Introduction to Multithreading in C++

Concurrency and parallelism

- The free lunch is over :(Obtaining additional performance from newer CPUs requires writing concurrent programs, which may or may not run in parallel.
- Writing concurrent code in modern languages is fairly easy*. C++ offers several elegant mechanism to do so.

* unless you want it to actually work

The challenges of concurrency

- Unfortunately, writing efficient and correct concurrent programs requires a deep understanding of the underlying architecture, memory model, language and compiler limitations, and synchronization.
- The biggest two challenges are correctness and scalability.
  - Correctness breaks with race conditions: elements of code whose behavior change depending on the order they're accessed by different threads.
  - Scalability breaks when elements of the code serialize on a single resource. This can be hidden inside a library!
- Both problems are significantly alleviated by reducing (or even eliminating) shared state.
- For correctness (not scalability), it's enough to eliminate shared **mutable** state.
  - You can make mutable state temporarily un-shared with various synchronization mechanisms.

The pthread model

- For simplicity in this class, we'll focus only on a single model of concurrency that is borrowed from the pthreads library (and implemented in its terms).
- POSIX threads (pthread) is a widely popular, relatively low-level library for concurrent programming.
- Its abstraction level is threads (not processes), which are shared-memory, low overhead (except creation and destruction), and offer a very simple API (too simple)?
- Pthreads offer various synchronization mechanisms, such as mutex, barrier, and Read/write lock.
- If you're curious and want to learn how to use the pthread library, start here.
- In C++, we use abstractions (classes) that wrap around pthreads and make it a little easier to use.
- Requirements to compile and run:
    - Add -pthread to link command
    - Set vb.cpus to 2 or 4 in VagrantFile

std::thread

- When a thread object is constructed, it creates a POSIX thread and attaches it to a task.
- We therefore pass a ‘callable’ to a thread’s constructor. A callable is one of:
  - A function pointer (the name of a function)
  - A function object (functor)
  - A lambda
- When the underlying callable completes its execution, the object is destructed, or if the object is destructed first, the task is aborted.
- If you want to let the thread run indefinitely, you can detach() from it.
- Or if you want to wait for its completion, you can join() it.

std::mutex

- Mutable shared state is the root of all data races.
- To protect shared state from concurrent mutation, we can use several mechanisms.
- One of the most generalized one is the mutual-exclusion variable, or mutex.
- The two main methods are lock() and unlock(). The former is exclusive: only one process will succeed (and the rest would either fail or block).
- The exclusivity is enforced by the underlying library (and often accelerated by the hardware).
- You can now wrap code that mutates shared state with a mutex so that there are no concurrent mutating access.

std::scoped_lock

- scoped_lock is a RAII encapsulation of a mutex, so that you don’t have to remember to unlock it (or fail to do so in case the program goes through another code path).
- At construction, a scoped_lock blocks until it can lock the mutex.
- At destruction, the scoped_lock automatically unlocks it.
- This is a more convenient, idiomatic, and correct way to protect a scope of code.

Mutex caveats

- Serializing access to data or code helps correctness (not ensuring it!), but reduces scalability.
- Deadlocks: combinations where A holds lock1 and wait for lock2, and B holds lock2 and waits for lock1 (or more complicated versions)
- Livelock: akin to a deadlock when both processes are continuously trying to make progress but repeatedly block each other.
- Starvation: one or more processes can’t make progress because of contention over a shared resource (for example, an unfair mutex that always favors another process).

std::atomic

- If we have a shared state that is a simple primitive type (like int), we can protect it with std::atomic instead of a mutex.
- `std::atomic<int> sum;` behaves much like `int sum;` --- but is guaranteed (by the hardware) to have atomic accesses only.
- For example, if two threads are trying to do `sum++;` concurrently, the atomic version guarantees no race conditions (such as the one shown here).
- But there's a small performance cost to this guarantee (much lower than a mutex's).
- Unfortunately, the hardware can only guarantee atomicity for small (~word size) types.

Overhead comparison

With even just two threads, we can see how costly different synchronization mechanisms are (on my Mac):

- No synchronization: ~0.16s for 100M iterations.
- std::atomic synchronization: ~1.42s.
- std::mutex: ~3.53s.

No synchronization is fastest. Unfortunately, it also doesn't work.

There's an entire sub-field of computer science dedicated to coming up with correct parallel algorithms for different problem with little or no synchronization.

Types of parallelism (granularity)

- The granularity of a parallel/concurrent task is the a measure of the amount of work performed by the task before it has to synchronize with other tasks.
- Fine-grained parallelism means a program is broken down into many small tasks.
  - Often leads to better load balancing.
  - Typically increases synchronization overhead and reduces efficiency.
  - Best for architectures with many processors and fast synchronization/communication.
  - Example of fine-grained parallelism: neurons in our brain.
- Coarse-grained parallelism involves splitting up the program into large tasks.
  - Opposite pros and cons of fine-grained parallelism.
- Medium-grained parallelism is a compromise between the two.
  - If granularity is too fine, the synchronization overhead can overwhelm the computation, leading to **worse** performance as we add threads, as we've seen in the synchronization benchmark.
  - If your program is flexible enough to have variable granularity, you can control the trade-offs between synchronization overhead (fine-grained) and load imbalance (coarse-grained).

The latest from the C++ standard

C++17 standardizes an easy way to parallelize calls to many of the STL algorithms, such as:

```
std::sort(std::execution::par, begin, end);  // multithreaded
```

Or:

```
std::min_element(std::execution::par_vec, begin, end); // multithreaded and/or SIMD vectorized
```

Unfortunately, most compilers haven't yet implemented this feature yet at this time.

More advanced features

- C++ offers more sophisticated mechanisms for concurrency, such as launching a task asynchronously, checking (later) for its return value, or tying it with a continuation task.
- C++20 will also offer coroutines (similar to goroutines) transactional memory, and barriers.

To read more

- Very brief primer on std::thread.
- Very brief primer on std::mutex.
- C++11 Multithreading tutorial.
- Wikipedia page on granularity.
- “Concurrency with Modern C++” by Rainer Grimm for the cutting edge language features.