Concurrency Summary

Chapter 1: Hello World of Concurrency

What is a thread?

1. Threads are like lightweight processes that each run independently of others and may run a different sequence of instructions.
2. All threads share the same address space, and most data can be accessed directly by all threads.

Why use concurrency?

1. Separate concerns – group related bits of code together.
2. Use concurrency for performance
    a. Task parallelism - Splitting a single task into multiple parts to reduce the total runtime.
    b. Data parallelism – Splitting the same array of data into multiple chunks and performing the same operation on those chunks.

Chapter 2: Managing Threads

Launching

1. When you launch a thread must decide whether you can wait for it to finish (join) or let it run on its own (detach).
    a. If you don't decide, its constructor will call terminate.
    b. Detached threads are sometimes called daemon threads.

Passing Parameters

1. Pass a callable, function pointer, or lambda, and pass arguments the way C# does (params[]) where reference arguments are wrapped in std::ref.
2. You can also have a thread invoke a method on an object by supplying the address of a nobject pointer as the second argument (with the third argument being the first argument to the mbmer function).
    a. See page 26.

Transferring ownership of a thread

1. Like unique_ptr, the object is no copyable, you can move a thread, though.
2. You can't move toa thread that is already running a process. std::terminate would be called.
3. Joinable_thread class is on page 29 (Listing 2.7)

Choosing the number of threads at runtime

1. Use std::thread::hardware_concurrency(). See page 32 for an example.

Chapter 3: Sharing data between threads

Problems with shared data

1. Sharing data between threads is dangerous only when there is one modifying thread. If all shared data is read-only, there is no problem.
    a. The issue is called a *race condition* which is defined as anything where the outcome depends on the relative ordering of execution of operations on two or more threads.
        i. CA race condition in the context of the modification of a single object is called a *data race*.

Protecting shared data

1. There are multiple ways to protect shared data.
    a. Lock-free programming
    b. Lock-less programming
    c. Software transactional model
    d. Mutual exclusion
        i. A mutex ensures only one thread has access to a specific piece of shared data at a time. Conceptually, a thread locks the mutex when it acquires it, and unlocks it when it releases it.
            1. Lock objects (e.g. lock_guard) provides RAII-based wrappers around constructs like std::mutex to prevent one from needing to call lock() and unlock().

Structuring code for protecting shared data

The following are best practices when using locks:

1. Do not pass objects protected by locks outside the scope of that lock.
2. Avoid nested locks
3. Avoid calling user supplied code while holding a lock (that user supplied code can contain more locks inside of it)

Deadlock

1. Dealock occurs when two threads contend for different locks in a form that doesn't allow either to progress.
    a. A way to avoid this is to always acquire locks in the same order.

Multiple readers one writer

1. You can support multiple readers and one writer with:
    a. Std::shared_lock<std::shared_mutex> (multiple readers) and std::scoped_lock<std::shared_mutex> (one writer)

Chapter 4: Synchronizing concurrent operations

Returning values from background threads

1. Launch an async task (std::async) and wait on the future (std::future)
    a. Std::async can be alled with std::launch::deferred or std::launch::async
2. You can also do so via std::promise and std::future (like TaskCompletionSource)
3. The original future will become invalid when a continuation is scheduled (via .then()). You can check for the validity via (.valid())
    a. The key about a continuation is that it's called on an unspecified thread when the original future is ready. This gives the implementation freedom to run on a thread pool or another library-managed thread.
4. There are similar concepts like when_all and when_any

Waiting with a time limit

1. There are two time limits, a duration-based limit and an absolute timeout based limit (corresponding to _for and _until).
    a. Std::future contains `wait_for` and `wait_until` methods that takes in a timespan and returns a status (std::future_status::timeout, std::future_status::ready, etc).
        i. For example, cv.wait_until(lock, std::chrono::steady_clock::now() + std::chrono::milliseconds(500)
    b. Std::this_thread contains `sleep_until` and `sleep_for`

Latches and barriers in the TS

1. A *latch* is a lightweight synchronization object that becomes ready when its count is decremented to zero.
    a. With a latch, each thread can decrement the count once, zero times, or more than once.
2. A *barrier* each thread must arrive at it before they're all released.
    a. Barriers can be reused (they're like manual reset events, except it happens automatically once all threads make it), latches cannot.
    b. Barriers also require that the thread that waits on it is part of the synchronization group.
    c. The barrier case involves a situation where each thread is operating on its own data independently, but the combined result must be shared between all threads (uses all individual pieces of data).
    d. A flex_barrier also exists, albeit it's a bit more flexible (allows you to change the thread barrier count in the serial callback, as well as it takes a lambda for one of the threads to execute when all threads have called arrive_and_wait())

Chapter 5: The memory model and operations on atomic types

- The standard atomic types are not copyable or assignable in the traditional sense of the word.
    - They have member functions such as *load*, *store*, *exchange*, *compare_exchange_weak*, and *compare_exchange_strong*. They also support compound assignment operators (+=, -=, *=, |=) as well as compound assignment operators where appropriate (++ and --).

        - Unlike fetch_add (below), the compound assignment operators return the new value.
        - The return value from these oerators is either the value stored (in the case of assignment) or the value prior to the operation (in the case of the named functions, like fetch_add).
            - Fetch_add stores the latest value and returns the previously held value.
            - Because this is a read-modify-write operatoion, they can have any memory_ordering tags,
    - Each operation on the atomic types has an optional memory-ordering argument of type (memory_order)
        - There are six types: relaxed, acquire, consume, acq_rel, release, seq_cst
            - These are divided into three categories:
                - Store – relaxed, release, seq_cst
                - Load – relaxed, consume, acquire, seq_cst
                - Read-modify-write – relaxed, consume, acquire, release, acq_rel, seq_cst
- Atomic_flag is the simplest atomic type, it's a Boolean flag.
    - This type must be initialized with ATOMIC_FLAG_INIT. It is the only atomic type that needs to be initialized like this, and it's also the only one that is guaranteed to be lock-free.
        - The two most important methods on this is *clear* and *test_and_set* (which returns the old value). This type is ideal for writing a spinlock.
- For compare_exchange, there are two types:
    - Compare_exchange_weak – Can fail spuriously despite the original value stored in the variable being equal to the expected value. In this case, false is returned. Hence compare_exchange_weak is usually used in a loop.
    - Compare_exchange_strong – In compare_exchange_weak, the expected value is reloaded every time.
        - Both compare_exchange functions take two memory ordering parameters that allow for different semantics on the case of success and failure.
- You can use user-defined types with atomic, but they must have a tricial copy assignment operator. This means no virtual functions or base classes and a compiler generated copy-assignment operator.

Synchronizaing operations and enforcing ordering

- The memory model is enforced through happens-before and synchronizes-with concepts.
- There are six memory orders for atomic operations as mentioned before.
    - Sequentiallly consistent – A single-view of the world, seeing memory as one global object as if all threads in a multithreaded program were 1 thread. In a multiprocessor system, there are extensive and expensive performance penalties here.
        - Outside of this memory ordering, different threads have different views of the same operations. Threads also don't need to agree with the ordering of events.

    - Relaxed – the modification order between each variable is the only thing shared between threads that are using memory_order_relaxed. There is no ordering guarantees relating to visibility of values arising from operations on different variables.
    - Acquire-release ordering – loads are acquire, stores are release, and read-modifyu-write are acq_rel
        - A release operation synchronizes with an acquire operation that reads the value written from the same thread. Read page 158.
    - Just don't use consume since the C++17 standard says not to. It has something to do with data dependencies.

Chapter 8: Designing concurrent code

Data contention and cache ping-pong

- If two threads are executing concurrently on different processors and theyre both reading the same data, this usually wont cause a problem. The data will be copied into both the processors respective caches and both processors can proceed.
- If one of the threads modifies the data, this change then has to propagate to the cache on the other core, which takes time. Depending on the nature of the operations of the two threads, and the memory ordering, this modification may cause the second processor to stop in its track and wait for the change to propagate through the memory hardware.
    - When processors need to wait on each other, this is called high contention.

False sharing

- Processors don't generally deal in individual memory locations, instead tey deal with cache lines. These blocks are usually 64 bytes in size but it depends on the processor.
    - Because the processor deals with cache-line-sized blocks of memory, small data items in adjacent memory locations are located on the same line.
    - This is usually a performance increase since, when the cpu accesses data in one region, it most likely needs data in surrounding closeby regions.
        - If the data in the cacheline are unrelated and need to be accessed by different threads, this can be a major cause of performance problems.
- Suppose we have an array of int values and a set of threads that each access their own entry in the array but do so repeatedly. Since int is typically smaller than a cache line, quite a few of those array entries will be on the same cache line. Here the cache hardwarehas to play cache ping-pong.
    - Every time the thread accessing entry 0 needs to update the value, ownership of the cacheline needs to be transferred to the processor running that thread, only to be transferred to the cache for the processor running the thread for entry 1 that needs to update its own data item.
        - The cache line is shared, even though none of the data is, hence the term false-sharing.
            - To prevent this, ensure that data that belongs to different threads is at least std::hardware_destructive_interference_size bytes away from each other (or use thread_local).

How close is your data?

- If your data is spread far apart, the need to go to cache to retrieve data gets even worse when you are oversubscribed and you have more than threads than cores (as task sqitching requires each thread to reload its cache, transferring required lines in)

Chapter 10: Parallel Algorithms

- There are three parallel algorithm execution policies you can specify
    - Sequenced_policy – no parallelism. They must be performed on same thread and not interleaved. The precise order is unspecified. This differs from an algorithm without an execution policy.
    - Parallel_policy – Will run execution across a number of threads. The precise order in each thread is not specified. Be careful of data rases here. For example, this is illegal: std::for_each(std::execution::par, v.begin(), v.end(), [&](int& x) { x =++count; });
    - Parallel_unsequenced policy – The most restrictive. Operations can be interleaved on a single thread such that a second operation is started on a thread before the first is finished. There can be no shared state, even if protected.
- All execution policies will call terminate if there are uncaught exceptions. You can find these policies in the <execution> header