Concurrency II

CIS 198 Lecture 11

More Concurrency Primitives!

- RwLock
- Barrier
- CondVar
- Once

`std::sync::RwLock<T>`

- A reader-writer lock.
- Similar to a `Mutex`, but has separate modes for reading and writing access.
- Allows data access to *many* readers or *one* writer.
  - Trying to acquire a lock may block depending on the borrow state of the container.
- Inner `T` must be `Send + Sync`.
- May also be poisoned like a `Mutex`.
  - Poisoning only occurs if a writer panics.

`std::sync::RwLock<T>`

- Useful if you have many threads requesting reads and few requesting writes.
- `RwLock` allows safe, concurrent access with better performance (fewer lockouts; more concurrency) in some situations.
- Compare with a `Mutex`, which locks *everyone else* out.
    - Not strictly better; `RwLock` only works on `Sync` types - otherwise, you really need mutual exclusion.
- Compare with a `RefCell`, which also encodes the single-writer-or-multiple-reader idea, but panics instead of blocking on conflicts and is never thread-safe (`Sync`).
- As with `Mutex`, an `RwLock` could cause deadlock under certain conditions.

`std::sync::Barrier`

- A type to allow multiple threads to synchronize on a computation.
- Created for a fixed number of threads.
- Threads may call `wait` on a copy of the `Barrier` to "report" their readiness.
- Blocks `n - 1` threads and wakes all blocked threads when the `n`th thread reports.

`std::sync::Barrier`

- Another primitive to help ensure consistent world views!
- Recall from last lecture: how do you ensure all threads see the same data?
  - For example, readers may want to be guaranteed that all writes have completed before they read.
  - Able to synchronize on this action with a `Barrier`.

`std::sync::Condvar`

- A "condition variable".
- Blocks a thread so that it consumes no CPU time while waiting for an event.
- Generally associated with a `Mutex` wrapping some boolean predicate, which is the blocking predicate.
- Logically similar to having a thread do this:

```
while !predicate { }
```

`std::sync::Condvar`

- A nice way to put a thread "on hold".
- Does not require holding any locks.
- We could have the thread spin in a loop as above, but that would waste CPU time.
  - The waiting thread has to be scheduled to spin-wait.
  - Repeatedly checking the predicate might also require acquiring a lock, which might cause other thread to deadlock.

`std::sync::Condvar`

- Motivating example: producer/consumer problem.
  - Producer threads add items to a bounded queue
  - Consumer threads take them off.
- Full queue: producer threads should wait until it has space.
- Empty queue: consumer threads should wait until it has items.
- Both producers & consumers can wait on a condition variable.
  - Threads can be woken when the queue is ready.

`std::sync::Once`

- A primitive to run a one-time global initialization.
- Regardless of how many times `once.call_once(function)` is called, only the first one will execute.
  - After that, it's guaranteed that the initialization was run, and all memory writes performed are visible from all threads.
- Allows the code to attempt initialization more than once (e.g. one per thread).
- Useful for doing one-time initialization to call foreign functions (e.g. functions from a C library).

Atomics

- Instruction Reordering
- Atomicity
- Rust Atomics
- Data Access
- Atomic Memory Ordering

[1] Some section content borrowed from The Rustonomicon

Instruction Reordering

- Ordinarily, the compiler may reorder instructions in your program semi-arbitrarily.
    - Unless there are compiler bugs, your code should at least have sequential consistency.
- Instructions may also be optimized out by the compiler.
- In single-threaded code, these optimizations are fine.
- In multi-threaded code, we might have relied on `x` being equal to `1` at some point, and this optimization may be bogus!

```
x = 1;
y = 2;  =====>  x = 3;
x = 3;          y = 2;
```

Hardware Reordering

- Even if the compiler behaves, the CPU may decide to shake things up.
- Due to memory hierarchies, hardware doesn't guarantee that all threads see the same view.
    - Events may occur in one order on thread 1, and in a totally different order on thread 2.

Hardware Reordering

- This code has two possible final states:
  - `y = 3` (thread 1 writes to y after thread 2 reads y)
  - `y = 6` (thread 1 writes to y before thread 2 reads y)
- Hardware reordering enables a third final state: y = 2.
  - Here, thread 2 read x, didn't read y = 3, and then wrote y = 2 * 1.

```
Initially: x = 0, y = 1;

Thread 1  |  Thread 2
----------+-------------
y = 3;    |  if x == 1 {
x = 1;    |      y *= 2;
          |  }
```

Hardware Reordering

- In order to ensure events happen in the order we want them to, the compiler needs to emit special CPU instructions.
- Even so, different CPUs have different guarantees surrounding these instructions.
- Some CPUs guarantee strong event ordering by default, others weak ordering.
  - Strong ordering guarantees causes less instruction reordering.
- Different hardware may have different performance depending on the type of guarantees provided.

Atomics

- What are atomics?
- An operation is *atomic* if it appears to occur "instantly" from the program's perspective.
    - Not time-instant, but *instruction-instant*.
- An atomic operation is not divisible into sub-operations.
- *Atomicity* is an important property to ensure that certain operations will not be broken up by thread interleaving.
- Atomic operations are a compiler intrinsic, and are handled by special CPU instructions.

`std::sync::atomic`

- Rust's atomic primitives define this behavior at the type level.
- Four kinds: `AtomicUsize`, `Isize`, `Bool`, `Ptr`
- Standard atomics are safe to share between threads (`Sync`).
- Most non-primitive types in `std::sync` use these primitives (`Arc`, `Mutex`, etc.).
  - If you want to build your own thread-safe types, you probably need to use atomics.

Data Access

- An atomic type's value may not be used by simple member variable access.
  - e.g., you can't just access the value of an `AtomicUsize` freely.
  - Atomic variables are not true *language* primitives like `usize`.
- Instead, you must load from the struct to read its value, and store a value to update it.
- More complex operations may also be performed (e.g., swap, fetch-and-add).
  - We'll see why these are useful later.

Atomic Memory Ordering

- All operations on atomics require an `Ordering`.
- An `Ordering` describes how the compiler & CPU may reorder instructions surrounding atomic operations.
- Rust uses the same memory orderings as LLVM, and the same atomic model as C11.
- You won't usually have to worry about this unless you're building concurrency libraries from the ground up.

Concurrency Applications

Multithreaded Networking

- Networking lends itself well to multithreading.
- Socket operations are often blocking, so running several operations in parallel is convenient.
- A program can listen on a socket on one thread, and send over a socket on another.

Multithreaded Networking

- A server listening for client connections might want to spawn one thread per connection.
    - Each client thread will block until it receives a message.
    - Messages can be relayed to the main thread using channels!
- The main thread could also use a channel to relay messages back to clients.

Multithreaded Networking

- This model unfortunately does not scale well.
  - Spawning threads is expensive.
  - Destroying threads is expensive.
- A more complex model using thread pools* is often necessary.
  - A set of threads is pre-allocated, e.g. one thread per CPU core.
  - Rather than blocking, each thread will poll each client socket (asynchronously).
  - Data may be available immediately as a consequence.

[1] More on these in a bit.

Rayon

- A data parallelism library developed by Niko Matsakis, inspired by Cilk.
- Two primary features:
  - Parallel iterators: take iterator chains and execute them in parallel.
  - `rayon::join`, which converts recursive divide-and-conquer iterators to execute in parallel.
- Statically guarantees data race safety!
  - This may seem to contradict what we said last week, but Rayon achieves this property via allowing particular, limited data access per thread.
- Still pretty experimental, so user beware.

Parallel Iterators

- Parallel iterators are really easy!
  - (As long as everything is `Send` and `Sync`)
- Just use `par_iter()` or `par_iter_mut()` instead of the non-`par_` variants.
- Rayon also provides a number of parallel iterator adapters (`map`, `fold`, `filter`, etc.).

```rust
// Increment all values in a slice
use rayon::prelude::*;
fn increment_all(input: &mut [i32]) {
    input.par_iter_mut()
         .for_each(|p| *p += 1);
}
```

Recursive Divide-And-Conquer

- Recursive divide-and-conquer problems can be parallelized using Rayon's `join` method.
- Parallel iterators are built on this method, and `par_iter()` abtracts over it.
- `join` takes two closures and potentially runs them in parallel.
  - Lets the program decide on the use of parallelism dynamically.
  - Effectively an annotation to say "run this in parallel if you can."

Recursive Divide-And-Conquer

- We can rewrite `increment_all()` using `join()`.
- Even though this actually complicates the implementation.

```
// Increment all values in slice.
fn increment_all(slice: &mut [i32]) {
    if slice.len() < 1000 {
        for p in slice { *p += 1; }
    } else {
        let mid_point = slice.len() / 2;
        let (left, right) = slice.split_at_mut(mid_point);
        rayon::join(|| increment_all(left),
                    || increment_all(right));
    }
}
```

Recursive Divide-And-Conquer

- `join()` can also be used to implement things like parallel quicksort:

```
fn quick_sort<T:PartialOrd+Send>(v: &mut [T]) {
    if v.len() <= 1 {
        return;
    }

    let mid = partition(v); // Choose some partition
    let (lo, hi) = v.split_at_mut(mid);
    rayon::join(|| quick_sort(lo), || quick_sort(hi));
}
```

Recursive Divide-And-Conquer

- Be careful: `join` is not the same as just spawning two threads (one per closure).
- If all CPUs are already busy, Rayon will opt to run the closures in sequence.
- `join` is designed to have low overhead, but may have performance implications on small workloads.
  - You may want to have a serial fallback for smaller workloads.
  - Parallel iterators already have this in place, but `join` is lower-level.

Safety

- Basic examples that would intuitively violate borrowing rules are invalid in Rayon:

```
// This fails to compile, since both threads in `join`
// try to borrow `slice` mutably.
fn increment_all(slice: &mut [i32]) {
    rayon::join(|| process(slice), || process(slice));
}
```

Safety

- Rayon is guaranteed to be free of data races.
- Rayon *itself* cannot cause deadlocks.
- As a consequence, you can't use any types with Rayon which are not thread-safe.
- The Rayon docs suggest these conversions for thread-unsafe types:
  - `Cell` -> `AtomicUsize`, `AtomicBool`, etc.
  - `RefCell` -> `RwLock`
  - `Rc` -> `Arc`
- However, there are some nuances to these conversions, and they can't be done blindly.
  - This is true when using concurrency in general.

Safety

- Using `(Ref)Cell`-like structures in parallel has some pitfalls due to code interleaving.
- Something like an `Rc<Cell<usize>>` can't be blindly logically converted to an `Arc<AtomicUsize>`.
- Consider this example, where `ts` is an `Arc<AtomicUsize>`:

```
Thread 1                        |  Thread 2
--------------------------------+--------------------------------
let value =                     |
  ts.load(Ordering::SeqCst);    |
// value = X                    |  let value =
                                |    ts.load(Ordering::SeqCst);
                                |  // value = X
ts.store(value+1);              |  ts.store(value+1);
// ts = X+1                     |  // ts = X+1
```

Safety

- Above, `ts` only gets incremented by 1, but we expect it to get incremented twice.
- To guarantee that this happens, we need something better than just `load` and `store`.
- `fetch_add` is more appropriate (and correct!) in this case.
  - This method performs an atomic load & store in one operation.
- Atomicity of operations needs to be accounted for in these cases.
- Once again, concurrency primitives are not magic bullets.

Safety

- Why doesn't Rust automatically protect from the above example? Isn't that the whole point of Rust?
- The type system prevents you from making basic mistakes, but thread interleaving is sometimes useful.
- Example: a shortest-route search in parallel that terminates a thread when it goes beyond the current shortest path length.

Internals

- Rayon is built on the concept of *work stealing*.
- Conceptually, there is a pool of threads in memory, typically one per CPU, that can run tasks.
- When you call `join(a, b)`, `a` is started, and `b` gets put on a queue of pending work.
- Any idle threads will scan the queue of pending tasks and choose one to execute.
- Once `a` completes, the thread that ran `a` checks to see if `b` was taken off the queue, and runs it if not.

Scoped Threads

- Threads which are guaranteed to terminate before the current stack frame goes away.
- The thread is "scoped" to the stack frame in which it was created.
- Such threads can access their parent stack frame directly.
  - This data is guaranteed to always be valid from the thread's view.
- Simple, right? 😼

Scoped Threads

- Rust does not have a standard scoped thread library anymore.
- Instead, there are three notable ones:
  - scoped_threadpool
  - scoped_pool
  - crossbeam
- These are all relatively* similar, so we'll look at some high-level features from each.

*Your personal value of relativity may vary

Scoped Threads

- Crossbeam's scoped threads work a bit differently.
- `crossbeam::Scope::defer(function)` schedules some code to be executed at the end of its scope.
- `crossbeam::Scope::spawn` creates a standalone scoped thread, tied to a parent scope.
    - Created just like regular threads.
- A scoped thread mandates that its parent scope **must** wait for it to finish before its scope exits.

Scoped Threads

- When you spawn a std thread, the function the thread executes must have the `'static` lifetime.
  - This ensures that the thread lives long enough.
- Because `Scope` is tied to its parent thread's liftetime, the function the thread executes need only have some `'a` lifetime of its parent!
- This avoids the need to:
  - Explicitly join on threads from some main thread.
  - Potentially put some data in a heavy `Arc` wrapper just to share it with local threads.

Scoped Threads

- Crossbeam scoped threads can also defer their destruction until after they would have been destructed.
- Crossbeam scoped threads are also totally black magic, you guys.
- Crossbeam also has a few other concurrency features, but they're somewhat out of scope here.

Thread Pools

- Two of these libraries provide *thread pools*.
- Thread pools are a collection of threads that can be scheduled to a set of tasks.
- Threads are created up-front and stored in memory in the pool, and can be dispatched as needed.

Thread Pools

- Threads are not destroyed when they complete, but are saved for re-use.
- *Ad hoc* thread creation & destruction can be very expensive.
- Choosing a good thread pool size is often an indeterminate problem.
  - Pool size is best chosen based on system requirements.

Thread Pools

- `scoped_pool` provided pools with both scoped and unscoped threads.
- `scoped_threadpool` only has scoped pools.
- Unscoped threads will dispatch from the pool, but will not be waited on to complete.
- Scoped threads *must* be waited on by the pool.

Thread Pools

- One nice feature of Rust is that thread lifetimes can be expressed explicitly.
- It's very easy to statically reason about how long threads will live.
    - Can also enforce guarantees about what thread scopes are tied to what lifetimes.

Interlude: Thread Unsoundness Bug

- Remember how we said scoped threads used to be in `std`?
- Before Rust 1.0, a serious bug was discovered with scoped threads and a few other types.
- In short, it was possible to create a scoped thread, leak it with `mem::forget`, and have it end up accessing freed memory.
- See discussion here and here.

Eventual

- A library for performing asynchronous computations.
- Employs `Future`s and `Stream`s, which represent asynchronous computations.
  - Futures are similar to `Promise`s (in JavaScript).

> The term *promise* was proposed in 1976 by Daniel P. Friedman and David Wise, and Peter Hibbard called it *eventual*. A somewhat similar concept *future* was introduced in 1977 in a paper by Henry Baker and Carl Hewitt. —Wikipedia

Futures & Streams

- A `Future` is a proxy for a value which is being computed asynchronously.
- The computation behind a `Future` may be run in another thread, or may be kicked off as a callback to some other operation.
- You can think of a `Future` like a `Result` whose value is computed asynchronously.
- Futures can be polled for their information, blocked on, ANDed together with other Futures, etc.
- `Stream`s are similar to `Future`s, but represent a sequence of values instead of just one.

Futures & Streams

```rust
extern crate eventual;
use eventual::*;

let f1 = Future::spawn(|| { 1 });
let f2 = Future::spawn(|| { 2 });
let res = join((f1, f2))
            .and_then(|(v1, v2)| Ok(v1 + v2))
            .await().unwrap();
println!("{}", res); // 3
```

Futures

- Futures are useful for offloading slow computational tasks into the "background".
- Easier than running a raw thread with the desired computation.
- Example: Requesting large images from an HTTP server might be slow, so it might be better to fetch images in parallel asynchronously.