- tick period of the clock is specified as a fractional number of seconds
  - given by the period member typedef of the clock
  - clock that ticks 25 times per second has a period of std::ratio<1,25>
  - clock that ticks every 2.5 seconds has a period of std::ratio<5,2>
  - period may be specified as the average tick period, smallest possible tick period, or some other value that the library writer deems appropriate if the tick period of a clock can't be known until runtime or may vary during a given run of the application
  - no guarantee that the observed tick period in a given run of the program matches the specified period for that clock
- clock is said to be a steady clock if it ticks at a uniform rate and can't be adjusted
  - (regardless of if rate matches the period)
  - is_steady static data member of the clock class is true if the clock is steady
  - std::chrono::system_clock will not be steady, because the clock can be adjusted, even if such adjustment is done automatically to take account of local clock drift
    - clock adjustment may cause a call to now() to return a value earlier than that returned by a prior call to now()
    - violates requirement for a uniform tick rate
  - steady clocks are important for timeout calculations
    - std::chrono::steady_clock
- other clocks
  - std::chrono::system_clock
    - represents the "real time" clock of the system
    - provides functions for converting its time points to and from time_t values
  - std::chrono::high_resolution_clock
    - provides smallest possible tick period (highest possible resolution) of all the library-supplied clocks
    - may actually be a typedef to one of the other clocks
- defined in the library header

Durations

- simplest part of the time support
  - std:: chrono::duration<> class template
  - first template parameter is the type of the representation (such as int, long, or double)

- second is a fraction specifying how many seconds each unit of the duration represents
  - e.g. a number of minutes stored in a short is std::chrono::duration<short,std::ratio<60,1>>
    - 60 seconds in a minute
  - e.g. count of milliseconds stored in a double is std::chrono::duration<double,std::ratio <1,1000>>
    - millisecond is 1/1000 of a second
- all the C++ time-handling facilities used by the Thread Library are in the std::chrono namespace
- predefined typedefs in the std::chrono namespace for various durations: nanoseconds, microseconds, milliseconds, seconds, minutes, and hours
  - all use a sufficiently large integral type for the representation chosen such that a duration of over 500 years in the appropriate units can be represented
- also typedefs for all the SI ratios from std::atto (10–18) to std::exa (1018)
  - (more, if platform has 128-bit integer types)
  - use when specifying custom durations such as std::duration<double,std::centi> for a count of 1/100 of a second represented in a double
- conversion between durations is implicit where it does not require truncation of the value
  - converting hours to seconds is OK
  - converting seconds to hours is not
- explicit conversions can be done with std::chrono::duration_cast<>

```
std::chrono::milliseconds ms(54802);
std::chrono::seconds s = std::chrono::duration_cast<std::chrono::seconds>(ms);
```

- result is truncated rather than rounded
  - s will have a value of 54
- durations support arithmetic
  - can add and subtract durations to get new durations
  - multiply or divide by a constant of the underlying representation type (the first template parameter)
  - e.g. `5 * seconds(1)` same as `seconds(5)` or `minutes(1) - seconds(55)`
- count of the number of units in the duration can be obtained with the count() member function
  - `std::chrono::milliseconds(1234).count()` is 1234
- duration-based waits are done with instances of std::chrono::duration<>
  - can wait for up to 35 milliseconds for a future to be ready

```
std::future<int> f=std::async(some_task);
if (f.wait_for(std::chrono::milliseconds(35)) == std::future_status::ready) {
```

```
        do_something_with(f.get());
}
```

- wait functions all return a status to indicate whether the wait timed out or the waitedfor event occurred
  - waiting for a future
    - returns std::future_status::timeout if the wait times out
    - returns std::future_status::ready if the future is ready
    - returns std::future_status::deferred if the future's task is deferred
- time for a duration-based wait is measured using a steady clock internal to the library
  - 35 milliseconds means 35 milliseconds of elapsed time
    - even if the system clock was adjusted (forward or back) during the wait
  - vagaries of system scheduling and the varying precisions of OS clocks means that the actual time between the thread issuing the call and returning from it may be much longer than 35 ms

Time points

- time point for a clock is represented by an instance of the std::chrono::time_point<> class template
  - specifies which clock it refers to as the first template parameter and the units of measurement (a specialization of std::chrono::duration<>) as the second template parameter
  - value of a time point is the length of time (in multiples of the specified duration) since a specific point in time called the epoch of the clock
- epoch of a clock is a basic property but not something that's directly available to query or specified by the C++ Standard
- typical epochs include 00:00 on January 1, 1970 and the instant when the computer running the application booted up
- clocks may share an epoch or have independent epochs
  - time_point typedef in one class may specify the other as the clock type associated with the time_point if two clocks share an epoch
  - can get the time_since_epoch() for a given time_point but can't find out when the epoch is
    - returns a duration value specifying the length of time since the clock epoch to that particular time point
- e.g. `std::chrono::time_point<std::chrono::system_clock, std::chrono::minutes>`
  - holds the time relative to the system clock but measured in minutes as opposed to the native precision of the system clock (which is typically seconds or less)

- can add durations and subtract durations from instances of std::chrono::time_point<> to produce new time points
  - `std::chrono::high_resolution_clock::now() + std::chrono::nanoseconds(500)`

    - returns a time 500 nanoseconds in the future
  - good for calculating an absolute timeout when the maximum duration of a block of code is known
  - multiple calls to waiting functions within it or nonwaiting functions that precede a waiting function but take up some of the time budget
- can also subtract one time point from another that shares the same clock
  - result is a duration specifying the length of time between the two time points
  - useful for timing blocks of code

```
auto start = std::chrono::high_resolution_clock::now();
do_something();
auto stop = std::chrono::high_resolution_clock::now();
std::cout << "do_something() took " << std::chrono::duration<double,std::chrono::s
          << " seconds\n";
```

- clock parameter of a std::chrono::time_point<> instance does more than just specify the epoch
- when passing the time point to a wait function that takes an absolute timeout, the clock parameter of the time point is used to measure the time
- has important consequences when the clock is changed
  - wait tracks the clock change and won't return until the clock's now() function returns a value later than the specified timeout
  - may reduce the total length of the wait (as measured by a steady clock) if the clock is adjusted forward
  - may increase the total length of the wait if it's adjusted backward
- time points are used with the _until variants of the wait functions
  - typical use case is as an offset from some-clock::now() at a fixed point in the program
  - time points associated with the system clock can be obtained by converting from a time_t using the std::chrono::system_clock::to_time_point() static member function for scheduling operations at a user-visible time
- Listing 4.11
- recommended way to wait for condition variables with a time limit
  - if not passing a predicate to the wait
  - overall length of the loop is bounded
  - need to loop when using condition variables if the predicate isn't passed in to handle spurious wakeups

- might end up waiting almost the full length of time before a spurious wake, and the next time through the wait time starts again if using wait_for()
  - may repeat any number of times, making the total wait time unbounded

Functions that accept timeouts

- add a delay to the processing of a particular thread
  - doesn't take processing time away from other threads when it has nothing to do
  - two functions that handle this are std::this_thread::sleep_for() and std::this_thread::sleep_until()
  - work like a basic alarm clock
    - thread goes to sleep either for the specified duration (with sleep_for()) or until the specified point in time (with sleep_until())
    - sleep_for() makes sense for cases where something must be done periodically and the elapsed time is what matters
    - sleep_until() allows for scheduling the thread to wake at a particular point in time
- sleeping isn't the only facility that takes a timeout
  - can use timeouts with condition variables and futures
  - can use timeouts when trying to acquire a lock on a mutex if the mutex supports it
- std::mutex and std::recursive_mutex don't support timeouts on locking
- use std::timed_mutex or std::recursive_timed_mutex
  - support try_lock_for() and try_lock_until() member functions that try to obtain the lock within a specified time period or before a specified time point
- Table 4.1

4.4 Using synchronization of operations to simplify code

- synchronization facilities
  - allow a more functional (in the sense of functional programming) approach to programming concurrency
    - each task can be provided with the data it needs rather than sharing data directly between threads
    - result can be passed to any other threads that need it through the use of futures

Functional programming with futures

- focus on idea that if function a function is invoked twice with the same parameters, the result is exactly the same
- pure function doesn't modify any external state
- simplifies reasoning especially when concurrency is involved
  - removes shared memory == no race conditions and no need to protect shared data with mutexes either

- when most functions are pure impure functions that actually do modify the shared state stand out more
  - easier to reason about how they fit into the overall structure of the application
- C++ is a multiparadigm language
  - possible to write programs in a functional style
    - lambda functions
    - std::bind from Boost and TR1
    - automatic type deduction for variables
    - futures
      - future can be passed around between threads to allow the result of one computation to depend on the result of another without any explicit access to shared data

FP-style quicksort

- given a list of values
  - take an element to be the pivot element
  - partition the list into two sets
    - less than the pivot
    - greater than or equal to the pivot
  - sorted copy of the list is obtained by sorting the two sets and returning the sorted list of values less than the pivot, followed by the pivot, followed by the sorted list of values greater than or equal to the pivot
- FP-style sequential implementation is shown
  - takes and returns a list by value rather than sorting in place like std::sort() does
- Listing 4.12
- interface is FP-style
  - using FP-style throughout leads to a lot of copying
  - use imperative style for the internals
  - take the first element as the pivot by slicing it off the front of the list using splice()
    - can potentially result in a suboptimal sort (in terms of numbers of comparisons and exchanges)
    - doing anything with a std::list can be time consuming because of the list traversal
  - splice it directly into the list because it is in result
  - also use it for comparisons
    - take a reference to it to avoid copying
  - then use std::partition to divide the sequence into those values less than the pivot and those not less than the pivot
    - easiest way to specify the partition criteria is to use a lambda function
    - use a reference capture to avoid copying the pivot value
    - std::partition() rearranges the list in place and returns an iterator marking the first element that's not less than the pivot value

- if using recursion to sort the two "halves" need to create two lists
    - can do this by using splice() again to move the values from input up to the divide_point into a new list
    - leaves the remaining values alone in input
- then sort the two lists with recursive calls
    - use std::move() to pass the lists in
    - can avoid copying by using std::move9) to pass the lists in
- can use splice() again to piece the result together in the right order

FP-style parallel quicksort

- same operations as before except that some of them now run in parallel
- Listing 4.13
- sort lower portion on another thread using std::async() rather than current thread
- upper portion of the list is sorted with direct recursion as before
- can take advantage of the available hardware concurrency by recursively calling parallel_quick_sort()
    - recursing down three times if std::async() starts a new thread every time leads to 8 running threads
    - recurse down 10 times (for ~1000 elements) leads to 1024 threads running if the hardware can handle it
        - if the library decides there are too many spawned tasks (perhaps because the number of tasks has exceeded the available hardware concurrency), it may switch to spawning the new tasks synchronously
        - will run in the thread that calls get() rather than on a new thread
            - avoids the overhead of passing the task to another thread when this won't help the performance
    - NOTE: conformant to standard for an implementation of std::async to start a new thread for each task (even in the face of massive oversubscription) unless std::launch::deferred is explicitly specified or to run all tasks synchronously unless std::launch::async is explicitly specified
        - when relying on the library for automatic scaling best to check the documentation for current implementation to see what behavior it exhibits
- can also write custom spawn_task() function as a simple wrapper around std::packaged_task and std::thread rather than using std::async()
    - create a std::packaged_task for the result of the function call
    - get the future from it
    - run it on a thread
    - return the future
    - doesn't offer much advantage and would likely lead to massive oversubcription
    - sets up abilitiy to migrate to a more sophisticated implementation that adds the task to a queue to be run by a pool of worker threads

- worth it over using std::async only if programmer really knows what they're doing and want complete control over the way the thread pool is built and executes tasks
- can just splice new_higher because of direct recursion to get new_higher
    - new_lower is now a std::future<std::list> rather than just a list
        - need to call get() to retrieve the value before splice() can be called
        - waits for the background task to complete and moves the result into the splice() call
            - get() returns an rvalue reference to the contained result
            - can be moved out
- still isn't an ideal parallel implementation of Quicksort even assuming that std::async() makes optimal use of the available hardware concurrency
    - std::partition does a lot of the work but is still sequential
    - check the academic literature for fastest possible parallel implementation
- Listing 4.14
- another paradigm is CSP (Communicating Sequential Processes) where threads are conceptually entirely separate
    - no shared data but with communication channels that allow messages to be passed between them
    - Erlang (http://www.erlang.org/) and by the MPI (Message Passing Interface) (http://www.mpi-forum.org/) environment commonly used for high-performance computing in C and C++

Synchronizing operations with message passing

- idea of CSP
    - no shared data
    - each thread can be reasoned about entirely independently
    - consider how it behaves in response to the messages that it received
- each thread is a state machine
    - when it receives a message, it updates its state in some manner and maybe sends one or more messages to other threads, with the processing performed depending on the initial state
- can formalize this and implement a Finite State Machine model
- state machine can be implicit in the structure of the application
- separation into independent processes has the potential to remove much of the complication from shared-data concurrency and therefore make programming easier
- true communicating sequential processes have no shared data
    - all communication passed through message queues
    - not possible to enforce this requirement because C++ threads share an address space
- requires discipline as application or library authors
    - ensure that we don't share data between the threads

- message queues must be shared in order for the threads to communicate
    - details can be wrapped in the library
- state machines
    - in each state the thread waits for an acceptable message, which it then processes
    - may result in transitioning to a new state, and the cycle continues
    - can implement collected states with a class that has a member function to represent each state
    - each member function can then wait for specific sets of incoming messages and handle them when they arrive, possibly triggering a switch to another state
    - each distinct message type is represented by a separate struct
- all the necessary synchronization for the message passing is entirely hidden inside the message-passing library (see appendix C)
- Listing 4.15
- implementation described here is simplified from the real logic
    - no need to think about synchronization and concurrency issues, just which messages may be received at any given point and which messages to send
    - state machine runs on a single thread
    - other parts of the system run on separate threads
- style of program design is called the Actor model
    - several discrete actors in the system (each running on a separate thread)
    - send messages to each other to perform the task at hand
    - no shared state except that directly passed via messages
- NOTE: CSP is distinct from the Actor model in that it formalizes and abstracts the message channels
- Listing 4.16
- simplifies the task of designing a concurrent system
    - each thread can be treated entirely independently
- example of using multiple threads to separate concerns and as such requires explicitly deciding how to divide the tasks between threads

4.5 Summary

- synchronizing operations between threads is an important part of writing an application that uses concurrency
    - no synchronization = threads are essentially independent and might as well be written as separate applications that are run as a group because of their related activities
- methods for synchronization
    - basic condition variables
    - futures
    - promises

- packaged tasks
- ways of approaching synchronization issues
  - functional-style programming
    - each task produces a result entirely dependent on its input rather than on the external environment
  - message passing
    - communication between threads is via asynchronous messages sent through a messaging subsystem that acts as an intermediary

Ch. 5 - The C++ memory model and operations on atomic types

- new (as of C++11) multithreading-aware memory model is one of the most important added features
  - defines exactly how the fundamental building blocks work for reliability
- do not need details if using mutexes to protect data and condition variables or futures to signal events
- only when getting "close to the machine" that precise details of the memory model matter
- C++ is a systems programming language at its core
  - goal - no need for a lower-level language
- programmers should be provided with enough flexibility within C++ to do whatever they need without the language getting in the way
  - atomic types and operations allow just that, providing facilities for low-level synchronization operations that will commonly reduce to one or two CPU instructions

5.1 Memory model basics

- two components to the memory model
  - basic structural aspects
    - relate to how data is laid out in memory
  - concurrency aspects
- structural aspects are important for concurrency
  - especially low-level atomic operations
- objects and memory locations

Objects and memory locations

- data in a C++ program is objects
  - fundamental or primitive types do not function like objects in Smalltalk, e.g.

- C++ Standard defines an object as "a region of storage"
  - assigns properties to these objects
    - type
    - lifetime
  - some are fundamental types e.g. int or float
  - others are instances of user-defined classes
- some objects (e.g. arrays, instances of derived classes, and instances of classes with non-static data members) have subobjects, some don't
- object is stored in one or more memory locations
  - each memory location is either an object (or subobject), a scalar type, or a sequence of adjacent bit fields
  - NOTE: although adjacent bit fields are distinct objects, they're still counted as the same memory location
- example of struct
  - entire struct is one object
  - consists of several subobjects, one for each data member
  - bit fields share a memory location
  - std::string object consists of several memory locations internally
  - each member has its own memory location
  - zero-length bit field separates next bit field into its own memory location
- to note
  - every variable is an object, including those that are members of other objects
  - every object occupies at least one memory location
  - variables of fundamental type such as int or char are exactly one memory location, whatever their size, even if they're adjacent or part of an array
  - adjacent bit fields are part of the same memory location

Objects, memory locations, and concurrency

- everything hinges on memory locations for multithreaded applications in C++
  - if two threads access separate memory locations everything works fine
  - if two threads access the same memory location
    - OK if neither thread is updating the memory location
      - read-only data doesn't need protection or synchronization
    - there's a potential for a race condition if either thread is modifying the data
      - has to be an enforced ordering between the accesses in the two threads to avoid the race condition
      - can use mutexes
        - if the same mutex is locked prior to both accesses, only one thread can access the memory location at a time, so one must happen before the other

- can use the synchronization properties of atomic operations either on the same or other memory locations to enforce an ordering between the accesses in the two threads use of atomic operations to enforce an ordering is described in section 5.3.
- if more than two threads access the same memory location, each pair of accesses must have a defined ordering
    - with no enforced ordering between two accesses to a single memory location from separate threads
        - one or both of those accesses is not atomic
        - one or both is a write
        - results in a data race and causes undefined behavior
- according to the language standard, once an application contains any undefined behavior there are no guarantees for program functionality
- can avoid undefined behavior by using atomic operations to access the memory location involved in a race
    - doesn't prevent the race itself
    - which atomic operation touches memory location first is still not specified
    - does ensure no undefined behavior is encountered

Modification orders

- every object in a C++ program has a defined modification order composed of all the writes to that object from all threads in the program, starting with the object's initialization
    - in most cases order will vary between runs
    - in any given execution of the program all threads in the system must agree on an order
    - programmer is responsible for making certain that there's sufficient synchronization to ensure that threads agree on the modification order of each variable if object is not an atomic type
    - have a data race and undefined behavior if different threads see distinct sequences of values for a single variable
- compiler is responsible for ensuring synchronization when using atomics
    - certain kinds of speculative execution aren't permitted
        - once a thread has seen a particular entry in the modification order
            - subsequent reads from that thread must return later values
            - subsequent writes from that thread to that object must occur later in the modification order
            - a read of an object that follows a write to that object in the same thread must either return the value written or another value that occurs later in the modification order of that object
    - all threads must agree on the modification orders of each indi-vidual object in a program
    - don't necessarily have to agree on the relative order of operations on separate objects

5.2 Atomic operations and types in C++

- atomic operation == indivisible operation
  - can't observe an operation half-done from any thread in the system
- load will retrieve either the initial value of the object or the value stored by one of the modifications for a load where all modifications are atomic
- nonatomic operation might be seen as half-done by another thread
  - for a store the value observed by another thread might be neither the value before the store nor the value stored but something else
  - for a load, it might retrieve part of the object, have another thread modify the value, and then retrieve
  - problematic race condition but it may constitute a data race and cause undefined behavior

The standard atomic types

- header
- all operations on such types are atomic
- only operations on these types are atomic in the sense of the language definition
  - can use mutexes to make other operations appear atomic
- standard atomic types themselves might use such emulation
  - they (almost) all have an is_lock_free() member function
    - allows the user to determine whether operations on a given type are done directly with atomic instructions or done by using a lock internal to the compiler and library
- only type that doesn't provide an is_lock_free() member function is std::atomic_flag
  - simple Boolean flag
  - operations are required to be lock-free
  - can use that to implement a simple lock and thus implement all the other atomic types using that as a basis
  - objects of type std::atomic_flag are initialized to clear
  - can then either be queried and set (with the test_and_set() member function) or cleared (with the clear() member function)
  - no assignment
  - no copy construction
  - no test and clear
  - no other operations
- remaining atomic types are all accessed through specializations of the std::atomic<> class template
  - more full-featured but may not be lock-free

- on most popular platforms it's expected that the atomic variants of all the built-in types are lock-free
  - not required
  - interface of each specialization reflects the properties of the type
- can also use the set of names to refer to the implementation-supplied atomic types
- alternative type names may refer either to the corresponding std::atomic<> specialization or to a base class of that specialization (due to history)
  - mixing these alternative names with direct naming of std::atomic<> specializations in the same program can lead to nonportable code
- table 5.1
- C++ Standard Library also provides a set of typedefs for the atomic types corresponding to the various nonatomic Standard Library typedefs such as std::size_t
- pattern
  - for a standard typedef T, the corresponding atomic type is the same name with an atomic_ prefix
  - for built-in types
    - signed -> s
    - unsigned -> u
    - long long -> llong
- table 5.2
- generally simpler to use std::atomic
- atomic types are not copyable or assignable
  - no copy constructors or copy assignment operators
  - support assignment from and implicit conversion to the corresponding built-in types
  - have load() and store(), exchange(), compare_exchange_weak(), and compare_exchange_strong() member functions
  - support the compound assignment operators where appropriate: +=, -=, *=, |=,
  - integral types and std::atomic<> specializations for pointers support ++ and --
  - operators also have corresponding named member functions with the same functionality
    - fetch_add()
    - fetch_or()
    - etc.
  - return value from the assignment operators and member functions is either the value stored (in the case of the assignment operators) or the value prior to the operation (in the case of the named functions)
    - avoids potential problems from such assignment operators returning a reference to the object being assigned to
- std::atomic<> has a primary template that can be used to create an atomic variant of a user-defined type
  - operations are limited to
    - load()

- store()
- assignment from and conversion to the user-defined type
- exchange()
- compare_exchange_weak
- compare_exchange_strong().
- each of the operations has an optional memory-ordering argument
  - used to specify the required memory-ordering semantics
  - store operations
    - memory_order_relaxed
    - memory_order_release
    - memory_order_seq_cst
  - load operations
    - memory_order_relaxed
    - memory_order_consume
    - memory_order_acquire
    - memory_order_seq_cst
  - read-modify-write operations
    - memory_order_relaxed
    - memory_ order_consume
    - memory_order_acquire
    - memory_order_release
    - memory_order_acq_rel
    - memory_order_seq_cst
- default is memory_order_seq_cst

Operations on std::atomic_flag

- can be in one of two states
  - set
  - clear
- intended as a building block only
- only use under very special circumstances
- must be initialized with ATOMIC_FLAG_INIT
  - initializes the flag to a clear state
  - applies wherever the object is declared to whatever scope it has
  - only atomic type to require such special treatment for initialization
    - guaranteed to be lock-free
- guaranteed to be statically initialized if it has static storage duration
  - no initialization-order issues - will always be initialized by the time of the first operation on the flag

- only three things to do with it
  - destroy it - destructor
  - clear it - clear() member function
  - set it and query the previous value - test_and_set() member function
  - clear() and test_and_set() member functions can have a memory order specified
- can't copy-construct another std::atomic_flag object from the first
- can't assign one std::atomic_flag to another
- true for all the atomic types
  - operations on an atomic type are defined as atomic
  - assignment and copy-construction involve two objects
    - single operation on two distinct objects can't be atomic
- std::atomic_flag ideally suited to use as a spinlock mutex
  - flag is clear and the mutex is unlocked
  - loop on test_and_set() until the old value is false to lock the mutex
    - indicates that current thread set the value to true
  - clear the flag to unlock the mutex
  - can be used with std::lock_guard<>
- Listing 5.1
- does a busy-wait in lock()
  - bad choice if there is going to be any degree of contention
  - enough to ensure mutual exclusion
- can't be used as a general Boolean flag
  - doesn't have a simple nonmodifying query operation
  - better off using std::atomic

Operations on std::atomic

- more full-featured Boolean flag than std::atomic_flag
  - not copy-constructible or copy-assignable
  - can construct it from a nonatomic bool
  - can assign to instances of std::atomic from a nonatomic bool
- NOTE: the assignment operator from a nonatomic bool differs from the general convention of returning a reference to the object it's assigned to
  - returns a bool with the value assigned instead
- common pattern with std::atomic<> types
  - assignment operators return values (of the corresponding nonatomic type) rather than references
- writes (of either true or false) are done by calling store()
  - memory-order semantics can be specified
  - test_and_set() -> more general exchange() member function
    - allows replacement of the stored value and atomically retrieves the original value

- std::atomic also supports a plain nonmodifying query of the value with an implicit conversion to plain bool or with an explicit call to load()
- store() is a store operation
- load() is a load operation
- exchange() is a read-modify-write operation
- std::atomic introduces another read-modify-write operation to store a new value if the current value is equal to an expected value

Storing a new value (or not) depending on the current value

- compare/exchange operation
    - compare_exchange_weak()
    - compare_exchange_strong()
    - main component of programming with atomic types
        - compares the value of the atomic variable with a supplied expected value and stores the supplied desired value if they're equal
        - expected value is updated with the actual value of the atomic variable if the values aren't equal
        - return type of the compare/exchange functions is a bool, which is true if the store was performed and false otherwise
- store might not be successful even if the original value was equal to the expected value for compare_exchange_weak()
    - value of the variable is unchanged and the return value of compare_exchange_weak() is false
    - most likely to happen on machines that lack a single compare-and-exchange instruction
        - processor can't guarantee that the operation has been done atomically
        - thread performing the operation might be switched out in the middle of the necessary sequence of instructions and another thread scheduled in its place
        - called a spurious failure
- compare_exchange_weak() is typically be used in a loop to handle spurious failures

```
bool expected = false;
extern atomic<bool> b; // set somewhere else
while (!b.compare_exchange_weak(expected,true) && !expected);
```

- looping as long as flag is still false
    - flag indicates that the compare_exchange_weak() call failed spuriously
- compare_exchange_strong() is guaranteed to return false only if the actual value wasn't equal to the expected value
    - can eliminate the need for loops like the one shown where you just want to know whether you successfully changed a variable or whether another thread got there first

- to change the variable regardless of the initial value is update of expected becomes useful
  - possibly with an updated value that depends on the current value
  - each time through the loop, expected is reloaded
  - compare_exchange_weak() or compare_exchange_strong() call should be successful the next time around the loop so if no other thread modifies the value in the meantime
- may be beneficial to use compare_exchange_weak() in order to avoid a double loop on platforms where compare_exchange_weak() can fail spuriously (and so compare_exchange_strong() contains a loop) if the calculation of the value to be stored is simple
- may make sense to use compare_exchange_strong() to avoid having to recalculate the value to store when the expected value hasn't changed if the calculation of the value to be stored is itself time consuming
- can make a difference for the larger atomic types but not so important for std::atomic
- can take two memory-ordering parameters
  - allows for the memory-ordering semantics to differ in the case of success and failure
  - might be desirable for a successful call to have memory_order_acq_rel semantics whereas a failed call has memory_order_relaxed semantics
- failed compare/exchange doesn't do a store, so it can't have memory_ order_release or memory_order_acq_rel semantics
  - not permitted to supply these values as the ordering for failure
- can't supply stricter memory ordering for failure than for success
  - must specify memory_order_acquire or memory_order_seq_cst for success to use either semantics for failure
- failure ordering assumed to be the same as that for success if not specified
  - release part of the ordering is stripped
    - memory_order_ release becomes memory_order_relaxed
    - memory_order_acq_rel becomes memory_order_acquire
  - default to memory_order_seq_cst as usual
    - provides the full sequential ordering for both success and failure
- std::atomic may not be lock-free (unlike std::atomic_flag)
  - implementation may have to acquire a mutex internally in order to ensure the atomicity of the operations
  - can use the is_lock_free() member function to check whether operations on std::atomic are lock-free

Operations on std::atomic<T*>: pointer arithmetic

- atomic form of a pointer to some type T is std::atomic<T*>
  - operates on values of the corresponding pointer type rather than bool values
  - neither copy-constructible nor copy-assignable

- can be both constructed and assigned from the suitable pointer values
  - is_lock_free()
  - load()
  - store()
  - exchange()
  - compare_exchange_weak()
  - compare_exchange_strong()
  - take and return T* rather than bool
- new operations provided by std::atomic<T*> are the pointer arithmetic operations
- basic operations are provided by the fetch_add() and fetch_sub() member functions
  - do atomic addition and subtraction on the stored address
  - operators += and -=, and both pre- and post-increment and decrement with ++ and --
  - operators work just as expected
  - fetch_add() and fetch_sub() are slightly different in that they return the original value
    - operation is also known as exchange-and-add
    - it's an atomic read-modify-write operation
    - return value is a plain T* value rather than a reference to the std::atomic<T*> object

```
class Foo{};
Foo some_array[5];
std::atomic<Foo*> p(some_array);
Foo* x=p.fetch_add(2);
assert(x==some_array);
assert(p.load()==&some_array[2]);
x=(p-=1);
assert(x==&some_array[1]);
assert(p.load()==&some_array[1]);
```

- function forms also allow the memory-ordering semantics to be specified as an additional function call argument

```
p.fetch_add(3,std::memory_order_release);
```

- fetch_add() and fetch_sub() can have any of the memory-ordering tags and can participate in a release sequence because both are read-modify-write operations
- specifying the ordering semantics isn't possible for the operator forms
  - no way of providing the information
  - always have memory_order_seq_cst semantics

Operations on standard atomic integral types

- atomic integral types
  - usual set of operations (load(), store(), exchange(), compare_ exchange_weak(), and compare_exchange_strong()
  - fetch_add(), fetch_sub(), fetch_and(), fetch_or(), fetch_xor()
  - compound-assignment forms of these operations (+=, -=, &=, l=, and ^=)
  - pre- and post-increment and decrement (++x, x++, --x, and x--)
  - only division, multiplication, and shift operators are missing
- typically used either as counters or as bitmasks
  - missing operators are not a big loss
- additional operations can easily be done using compare_exchange_weak() in a loop
- semantics match closely to fetch_add() and fetch_sub() for std::atomic<T*>
  - named functions atomically perform their operation and return the old value
  - compound-assignment operators return the new value
  - pre- and post- increment and decrement work as usual
    - result is a value of the associated integral type in both cases

The std::atomic<> primary class template

- allows a user to create an atomic variant of a user-defined type
  - can't use any user-defined type with std::atomic<>
  - type must
    - have a trivial copy-assignment operator
      - must not have any virtual functions or virtual base classes and must use the compiler-generated copy-assignment operator
    - every base class and non-static data member of a user-defined type must also have a trivial copy-assignment operator
    - permits the compiler to use memcpy() or an equivalent operation for assignment operations
      - no user-written code to run
    - type must be bitwise equality comparable
      - must be able to compare instances for equality using memcmp()
      - required in order for compare/exchange operations to work
- compiler isn't going to be able to generate lock-free code for std::atomic, so it will have to use an internal lock for all the operations
  - would require passing a reference to the protected data as an argument to a user-supplied function if user-supplied copy-assignment or comparison operators were permitted
  - violates previous guideline - don't pass pointers and references to protected data outside the scope of the lock by passing them as arguments to user-supplied functions

- library is entirely at liberty to use a single lock for all atomic operations that need it
  - allowing user-supplied functions to be called while holding that lock might cause deadlock or cause other threads to block because a comparison operation took a long time
- restrictions increase the chance that the compiler will be able to make use of atomic instructions directly for std::atomic
  - can just treat the user-defined type as a set of raw bytes
  - more likely to make a particular instantiation lock-free
- built-in floating point types satisfy the criteria for use with memcpy and memcmp
  - can use std::atomic or std::atomic
  - behavior may be surprising in the case of compare_exchange_strong
  - operation may fail even though the old stored value was equal in value to the comparand if the stored value had a different representation
  - NOTE: no atomic arithmetic operations on floating-point values
- will get similar behavior with compare_exchange_ strong if using std::atomic<> with a user-defined type that has an equality-comparison operator defined
  - if operator differs from the comparison using memcmp operation may fail because the otherwise-equal values have a different representation
- most common platforms will be able to use atomic instructions for std::atomic if your UDT is the same size as (or smaller than) an int or a void*
  - some platforms will also be able to use atomic instructions for user-defined types that are twice the size of an int or void*
  - platforms are typically those that support a so-called double-word-compare-and-swap (DWCAS) instruction corresponding to the compare_exchange_xxx functions
  - can be helpful when writing lock-free code
- can't create a std::atomic<std:: vector>
  - can use it with classes containing counters or flags or pointers or even just arrays of simple data elements
  - more complex the data structure, the more likely you'll want to do operations on it other than simple assignment and comparison
  - better off using a std::mutex to ensure that the data is appropriately protected for the desired operations
- interface of std::atomic is limited to the set of operations available for std::atomic when instantiated with a user-defined type T
  - load()
  - store()
  - exchange()
  - compare_exchange_weak()
  - compare_exchange_strong()
  - assign- ment from and conversion to an instance of type T

Free functions for atomic operations

- equivalent nonmember functions for all the operations on the various atomic types
- table 5.3
- named after the corresponding member functions but with an atomic_ prefix (for example, std::atomic_load())
  - overloaded for each of the atomic types
  - come in two varieties when a memory-ordering tag can be specified
    - one without the tag
    - one with an _explicit suffix and an additional parameter or parameters for the memory-ordering tag or tags
- all the free functions take a pointer to the atomic object as the first parameter
- free functions are designed to be C-compatible
  - use pointers rather than references in all cases
- operations on std::atomic_flag do not follow pattern
  - std::atomic_flag_test_and_set(), std::atomic_flag_clear()
  - additional variants that specify the memory ordering alos have the _explicit suffix
  - std::atomic_flag_test_and_set_explicit() and std::atomic_ flag_clear_explicit()
- Standard Library also provides free functions for accessing instances of std::shared_ptr<> atomically
- C++ Standards Committee felt it was sufficiently important to provide these extra functions. The
- atomic operations available are load, store, exchange, and compare/exchange
  - provided as overloads of the same operations on the standard atomic types
  - take a std::shared_ptr<>* as the first argument

```
std::shared_ptr<my_data> p;
void process_global_data()
{
    std::shared_ptr<my_data> local=std::atomic_load(&p);
    process_data(local);
}
void update_global_data()
{
    std::shared_ptr<my_data> local(new my_data);
    std::atomic_store(&p,local);
}
```

- _explicit variants are provided to allow specification of the desired memory ordering
  - std::atomic_is_lock_free() function can be used to check whether the implementation uses locks to ensure the atomicity

- standard atomic types do more than just avoid the undefined behavior associated with a data race
    - allow user to enforce an ordering of operations between threads
    - enforced ordering is the basis of the facilities for protecting data and synchronizing operations such as std::mutex and std::future<>

5.3 Synchronizing operations and enforcing ordering

- two threads
    - one populating a data structure to be read by the second
    - to avoid a problematic race condition
        - first thread sets a flag to indicate that the data is ready
        - second thread doesn't read the data until the flag is set
- Listing 5.2
- sharing data between threads becomes impractical
    - every item of data is forced to be atomic
- undefined behavior to have nonatomic reads and writes accessing the same data without an enforced ordering
- required enforced ordering comes from the operations on the std::atomic variable data_ready
    - provide the necessary ordering by virtue of the memory model relations
    - happens-before and synchronizes-with
        - write of the data happens-before the write to the data_ready flag
        - read of the flag B happens-before the read of the data when the value read from data_ready B is true
        - write synchronizes-with read -> happens-before relationship
            - happens-before is transitive
        - write to the data d happens-before the write to the flag
        - happens-before the read of the true value from the flag B
        - happens-before the read of the data
        - write of the data happens-before the read of the data

The synchronizes-with relationship

- can only get between operations on atomic types
- operations on a data structure might provide this relationship if the data structure contains atomic types and the operations on that data structure perform the appropriate atomic operations internally

- a suitably tagged atomic write operation W on a variable x synchronizes-with a suitably tagged atomic read operation on x
  - reads the value stored by either
    - that write (W)
    - a subsequent atomic write operation on x by the same thread that performed the initial write W
    - a sequence of atomic read-modify-write operations on x (such as fetch_add() or compare_exchange_weak()) by any thread, where the value read by the first thread in the sequence is the value written by W
- all operations on atomic types are suitably tagged by default
- if thread A stores a value and thread B reads that value, there's a synchronizes-with relationship between the store in thread A and the load in thread B
- C++ memory model allows various ordering constraints to be applied to the operations on atomic types
  - this is the tagging

The happens-before relationship

- basic building block of operation ordering in a program
  - specifies which operations see the effects of which other operations for a single thread
- if one operation is sequenced before another, then it also happens-before it
  - if operation (A) occurs in a statement prior to another (B) in the source code, then A happens-before B
- if the operations occur in the same statement, in general there's no happens-before relationship between them
  - i.e. unordered or ordering is unspecified
- Listing 5.3
- circumstances where operations within a single statement are sequenced
  - built-in comma operator is used
  - result of one expression is used as an argument to another expression
- operations within a single statement are nonsequenced in general
  - no sequenced-before (and thus no happens-before) relationship between them
- all operations in a statement happen before all of the operations in the next statement
- interaction between threads
  - if operation A on one thread inter-thread happens-before operation B on another thread, then A happens-before B
- new relationship -> inter-thread happens-before
  - if operation A in one thread synchronizes-with operation B in another thread, then A inter-thread happens-before B
  - also transitive
    - if A inter-thread happens-before B and B inter-thread happens-before C, then A inter-thread happens-before C

- also combines with the sequenced-before relation
    - if operation A is sequenced before operation B, and operation B inter-thread happens-before operation C, then A inter-thread happens-before
- if A synchronizes-with B and B is sequenced before C, then A inter-thread happens-before C
- sequenced-before and synchronizes-with combined mean that if a series of changes are made to data in a single thread, only one synchronizes-with relationship is needed for the data to be visible to subsequent operations on the thread that executed
- additional nuances with data dependency

Memory ordering for atomic operations

- memory_order_relaxed
- memory_order_consume
- memory_order_acquire
- memory_order_release
- memory_order_acq_rel
- memory_order_seq_cst
- default for all operations on atomic types is memory_order_seq_cst
    - most stringent of the available options
- six ordering options represent three models
    - sequentially consistent ordering
        - memory_order_seq_cst
    - acquire-release ordering
        - memory_order_consume
        - memory_order_acquire
        - memory_order_release
        - memory_order_acq_rel
    - relaxed ordering
        - memory_order_relaxed
- memory-ordering models can have varying costs on different CPU architectures
    - systems based on architectures with fine control over the visibility of operations by processors other than the one that made the change
        - additional synchronization instructions can be required for sequentially consistent ordering over acquire-release ordering or relaxed ordering and for acquire-release ordering over relaxed ordering
    - if systems have many processors additional synchronization instructions may take a significant amount of time

- CPUs that use the x86 or x86-64 architectures (Intel and AMD) don't require any additional instructions for acquire-release ordering beyond those necessary for ensuring atomicity
  - sequentially-consistent ordering doesn't require any special treatment for load operations
    - small additional cost on stores
- allows experts to take advantage of the increased performance of the more fine-grained ordering relationships
  - allows the use of the default sequentially-consistent ordering for cases that are less critical
    - easier to reason about

Sequentially consistent ordering

- implies that the behavior of the program is consistent with a simple sequential view of the world
- behavior of a multithreaded program is as if all these operations were performed in some particular sequence by a single thread
  - easiest memory ordering to understand
- can write down all the possible sequences of operations by different threads
  - eliminate those that are inconsistent
  - verify that your code behaves as expected in the others
- operations can't be reordered
- point of view of synchronization
  - sequentially consistent store synchronizes-with a sequentially consistent load of the same variable that reads the value stored
- any sequentially consistent atomic operations done after that load must also appear after the store to other threads in the system using sequentially consistent atomic operations
- constraint doesn't carry forward to threads that use atomic operations with relaxed memory orderings
  - can still see the operations in a different order
  - must use sequentially consistent operations on all threads in order to get the benefit
- can impose a noticeable performance penalty on a weakly ordered machine with many processors
  - possibly requires extensive and expensive synchronization operations between the processors
- some processor architectures (x86 and x86-64 architectures) offer sequential consistency relatively cheaply
- Listing 5.4
- implied ordering relationship between a load of a variable that returns false and the store to that variable
  - because the semantics of memory_order_seq_cst require a single total ordering over all operations tagged memory_order_seq_cst

Non-sequentially consistent memory orderings

- no longer a single global order of events
    - different threads can see different views of the same operations
    - mental model must be drastically changed
- have to account for things happening truly concurrently
- threads don't have to agree on the order of events
- compiler can reorder the instructions
- threads can disagree on the order of events because of operations in other threads in the absence of explicit ordering constraints
    - different CPU caches and internal buffers can hold different values for the same memory
- threads don't have to agree on the order of events!!!
- only requirement is that all threads agree on the modification order of each individual variable
    - operations on distinct variables can appear in different orders on different threads
    - values seen must be consistent with any additional ordering constraints imposed

Relaxed ordering

- (operations on atomic types performed with relaxed ordering)
- don't participate in synchronizes-with relationships operations on the same variable within a single thread
    - still obey happens-before relationships
    - almost no requirement on ordering relative to other threads
- accesses to a single atomic variable from the same thread can't be reordered
    - once a given thread has seen a particular value of an atomic variable, a subsequent read by that thread can't retrieve an earlier value of the variable
- modification order of each variable is the only thing shared between threads that are using memory_order_relaxed when theres is no additional synchronization
- relaxed operations on different variables can be freely reordered provided they obey any happens-before relationships they're bound by (for example, within the same thread)
    - don't introduce synchronizes-with relationships
- Listing 5.6
- launching a thread is an expensive operation
    - need an explicit delay to ensure ordering
- many possible valid outcomes for relaxed operations

Understanding relaxed ordering

- separated units of execution with a shared log of data
    - can message between UEs and log holder and either load or store (read or write) to entries in a log
    - writes append to log
    - messages to read result in a read from the log
    - first read could be any value in log
        - no subsequent read will be any higher than initial read
    - write and subsequent read will result in either the value written or a value further down log
    - reads can result in repeated reads of the same value
    - after a write, all subsequent reads will return the value written until a new value is written
    - additional threads can act on same log
    - any log action is mutually exclusive, i.e. only one UE can be serviced at a time
    - a write from one UE does not mean that value needs to be available to read from another UE
        - reads for a UE can return any of the numbers after the previous value read for that UE in the log at any time
        - some sort of implicit bookkeeping for previous read for each UE
- shared log of data == shared atomic variables
    - each variable has its own modification order but there's no relationship between them at all
    - what happens when every operation uses memory_order_relaxed if each UE is a thread
- additional operations on variables/logs
    - exchange -> write a value and tell the value at the bottom of the log
    - compare-and-exchange (strong) -> write a value and tell if the replaced value is equal, otherwise tell the replaced value
- relaxed atomic operations are difficult to deal with
    - must be used in combination with atomic operations that feature stronger ordering semantics in order to be useful for inter-thread synchronization
- avoid relaxed atomic operations unless they're absolutely necessary
    - use them only with extreme caution

Acquire-release ordering

- no total order of operations
    - introduces some synchronization
    - atomic loads are acquire operations (memory_order_acquire)
    - atomic stores are release operations (memory_order_release)

- atomic read-modify-write operations (such as fetch_add() or exchange()) are either acquire, release, or both (memory_order_acq_rel)
- synchronization is pairwise
    - a release operation synchronizes-with an acquire operation that reads the value written between the thread that does the release and the thread that does the acquire
- different threads can still see different orderings
    - orderings are restricted
- Listing 5.7
- Listing 5.8
- adding to log model in previous section
    - every store that's done is part of some batch of updates
        - a write includes a batch ID
        - the last store in a batch is also recorded as the last store of the batch
        - the UE ID initiating the batch operation is recorded as well
    - models a store-release operation
    - batch numbers are monotonically increasing (between single UE's?)
    - for loads
        - can either just ask for a value (relaxed load) -> value is returned
        - can ask for a value and information about whether it's the last in a batch (load-acquire)
        - given the batch information
            - if value wasn't the last in a batch UE is informed that it is not last in the batch
            - otherwise, UE is informed it is the last in the batch
    - acquire-release semantics
        - when requesting a load, if the log manager is informed of all the batches known to UE, the UE will either be returned the last value from any of the known batches or a value further down the list
        - how semantics are modeled
            - thread a is writing to two variables, x and y, as part of a batch, writing x then y
            - thread b is reading y and then reading x
                - reads y and batch information in a loop until the expected value is returned
                - reads x but tells log manager of all batches from thread a that the thread was informed about while reading y
            - inter-thread happens-before is transitive
                - if A inter-thread happens-before B and B inter-thread happens-before C, then A inter-thread happens-before C
            - means that acquire-release ordering can be used to synchronize data across several threads, even when the "intermediate" threads haven't actually touched the data

Transitive synchronization with acquire-release ordering

- need at least three threads to demonstrate transitive ordering
    - thread 1 modifies some shared variables and does a store-release to one of them
    - thread 2 then reads the variable subject to the store-release with a load-acquire and performs a store-release on a second shared variable
    - thread 3 does a load-acquire on that second shared variable provided that the load-acquire operations see the values written by the store-release operations
    - thread 3 can read the values of the other variables stored by the first thread, even if the intermediate thread didn't touch any of them to ensure the synchronizes-with relationships
- Listing 5.9
- thread 2 only touches variable 1 and 2
    - enough for synchronization between thread 1 and thread 3
    - stores to data from thread 1 happens-before the store to variable 1
        - they're sequenced-before it in the same thread
    - load from variable 1 is in a while loop so it will eventually see the value stored from thread 1
        - forms the second half of the release-acquire pair
    - store to variable 1 happens-before the final load from 1 in the while loop
        - load is sequenced-before (and thus happens-before) the store to variable 2
            - forms a release-acquire pair with the final load from the while loop in thread 3
    - store to variable 2 thus happens-before the load, which happens-before the loads from data
    - stores to data happen-before the store to variable 1, which happens-before the load from variable 1, which happens-before the store to variable 2, which happens-before the load from variable 2, which happens-before the loads from data
        - stores to data in thread 1 happen-before the loads from data in thread 3
    - could combine variables 1 and 2 into a single variable by using a read-modify-write operation with memory_order_acq_rel in thread 2
- code sample
- important to pick which semantics when using read-modify-write operations
- if mixing acquire-release operations with sequentially consistent operations
    - sequentially consistent loads behave like loads with acquire semantics
    - sequentially consistent stores behave like stores with release semantics
- sequentially consistent read-modify-write operations behave as both acquire and release operations
- relaxed operations are still relaxed but are bound by the additional synchronizes-with and consequent happens-before relationships introduced through the use of acquire-release semantics

- locking a mutex is an acquire operation, and unlocking the mutex is a release operation
  - must ensure that the same mutex is locked when reading a value as was locked when writing it
- acquire and release operations have to be on the same variable to ensure an ordering
- exclusive nature of a lock with a mutex means that the result is indistinguishable from what it would have been had the lock and unlock been sequentially consistent operations
  - using acquire and release orderings on atomic variables to build a simple lock behavior will appear sequentially consistent, even though the internal operations are not
- pair-wise synchronization of acquire-release ordering has the potential for a much lower synchronization cost than the global ordering required for sequentially consistent operations
  - trade-off is the mental cost required to ensure that the ordering works correctly and that the non-intuitive behavior across threads isn't problematic

Data dependency with acquire-release ordering and memory_order_consume

- memory_order_consume requires special consideration
  - based on data dependencies
  - introduces data-dependency nuances to the inter-thread happens-before relationship
- carries-a-dependency-to
  - applies strictly within a single thread and essentially models the data dependency between operations
  - if the result of an operation A is used as an operand for an operation B, then A carries-a-dependency-to B
  - if the result of operation A is a value of a scalar type such as an int, then the relationship still applies
  - if the result of A is stored in a variable, and that variable is then used as an operand for operation B this operation is also transitive, so if A carries-a-dependency-to B, and B carries-a-dependency-to C, then A carries-a-dependency-to C
- dependency-ordered-before
  - can apply between threads
  - introduced by using atomic load operations tagged with memory_order_consume
  - special case of memory_order_acquire that limits the synchronized data to direct dependencies
  - a store operation A tagged with memory_order_release, memory_order_acq_rel, or memory_order_seq_cst is dependency-ordered-before a load operation B tagged with memory_order_consume if the consume reads the value stored
    - in contrast with the synchronizes-with relationship that occurs if the load uses memory_order_acquire
  - if this operation B then carries-a-dependency-to some operation C, then A is also dependency-ordered-before C
  - if A is dependency-ordered-before B, then A also inter-thread happens-before B

- important use is where the atomic operation loads a pointer to some data
  - memory_order_consume on the load and memory_order_release on the prior store
  - ensures that the pointed-to data is correctly synchronized
  - doesn't impose any synchronization requirements on any other nondependent data
- Listing 5.10
- sometimes the overhead of carrying the dependency around is not desirable
  - compiler should be able to cache values in registers and reorder operations to optimize the code rather than dealing with dependencies
  - can use std::kill_dependency() to explicitly break the dependency chain
- std::kill_dependency()
  - function template that copies the supplied argument to the return value but breaks the dependency chain in doing so
  - e.g. with a global read-only array
    - use std::memory_order_consume when retrieving an index into that array from another thread,
    - can use std::kill_dependency() to let the compiler know that it doesn't need to reread the contents of the array entry
- code sample
- shouldn't normally use std::memory_order_consume at all in simple scenarios
  - can call on std::kill_dependency() in a situation with more complex code
  - should only be used with care and where profiling has demonstrated the need for the optimization

Release sequences and synchronizes-with

- can get a synchronizes-with relationship between a store to an atomic variable and a load of that atomic variable from another thread
  - even when there's a sequence of read-modify-write operations between the store and the load
  - need all the operations to be suitably tagged
- if a store is tagged with memory_order_release, memory_order_acq_rel, or memory_order_seq_cst
  - load is tagged with memory_order_consume, memory_order_acquire, or memory_order_seq_cst
  - each operation in the chain loads the value written by the previous operation
  - chain of operations constitutes a release sequence
    - initial store synchronizes-with the final load for memory_order_acquire or memory_order_seq_cst
    - initial store is dependency-ordered-before the final load for memory_order_consume
- any atomic read-modify-write operations in the chain can have any memory ordering (even memory_order_relaxed)

- Listing 5.11
- see listing and description for more details

Fences

- operations that enforce memory-ordering constraints without modifying any data
  - typically combined with atomic operations that use the memory_order_relaxed ordering constraints
- global operations and affect the ordering of other atomic operations in the thread that executed the fence
- commonly called memory barriers
  - put a line in the code that certain operations can't cross
- relaxed operations on separate variables can usually be freely reordered by the compiler or the hardware
  - fences restrict this freedom and introduce happens-before and synchronizes-with relationships that weren't present before
- Listing 5.12
- general idea with fences
  - if an acquire operation sees the result of a store that takes place after a release fence, the fence synchronizes-with that acquire operation
  - if a load that takes place before an acquire fence sees the result of a release operation, the release operation synchronizes-with the acquire fence
- can have fences on both sides
  - if a load that takes place before the acquire fence sees a value written by a store that takes place after the release fence, the release fence synchronizes-with the acquire fence
- NOTE: the synchronization point is the fence itself even though the fence synchronization depends on the values read or written by operations before or after the fence
- code sample
- real benefit to using atomic operations to enforce an ordering is that they can enforce an ordering on nonatomic operations and thus avoid the undefined behavior of a data race

Ordering nonatomic operations with atomics

- replace x from listing 5.12 with an ordinary nonatomic bool the behavior is guaranteed to be the same
- Listing 5.13
- fences still provide an enforced ordering of the stores and loads
  - still a happens-before relationship between the store to x and the
  - some stores/loads still have to be atomic
- not just fences that can order nonatomic operations

- memory_order_release/memory_order_consume pair ordering nonatomic accesses to a dynamically allocated object
- many of the examples in this chapter could be rewritten with some of the memory_order_relaxed operations replaced with plain nonatomic operations instead
- ordering of nonatomic operations through the use of atomic operations is where the sequenced-before part of happens-before becomes so important
  - if a nonatomic operation is sequenced-before an atomic operation, and that atomic operation happens-before an operation in another thread
    - nonatomic operation also happens-before that operation in the other thread
- basis for the higher-level synchronization facilities in the C++ Standard Library
  - mutexes and condition variables
- consider spinlock mutex from listing 5.1 - see text for description
- basic principle is the same for mutexes even though various implementations will have different internal operations
  - lock() is an acquire operation on an internal memory location
  - unlock() is a release operation on that same memory location

5.4 Summary

- low-level details of
  - C++11 memory model
  - atomic operations that provide the basis for synchronization between threads
- basic atomic types provided by specializations of the std::atomic<> class template
- generic atomic interface provided by the primary std::atomic<> template
- operations on these types
- complex details of the various memory-ordering options
- fences and how they can be paired with operations on atomic types to enforce an ordering
- how the atomic operations can be used to enforce an ordering between nonatomic operations on separate threads

Ch. 6 - Designing lock-based concurrent data structures

- the choice of data structure to use for a programming problem can be key
- also required for parallel programming problems
- if a data structure is to be accessed from multiple threads
  - either must be completely immutable so the data never changes and no synchronization is necessary
  - program must be designed to ensure that changes are correctly synchronized between threads

- can use a separate mutex and external locking to protect the data
- another is to design the data structure itself for concurrent access

6.1 What does it mean to design for concurrency?

- designing a data structure for concurrency means that multiple threads can access the data structure concurrently
    - perform the same or distinct operations
    - each thread will see a self-consistent view of the data structure
    - no data will be lost or corrupted
    - all invariants will be upheld
    - no problematic race conditions
- e.g. thread-safe
- data structure will be safe only for particular types of concurrent access
    - may be possible to have multiple threads performing one type of operation on the data structure concurrently
        - another operation requires exclusive access by a single thread
    - may be safe for multiple threads to access a data structure concurrently if they're performing different actions
        - multiple threads performing the same action would be problematic
- means providing the opportunity for concurrency to threads accessing the data structure
    - mutex provides mutual exclusion
        - only one thread can acquire a lock on the mutex at a time
    - protects a data structure by explicitly preventing true concurrent access to the data it protects
    - e.g. serialization
- some data structures have more potential for true concurrency than others
    - smaller the protected region
    - fewer operations serialized
    - greater the potential for concurrency

6.1.1 Guidelines for designing data structures for concurrency

- in general
    - ensure that no thread can see a state where the invariants of the data structure have been broken by the actions of another thread
    - take care to avoid race conditions inherent in the interface to the data structure by providing functions for complete operations rather than for operation steps
    - pay attention to how the data structure behaves in the presence of exceptions to ensure that the invariants are not broken

- minimize the opportunities for deadlock when using the data structure by restricting the scope of locks and avoiding nested locks where possible
- what constraints to put on the users of the data structure
  - which functions are safe to call from other threads if one thread is accessing the data structure through a particular function
- generally constructors and destructors require exclusive access to the data structure
  - up to the user to ensure that they're not accessed before construction is complete or after destruction has started
  - if the data structure supports assignment, swap(), or copy construction then need to decide
    - whether these operations are safe to call concurrently with other operations
    - whether they require the user to ensure exclusive access even though the majority of functions for manipulating the data structure may be called from multiple threads concurrently without problem
- enabling genuine concurrent access
  - can the scope of locks be restricted to allow some parts of an operation to be performed outside the lock?
  - can different parts of the data structure be protected with different mutexes?
  - do all operations require the same level of protection?
  - can a simple change to the data structure improve the opportunities for concurrency without affecting the operational semantics?
- how to minimize the amount of serialization that must occur and enable the greatest amount of true concurrency?
- not uncommon for concurrent access from multiple threads that just read
  - a thread that can modify the data structure must have exclusive access
  - supported by using constructs like boost::shared_mutex
- common for a data structure to support concurrent access from threads performing different operations while serializing threads that try to perform the same operation
- simplest thread-safe data structures typically use mutexes and locks to protect the data

6.2 Lock-based concurrent data structures

- all about ensuring that the right mutex is locked when accessing the data and ensuring that the lock is held for a minimum amount of time
- issues are compounded if using separate mutexes to protect separate parts of the data structure
  - also the possibility of deadlock if the operations on the data structure require more than one mutex to be locked

6.2.1 A thread-safe stack using locks

- Listing 6.1 A class definition for a thread-safe stack

- basic thread safety is provided by protecting each member function with a lock on the mutex
  - ensures that only one thread is actually accessing the data at any one time
  - no thread can see a broken invariant if each member function maintains the invariants
  - potential for a race condition between empty() and either of the pop() functions
    - race condition isn't problematic because code explicitly checks for the contained stack being empty while holding the lock in pop()
  - avoid a potential race condition by returning the popped data item directly as part of the call to pop()
- potential sources of exceptions
  - locking a mutex may throw an exception
    - likely to be rare (because it indicates a problem with the mutex or a lack of system resources)
    - safe because no data has been modified since it is also the first operation in each member function
  - unlocking a mutex can't fail so it is always safe
  - use of std::lock_guard<> ensures that the mutex is never left locked
  - call to data.push() may throw an exception if either copying/moving the data value throws an exception or not enough memory can be allocated to extend the underlying data structure
  - might throw an empty_stack exception in the first overload of pop()
    - safe because nothing has been modified
  - creation of res might throw an exception
    - call to std::make_shared might throw because it can't allocate memory for the new object and the internal data required for reference counting
    - copy constructor or move constructor of the data item to be returned might throw when copying/moving into the freshly allocated memory
    - C++ runtime and Standard Library ensure that there are no memory leaks and the new object (if any) is correctly destroyed
    - safe because underlying stack has not been modified
  - empty() doesn't modify any data, so that's exception-safe
- opportunities for deadlock here
  - call user code while holding a lock
    - copy constructor or move constructor
    - copy assignment or move assignment operator on the contained data items
    - potentially a user-defined operator new
  - possibility of deadlock if these functions either call member functions on the stack that the item is being inserted into or removed from or require a lock of any kind and another lock was held when the stack member function was invoked
  - OK to require that users of the stack be responsible for ensuring this
  - safe for any number of threads to call the stack member functions because all the member functions use a std::lock_guard<> to protect the data

- only member functions that aren't safe are the constructors and destructors
    - object can be constructed only once and destroyed only once
    - user must ensure that other threads aren't able to access the stack until it's fully constructed and must ensure that all threads have ceased accessing the stack before it's destroyed
- only one thread is ever actually doing any work in the stack data structure at a time
    - serialization of threads can limit the performance of an application where there's significant contention
        - thread isn't doing any useful work while waiting for the lock on the stack
- doesn't provide any means for waiting for an item to be added
    - if a thread needs to wait, it must periodically call empty() or just call pop() and catch the empty_stack exceptions
    - a waiting thread must either consume precious resources checking for data or the user must write external wait and notification code
        - might render the internal locking unnecessary and wasteful

A thread-safe queue using locks and condition variables

- interface differs from that of the standard container adaptor because of the constraints of writing a data structure that's safe for concurrent access from multiple threads
- Listing 6.2 The full class definition for a thread-safe queue using condition variables
- two overloads of try_pop() don't throw an exception if the queue is empty
    - return either a bool value indicating whether a value was retrieved or a NULL pointer if no value could be retrieved by the pointer-returning overload f
- if more than one thread is waiting when an entry is pushed onto the queue, only one thread will be woken by the call to data_cond.notify_one()
    - if that thread then throws an exception in wait_and_pop(), such as when the new std::shared_ptr<> is constructed, none of the other threads will be woken
    - call can be replaced with data_cond.notify_all()
        - will wake all the threads but at the cost of most of them then going back to sleep when they find that the queue is empty
    - have wait_and_pop() call notify_one() if an exception is thrown
        - another thread can attempt to retrieve the stored value
    - move the std::shared_ptr<> initialization to the push() call and store std::shared_ptr<> instances rather than direct data values
        - copying the std::shared_ptr<> out of the internal std::queue<> can't throw an exception
        - wait_and_pop() is safe
- Listing 6.3 A thread-safe queue holding std::shared_ptr<> instances
- consequences of holding the data by std::shared_ptr<>
    - pop functions that take a reference to a variable to receive the new value now have to dereference the stored pointer

- pop functions that return a std::shared_ptr<> instance can just retrieve it from the queue before returning it to the caller
- allocation of the new instance can be done outside the lock in push() if the data is held by std::shared_ptr<>
  - previously had to be done while holding the lock in pop()
  - memory allocation is typically quite an expensive operation
  - because it reduces the time the mutex is held, allowing other threads to perform operations on the queue in the meantime
- use of a mutex to protect the entire data structure limits the concurrency
  - multiple threads might be blocked on the queue in various member functions
  - only one thread can be doing any work at a time
- have essentially one data item that's either protected or not by using the standard container
- can provide more fine-grained locking and thus allow a higher level of concurrency by taking control of the detailed implementation of the data structure

A thread-safe queue using fine-grained locks and condition variables

- need to look inside the queue at its constituent parts and associate one mutex with each distinct data item
- simplest data structure for a queue is a singly linked list
- data items are removed from the queue by replacing the head pointer with the pointer to the next item and then returning the data from the old head
- items are added to the queue at the other end
  - queue also contains a tail pointer, which refers to the last item in the list
- new nodes are added by changing the next pointer of the last item to point to the new node and then updating the tail pointer to refer to the new item
- when the list is empty, both the head and tail pointers are NULL
- Listing 6.4 A simple single-threaded queue implementation
- uses std::unique_ptr to manage the nodes
  - ensures that they (and the data they refer to) get deleted when they're no longer needed, without having to write an explicit delete
- ownership chain is managed from head, with tail being a raw pointer to the last node
- works fine in a single-threaded context
- problems when using fine-grained locking in a multi-threaded context
  - could use two mutexes for head and tail
  - push() can modify both head f and tail g, so it would have to lock both mutexes
  - critical problem is that both push() and pop() access the next pointer of a node
    - push() updates tail->next and try_pop() reads head->next
    - single item in the queue -> head==tail
      - both head->next and tail->next are the same object
      - requires protection

- can't tell if it's the same object without reading both head and tail
- have to lock the same mutex in both push() and try_pop()

Enabling concurrency by separating data

- solve by preallocating a dummy node with no data to ensure that there's always at least one node in the queue to separate the node being accessed at the head from that being accessed at the tail
- for an empty queue, head and tail now both point to the dummy node rather than being NULL
    - try_pop() doesn't access head->next if the queue is empty
- head and tail now point to separate nodes if one node is added to the queue
    - no race on head->next and tail->next
- have to add an extra level of indirection to store the data by pointer in order to allow the dummy nodes
- Listing 6.5 A simple queue with a dummy node
- see text for description
- push() now accesses only tail, not head
- try_pop() accesses both head and tail, but tail is needed only for the initial comparison, so the lock is short-lived
- dummy node means try_pop() and push() are never operating on the same node, so you no longer need an overarching mutex
    - one mutex for head and one for tail
- hold the locks for the smallest possible length of time
- for push -> mutex needs to be locked across all accesses to tail, which means you lock the mutex after the new node is allocated and before assigning the data to the current tail node
    - lock needs to be held until the end of the function
- for try_pop()
    - need to lock the mutex on head and hold it until finished with head
    - the mutex to determine which thread does the popping so pop first
    - once head is changed, can unlock the mutex
        - doesn't need to be locked when returning the result
    - leaves access to tail needing a lock on the tail mutex
        - need to access tail only once
        - can just acquire the mutex for the time it takes to do the read
        - best done by wrapping it in a function
    - clearer to wrap code that needs the head mutex locked in a function also because it is only a subset of the member
- Listing 6.6 A thread-safe queue with fine-grained locking
- invariants
    - tail->next == nullptr

- tail->data == nullptr
- head == tail implies an empty list
- A single element list has head->next == tail
- for each node x in the list
  - where
    - x!=tail
    - x->data points to an instance of T
    - x->next points to the next node in the list
  - x->next==tail implies x is the last node in the list
  - following the next nodes from head will eventually yield tail
- for push
  - only modifications to the data structure are protected by tail_mutex
  - uphold the invariant because the new tail node is an empty node and data and next are correctly set for the old tail node, which is now the last real node in the list
- for try_pop()
  - lock on tail_mutex is necessary to protect the read of tail itself
  - also necessary to ensure that a data race doesn't result from reading the data from the head
    - possible for a thread to call try_pop() and a thread to call push() concurrently without the mutex
      - no defined ordering on their operations
  - each member function holds a lock on a mutex but they hold locks on different mutexes and they potentially access the same data
    - all data in the queue originates from a call to push()
    - because the threads would be potentially accessing the same data without a defined ordering this would be a data race and undefined behavior
  - lock on the tail_mutex in get_tail() solves problems
    - defined order between the two calls because call to get_tail() locks the same mutex as the call to push()
    - either
      - call to get_tail() occurs before the call to push() -> sees the old value of tail
      - occurs after the call to push() -> sees the new value of tail and the new data attached to the previous value of tail
  - important that the call to get_tail() occurs inside the lock on head_mutex
    - otherwise call to pop_head() could be stuck in between the call to get_tail() and the lock on the head_mutex
- see text for more description of invariant handling
- actual possibilities for concurrency
  - data structure actually has considerably more potential for concurrency
    - locks are more fine-grained and more is done outside the locks

Waiting for an item to pop

- wait_and_pop() implementation in an identical interface with fine-grained locking

    - just adding the data_cond.notify_one() call at the end of the function to modify push()
        - using fine-grained locking because you want the maximum possible amount of concurrency
        - leaving the mutex locked across the call to notify_one() then if the notified thread wakes up before the mutex has been unlocked, it will have to wait for the mutex
        - unlocking the mutex before you call notify_one(), then the mutex is available for the waiting thread to acquire when it wakes up (assuming no other thread locks it first)
    - might be important in some cases

- wait_and_pop()

    - have to decide where to wait, what the predicate is, and which mutex needs to be locked
    - condition is "queue not empty" (represented by head != tail)
        - would require both head_mutex and tail_mutex to be locked
        - only need to lock tail_mutex for the read of tail and not for the comparison itself
        - making the predicate head!=get_tail()
            - only need to hold the head_mutex
            - can use lock on that for the call to data_cond.wait()
    - with wait logic the implementation is the same as try_pop()

- second overload of try_pop() and corresponding wait_and_pop()

    - potential exception-safety issue arises from just replacing the return of the std::shared_ptr<> retrieved from old_head with a copy assignment to the value parameter
    - data item has been removed from the queue and the mutex unlocked
        - just need to return the data to the caller
    - if the copy assignment throws an exception data item is lost because it can't be returned to the queue in the same place
    - could use no-throw move-assignment operator or a no-throw swap operation for type T if provided but this is not general
    - have to move the potential throwing inside the locked region, before the node is removed from the list
        - need an extra overload of pop_head() that retrieves the stored value prior to modifying the list

- empty() is trivial

    - just lock head_mutex and check for head == get_tail()

- Listing 6.7 A thread-safe queue with locking and waiting: internals and interface

- Listing 6.8 A thread-safe queue with locking and waiting: pushing new values

- Listing 6.9 A thread-safe queue with locking and waiting: wait_and_pop()

- Listing 6.10 A thread-safe queue with locking and waiting: try_pop() and empty()

- unbounded queue

  - threads can continue to push new values onto the queue as long as there's available memory, even if no values are removed

- bounded queue

  - maximum length of the queue is fixed when the queue is created
  - once full attempts to push further elements onto the queue will either fail or block until an element has been popped from the queue to make room
  - useful for ensuring an even spread of work when dividing work between threads based on tasks to be performed
    - prevents the thread(s) populating the queue from running too far ahead of the thread(s) reading items from the queue

- unbounded queue implementation can easily be extended to limit the length of the queue by waiting on the condition variable in push()

  - wait for the queue to have fewer than the maximum number of items

6.3 Designing more complex lock-based data structures

- most data structures support a variety of operations
  - can then lead to greater opportunities for concurrency
  - also makes the task of protecting the data that much harder because the multiple access patterns need to be taken into account
- precise nature of the various operations that can be performed is important when designing such data structures for concurrent access

6.3.1 Writing a thread-safe lookup table using locks

- associative containers
  - std::map<>
  - std::multimap<>
  - std::unordered_map<>
  - std::unordered_ multimap<>
- lookup table might be modified rarely

- interfaces of the standard containers aren't suitable when the data structure is to be accessed from multiple threads concurrently
    - there are inherent race conditions in the interface design
    - need to be cut down and revised
- biggest problem with the std::map<> interface are the iterators
    - possible to have an iterator that provides safe access into a container even when other threads can access (and modify) the container
    - difficult
    - have to deal with issues such as another thread deleting the element that the iterator is referring to
- skip the iterators given that the
- worth designing the interface from the ground up
    - interface to std::map<> is heavily iterator-based
- operations on a lookup table
    - add a new key/value pair
    - change the value associated with a given key
    - remove a key and its associated value
    - obtain the value associated with a given key if any
- container-wide operations
    - check on whether the container is empty
    - snapshot of the complete list of keys
    - snapshot of the complete set of key/value pairs
- all of these operations are safe if sticking to basic approach
    - don't return references
    - put a simple mutex lock around the entirety of each member function
- operations either come before some modification from another thread or come after it
    - biggest potential for a race condition is when a new key/value pair is being added
        - if two threads add a new value, only one will be first
        - second will therefore fail
- one possibility is to combine add and change into a single member function
- what happens when there is no value for an input key?
- one option is to allow the user to provide a "default" result that's returned in the case when the key isn't present
    - default-constructed instance of mapped_type could be used if the default_value wasn't explicitly provided
- could also be extended to return a std::pair<mapped_type,bool> instead of just an instance of mapped_type
    - bool indicates whether the value was present
- another option is to return a smart pointer referring to the value and return NULL if there was no value to return

- simple lock around every member function would waste the possibilities for concurrency provided by the separate functions for reading the data structure and modifying it
- one option is to use a mutex that supports multiple reader threads or a single writer thread
  - boost::shared_mutex
  - would improve the possibilities for concurrent access
  - only one thread could modify the data structure at a time

Designing a map data structure for fine-grained locking

- three common ways of implementing an associative container
  - binary tree, such as a red-black tree
  - sorted array
  - hash table
- binary tree doesn't provide much potential for extending the opportunities for concurrency
  - every lookup or modification has to start by accessing the root node
- sorted array is worse
  - can't tell in advance where in the array a given data value is going
  - need a single lock for the whole array
- hash table
  - assuming a fixed number of buckets, which bucket a key belongs to is purely a property of the key and its hash function
  - can safely have a separate lock per bucket
  - increase the opportunities for concurrency N-fold, where N is the number of buckets if using a mutex that supports multiple readers or a single writer
  - need a good hash function for the key
  - can use for this purpose std::hash<> template
    - specialized for the fundamental types such as int and common library types such as std::string
    - user can specialize it for other key types
- take the type of the function object to use for doing the hashing as a template parameter
  - user can choose whether to specialize std::hash<> for their key type or provide a separate hash function
- Listing 6.11 A thread-safe lookup table
- uses a std::vector<std::unique_ptr<bucket_type>> to hold the buckets
  - allows the number of buckets to be specified in the constructor
  - default to 19 (arbitrary prime number)
  - hash tables work best with a prime number of buckets
  - each bucket is protected with an instance of boost:shared_mutex to allow many concurrent reads or a single call to either of the modification functions per bucket
  - get_bucket() function can be called without any locking because the number of buckets is fixed

- the bucket mutex can be locked either for shared (read-only) ownership or unique (read/write) ownership
- all three functions make use of the find_entry_for() member function on the bucket to determine whether the entry is in the bucket
- each bucket contains just a std::list<> of key/value pairs
  - adding and removing entries is easy
- exception safety?
  - value_for is fine
  - remove_mapping modifies the list with the call to erase
    - guaranteed not to throw
  - add_or_update_mapping
    - might throw in either of the two branches of the if
    - push_back is exception-safe and will leave the list in the original state if it throws
    - problem is with the assignment in the case where you're replacing an existing value
      - relying on it leaving the original unchanged if it throws
      - doesn't affect the data structure as a whole and is entirely a property of the user-suppliedtype
      - can safely leave it up to the user to handle this
- retrieving a snapshot of the current state into
  - requires locking the entire container in order to ensure that a consistent copy of the state is retrieved
  - no opportunity for deadlock if buckets are locked them in the same order every time
- Listing 6.12 Obtaining contents of a threadsafe_lookup_table as a std::map<>
- increases the opportunity for concurrency of the lookup table as a whole by locking each bucket separately and by using a boost::shared_mutex to allow reader concurrency on each bucket

6.3.2 Writing a thread-safe list using locks

- lifetime of an STL-style iterator is completely outside the control of the container
  - bad idea to support
- alternative is to provide iteration functions such as for_each as part of the container itself
- for_each must call user-supplied code while holding the internal lock to do anything useful
  - must also pass a reference to each item to this user-supplied code in order for the user-supplied code to work on this item
  - could avoid this by passing a copy of each item to the user-supplied code
    - expensive if the data items were large
  - leave it up to the user to ensure that they don't cause deadlock by acquiring locks in the user-supplied operations and don't cause data races by storing the references for access outside the locks

- operations to supply for the list
  - add an item to the list
  - remove an item from the list if it meets a certain condition
  - find an item in the list that meets a certain condition
  - update an item that meets a certain condition
  - copy each item in the list to another container
- add operations such as
  - positional insert (unnecessary for your lookup table)
- one mutex per node
  - benefit here is that operations on separate parts of the list are truly concurrent
    - each operation holds only the locks on the nodes it's actually interested in and unlocks each node as it moves on to the next
- Listing 6.13 A thread-safe list with iteration support
- singly linked list
  - each entry is a node structure
- a default-constructed node is used for the head of the list, which starts with a NULL next pointer
- new nodes are added with the push_front() function
  - new node is constructed -> allocates the stored data on the heap while leaving the next pointer as NULL
  - acquire the lock on the mutex for the head node in order to get the appropriate next value and insert the node at the front of the list by setting head.next to point to the new node
  - lock one mutex in order to add a new item to the list
    - no risk of deadlock
  - slow memory allocation happens outside the lock
- for_each()
  - takes a Function of some type to apply to each element in the list
  - takes this function by value and will work with either a genuine function or an object of a type with a function call operator
  - function must accept a value of type T as the sole parameter
- hand-over-hand locking
  - lock the mutex on the head node
    - then safe to obtain the pointer to the next node (using get() because you're not taking ownership of the pointer)
  - if that pointer isn't NULL
    - lock the mutex on that node in order to process the data
  - once that node is locked
    - release the lock on the previous node and call the specified function

- once the function completes
    - update the current pointer to the node you just processed and move the owership of the lock from next_lk out to lk
- for_each passes each data item directly to the supplied Function
    - can use this to update the items if necessary or copy them into another container
- find_first_if()
    - is similar to for_each()
    - difference is that the supplied Predicate must return true to indicate a match or false to indicate no match
    - once matched, just return the found data rather than continuing to search
    - could do this with for_each(), but it would needlessly continue processing the rest of the list even once a match had been found
- remove_if()
    - has to actually update the list
    - if the Predicate returns true
        - remove the node from the list by updating current->next
        - can then release the lock held on the mutex for the next node
        - node is deleted when the std::unique_ptr you moved it into goes out of scope 2)
        - don't update current because you need to check the new next node
    - if the Predicate returns false
        - just move on as before
- no deadlocks or race conditions
- iteration is always one way, always starting from the head node, and always locking the next mutex before releasing the current one
    - no possibility of different lock orders in different threads
- only potential candidate for a race condition is the deletion of the removed node in remove_if ()
    - this is done after the mutex is unlocked (undefined behavior to destroy a locked mutex)
    - safe
        - still hold the mutex on the previous node (current)
        - no new thread can try to acquire the lock on the node you're deleting
- different threads can be working on different nodes in the list at the same time
    - mutex for each node must be locked in turn, the threads can't pass each other
    - if one thread is spending a long time processing a particular node, other threads will have to wait when they reach that particular node
- improves the possibilities for concurrency over using a single mutex

6.4 Summary

- designing data structures for concurrency

- provides guidelines for doing so
- stack
- queue
- hash map
- linked list
- implement them in a way designed for concurrent access
- using locks to protect data and prevent data races

Ch. 7 - Designing lock-free concurrent data structures

- implementations of data structures designed for concurrency without using locks
- techniques for managing memory in lock-free data structures
- simple guidelines to aid in the writing of lock- free data structures
- incorrect use of locks can lead to deadlock
- granularity of locking can affect the potential for true concurrency
- potential to avoid these problems by writing data structures that are safe for concurrent access without locks
- lock-free data structure
- memory-ordering properties of the atomic operations can be used to build lock-free data structures
- take care when designing such data structures
    - hard to get right
    - conditions that cause the design to fail may occur very rarely

7.1 Definitions and consequences

- algorithms and data structures that use mutexes, condition variables, and futures to synchronize the data are called blocking data structures and algorithms
- application calls library functions that will suspend the execution of a thread until another thread performs an action
    - blocking calls because the thread can't progress past this point until the block is removed
    - OS will suspend a blocked thread completely (and allocate its time slices to another thread) until it's unblocked by the appropriate action of another thread
        - unlocking a mutex
        - notifying a condition variable
        - making a future ready

- data structures and algorithms that don't use blocking library functions are said to be nonblocking
- nonblocking != lock-free

Types of nonblocking data structures

- Listing 7.1 Implementation of a spin-lock mutex using std::atomic_flag
- code doesn't call any blocking functions
    - lock() just keeps looping until the call to test_and_set() returns false
- any code that uses this mutex to protect shared data is nonblocking
    - not lock-free
    - still a mutex and can still be locked by only one thread at a time

Lock-free data structures

- more than one thread must be able to access the data structure concurrently
    - don't have to be able to perform the same operations
    - if one of the threads accessing the data structure is suspended by the scheduler midway through its operation the other threads must still be able to complete their operations without waiting for the suspended thread
- algorithms that use compare/exchange operations on the data structure often have loops in them
    - reason for using a compare/exchange operation is that another thread might have modified the data in the meantime
        - code will need to redo part of its operation before trying the compare/exchange again
    - can still be lock-free if the compare/exchange would eventually succeed if the other threads were suspended
    - essentially have a spin lock otherwise
        - nonblocking but not lock-free
- lock-free algorithms with such loops can result in one thread being subject to starvation
    - other thread might make progress while if another thread performs operations with the "wrong" timing
    - first thread continually has to retry its operation
- data structures that avoid this problem are wait-free as well as lock-free

Wait-free data structures

- wait-free data structure is lock-free
    - additional property that every thread accessing the data structure can complete its operation within a bounded number of steps, regardless of the behavior of other threads
- algorithms that can involve an unbounded number of retries because of clashes with other threads are thus not wait-free

- writing wait-free data structures correctly is extremely hard
- have to ensure that each operation can be performed in a single pass and that the steps performed by one thread don't cause an operation on another thread to fail
  - can make the overall algorithms for the various operations considerably more complex
- need some pretty good reasons to write one
  - need to be sure that the benefit outweighs the cost

The pros and cons of lock-free data structures

- primary reason for using lock-free data structures is to enable maximum concurrency
- potential for one thread to have to block and wait for another to complete its operation before the first thread can proceed with lock-based containers
  - preventing concurrency through mutual exclusion is the entire purpose of a mutex lock
- some thread makes progress with every step with a lock-free data structure
- every thread can make forward progress, regardless of what the other threads are doing with a wait-free data structure
  - no need for waiting
- robustness
  - if a thread dies while holding a lock, that data structure is broken forever with lock-based
  - if a thread dies partway through an operation on a lock-free data structure, nothing is lost except that thread's data
    - other threads can proceed normally
  - must be careful to ensure that the invariants are upheld or choose alternative invariants that can be upheld also if threads can't be excluded from accessing the data structure
  - must pay attention to the ordering constraints imposed on the operations
- must use atomic operations for the modifications to avoid the undefined behavior associated with a data race
- must also ensure that changes become visible to other threads in the correct order
- is the possibility of live locks instead
  - live lock occurs when two threads each try to change the data structure
  - each thread the changes made by the other require the operation to be restarted
  - both threads loop and try again
- live locks are typically short lived because they depend on the exact scheduling of threads
  - hurt performance rather than cause long-term problems
- wait-free code can't suffer from live lock because there's always an upper limit on the number of steps needed to perform an operation
- algorithm is likely more complex than the alternative
- another downside of lock-free and wait-free code
  - can increase the potential for concurrency of operations on a data structure reduce the time an individual thread spends waiting

- may decrease overall performance
- atomic operations used for lock-free code can be much slower than nonatomic operations
  - likely more of them in a lock-free data structure than in the mutex locking code for a lock-based data structure
- hardware must synchronize data between threads that access the same atomic variables
  - cache ping-pong associated with multiple threads accessing the same atomic variables can be a significant performance drain
- important to check the relevant performance aspects both with a lock-based data structure and a lock-free one before committing either way
  - worst-case wait time
  - average wait time
  - overall execution time
  - others

7.2 Examples of lock-free data structures

- TODO: add back sequential steps here
- rely on the use of atomic operations and the associated memory-ordering guarantees in order to ensure that data becomes visible to other threads in the correct order
- default memory_order_seq_cst memory ordering for all atomic operations
  - all memory_order_seq_cst operations form a total order
- reduce some of the ordering constraints to memory_order_acquire for later examples
  - memory_order_release
  - memory_order_relaxed
- only std::atomic_flag is guaranteed not to use locks in the implementation
- simple lock-based data structure might actually be more appropriate on these platforms
  - must identify program's requirements and profile the before choosing an implementation various options that meet those requirements
- writing a thread-safe stack without locks
  - important to ensure that once a value is added to the stack it can safely be retrieved immediately by another thread
  - also important to ensure that only one thread returns a given value
- what works fine for adding a node in a single-threaded context is not enough if other threads are also modifying the stack in a multithreaded context
  - sequential steps
    a. create a new node
    b. set the node's next pointer to the current head
    c. set the head pointer to the new node
  - race condition -> a second thread could modify the value of head between when thread reads it in step 2 and you update it in step 3

- results in the changes made by that other thread being discarded or even worse consequences
  - NOTE: that once head has been updated to point to your new node, another thread could read that node
    - necessary for new node to thoroughly prepared before head is set to point to it
      - cannot modify it afterward
- Listing 7.2 Implementing push() without locks
- to handle race condition use an atomic compare/exchange operation at step 3 to ensure that head hasn't been modified since you read it in step 2
  i. create a new node
  ii. set the node's next pointer to the current head
  iii. set the head pointer to the new node
  - ensured that the node by populating the data in the node structure itself from the node constructor
  - use compare_exchange_weak() to ensure that the head pointer still has the same value as you stored in new_node->next
    - set it to new_node if so
  - if it returns false to indicate that the comparison failed the value supplied as the first parameter (new_node->next) is updated to the current value of head (highlights useful part compare/exchange functionality)
  - don't have to reload head each time through the loop
    - compiler handles
  - can use compare_exchange_weak because looping directly on failure
    - compare_exchange_strong may be better on some architectures
  - only place that can throw an exception is the construction of the new node
    - will clean up after
    - list hasn't been modified yet
    - safe
  - no problematic race conditions here
    - build the data to be stored as part of the node
    - use compare_exchange_weak() to update
    - node is on the list and ready for use once the compare/exchange succeeds
  - no locks, so no possibility of deadlock
- simplified view of pop (sequential process)
  i. read the current value of head
  ii. read head->next
  iii. set head to head->next
  iv. return the data from the retrieved node
  v. delete the retrieved node

- both of two threads might read the same value of head at step 1 if there are two threads removing items from the stack
    - if one thread then proceeds all the way through to step 5 before the other gets to step 2, the second thread will be dereferencing a dangling pointer
    - memory management is one of the biggest issues in writing lock-free code
        - for now just leave out step 5 and leak the nodes
- if two threads read the same value of head, they'll return the same node
    - violates the intent of the stack data structure
    - use compare/exchange to update head
        - if the compare/exchange fails, either a new node has been pushed on or another thread just popped the node you were trying to pop
    - need to return to step 1 (although the compare/exchange call rereads head for you) either way
    - can be sure current thread is the only thread that's popping the given node off once the compare/exchange call succeeds
        - can safely execute step 4
- code for first cut at pop
- problems aside from the leaking node
    - doesn't work on an empty list
        - if head is a null pointer, it will cause undefined behavior as it tries to read the next pointer
            - fixed by checking for nullptr in the while loop and either throwing an exception on an empty stack or returning a bool to indicate success or failure
    - second problem is an exception-safety issue
        - if an exception is thrown when copying the return value, the value is lost
        - cannot pass in a reference to the result in this case
            - can only safely copy the data once sure current thread is the only thread returning the node
                - means the node has already been removed from the queue
            - passing in the target for the return value by reference is no longer an advantage
                - might as well just return by value
        - have to use the other option to return a (smart) pointer to the data value to return the value safely
            - can just return nullptr to indicate that there's no value to return
                - requires that the data be allocated on the heap
        - no to do the heap allocation as part of the pop() because the heap allocation might throw an exception
            - can allocate the memory when you push() the data onto the stack
                - have to allocate memory for the node anyway
        - returning a std::shared_ptr<> won't throw an exception so pop() is now safe
- Listing 7.3 A lock-free stack that leaks nodes

Managing memory in lock-free data structures

- TODO: check section number here
- opted to leak nodes in order to avoid the race condition where one thread deletes a node while another thread still holds a pointer to it that it's just about to dereference
  - leaking memory isn't acceptable in any sensible C++ program
- basic problem is
- basic problem - need to free a node but you can't do so until certain there are no other threads that still hold pointers to it
  - OK if only one thread ever calls pop() on a particular stack instance
    - push() doesn't touch the node once it's been added to the stack
    - thread that called pop() must be the only thread that can touch the node
      - it can safely delete it
  - to handle multiple threads calling pop() on the same stack instance
    - need some way to track when it's safe to delete a node
    - means there is a need to write a special-purpose garbage collector just for nodes
    - only requires checking for nodes which are accessed from pop()
      - not worried about nodes in push()
        - only accessible from one thread until they're on the stack
    - safe to delete all the nodes currently awaiting deletion if there are no threads calling pop()
      - can delete all nodes added to the "to delete" list when there are no threads calling pop() if they've been added when the data was extracted
    - count nodes to ensure no threads are calling pop
      - increment a counter on entry and decrement counter on exit
      - safe to delete the nodes from the "to be deleted" list when the counter is zero
      - has to be an atomic counter so it can safely be accessed from multiple threads
- Listing 7.4 Reclaiming nodes when no threads are in pop()
- Listing 7.5 The reference-counted reclamation machinery
- see text for code description
- deleting a node is potentially a time-consuming operation
  - want the window in which other threads can modify the list to be as small as possible
  - may never be such a quiescent state in high-load situations

7.2.3 Detecting nodes that can't be reclaimed using hazard pointers

- hazard pointers
  - technique discovered by Maged Michael

- deleting a node that might still be referenced by other threads is hazardous if other threads do indeed hold references to that node and proceed to access the node through that reference results in undefined behavior
- thread first sets a hazard pointer to reference an object that another thread might want to delete if it is going to access that object
  - informs the other thread that deleting the object would indeed be hazardous
  - hazard pointer is cleared once the object is no longer needed
  - must first check the hazard pointers belonging to the other threads in the system when it wants to delete an object
  - object can safely be deleted if none of the hazard pointers reference the object
  - list of objects that have been left until later is checked periodically to see if any of them can be deleted

- need a location to store the pointer to the object being accessed, the hazard pointer

  - must be visible to all threads
  - need one of these for each thread that might access the data structure

- assume there is a function get_hazard_pointer_for_current_thread() that returns a reference to hazard pointer

  - set it when you read a pointer that you intend to dereference
  - to do this in a while loop to ensure that the node hasn't been deleted between the reading of the old head pointer and the setting of the hazard pointer
  - during this window no other thread knows this particular node is being accessed
  - if the old head node is going to be deleted, head itself must have changed
  - can check this and keep looping until you know that the head pointer still has the same value you set your hazard pointer

- using hazard pointers like this relies on the fact that it's safe to use the value of a pointer after the object it references has been deleted

  - technically undefined behavior if you are using the default implementation of new and delete
  - either need to ensure that your implementation permits it
  - need to use a custom allocator that permits such usage

- can proceed with the rest of pop() once hazard pointer is set knowing that no other thread will delete the nodes

  - need to update the hazard pointer before you dereference every time the old head is read

- can clear your hazard pointer once the node has been extracted from the list, you

- if there are no other hazard pointers referencing your node, you can safely delete it

- otherwise, add it to a list of nodes to be deleted later

- Listing 7.6 An implementation of pop() using hazard pointers

- Listing 7.7 A simple implementation of get_hazard_pointer_for_ current_thread()

- Listing 7.8 A simple implementation of the reclaim functions

- TODO: find where 7.2.4 starts

Better reclamation strategies using hazard pointers

- Listing 7.9 A lock-free stack using a lock-free std::shared_ptr<> implementation
- Listing 7.10 Pushing a node on a lock-free stack using split reference counts

7.2.5 Applying the memory model to the lock-free stack

7.2.6 Writing a thread-safe queue without locks

- Listing 7.13 A single-producer, single-consumer lock-free queue

Handling multiple threads in push()

- Listing 7.15 Implementing push() for a lock-free queue with a reference-counted tail
- Listing 7.16 Popping a node from a lock-free queue with a reference-counted tail
- Listing 7.17 Releasing a node reference in a lock-free queue
- Listing 7.18 Obtaining a new reference to a node in a lock-free queue
- Listing 7.19 Freeing an external counter to a node in a lock-free queue

Making the queue lock-free by helping out another thread

- Listing 7.20 pop()modifiedtoallowhelpingonthepush()side
- Listing 7.21 A sample push() with helping for a lock-free queue

7.3 Guidelines for writing lock-free data structures

- general guidelines regarding concurrent data structures still apply
  - need more than that

7.3.1 Guideline: use std::memory_order_seq_cst for prototyping

- std::memory_order_seq_cst is much easier to reason about than any other memory ordering because all such operations form a total order
- all the examples started with std::memory_order_seq_cst and only relaxed the memory-ordering constraints once the basic operations were working
  - using other memory orderings is an optimization

- avoid doing it prematurely
- can only determine which operations can be relaxed when you can see the full set of code that can operate on the guts of the data structure
- just running the code isn't enough to test
  - having an algorithm checker that can systematically test all possible combinations of thread visibilities that are consistent with the specified ordering guarantees (such things do exist) is the only way to have guarantees about the code

7.3.2 Guideline: use a lock-free memory reclamation scheme

- one of the biggest difficulties with lock-free code is managing memory
  - essential to avoid deleting objects when other threads might still have references to them
  - delete the object as soon as possible in order to avoid excessive memory consumption
- three techniques for ensuring that memory can safely be reclaimed
  - waiting until no threads are accessing the data structure and deleting all objects that are pending deletion
  - using hazard pointers to identify that a thread is accessing a particular object
  - reference counting the objects so that they aren't deleted until there are no outstanding references
- key idea is to use some method to keep track of how many threads are accessing a particular object and only delete each object when it's no longer referenced from anywhere
- many other ways of reclaiming memory in lock-free data structures
  - ideal scenario for using a garbage collector
  - recycle nodes and only free them completely when the data structure is destroyed
    - memory never becomes invalid because the nodes are reused so some of the difficulties in avoiding undefined behavior go away

7.3.3 Guideline: watch out for the ABA problem

- ABA problem is something to be wary of in any compare/exchange–based algorithm
- outline
  i. thread 1 reads an atomic variable x and finds it has value A
  ii. thread 1 performs some operation based on this value
     - dereferencing it (if it's a pointer)
     - doing a lookup
     - etc.
  iii. thread 1 is stalled by the operating system
  iv. another thread performs some operations on x that changes its value to B
  v. a thread then changes the data associated with the value A such that the value held by thread 1 is no longer valid this may be as drastic as freeing the pointed-to memory or just changing an associated value

vi. a thread then changes x back to A based on this new data if this is a pointer
  - may be a new object that just happens to share the same address as the old one

vii. thread 1 resumes and performs a compare/exchange on x,
  - comparing against A the compare/exchange succeeds (because the value is indeed A)
  - this is the wrong A value the data originally read at step 2 is no longer valid
  - thread 1 has no way of telling and will thus corrupt the data structure

- easy to write lock-free algorithms that suffer from this problem
- most common way to avoid the problem is to include an ABA counter alongside the variable x
  - compare/exchange operation is then done on the combined structure of x plus the counter as a single unit
  - every time the value is replaced, the counter is incremented
  - even if x has the same value, the compare/exchange will fail if another thread has modified x
- ABA problem is particularly prevalent in algorithms that use free lists or otherwise recycle nodes rather than returning them to the allocator

7.3.4 Guideline: identify busy-wait loops and help the other thread

- a busy-wait loop is effectively a blocking operation and might as well use mutexes and locks
- can remove the busy-wait by modifying the algorithm so that the waiting thread performs the incomplete steps if it's scheduled to run before the original thread completes the operation
  - operation is no longer blocking
- queue example
  - required changing a data member to be an atomic variable rather than a nonatomic variable
  - using compare/exchange operations to set it
- might require more extensive changes in more complex data structures it

7.4 Summary

- simple implementations of various lock-free data structures
  - stack
  - queue
- must take care with the memory ordering on atomic operations to ensure that there are no data races and that each thread sees a coherent view of the data structure
- memory management becomes much harder for lock-free data structures than lock-based ones
  - mechanisms for handling it

- how to avoid creating wait loops by helping the thread that is being waited for to complete its operation
- difficult and easy to make mistakes
- have scalability properties that are important in some situations
- better equipped to
    - design your own lock-free data structure
    - implement one from a research paper
    - find the bug in the one your former colleague wrote just before leaving the company
- need to think about the data structures used when data shared between threads
    - how the data is synchronized between threads
- can encapsulate that responsibility in the data structure by designing data structures for concurrency

Ch. 8 - Designing concurrent code

- techniques for dividing data between threads
- factors that affect the performance of concurrent code
- how performance factors affect the design of data structures
- exception safety in multithreaded code
- scalability
- example implementations of several parallel algorithms
- more to designing concurrent code than the design and use of basic data structures
- same principles apply at all scales of an application
- vital to think carefully about the design of concurrent code
- even more factors to consider with multithreaded code than with sequential code
- usual encapsulation, coupling, and cohesion
- also need to consider
    - which data to share
    - how to synchronize accesses to that data
    - which threads need to wait for which other threads to complete certain operations
    - etc
- high-level (but fundamental) considerations of how many threads to use
- which code to execute on which thread
- how this can affect the clarity of the code
- low-level details of how to structure the shared data for optimal performance

8.1 Techniques for dividing work between threads

- decide how many threads to use and what tasks they should be doing

- decide whether to have "generalist" threads that do whatever work is necessary at any point in time or "specialist" threads that do one thing well, or some combination
- always need to make these decisions when dealing with concurrency
    - will have a crucial effect on the performance and clarity of the code

Dividing data between threads before processing begins

- easiest algorithms to parallelize are simple algorithms such as std::for_each that perform an operation on each element in a data set
- to parallelize
    - assign each element to one of the processing threads
- how the elements are best divided for optimal performance depends very much on the details of the data structure
- simplest means of dividing the data is to allocate the first N elements to one thread, the next N elements to another thread, and so on,
- no matter how the data is divided, each thread then processes just the elements it has been assigned without any communication with the other threads until it has completed its processing
- common pattern in Message Passing Interface (MPI) or OpenMP2 frameworks
    - task is split into a set of parallel tasks, the worker threads run these tasks independently, and the results are combined in a final reduction step
- both the parallel tasks and the final reduction step are accumulations
- for a simple for_each, the final step is a no-op because there are no results to reduce
    - identifying this final step as a reduction is important
        - naive implementation will perform this reduction as a final serial step
        - step can often be parallelized also
- accumulate actually is a reduction operation itself
    - could be modified to call itself recursively where the number of threads is larger than the minimum number of items to process on a thread
- worker threads could be made to perform some of the reduction steps as each one completes its task, rather than spawning new threads each time
- sometimes the data can't be divided neatly up front because the necessary divisions become apparent only as the data is processed
- apparent with recursive algorithms such as Quicksort
    - need a different approach

8.1.2 Dividing data recursively

- need to make use of the recursive nature to parallelize Quicksort
- recursive calls are entirely independent
    - access separate sets of elements
    - prime candidates for concurrent execution

- used std::async() to spawn asynchronous tasks for the lower chunk at each stage
- spawning a new thread for each recursion would quickly result in a lot of threads
- having too many threads might actually slow down the application
    - also a possibility of running out of threads if the data set is very large
- need to keep better control on the number of threads
- std::async() can handle this in simple cases
- one alternative is to use the std::thread::hardware_concurrency() function to choose the number of threads,
    - just push the chunk to be sorted onto a thread-safe stack rather than starting a new thread for the recursive calls
    - if a thread has nothing else to do, either because it has finished processing all its chunks or because it's waiting for a chunk to be sorted, it can take a chunk from the stack and sort that
- NOTE: this highlights an important flip in thinking -> managing threads and feeding work (push model) vs. spawning threads and allowing them to grab work when necessary (pull model)
- Listing 8.1 Parallel Quicksort using a stack of pending chunks to sort
- have to wait for lower chunk to be ready because it might be handled by another thread
    - try to process chunks from the stack on current thread while it is waiting
        - pops a chunk off the stack and sort it storing the result in a promise
        - promise is then picked up by the thread that posted the chunk on the stack
- freshly spawned threads sit in a loop trying to sort chunks off the stack while the end_of_data flag isn't set
- in between checking, they yield to other threads to give them a chance to put some more work on the stack
- code relies on the destructor of your sorter class to tidy up threads
- no longer have the problem of unbounded threads that you have with a spawn_task that launches a new thread
- no longer relying on the C++ Thread Library to choose the number of threads
- limit the number of threads to the value of std::thread::hardware_concurrency() in order to avoid excessive task switching
- another potential problem
    - the management of these threads and the communication between them add quite a lot of complexity to the code
- all threads access the stack to add new chunks and remove chunks even though the threads are processing separate data elements
    - heavy contention can reduce performance
    - even with a lock-free (and hence nonblocking) stack
- approach is a specialized version of a thread pool
    - set of threads that each take work to do from a list of pending work, do the work, and then go back to the list for more

- if the data is dynamically generated or is coming from external input, this approach doesn't work
    - might make more sense to divide the work by task type rather than dividing based on the data

8.1.3 Dividing work by task type

- make the threads specialists
    - each performs a distinct task,
    - threads may or may not work on the same data
    - if they do it's for different purposes
- separating concerns with concurrency
    - each thread has a different task, which it carries out independently of other threads
    - other threads may occasionally give the current thread data or trigger events that it needs to handle
    - in general each thread focuses on doing one thing well
- each piece of code should have a single responsibility

Dividing work by task type to separate concerns

- migrating from single responsibility in sequential code to concurrent code
    - if everything is independent and the threads have no need to communicate with each other, then it is as easy as passing various tasks to different threads
- two big dangers with separating concerns with multiple threads
    - possibly end up separating the wrong concerns
        - check for
            - is there a lot of data shared between the threads
            - do the different threads end up waiting for each other
        - i.e. too much communication between threads
        - look at the reasons for the communication
            - if all the communication relates to the same issue, maybe that should be the key responsibility of a single thread and extracted from all the threads that refer to it
            - if two threads are communicating a lot with each other but much less with other threads maybe they should be combined into a single thread
- don't have to be limited to completely isolated cases
    - can divide the work so each thread performs one stage from the overall sequence if multiple sets of input data require the same sequence of operations to be applied

Dividing a sequence of tasks between threads

- can use a pipeline to exploit the available concurrency of the system if the task consists of applying the same sequence of operations to many independent data items
    - data flows in at one end through a series of operations (pipes) and out at the other end

- create a separate thread for each stage in the pipeline—one thread for each of the operations in the sequence
  - when the operation is completed, the data element is put on a queue to be picked up by the next thread
- allows the thread performing the first operation in the sequence to start on the next data element while the second thread in the pipeline is working on the first element
- alternative to just dividing the data between threads
- appropriate in circumstances where the input data itself isn't all known when the operation is started

8.2 Factors affecting the performance of concurrent code

- need to know what factors are going to affect the performance when using concurrency in order to improve the performance of code on systems with multiple processors
- need to ensure use of multiple doesn't adversely affect performance even if when just using multiple threads to separate concerns
- many factors affect the performance of multithreaded code

How many processors?

- the number (and structure) of processors is the first big factor that affects the performance of a multithreaded application
- may know exactly what the target hardware is and can thus design with this in mind
  - allows for taking real measurements on the target system or an exact duplicate
- general don't have this advantage
- may be a similar system, but the differences can be crucial
  - behavior and performance characteristics of a concurrent program can vary considerably under such different circumstances
  - need to think carefully about what the impact may be and test things where possible
- oversubscription - having more threads actually running (and not blocked, waiting for something) than available based on hardware
  - application will waste processor time switching between the threads
- use std::thread::hardware_concurrency() to allow applications to scale the number of threads in line with the number of threads the hardware can run concurrently
  - code doesn't take into account any of the other threads that are running on the system unless you explicitly share that information
  - worst case - if multiple threads call a function that uses std::thread::hardware_concurrency() for scaling at the same time, there will be huge oversubscription
    - std::async() avoids this problem because the library is aware of all calls and can schedule appropriately

- careful use of thread pools can also avoid this problem
- still subject to the impact of other applications running at the same time even if you take into account all threads running in your application
- use of multiple CPU-intensive applications simultaneously is rare on single-user systems
- some domains where it's more common
- systems designed to handle this scenario typically offer mechanisms to allow each application to choose an appropriate number of threads
  - outside the scope of the C++ Standard
- one option is for a std::async()-like facility to take into account the total number of asynchronous tasks run by all applications when choosing the number of threads
- another is to limit the number of processing cores that can be used by a given application
  - limit should be reflected in the value returned by std::thread::hardware_concurrency() on such platforms, although this isn't guaranteed
- consult system documentation to see what options are available
- ideal algorithm for a problem can depend on the size of the problem compared to the number of processing units
- massively parallel system with many processing units, an algorithm that performs more operations overall may finish more quickly than one that performs fewer operations, because each processor performs only a few operations

8.2.2 Data contention and cache ping-pong

- problem
  - multiple processors trying to access the same data if two threads are executing concurrently on different processors and they're both reading the same data
    - usually OK - data will be copied into their respective caches, and both processors can proceed
  - modification of data has to propagate to the cache on the other core, which takes time
  - such a modification may cause the second processor to stop in its tracks and wait for the change to propagate through the memory hardware
    - depending on the nature of the operations on the two threads, and the memory orderings used for the operations
  - exact timing depends primarily on the physical structure of the hardware but may be very slow
- high contention
  - if processors rarely have to wait for each other, you have low contention
- data passed back and forth between the caches many times is called cache ping-pong
  - can seriously impact the performance of the application
  - if a processor stalls because it has to wait for a cache transfer, it can't do any work in the meantime, even if there are other threads waiting that could do useful work

- problem example - acquiring a mutex in a loop
  - in order to lock the mutex, another thread must transfer the data that makes up the mutex to its processor and modify it
  - when it's done, it modifies the mutex again to unlock it, and the mutex data has to be transferred to the next thread to acquire the mutex
  - transfer time is in addition to any time that the second thread has to wait for the first to release
  - if the data and mutex really are accessed by more than one thread
    - adding more cores and processors to the system makes it more likely that to get high contention and one processor having to wait for another
  - using multiple threads to process the same data more quickly, the threads are competing for the data and thus competing for the same mutex
- effects of contention with mutexes are usually different from the effects of contention with atomic operations
  - use of a mutex naturally serializes threads at the operating system level rather than at the processor level
  - operating system can schedule another thread to run while one thread is waiting for the mutex if there are enough threads ready to run
  - processor stall prevents any threads from running on that processor
  - will still impact the performance of those threads that are competing for the mutex
    - can only run one at a time
  - rarely updated data structure can be protected with a single-writer, multiple-reader mutex
- cache ping-pong effects can nullify the benefits of such a mutex if the workload is unfavorable
  - all threads accessing the data (even reader threads) still have to modify the mutex itself
  - contention on the mutex itself increases as the number of processors accessing the data goes up
    - cache line holding the mutex must be transferred between cores
    - potentially increases the time taken to acquire and release locks to undesirable levels
- techniques to handle
  - spread out the mutex across multiple cache lines
  - must implement mutex otherwise subject to whatever the system provides
- avoid cache ping-pong by doing everything to reduce the potential for two threads competing for the same memory location
  - can still get cache ping-pong even if a particular memory location is only ever accessed by one thread
  - due to an effect known as false sharing

8.2.3 False sharing

- caches deal in blocks of memory called cache lines rather than individual memory locations
    - typically 32 or 64 bytes in size
    - exact details depend on the architecture
- small data items in adjacent memory locations will be in the same cache line
    - better for performance of the application if a set of data accessed by a thread is in the same cache line
    - can cause performance problems if the data items in a cache line are unrelated and need to be accessed by different threads
- array of int values and a set of threads that each access and modify their own entry in the array repeatedly
    - many array entries will be in the same cache line
    - cache hardware still ping-pongs even though each thread only accesses its own array entry
    - ownership of the cache line needs to be transferred to the processor running that thread every time the thread accessing entry 0 needs to update the value
    - then transferred to the cache for the processor running the thread for entry 1 when that thread needs to update its data item
    - none of the data is shared even though the cache line is
    - false sharing
- structure the data so that data items to be accessed by the same thread are close together in memory
    - more likely to be in the same cache line
    - data accessed by separate threads are far apart in memory and thus more likely to be in separate cache lines

8.2.4 How close is your data?

- false sharing is caused by having data accessed by one thread too close to data accessed by another
- data proximity
    - another pitfall associated with data layout directly impacts the performance of a single thread
    - likely data accessed by a single thread lies in separate cache lines if the data is spread out in memory
    - data accessed by a single thread is more likely to lie on the same cache line if it is close together in memory
    - if data is spread out, more cache lines must be loaded from memory onto the processor cache,

- can increase memory access latency and reduce performance compared to data that's located close together
- increased chance that a given cache line containing data for the current thread also contains data that's not for the current thread if the data is spread out
- important with single-threaded code but also impacts task switching
  - each core runs multiple threads if there are more threads than cores in the system
  - increases the pressure on the cache when attempting to avoid false sharing
  - processor is more likely to have to reload the cache lines if each thread uses data spread across multiple cache lines than if each thread's data is close together in the same cache line when switching threads
  - OS might also choose to schedule a thread on one core for one time slice and then on another core for the next time slice
    - requires transferring the cache lines for that thread's data from the cache for the first core to the cache for the second
    - more time consuming for more cache lines that need transferring
- OSes typically avoid this when they can but it does happen
- task-switching problems are prevalent when lots of threads are ready to run as opposed to waiting i.e. oversubscription

8.2.5 Oversubscription and excessive task switching

- typical to have more threads than processors in multithreaded systems
  - unless running on massively parallel hardware
- threads often spend time waiting for external I/O to complete or blocked on mutexes or waiting for condition variables and so forth
  - num threads vs processors is generally not a problem
  - having extra threads enables the application to perform useful work rather than having processors sitting idle while the threads wait
  - not always good
- will be more threads ready to run than there are available processors if there are too many additional threads
  - operating system will have to start task switching quite heavily in order to ensure they all get a fair time slice
  - can increase the overhead of the task switching as well as compound any cache problems resulting from lack of proximity
- oversubscription can arise
  - when there is a task that repeatedly spawns new threads without limits
  - where the natural number of threads when separating by task type is more than the number of processors and the work is naturally CPU bound rather than I/O bound
- spawning too many threads because of data division can limit the number of worker threads
- not a lot that can be done if the oversubscription is due to the natural division of work

- choosing the appropriate division may require more knowledge of the target platform
    - only worth doing if performance is unacceptable and it can be demonstrated that changing the division of work does improve performance
- cost of cache ping-pong can vary quite considerably between two single-core processors and a single dual-core processor

8.3 Designing data structures for multithreaded performance

- using techniques for dividing work between threads and knowledge of various factors that can affect the performance of code
    - layout of the data used by a single thread can have an impact
- REMEMBER:
    - contention
    - false sharing
    - data proximity
- each can have a big impact on performance
    - can often improve performance just by altering the data layout or changing which data elements are assigned to which thread

8.3.1 Dividing array elements for complex operations

- for matrix multiplication need to pay careful attention to the data access patterns in order to get optimal performance
    - many ways work can be divided between threads
- with more rows/columns than available processors
    - could have each thread calculate the values for a number of columns in the result matrix
    - could have each thread calculate the results for a number of rows
    - could have each thread calculate the results for a rectangular subset of the matrix
- better to access contiguous elements from an array rather random access
    - reduces cache usage and the chance of false sharing
- each thread needs to read every value from the first matrix and the values from the corresponding columns in the second matrix when having each thread handle a set of columns
    - only have to write the column values
    - accessing N elements from the first row, N elements from the second, and so forth given that the matrices are stored with rows (where N is the number of columns being processed)
    - should access adjacent columns since other threads will be accessing the other elements of each row
        - N elements from each row are adjacent
    - minimizes false sharing

- no false sharing if the space occupied by N elements is an exact number of cache lines
    - threads will be working on separate cache lines
- each thread needs to read every value from the second matrix and the values from the corresponding rows of the first matrix if it handles a set of rows
    - only has to write the row values
    - matrices are stored with the rows contiguous
    - accessing all elements from N rows
- thread is the only thread writing to N rows when choosing adjacent rows
    - has a contiguous block of memory that's not touched by any other thread
    - probably an improvement over having each thread handle a set of columns
    - only possibility of false sharing is for the last few elements of one block with the first few of the next
- time it!
- third option
    - divide into rectangular blocks
- can be viewed as dividing into columns and then dividing into rows
    - has the same false-sharing potential as division by columns
- advantage to rectangular division from the read side
    - don't need to read the entirety of either source matrix
    - only need to read the values corresponding to the rows and columns of the target rectangle
- important to profile various options on the target architecture
- same principles apply to any situation when working with large blocks of data to divide between threads
    - look at all the aspects of the data access patterns carefully
    - identify potential causes of performance hits
- may be similar circumstances in your problem domain where changing the division of work can improve performance without requiring any change to the basic algorithm

8.3.2 Data access patterns in other data structures

- considerations when optimizing the data access patterns data structures
    - try to adjust the data distribution between threads so that data that's close together is worked on by the same thread
    - try to minimize the data required by any given thread
    - try to ensure that data accessed by separate threads is sufficiently far apart to avoid false sharing
- not easy to apply to other data structures
    - binary trees are inherently difficult to subdivide in any unit other than a subtree
    - may not be useful, depending on how balanced the tree is

- nature of the trees means that the nodes are likely dynamically allocated and thus end up in different places on the heap
- having data end up in different places on the heap isn't a problem in itself
  - means that the processor has to keep more things in cache
  - can be beneficial if multiple threads need to traverse the tree
    - all need to access the tree nodes
  - processor only has to load the data from memory if it's actually needed if the tree nodes only contain pointers to the real data held at the node
  - can avoid the performance hit of false sharing between the node data itself and the data that provides the tree structure if the data is being modified by the threads that need it
- similar issue with data protected by a mutex
  - simple class that contains a few data items and a mutex used to protect accesses from multiple threads
  - ideal for a thread that acquires the mutex if the mutex and the data items are close together in memory
    - data it needs may already be in the processor cache because it was just loaded in order to modify the mutex
  - if other threads
  - other threads will need access to that memory if they try to lock the mutex while it's held by the first thread
- mutex locks are typically implemented as a read-modify-write atomic operation on a memory location within the mutex to try to acquire the mutex
  - followed by a call to the operating system kernel if the mutex is already locked
  - read-modify-write operation may cause the data held in the cache by the thread that owns the mutex to be invalidated
  - thread isn't going to touch the mutex until it unlocks it so not a problem for mutex
  - thread that owns the mutex can take a performance hit because another thread tried to lock the mutex if the mutex shares a cache line with the data being used by the thread
- to test for this kind of false sharing
  - add large blocks of padding between the data elements that can be concurrently accessed by different threads
  - know that false sharing was a problem if it improves performance
    - leave the padding in or work to eliminate the false sharing in another way by rearranging the data accesses
- more than just the data access patterns to consider when designing for concurrency

8.4 Additional considerations when designing for concurrency

- consider
  - exception safety
  - scalability

- scalable if the performance (reduced speed of execution or increased throughput) increases as more processing cores are added to the system
  - ideal performance increase is linear
- without exception safety
  - can end up with broken invariants or race conditions
  - application might terminate unexpectedly because an operation threw an exception

8.4.1 Exception safety in parallel algorithms

- essential aspect of good C++ code
- parallel algorithms often require more care with regard to exceptions than normal sequential algorithms
- sequential algorithm only has to worry about ensuring related state is cleaned up to avoid resource leaks and broken invariants when an exception is thrown
  - can allow the exception to propagate to the caller
- when running operations on a separate thread in a parallel algorithm
  - exception can't be allowed to propagate because it's on the wrong call stack
  - application is terminated if a function spawned on a new thread exits with an exception
- Listing 8.2 A naive parallel version of std::accumulate (from listing 2.8)
- places where an exception can be thrown
  - anywhere where function is called that can throw or you perform an operation on a user-defined type that may throw
- no catch blocks
  - exceptions will be left unhandled and cause the library to call std::terminate() and abort the application

Adding exception safety

- start by addressing exceptions thrown on new threads
- trying to calculate a result to return while allowing for the possibility that the code might throw an exception
  - this is what the combination of std::packaged_task and std::future is designed for
- rearrange code to use std::packaged_task
- Listing 8.3 A parallel version of std::accumulate using std::packaged_task
- exceptions thrown in the worker threads are rethrown in the main thread
  - if more than one of the worker threads throws an exception, only one will be propagated
  - can use something like std::nested_exception to capture all the exceptions and throw that instead if it really matters
- remaining problem
  - leaking threads if an exception is thrown between when first thread is spawned and when they are all joined

- simplest solution is just to catch any exceptions, join with the threads that are still joinable(), and rethrow the exception
- works but results in ugly try-catch blocks and duplicate code
- see code for RAII-style resource clean-up in destructor similar to thread_guard extended for the whole vector of threads
- Listing 8.4 An exception-safe parallel version of std::accumulate

Exception safety with std::async()

- with std::async() library takes care of managing the threads
  - any threads spawned are completed when the future is ready
- NOTE: for exception safety, destructor will wait for the thread to complete when destroying the future without waiting for it
  - avoids the problem of leaked threads that are still executing and holding references to the data
- Listing 8.5 An exception-safe parallel version of std::accumulate using std::async
- uses recursive division of the data rather than pre-calculating the division of the data into chunks
  - simpler
  - exception safe
- captured by the future if the asynchronous call throws
  - call to get() will rethrow the exception

8.4.2 Scalability and Amdahl's law

- scalability
  - ensure that application can take advantage of additional processors in the system it's running on
- single-threaded application -> unscalable
- other extreme -> something like the SETI@Home3 project
- number of threads that are performing useful work will vary as the program runs for any given multithreaded program
- application may initially have only one thread which will then have the task of spawning all the others
- threads often spend time waiting for each other or waiting for I/O operations to complete
- have a processor sitting idle that could be doing useful work every time one thread has to wait for something unless there's another thread ready to take its place on the processor
- divide the program into "serial" sections where only one thread is doing any useful work and "parallel" sections where all the available processors are doing useful work
- "parallel" sections will theoretically be able to complete more quickly when application is on a system with more processors
- Amdahl's law

- tasks are rarely infinitely divisible in the way that would be required for the equation to hold
- rare for everything to be CPU bound in the way that's assumed
  - threads may wait for many things while executing
- worth looking at the overall design of the application to maximize the potential for concurrency and ensure that there's always useful work for the pro- cessors to be doing when using concurrency for performance
  - reduce the size of the "serial" sections
  - reduce the potential for threads to wait
- alternatively provide more data for the system to process
  - keep the parallel sections working
  - reduces the serial fraction
- scalability is about reducing the time it takes to perform an action or increasing the amount of data that can be processed in a given time as more processors are added
  - sometimes equivalent

8.4.3 Hiding latency with multiple threads

- threads frequently block while waiting for
  - I/O
  - to acquire a mutex
  - waiting for another thread to complete some operation
  - sleep for a period of time
- having blocked threads means wasting CPU time
- can make use of that spare CPU time by running one or more additional threads
- work stealing, context management within application, etc.
- at some point system will slow down again as it spends more and more time task switching
- important to measure performance before and after any change in the number of threads
  - optimal number of threads will be highly dependent on the nature of the work being done and the percentage of time the thread spends waiting
- might be possible to use up this spare CPU time without running additional threads
- might make sense to use asynchronous I/O if that's available if a thread is blocked because it's waiting for an I/O operation to complete
  - thread can perform other useful work while the I/O is performed in the background
- rather than blocking when a thread is waiting for another thread to perform an operation waiting thread might be able to perform that operation itself
- waiting thread might perform the task in entirety itself or another task that's incomplete if a thread is waiting for a task to be completed and that task hasn't yet been started by any thread
- sometimes pays to add threads to ensure that external events are handled in a timely manner rather than adding threads to ensure that all available processors are being used,

8.4.4 Improving responsiveness with concurrency

- systems and GUIs often process events in an event loop
- structure is generally the same even when details of APIs vary
    - wait for an event
    - processing event
    - wait for the next event
- can make long-running tasks hard to write in a single-threaded application
- get_event()/process() code must be called with reasonable frequency to ensure that use input is handled in a responsive way
    - task must periodically suspend itself and return control to the event loop
    - get_event()/process() code must be called from within the code at convenient points
    - each option complicates the implementation of the task
- can put the lengthy task on a whole new thread and leave a dedicated GUI thread to process the events
    - separation of concerns
- threads can then communicate through simple mechanisms rather than having to somehow mix the event-handling code in with the task code
- Listing 8.6 Separating GUI thread from task thread
- responsiveness is key to the user experience when using an application
- dedicated event-handling thread allows the GUI to handle GUI-specific messages without interrupting the execution of the time-consuming processing
    - still passes on the relevant messages where they do affect the long-running task
- most considerations will become second nature over time when working with multithreaded code

8.5 Designing concurrent code in practice

- extent to which each of the issues described previously will need to be considered will depend on the task
- implementations demonstrate particular techniques
    - not state-of-the-art implementations
    - more advanced implementations that make better use of the available hardware concurrency may be found in the academic literature on parallel algorithms or in specialist multithreading libraries such as Intel's Threading Building Blocks

8.5.1 A parallel implementation of std::for_each

- Listing 8.7 A parallel version of std::for_each template<typename Iterator,typename Func>
- Listing 8.8 A parallel version of std::for_each using std::async

8.5.2 A parallel implementation of std::find

8.5.3 A parallel implementation of std::partial_sum

Implementing the incremental pairwise algorithm for partial sums

- (SIMD) instructions allow processors to execute additions in lockstep
  - it is rare to be able to limit execution to architectures where this is definitely available
- must design code for the general case
  - explicitly synchronize the threads at each step
- barrier
  - synchronization mechanism that causes threads to wait until the required number of threads has reached the barrier
  - once all the threads have reached the barrier, they're all unblocked and may proceed
- none in C++11 Thread Library
- Listing 8.12 A simple barrier class
- Listing 8.13 A parallel implementation of partial_sum by pairwise updates

8.6 Summary

- various techniques for dividing work between threads
  - dividing the data beforehand
  - using a number of threads to form a pipeline
- issues surrounding the performance of multithreaded code from a low-level perspective
  - false sharing
  - data contention
  - how the patterns of data access can affect the performance of code
- additional considerations in the design of concurrent code
  - exception safety and scalability
- parallel algorithm implementations
- idea of a thread pool
  - preconfigured group of threads that run tasks assigned to the pool
  - a lot of thought goes into the design of a good thread pool

Ch. 9 - Advanced thread management

- thread pools
- handling dependencies between pool tasks
- work stealing for pool threads
- interrupting threads
- previously explicitly managing threads by creating std::thread objects for every thread
  - have to manage the lifetime of the thread objects
  - determine the number of threads appropriate to the problem and to the current hardware
  - etc.
- ideal ->
  - divide the code into the smallest pieces that can be executed concurrently
  - pass to the compiler and library
  - parallelize this for optimal performance
- might use several threads to solve a problem
  - require that they finish early if some condition is met
  - threads need to be
    - sent a stop request so that they can give up on the task they were given
    - clean up
    - finish ASAP
- mechanisms for managing threads and tasks
  - automatic management of the number of threads and the division of tasks between them

9.1 Thread pools

- impractical to have a separate thread for every task that can potentially be done in parallel with other tasks on most systems
  - want to take advantage of the available concurrency where possible
- thread pool allows this
  - tasks that can be executed concurrently are submitted to the pool
  - puts them on a queue of pending work
  - each task is taken from the queue by one of the worker threads
    - executes the task before looping back to take another from the queue
- several key design issues when building a thread pool
  - number of threads to use
  - most efficient way to allocate tasks to threads
  - possibility of waiting for a task to complete

9.1.1 The simplest possible thread pool

- fixed number of worker threads
    - typically the same number as the value returned by std::thread::hardware_concurrency ()
    - process work when there is work to do
- call a function to put work on the queue of pending work
- each worker thread
    - takes work off the queue
    - runs the specified task
    - goes back to the queue for more work
- simplest case -> no way to wait for the task to complete
    - programmer has to manage the synchronization
- Listing 9.1 Simple thread pool
- vector of worker threads
    - thread-safe queue to manage the queue of work
    - users can't wait for the tasks
    - can't return any values
    - use std::function<void()> to encapsulate tasks
    - submit() function then wraps whatever function or callable object is supplied inside a std::function<void()> and pushes it on the queue 1@
    - threads are started in the constructor
        - std::thread::hardware_concurrency() to tell how many concurrent threads the hardware can support
        - create that many threads running your worker_thread() member function
    - starting a thread can fail by throwing an exception
        - need to ensure that any threads you've already started are stopped and cleaned up
        - achieved with a try-catch block that sets the done flag when an exception is thrown
        - instance of the join_threads class (ch. 8) to join all the threads
    - works with the destructor
        - just set the done flag join_threads instance will ensure that all the threads have completed before the pool is destroyed
    - order of declaration of the members is important
        - done flag and the worker_queue must be declared before the threads vector
        - threads vector must be declared before the joiner
        - ensures that the members are destroyed in the right order;
            - can't destroy the queue safely until all the threads have stopped
    - worker_thread function is simple
        - sits in a loop waiting until the done flag is set e
        - pulls tasks off the queue and executes them

- calls std::this_thread::yield() if there are no tasks on the queue and give another thread a chance to put some work on the queue before it tries to take some off again the next time around
- simple thread pool will suffice for many purposes such
  - if the tasks are entirely independent and don't return any values or perform any blocking operations
- many circumstances where such a simple thread pool may not adequately address needs
- others where it can cause problems such as deadlock
- may be better served using std::async in the simple cases

9.1.2 Waiting for tasks submitted to a thread pool

- master thread always waited for the newly spawned threads to finish in the examples in chapter 8 that explicitly spawned threads
  - done after dividing the work between threads
  - ensures that the overall task was complete before returning to the caller
- need to wait for the tasks submitted to the thread pool to complete with thread pools
  - similar to the way that the std::async-based waited for the futures
- have to do this manually using condition variables and futures with the simple thread pool
  - adds complexity
- better to wait for the tasks directly by moving that complexity into the thread pool itself
- can wait for the tasks directly
  - have the submit() function return a task handle of some description
  - can then use to wait for the task to complete
- task handle wraps the use of condition variables or futures
  - simplifies code
- special case
  - main thread needs a result computed by the task
  - e.g. parallel_accumulate()
- can combine the waiting with the result transfer through the use of futures
- can't use std::function<> since std::packaged_task<> instances are not copyable, just movable
  - requires that the stored function objects are copy-constructible
  - must use a custom function wrapper that can handle move-only types
  - simple type-erasure class with a function call operator
  - only need to handle functions that take no parameters and return void
- Listing 9.2 A thread pool with waitable tasks
- modified submit() function returns a std::future<>
  - holds the return value of the task
  - allows caller to wait for the task to complete

- requires definite return type for the supplied function
    - std::result_of<FunctionType()>::type is the type of the result of invoking an instance of type FunctionType with no arguments
    - use the same std::result_of<> expression for the result_type typedef inside the function
    - wrap the function f in a std::packaged_task<result_type()>
- can get future from the std::packaged_task<> before pushing the task onto the queue and returning the future
- NOTE: have to use std::move() when pushing the task onto the queue
    - std::packaged_task<> isn't copyable
    - queue stores function_wrapper objects rather than std::function<void()> objects to handle this
- pool allows waiting for tasks and return of results
- Listing 9.3 parallel_accumulate using a thread pool with waitable tasks
- working with number of blocks to use rather than the number of threads
- need to divide the work into the smallest blocks worth working with concurrently to make the most use of the scalability of the thread pool
    - each thread will process many blocks when there are only a few threads in the pool
    - number of blocks processed in parallel will also grow as the number of threads grows with the hardware
    - NOTE: be careful when choosing the smallest blocks
        - inherent overhead to submitting a task to a thread pool, having the worker thread run it, and passing the return value through a std::future<>
        - for small tasks it's not worth overhead
    - code may run more slowly with a thread pool than with one thread if the task size is too small
    - don't have to worry about packaging the tasks, obtaining the futures, or storing the std::thread objects to join with the threads later if the block size is well chosen
    - just call submit() with the task
    - thread pool handles exception safety
        - any exception thrown by the task gets propagated through the future returned from submit()
        - thread pool destructor abandons any not-yet-completed tasks and waits for the pool threads to finish if the function exits with an exception
- works well for simple cases when the tasks are independent
- not good for situations where the tasks depend on other tasks also submitted to the thread pool

9.1.3 Tasks that wait for other tasks

- std::async for quicksort works well because each task is either running on its own thread or will be invoked when required
- alternative structure (ch 8) that used a fixed number of threads related to the available hardware concurrency
    - used a stack of pending chunks that needed sorting
    - each thread added a new chunk to the stack for one of the sets of data and then sorted the other one directly as each thread partitioned the data it was sorting
    - could end up in a situation where all of the threads were waiting for chunks to be sorted and no threads were actually doing any sorting
    - addressed by having the threads pull chunks off the stack and sort them while the particular chunk they were waiting for was unsorted
- same problem if substituting a simple thread pool for std::async
    - limited number of threads
    - might end up all waiting for tasks that haven't been scheduled because there are no free threads
- need to use a solution similar to the one used in ch 8
    - process outstanding chunks while waiting for current chunk to complete
    - don't have access to the task list if using the thread pool to manage the list of tasks and their association with threads
    - need to modify the thread pool to do this automatically
- simplest way to do this is to add a new function on the thread pool to run a task from the queue and manage the loop manually
- advanced thread pool implementations might add logic into the wait function or additional wait functions to handle this case
    - possibly prioritize the task being waited for
- Listing 9.4 An implementation of run_pending_task()
- Listing 9.5 A thread pool–based implementation of Quicksort
- simpler than the corresponding version from listing 8.1 because all the thread-management logic has been moved to the thread pool
- addressed the crucial deadlock-causing problem with tasks that wait for other tasks
    - still not ideal
- every call to submit() and every call to run_pending_task() accesses the same queue

9.1.4 Avoiding contention on the work queue

- every time a thread calls submit() on a particular instance of the thread pool, it has to push a new item onto the single shared work queue
    - worker threads are continually popping items off the queue in order to run the tasks
    - increasing contention on the queue

- cache ping-pong can be a substantial time sink even if when using a lock-free queue
- one way to avoid cache ping-pong is to use a separate work queue per thread
  - each thread then posts new items to its own queue and takes work from the global work queue only if there's no work on its own individual queue
- Listing 9.6 A thread pool with thread-local work queues
- std::unique_ptr<> to hold the thread-local work queue to keep non-pool threads from having one
  - initialized in the worker_thread() function before the processing loop
- submit() checks to see if the current thread has a work queue
  - if it does, it's a pool thread
    - can put the task on the local queue
  - otherwise need to put the task on the pool queue as before
- similar check in run_pending_task()
  - also need to check to see if there are any items on the local queue
  - can take the front one and process it
  - local queue can be a plain std::queue<> only ever accessed by the one thread
  - try the pool queue as before if there are no tasks on the local queue
- works fine for reducing contention
- can result in one thread having a lot of work on its queue while the others have no work do to when the distribution of work is uneven
  - defeats the purpose of using a thread pool
- solution -> allow the threads to steal work from each other's queues if there's no work in their queue and no work in the global queue

9.1.5 Work stealing

- queue must be accessible to the thread doing the stealing to allow a thread with no work to do to take work from another thread with a full queue
  - requires that each thread register its queue with the thread pool or be given one by the thread pool
  - must ensure that the data in the work queue is suitably synchronized and protected
    - invariants must be protected
- possible to write a lock-free queue that allows the owner thread to push and pop at one end while other threads can steal entries from the other
  - beyond scope of the book
- stick to using a mutex to protect the queue's data
  - little contention on the mutex if work stealing is a rare event
- Listing 9.7 Lock-based queue for work stealing
- simple wrapper around a std::deque<function_wrapper> that protects all accesses with a mutex lock

- each call to do_sort() pushes one item on the stack and then waits for it
    - ensure that the chunk needed for the current call to complete is processed before the chunks needed for the other branches by processing the most recent item first
    - reduces the number of active tasks and the total stack usage
    - try_steal() takes items from the opposite end of the queue to try_pop() in order to minimize contention
    - can use the techniques (ch 6 and 7) to enable concurrent calls to try_pop() and try_steal()
- Listing 9.8 A thread pool that uses work stealing
- demonstrates work integration of thread pool and work queue that permits stealing
- working thread pool that's good for many potential uses
    - still many ways to improve
- one aspect that hasn't been explored is the idea of dynamically resizing the thread pool to ensure that there's optimal CPU usage even when threads are blocked waiting for something such as I/O or a mutex lock

9.2 Interrupting threads

- often desirable to signal to a long-running thread that it's time to stop
    - worker thread for a thread pool and the pool is now being destroyed
    - work being done by the thread has been explicitly canceled by the user
- need to signal from one thread that another should stop before it reaches the natural end of its processing
    - need to do this in a way that allows that thread to terminate nicely rather than abruptly stopping
- C++11 Standard doesn't provide such a mechanism

9.2.1 Launching and interrupting another thread

- same interface as for std::thread with an additional interrupt() function
    - use std::thread to manage the thread itself internally
    - use some custom data structure to handle the interruption
    - need an interruption point for the thread
    - needs to be a simple function that can be called without any parameters (so no additional data is passed down)
        - interruption_point()
    - interruption-specific data structure needs to be accessible through a thread_local variable
        - set when the thread is started
        - when a thread calls interruption_point() it checks the data structure for the currently executing thread

- thread_local flag is the primary reason a plain std::thread can't be used to manage the thread
    - needs to be allocated in a way that the interruptible_thread instance and the thread can access it
    - can do this by wrapping the supplied function before you pass it to std::thread to actually launch the thread in the constructor
- Listing 9.9 Basic implementation of interruptible_thread
- supplied function is wrapped in a lambda function that holds a copy of the function and a reference to the local promise
    - sets the value of the promise to the address of the this_thread_interrupt_flag (declared thread_local) for the new thread before invoking the copy of the supplied function
    - calling thread then waits for the future associated with the promise to become ready and stores the result in the flag member variable
    - NOTE:
        - lambda is running on the new thread
        - has a dangling reference to the local variable
        - OK because the interruptible_thread constructor waits until the promise is no longer referenced by the new thread before returning
    - NOTE: implementation doesn't take account of handling joining with the thread, or detaching it
    - ensure that the flag variable is cleared when the thread exits, or is detached, to avoid a dangling pointer
- interrupt() function
    - have a valid pointer to an interrupt flag
    - have a thread to interrupt
    - can just set the flag
    - up to the interrupted thread to handle the interruption

9.2.2 Detecting that a thread has been interrupted

- setting the interruption flag doesn't do any good if the thread doesn't actually check whether it's being interrupted
- can do this with an interruption_point() function
    - call it at a point where it's safe to be interrupted
    - throw a thread_interrupted exception if the flag is set
    - can use such a function by calling it at convenient points within your code
- works but not ideal
    - best places for interrupting a thread are where it's blocked waiting for something
    - means the thread isn't running in order to call interruption_point()
- need a means for waiting for something in an interruptible fashion

9.2.3 Interrupting a condition variable wait

- can detect interruptions at carefully chosen places in your code with explicit calls to interruption_point()
- doesn't help do a blocking wait
    - e.g. waiting for a condition variable to be notified
- need a new function that can be overloaded for the various reasons to wait
    - interruptible_wait()
    - need to work out how to interrupt the waiting
- to be able to interrupt a wait on a condition variable
    - notify the condition variable once the interrupt flag has been set
    - put an interruption point immediately after the wait
    - have to notify all threads waiting on the condition variable in order to ensure that the thread of interest wakes up for this to work
- waiters have to handle spurious wake-ups
    - other threads would handle this the same as a spurious wake-up
        - wouldn't be able to tell the difference
- interrupt_flag structure needs to be able to store a pointer to a condition variable so that it can be notified in a call to set()
- Listing 9.10 A broken version of interruptible_wait for std::condition_variable
- assumes functions for setting and clearing an association of a condition variable with an interrupt flag
- checks for interruption
- associates the condition variable with the interrupt_flag for the current thread
- waits on the condition variable
- clears the association with the condition variable
- checks for interruption again
- interrupting thread will broadcast the condition variable and wake current thread from the wait if the thread is interrupted during the wait on the condition variable
    - can check for interruption
- code is broken
    - std::condition_ variable::wait() can throw an exception
        - might exit the function without removing the association of the interrupt flag
        - easily fixed with a structure that removes the association in its destructor
    - there's a race condition
        - if the thread is interrupted after the initial call to interruption_point(), but before the call to wait(), then it doesn't matter whether the condition variable has been associated with the interrupt flag
        - the thread isn't waiting and so can't be woken by a notify on the condition variable

- need to ensure that the thread can't be notified between the last check for interruption and the call to wait()
- only one way of doing that without delving into the internals of std::condition_variable
  - use mutex held by lk to protect it too
  - requires passing it in on the call to set_condition_variable()
  - creates problems
    - passing a reference to a mutex with unknown lifetime to another thread (the thread doing the interrupting) for that thread to lock (in the call to interrupt())
    - don't know whether that thread has locked the mutex already when it makes the call
  - has the potential for deadlock and the potential to access a mutex after it has already been destroyed
- too restrictive to not be able to reliably interrupt a condition variable wait
- can put a timeout on the wait
  - use wait_for() rather than wait() with a small timeout value (such as 1 ms)
  - puts an upper limit on how long the thread will have to wait before it sees the interruption (subject to the tick granularity of the clock)
  - waiting thread will see rather more "spurious" wakes resulting from the timeout
- Listing 9.11 Using a timeout in interruptible_wait for std::condition_variable
- 1 ms timeout can be completely hidden inside the predicate loop if the predicate being waited for is available
  - results in the predicate being checked more often than it might otherwise be
    - easily used in place of a plain call to wait()
- handles std::condition_variable waits
- need to handle std::condition_variable_any waits

9.2.4 Interrupting a wait on std::condition_variable_any

- std::condition_variable_any differs from std::condition_variable in that it works with any lock type rather than just std::unique_lockstd::mutex
  - makes interrupting easier
- can build custom lock type that locks/unlocks both the internal set_clear_mutex in the interrupt_flag and the lock supplied to the wait call
- Listing 9.12 interruptible_wait for std::condition_variable_any
- custom lock type acquires the lock on the internal set_clear_mutex when it's con- structed
  - sets the thread_cond_any pointer to refer to the std::condition_variable_any passed in to the constructor
  - Lockable reference is stored for later
    - must already be locked

- can check for an interruption without worrying about races
    - interrupt flag was set before the lock was acquired on set_clear_mutex if it is set at this point
    - unlock the Lockable object and the internal set_clear_mutex when the condition variable calls your unlock() function inside wait()
        - allows threads that are trying to interrupt current thread to acquire the lock on set_clear_mutex and check the thread_cond_any pointer once inside the wait() call but not before
- once wait() has finished waiting (either because it was notified or because of a spurious wake), it will call the lock() function
    - acquires the lock on the internal set_clear_mutex and the lock on the Lockable object
    - can check again for interruptions that happened during the wait() call before clearing the thread_cond_any pointer in your custom_lock destructor
    - also unlock the set_clear_mutex

9.2.5 Interrupting other blocking calls

- other blocking waits
    - mutex locks
    - waiting for futures
    - etc
- in general go for the timeout option used for std::condition_variable
    - no way to interrupt the wait short of aside from fulfilling the condition being waited for
        - (without access to the internals of the mutex or future)
- can loop within the interruptible_wait() function because what is being waited on is known
- code example

9.2.6 Handling interruptions

- interruption is just a thread_interrupted exception from the point of view of the thread being interrupted
    - can be handled just like any other exception
- can catch the interruption, handle it in some way, and then carry on regardless
    - if another thread calls interrupt() again, current thread will be interrupted again the next time it calls an interruption point
- may do this if thread is performing a series of independent tasks
    - interrupting one taskwill cause that task to be abandoned
    - thread can then move on to performing the next task in the list
- all usual exception-safety precautions must also be taken because thread_interrupted is an exception

- to ensure that resources aren't leaked and data structures are left in a coherent state when calling code that can be interrupted often
    - let the interruption terminate the thread
    - let the exception propagate up
- letting exceptions propagate out of the thread function passed to the std::thread constructor causes std::terminate() to be called
    - whole program will be terminated
- put catch block inside the wrapper used for initializing the interrupt_flag to avoid having to remember to put a catch (thread_interrupted) handler in every function you pass to interruptible_thread
    - makes it safe to allow the interruption exception to propagate unhandled
        - will then terminate just that individual thread

9.2.7 Interrupting background tasks on application exit

- main thread/background thread needs to run for the entire lifetime of an application (e.g. GUI example)
    - be started as part of the application initialization and left to run until the application is shut down
    - typically only when the machine itself is being shut down
        - application needs to run the whole time in order to maintain an up-to-date index
- need to close down the background threads in an orderly manner when the application is being shut down
    - can do this by interrupting them
- Listing 9.13 Monitoring the filesystem in the background
- background threads are launched at startup
    - main thread then proceeds with handling the GUI
    - background threads are interrupted when the user has requested that the application exit
    - main thread waits for each background thread to complete before exiting
    - background threads sit in a loop
        - check for disk changes and update the index
        - check for interruption by calling interruption_point()
- interrupt all the threads before waiting for any rather than interrupting each and then waiting for it before moving on to the next for concurrency
    - threads will likely not finish immediately when they're interrupted
        - have to proceed to the next interruption point and then run any destructor calls and exception-handling code necessary before exiting
    - cause the interrupting thread to wait by joining with each thread immediately
        - still has useful work it could do
    - interrupt the other threads only when there is no more work to do and then wait

- allows all the threads being interrupted to process their interruptions in parallel and potentially allows them to finish sooner
- interruption mechanism easily extended to add further interruptible calls or to disable interruptions across a specific block of code

9.3 Summary

- "advanced" thread-management techniques
  - thread pools
  - interrupting threads
- use of local work queues and work stealing to
  - reduce the synchronization overhead
  - potentially improve the throughput of the thread pool
- running other tasks from the queue while waiting for a subtask to complete can eliminate the potential for deadlock
- various ways of allowing one thread to interrupt the processing of another
  - use of specific interruption points
  - functions that perform what would otherwise be a blocking wait in a way that can be interrupted

Ch. 10 - Testing and debugging multithreaded applications

- concurrency-related bugs
- locating bugs through testing and code review
- designing multithreaded tests
- testing the performance of multithreaded code
- testing and debugging concurrent code is hard

10.1 Types of concurrency-related bugs

- any sort of bug in concurrent code
- concurrency-related bugs typically fall into two primary categories
  - unwanted blocking
  - race conditions

Unwanted blocking

- thread is blocked when it's unable to proceed because it's waiting for something

- why is this blocking unwanted?
    - because some other thread is also waiting for the blocked thread to perform some action, and so that thread in turn is blocked
- several variations
    - deadlock
        - one thread is waiting for another, which is in turn waiting for the first
        - if your threads deadlock, the tasks they're supposed to be doing won't get done
    - livelock
        - similar to deadlock in that one thread is waiting for another, which is in turn waiting for the first
        - key difference here is that the wait is not a blocking wait but an active checking loop, such as a spin lock
        - in serious cases, the symptoms are the same as deadlock except that the CPU usage is high because threads are still running but blocking each other
        - in not-so-serious cases, the livelock will eventually resolve because of the random scheduling, but there will be a long delay in the task that got livelocked, with a high CPU usage during that delay
    - blocking on I/O or other external input
        - If your thread is blocked waiting for exter- nal input,
        - thread can't proceed if it is blocked waiting for external input,even if the waited-for input is never going to come
        - undesirable to block on external input from a thread that also performs tasks that other threads may be waiting for

Race conditions

- most common cause of problems in multithreaded code
    - many deadlocks and livelocks only manifest because of a race condition
- not all race conditions are problematic
    - a race condition occurs anytime the behavior depends on the relative scheduling of operations in separate threads
- often cause the following types of problems
    - data races
        - the specific type of race condition that results in undefined behavior because of unsynchronized concurrent access to a shared memory location
        - usually occur through incorrect usage of atomic operations to synchronize threads or through access to shared data without locking the appropriate mutex.
    - broken invariants
        - e.g.
            - dangling pointers (because another thread deleted the data being accessed)
            - random memory corruption (due to a thread reading inconsistent values resulting from partial updates)

- double-free (such as when two threads pop the same value from a queue, and so both delete some associated data)
- the invariants being broken can be temporal- as well as value-based
  - if operations on separate threads are required to execute in a particular order, incorrect synchronization can lead to a race condition in which the required order is sometimes violated
- lifetime issues
  - separate category from broken invariants
  - thread outlives the data that it accesses, so it is accessing data that has been deleted or otherwise destroyed, and potentially the storage is even reused for another object
  - typically get lifetime issues where a thread references local variables that go out of scope before the thread function has completed, but they aren't limited to that scenario
  - potential for data to be destroyed before the thread has finished whenever the lifetime of the thread and the data it operates on aren't tied together in some way
- need to ensure that the call to join() can't be skipped if an exception is thrown when manually calling join() in order to wait for the thread to complete
- problematic race conditions that are the serious issues to beware of
  - application appears to hang and become completely unresponsive or takes too long to complete a task with deadlock and livelock
    - can often attach a debugger to the running process to identify which threads are involved in the deadlock or livelock and which synchronization objects they're fighting over
  - visible symptoms of the problem can manifest anywhere with data races, broken invariants, and lifetime
    - code may overwrite memory used by another part of the system that isn't touched until much later
    - fault will manifest in code completely unrelated to the location of the buggy code
      - possibly much later in the execution of the program
- difficulty of shared memory systems
  - any thread can overwrite the data being used by any other thread in the application

10.2 Techniques for locating concurrency-related bugs

- for finding bugs the most obvious and straightforward thing to do is look at the code
  - difficult to do in a thorough way
  - easy for programmer to read what they intended to write rather than what's actually there
- tempting to just give it a quickly read code when doing a code review
  - really need to go through the code thoroughly
    - think about the concurrency issues

10.2.1 Reviewing code to locate potential bugs

- review code throughly and get someone else to review it
- requires time
- most concurrency bugs require more than a quick glance
- take a break and come back to code
- try to explain how it works in detail
- write detailed notes
- ask questions about the code and explain the answers

Questions to think about when reviewing multithreaded code

- which data needs to be protected from concurrent access?
- how do you ensure that the data is protected?
- where in the code could other threads be at this time?
- which mutexes does this thread hold?
- which mutexes might other threads hold?
- are there any ordering requirements between the operations done in this thread and those done in another?
- how are those requirements enforced?
- is the data loaded by this thread still valid?
- could it have been modified by other threads?
- if you assume that another thread could be modifying the data, what would that mean and how could you ensure that this never happens?
- think about the relationships between the threads
- assume the existence of a bug related to a particular line of code
  - track down the cause
- consider every corner case and possible ordering
- public data and data for which other code can readily obtain a pointer or reference has to be heavily scrutinized
- must assume that other threads may have modified the shared data any time a mutex is released and then reacquired
- may unintentionally be done if the mutex locks aren't immediately visible (e.g. because they're internal to an object)

10.2.2 Locating concurrency-related bugs by testing

- testing multithreaded code is harder
  - precise scheduling of the threads is indeterminate and may vary from run to run
- correctly runs with the same input data may fail at other times if there's a race condition lurking in the code

- pays to design tests carefully
- each test should run the smallest amount of code that could potentially demonstrate a problem
    - isolate the code that's faulty if the test fails
- can help to think about how code should be tested when designing it
- worth eliminating the concurrency from the test in order to verify that the problem is concurrency-related
- particularly important when trying to track down a bug that occurs "in the wild" as opposed to being detected in test harnesses
- can eliminate concurrency as a cause if it is possible to reduce the application to a single thread
- if the problem goes away on a single-core system (even with multiple threads running) but is present on multicore systems or multiprocessor systems
    - definitely have a race condition and possibly a synchronization or memory-ordering issue
- structure of the test is important
- test environment is important
- think through all possible scenarios, e.g. number of threads calling various methods in various timing configurations
- consider additional factors about the test environment
    - what is meant by "multiple threads" in each case
    - whether there are enough processing cores in the system for each thread to run on its own core
    - which processor architectures the tests should be run on
    - how to ensure suitable scheduling for the "while" parts of the tests
- think about additional factors specific to the particular situation

10.2.3 Designing for testability

- in general, code is easier to test if the following factors apply:
    - the responsibilities of each function and class are clear
    - functions are short and to the point
    - tests can take complete control of the environment surrounding the code being tested
    - the code that performs the particular operation being tested is close together rather than spread throughout the system
- even more important to pay attention to the testability of multithreaded code
- one of the best ways to design concurrent code for testing is to eliminate the concurrency
- those parts of the application that operate on data that's being accessed by only that one thread can be tested using the normal single-threaded techniques

- dividing code into multiple blocks of read shared data/transform data/update shared data allows for testing the transform data portions using all the usual single-threaded techniques
  - hard problem of testing a multithreaded transformation will be reduced to testing the reading and updating of the shared data
- library calls can use internal variables to store state
  - becomes shared if multiple threads use the same set of library calls
  - not immediately apparent that the code accesses shared data

10.2.4 Multithreaded testing techniques

Brute-force testing

- stress the code to see if it breaks
  - run the code many times, possibly with many threads running at once
  - more times the code is run the more likely hidden bugs are to appear
- combine fine-grained tests with tests that have a larger scope (end-to-end or integration)
- brute-force testing might lead to false confidence
- worst example is where problematic circumstances can't occur on the test system because of the configuration of the particular system
  - classic example here is testing a multithreaded application on a single-processor system
- different processor architectures provide different synchronization and ordering facilities
  - atomic load operations are always the same, whether tagged memory_order_relaxed or memory_order_seq_cst on x86 and x86-64 architectures
  - relaxed memory ordering may work on systems with an x86 architecture
  - would fail on a system with a finer-grained set of memory-ordering instructions such as SPARC
- use careful thought for test design
  - choice of unit for the code being tested
  - design of the test harness
  - choice of testing environment
- ensure that as many of the code paths as possible are tested and as many of the possible thread interactions as feasible
- need to know which options are covered and which are left untested

Combination simulation testing

- run code with a special piece of software that simulates the real runtime environment of the code
  - similar to software that allows running multiple virtual machines on a single physical computer
    - characteristics of the virtual machine and its hardware are emulated by the supervisor software

- simulation software records the sequences of data accesses, locks, and atomic operations from each thread in addition to emulating the system
  - uses the rules of the C++ memory model to repeat the run with every permitted combination of operations to identify race conditions and deadlocks
- guaranteed to find all the problems the system is designed to detect
- takes a huge amount of time
- best reserved for fine-grained tests of individual pieces of code rather than an entire application
- relies on the availability of simulation software that can handle the operations used in your code
- third option is to use a library that detects problems as they occur in the running of the tests

Detecting problems exposed by tests with a special library

- can identify many problems by using a special implementation of the library synchronization primitives such as mutexes, locks, and condition variables
- can verify that the appropriate mutex was locked if it is possible to check which mutexes are locked when a particular piece of data is accessed
  - done by the calling thread when the data was accessed and report a failure if specific mutex was not lock
- allow the library to check by marking data in some way
- can also record the sequence of locks if more than one mutex is held by a particular thread at once
  - if another thread locks the same mutexes in a different order, this could be recorded as a potential deadlock even if the test didn't actually deadlock while running
- another type of special library where the implementations of the threading primitives such as mutexes and condition variables give the test writer control over which thread gets the lock when multiple threads are waiting or which thread is notified by a notify_one() call on a condition variable
  - can set up particular scenarios and verify that the code works as expected in those scenarios
- some facilities need to be supplied as part of the C++ Standard Library implementation
- others can be built on top of the Standard Library as part of the test harness

10.2.5 Structuring multithreaded test code

- issue
  - need to arrange for a set of threads to each be executing a chosen piece of code at a specified time
- need to identify the distinct parts of each test
  - the general setup code that must be executed before anything else
  - the thread-specific setup code that must run on each thread
  - the actual code for each thread that needs to be run concurrently

- the code to be run after the concurrent execution has finished
    - possibly includes assertions on the state of the code
- Listing 10.1 An example test for concurrent push() and pop() calls on a queue
- necessary to use something like this to have the best chance of testing what actually needs to be tested
    - requires a lot of boilerplate code
- actually starting a thread can be quite a time-consuming process so need control over when threads are run in relation to others
- relatively straightforward to create new tests in the same pattern once the basic structure is set
- readily extended to additional threads

10.2.6 Testing the performance of multithreaded code

- important to actually test code to confirm that the performance does improve
- scalability
    - don't want code that runs twice as fast on a dual-core machine but is actually slower on a 24-core machine
- limited by linearization points or required serial sections of code
- worth looking at the overall design of the code before starting to test
    - know what factor of improvement is being pursued
- contention between processors for access to a data structure can have a big performance impact
- something that scales nicely with the number of processors when that number is small may actually perform badly when the number of processors is much larger because of the huge increase in contention
- best to check the performance on systems with as many different configurations as possible
- ought to test on a single-processor system and a system with as many processing cores as will be available

10.3 Summary

- various types of concurrency-related bugs
    - deadlocks
    - livelocks
    - data races
    - other problematic race conditions
- techniques for locating bugs
- issues to think about during code reviews
- guidelines for writing testable code
- how to structure tests for concurrent code

- utility components that can help with testing