PNA Rust Project 4: Concurrency and parallelism

Task: Create a *multi-threaded*, persistent key/value store server and client with synchronous networking over a custom protocol.

Goals:

- Write a simple thread pool
- Use channels for cross-thread communication
- Share data structures with locks
- Perform read operations without locks
- Benchmark single-threaded vs multithreaded

Topics: thread pools, channels, locks, lock-free data structures, atomics, parameterized benchmarking.

Introduction

In this project you will create a simple key/value server and client that communicate over a custom protocol. The server will use synchronous networking, and will respond to multiple requests using increasingly sophisticated concurrent implementations. The in-memory index will become a concurrent data structure, shared by all threads, and compaction will be done on a dedicated thread, to reduce latency of individual requests.

Project spec

The cargo project, `kvs`, builds a command-line key-value store client called `kvs-client`, and a key-value store server called `kvs-server`, both of which in turn call into a library called `kvs`. The client speaks to the server over a custom protocol.

The interface to the CLI is the same as in the [previous project]. The difference this time is in the concurrent implementation, which will be described as we work through it..

The library interface is nearly the same except for two things. First this time all the `KvsEngine`, `KvStore`, etc. methods take `&self` instead of `&mut self`, and now it implements `Clone`. This is common with concurrent data structures. Why is that? It's not that we're not going to be writing immutable code. It *is* though going to be shared across threads. Why might that preclude using `&mut self` in the method signatures? If you don't know now, it will become obvious by the end of this project.

The second is that the library in this project contains a new *trait*, `ThreadPool`. It contains the following methods:

- `ThreadPool::new(threads: u32) -> Result<ThreadPool>`

  Creates a new thread pool, immediately spawning the specified number of threads.

  Returns an error if any thread fails to spawn. All previously-spawned threads are terminated.

- `ThreadPool::spawn<F>(&self, job: F) where F: FnOnce() + Send + 'static`

  Spawn a function into the threadpool.

  Spawning always succeeds, but if the function panics the threadpool continues to operate with the same number of threads — the thread count is not reduced nor is the thread pool destroyed, corrupted or invalidated.

By the end of this project there will be several implementations of this trait, and you will again perform benchmarking to compare them.

This project should not require any changes at all to the client code.

Project setup

Continuing from your previous project, delete your privous `tests` directory and copy this project's `tests` directory into its place. This project should contain a library named `kvs`, and two executables, `kvs-server` and `kvs-client`.

You need the following dev-dependencies in your `Cargo.toml`:

```toml
[dev-dependencies]
assert_cmd = "0.11"
criterion = "0.2.11"
crossbeam-utils = "0.6.5"
predicates = "1.0.0"
rand = "0.6.5"
tempfile = "3.0.7"
walkdir = "2.2.7"
panic-control = "0.1.4"
```

As with previous projects, add enough definitions that the test suite builds.

Background: blocking and multithreading

Until now you have serviced all of your requests, both read and write (e.g. "get" and "set"), on a single thread. In other words, all the requests in your database are *serialized*. Using a diagram we are going to repeat through this project, the flow of time looks like this:

```
    thread
           +  +---------+---------+---------+---------+
      T1   |  |   R1    |   R2    |   W1    |   W2    |
           +  +---------+---------+---------+---------+

              --> read/write reqs over time -->
```

Both read and write operations may require *blocking*. Blocking is when a thread stops execution while waiting for access to a resource, like data from a file, or a variable protected by a lock. While a thread is blocked on one task it can't make progress on another task. So in an I/O heavy system, any particular request might spend *most* of its time just waiting on the operating system and memory control to move data to and from the disk:

```
        +---------+----------------------------+---------+
    R1  | working | waiting for data ...       | working |
        +---------+----------------------------+---------+

        --> time -->
```

The simplest way to put the CPU back to work while one request is blocked is to service requests on multiple threads, so that ideally our requests are all processed concurrently, and — if we have enough CPUs — in parallel:

```
  thread
         +  +---------+
    T1   |  |   R1    |
         |  +---------+
    T2   |  |   R2    |
         |  +---------+
    T3   |  |   W1    |
         |  +---------+
    T4   |  |   W2    |
         +  +---------+

            --> read/write reqs over time -->
```

So that will be the focus of this project — to process requests in parallel.

Part 1: Multithreading

Your first try at introducing concurrency is going to be the simplest: spawning a new thread per incoming connection, and responding to the request on that connection, then letting the thread exit. What performance benefits will distributing work across threads provide? How do you expect latency will be affected? What about throughput?

The first step is to write a `ThreadPool` implementation for this naive approach, where `ThreadPool::spawn` will create a new thread for each spawned job. Call it `NaiveThreadPool` (it's not *really* even a thread *pool* since this implementation is not going to reuse threads between jobs, but it needs to conform to our trait for later comparisons).

We aren't focusing on a more sophisticated implementation now because simply integrating this solution into our existing design is going to take some effort. Note that the `ThreadPool::new` constructor takes a `threads` argument specifying the number of threads in the pool. In this implementation it will be unused.

Go ahead and implement this version of `ThreadPool` now, then we'll integrate it into the new `KvStore`.

Test cases to complete:

- `thread_pool::naive_thread_pool_*`

Part 2: Creating a shared KvsEngine

Before we can integrate the NaiveThreadPool into KvServer we have to make the KvsEngine trait and the KvStore implementation (for now you can ignore the SledKvsEngine from the previous project, but you can optionally re-implement it as an extension to this project).

Recall from the project spec that, this time, our KvsEngine takes self as &self instead of &mut self as previously, It also implements Clone, which must be done explicitly for each implementation, as well as Send + 'static, implicit properties of the definition of each implementation. More concretely, it looks like

```
pub trait KvsEngine: Clone + Send + 'static {
    fn set(&self, key: String, value: String) -> Result<()>;

    fn get(&self, key: String) -> Result<Option<String>>;

    fn remove(&self, key: String) -> Result<()>;
}
```

This gives us a lot of clues about the implementation strategy we're pursuing. First, think about why the engine needs to implement Clone when we have a multithreaded implementation. Consider the design of other concurrent data types in Rust, like [Arc]. Now think about why that makes us use &self instead of &mut self. What do you know about shared mutable state? By the end of this project be sure you understand the implications here — *this is what Rust is all about*.

In this model, KvsEngine behaves like a *handle* to another object, and because that object is shared between threads, it probably needs to live on the [heap], and because that shared state can't be mutable it needs to be protected by some synchronization primitive.

So, *move the data inside your implementation of* KvsEngine, KvStore *onto the heap using a thread-safe shared pointer type and protect it behind a lock of your choosing*.

Since SledKvsEngine implements KvsEngine it may also need to change.

At this point your single-threaded kvs-server should work once again, but now with a KvsEngine that can later be shared across threads.

Test cases to complete:

- kv_store::concurrent_*

Part 3: Adding multithreading to KvServer

Let's quickly review our architecture here: KvServer sets up a TCP socket and begins listening on it; when it receives a request it deserializes it and calls some implementation of the KvsEngine trait to store or retrieve data; then it sends back the response. The details of how KvsEngine works don't matter to KvServer.

So in the last project you probably created a loop vaguely like:

```
let listener = TcpListener::bind(addr)?;

for stream in listener.incoming() {
  let cmd = self.read_cmd(&stream);
  let resp = self.process_cmd(cmd);
  self.respond(&stream, resp);
}
```

Well, now you just need to do the same thing, but spawn all the work inside the loop into your `NaiveThreadPool`. The database query and the response are both handled on a different thread than the TCP listener. This offloads most of the hard work to other threads, allowing the recieving thread to process more requests. It should increase throughput, at least on multi-core machines.

Again, you should still have a working client/server key-value store, now multithreaded.

Part 4: Creating a real thread pool

So now that you've got your multithreaded architecture in place, it's time to write a real thread pool. You probably wouldn't write your own thread pool in practice as there exist thread pool crates that are well-tested, but it is a useful exercise to gain experience with concurrency in general. Later in this project you will, as we did with the engine in the previous project, abstract the thread pool and compare the performance of yours with an existing.

So, what is a thread pool?

It's nothing complicated. Instead of creating a new thread for every multithreaded job to be performed, a thread pool maintains a "pool" of threads, and reuses those threads instead of creating a new one.

But why?

It's entirely about performance. Reusing threads saves a small amount of performance, but when writing high-performance applications, every bit counts. Imagine what it takes to make a new thread:

You've got to have a call stack for that thread to run on. That call stack must be allocated. Allocations are pretty cheap, but not as cheap as no allocation. How that call stack is allocated depends on details of the operating system and runtime, but can involve locks and syscalls. Syscalls again are not *that* expensive, but they are expensive when we're dealing with Rust levels of performance — reducing syscalls is a common source of easy optimizations. That stack then has to be carefully initialized so that first [stack frame] contains the appropriate values for the base pointer and whatever else is needed in the stack's initial function prologue. In Rust the stack needs to be configured with a [guard page] to prevent stack overflows, preserving memory safety. That takes two more syscalls, to `mmap` and to `mprotect` (though on Linux in particular, those two syscalls are avoided).

That's just setting up the callstack. It's at least another syscall to create the new thread, at which point the kernel must do its own internal accounting for the new thread.

In Rust, the C [libpthread] library handles most of this complexity.

Then at some point the OS performs a [context switch] onto the new stack, and the thread runs. When the thread terminates all that work needs to be undone again.

With a thread pool, all that setup overhead is only done for a few threads, and subsequent jobs are simply context switches into existing threads in the pool.

So how do you build a thread pool?

There are many strategies and tradeoffs, but for this exercise you are going to use a single shared queue to distribute work to idle threads. That means that your "producer", the thread that accepts network connections, sends jobs to a single queue (or channel), and the "consumers", every idle thread in the pool, read from that channel waiting for a job to execute. This is the very simplest work scheduling strategy, but it can be effective. What are the downsides?

You have three important considerations here:

1) *which data structure to use to distribute the work* — it's going to be a queue, and there is going to be one sender ("producer"), the thread listening for TCP connections, and many recievers ("consumers"), the threads in the pool.

2) *how to deal with panicking jobs* — your pool runs arbitrary work items. If a thread panics, the thread pool needs to recover in some way.

3) *how to deal with shutdown* — when the `ThreadPool` object goes out of scope it needs to shut down every thread. It must not leave them idle.

These concerns are all intertwined since dealing with each of them may involve communication and synchronization between threads. Some solutions will be simple, the solutions to each of these working together gracefully; some solutions will be complex, the solutions being independent and convoluted. Choose your data structures carefully and use their capabilities wisely.

You will distribute work by sending messages over some concurrent queue type (a concurrent queue in Rust typically being a data structure with two connected types: sender types, and reciever types; and that can send between the two types any type that implements `Send` + `'static`).

Messages in Rust are typically represented as enums, with variants for each possible message that can be sent, like:

```
enum ThreadPoolMessage {
    RunJob(Box<FnOnce + Send + 'static>),
  Shutdown,
}
```

This tends to be a simpler and more efficient solution than trying to "juggle" multiple channels for different purposes. Of course, if there is only one type of message, an enum is not necessary. Now, the above example may or may not be the full set of messages you need to manage your thread pool, depending on the design. In particular, shutting down can often be done implicitly if your queue returns a result indicating that the sender has been destroyed.

There are many types of multithreaded queues. In Rust the most common is the [`mpsc`] channel, because it lives in Rust's standard library. This is a multi-producer, single consumer queue, so using it for your single-queue thread pool will require a lock of some kind. What's the downside of using a lock here? There are many other concurrent queue types in Rust, and each has pros and cons. If you are willing to take a lock on both producer and consumer sides, then you could even use a `Mutex<VecDeque>`, but there's probably no reason to do that in production when better solutions exist.

Historical note: the existence of channels in Rust's standard library is a bit of a curiosity, and is considered a mistake by some, as it betrays Rust's general philosophy of keeping the standard library minimal, focused on abstracting the operating system, and letting the crate ecosystem experiment with advanced data structures. Their presence is an artifact of Rust's development history and origins as a message-passing language like Go. Other libraries like [`crossbeam`] provide more sophisticated alternatives, and sometimes more suitable options 😉.

Your thread pool will need to deal with the case where the spawned function panics — simply letting panics destroy the threads in your pool would quickly deplete its available threads. So if a thread in your pool panics you need to make sure that the total number of threads doesn't decrease. So what should you do? You have at least two options: let the thread die and spawn another, or catch the panic and keep the existing thread running. What are the tradeoffs? You've got to pick one, but leave a comment in your code explaining your choice.

Some of the tools at your disposal are [`thread::spawn`], [`thread::panicking`], [`catch_unwind`], [`mpsc`] channels, [`Mutex`], crossbeam's MPMC channels, and `threads` [`JoinHandle`]. You may use any of these, but probably not all.

Create the `SharedQueueThreadPool` type, implementing `ThreadPool`.

Test cases to complete:

- `shared_queue_thread_pool_*`

Replace the `NaiveThreadPool` used by `KvServer` with `SharedQueueThreadPool`. Again your `kvs-server` should still work the same as previously, now with a slightly more clever multithreading model. This time you'll want to call the thread pool constructor with an appropriate number of threads. For now you can create a thread per CPU, using the [`num_cpus`] crate. We'll revisit the number of threads later.

Part 5: Abstracted thread pools

As in the previous project where you created a `KvsEngine` abstraction to compare different implementations, now you are going to use the `ThreadPool` abstraction to do the same.

If you haven't already, add a second type parameter to `KvServer` to represent the `ThreadPool` implementation, the constructor to accept the thread pool as its second argument, and use that threadpool to distribute the work.

Finally create one more `ThreadPool` implementation, `RayonThreadPool`, using the `ThreadPool` type from the [`rayon`] crate.

Rayon's thread pool uses a more sophisticated scheduling strategy called "work stealing", and we'll expect it to perform better than ours, but who knows until we try!

Part 6: Evaluating your thread pool

Now you are going to write *six* benchmarks, one write-heavy workload comparing performance of `SharedQueueThreadPool` with varying numbers of threads, one read-heavy workload comparing the performance of `SharedQueueThreadPool` with varying number of threads; two more that use `RayonThreadPool` instead of of `SharedQueueThreadPool`, and finally, yet two more that use `RayonThreadPool` in conjunction with `SledKvsEngine`.

It's not as much as work as it sounds — four of them are essentially duplicates of the first two.

Note: the next two sections describe a fairly complex set of benchmarks. They can be written (probably… nobody has done it yet), but it may be challenging both to understand and to write efficiently. These sections do introduce some useful criterion features, but if it's too overwhelming it's ok to skip [forward] (and file a bug about what didn't work for you). On the other hand, the difficulty here may present a good learning opportunity. Finally, implementing these benchmarks as described requires a way to shutdown `KvsServer` programmatically (i.e. without sending `SIGKILL` and letting the OS do it), which we have not previously discussed.

So as part of this you will need to make sure the `SledKvsEngine` implementation you wrote as part of the previous project works again in this multithreaded context. It should be trivial as sled can be cloned and sent between threads, just like your engine.

Hopefully the results will be interesting.

Again you will use criterion.

These are going to be *parameterized* benchmarks, that is, single benchmarks that are run multiple times with different parameters. Criterion calls these benchmarking with inputs. And the parameter to you benchmarks will be the number of threads in the thread pool.

What you are attempting to test is the throughput of your server under various conditions. You will be sending many requests concurrently, waiting for the responses, then ending. One thing you should be curious about here is how the number of threads affects your throughput compared to the number of CPUs on your machine; how your threadpool compares to rayon's; and how your `KvStore` compares to `SledKvsEngine` in a multithreaded context.

This will be somewhat complicated by the fact that your `KvsClient` is (probably) blocking, that is, it sends a request then waits for a response. If it was non-blocking, then you could send *many* requests without waiting for responses, then collect the responses later. With a blocking `KvsClient` you will need to send each request in its own thread in order to saturate the server's capacity.

When benchmarking it is important to understand exactly what code you are trying to measure, and to the greatest extent possible only measure that code. A benchmarker like criterion runs a single piece of code many times in a loop, measuring the time it takes through each loop. As such we want to put only the code we want to measure in the loop, and leave as much outside of the loop as we can.

So take this simple example of a criterion benchmark with inputs:

```
let c = Criterion::default();
let inputs = &[1, 2, 3, 4, 5];

c.bench_function_over_inputs("example", |b, &&num| {
    b.iter(|| {
        // important measured work goes here
  });
}, inputs);
```

That `iter` calls your closure many times, measuring each iteration. But since you are going to need to set up a lot of threads beforehand, that is work that you don't want to measure. If you can do the setup only once for multiple iterations then the setup can go outside the closure, like

```
let c = Criterion::default();
let inputs = &[1, 2, 3, 4, 5];

c.bench_function_over_inputs("example", |b, &&num| {
    // do setup here
    b.iter(|| {
        // important measured work goes here
  });
}, inputs);
```

The code inside the `b.iter` closure is what is measured, setup goes before.

If the setup can't go before the loop, then another strategy is to make the amount of setup work for smaller than the amount of work you actually want to measure, by e.g. adding loops. Also consider that "teardown" of the benchmark, which will often mostly consist of running `drop` implementations, also has a cost.

If you have a blocking client, you are going to need many threads for your clients, and you only have the opportunities to create those threads once, prior to the many iterations of your loop. So you'll need to set up a bunch of reusable threads before you iterate over the benchmark. Fortunately you have the perfect tool for that in your `SharedQueueThreadPool`. Set that up with a thread per request, and pair it with some channels to report back that the response is received, and you will have a suitable benchmark harness.

Ok, now to the first two benchmarks

We have said that this is a parameterized benchmark, and the parameter to the benchmark is the number of CPUs to use in the server's thread pool. We want to see what the throughput is like with just 1 thread, with 2, with 4, and then for every even number up to 2x the number of cores in your computer. Why 2x? Well, there may be benefits to having more threads than cores, and you are going to find out experimentally.

For the write-heavy workload, during setup (the part that runs before the call to `b.iter(...)`), create the `KvServer<KvStore, SharedQueueThreadPool>`, with the thread pool containing the parameterized number of threads. Then write a workload that sets 1000 unique keys of the same length, all to the same value. Note that though the keys are different, for consistent results they need to be the same keys every benchmark loop.

Then after each thread sets a value they should also `assert!` that the call was successful (to ensure there are no bugs under load), then indicate that it is done. When all threads are done the benchmarking thread continues and finishes that iteration. An obvious way to implement this signaling of completion is for each thread to send a message back to the benchmarking thread, but keep in mind that the signaling code is overhead unrelated to the code you are trying to measure, so it needs to do a minimal amount of work. Can you do it with only one message, or maybe with some other concurrent type that only signals the benchmarking thread once?

Call this benchmark `write_queued_kvstore` (or whatever).

For the read-heavy workload, during setup, create the `KvServer<KvStore, SharedQueueThreadPool>`, with the thread pool containing the parameterized number of threads, and create your client thread pool containing 1000 threads. Still in the setup phase, create yet another client and initialize 1000 unique keys of the same length, all to identical values.

Then, during the benchmarking loop, from the client, spawn 1000 jobs that retrieve those same key/value pars, and then `assert!` that the result is correct. Finally, as before, send a message back to the benchmarking thread indicating that read is complete.

Call this benchmark `read_queued_kvstore` (or whatever).

Whew. That was a lot of work.

So you can run this set of criterion benchmarks as usual with `cargo bench`.

But this time you are going to do more. Since you are running the same benchmark over multiple parameters, representing the number of threads in your threadpool, what would be really nice is if we could see the effect of different thread counts in a nice graph.

Oh, hey — criterion does that!

Go back and read about benchmarking with inputs. It explains how to see the graph of your benchmark against its inputs. What do you notice? What happens as your number of threads approaches the number of CPUs on your machine? What happens as the number of threads exceeds the number of threads on your machine? What do you think accounts for the trend you see? The results depend on many factors, so your results might be different from anybody elses.

That's a good reason to always benchmark, and not speculate about performance. We can make educated guesses, but we don't know until we test.

<!-- TODO: not sure if this would actually improve perf

Extension 1: Alternatives to the thousand thread approach

As described above, to write your benchmarks, you will need to spawn 1000 threads, one for each client that is generating load against your server. This is a necessity as `KvsClient`'s `get` and `set` methods *block* waiting for the result of the operation. This is going to cause a lot of overhead, and it is very likely to impact the quality of your benchmarks. The overhead here comes not from spawning and destroying the threads, since you've placed that work outside of the benchmarking loop via your `ThreadPool` setup. *But*, because each requested is generated on a different thread, that means that every request is going to require a context switch into and out of the kernel as those threads are scheduled.

It would be better if a single thread could issue many requests at once, then later wait for their results.

This is simple *asynchronous* style of programming, which is *not* the topic of this project, but *is* the topic of the next.

For this project though, if you want to create a more efficient benchmark, there is a simple way to do it, by having the "set" method return a handle that can later be waited on.

So imagine that your `KvsClient` API today looks like:

``rust pub fn get(&mut self, key: String) -> Result<Option>; pub fn set(&mut self, key: String, value: String) -> Result<()>; pub fn remove(&mut self, key: String) -> Result<()>;

```
If you instead added a new set of methods:

``rust
pub fn get_async(&mut self, key: String) -> Result<QueryHandle>;
pub fn set_async(&mut self, key: String, value: String) ->
Result<QueryHandle>;
pub fn remove_async(&mut self, key: String) -> Result<QueryHandle>;
pub fn wait_for_result(&mut self, q: QueryHandle) -> Result<QueryResult>;
```

then you could, e.g. issue many queries at once, store the handles in a vector, each containing an open TCP stream, then later wait on each result in turn. That would let your benchmarking client thread pool contain far fewer threads (probably one per CPU would be persisent).

We won't explore this solution at length here, but you might want to experiment in this direction, particularly if you find the comparisions between your benchmarks are not interesting.

TODO: Can we explain how to use perf to measure context switch time?

-->

Part 7: Evaluating other thread pools and engines

Ok. You've gotten the most difficult part of this benchmarking exercise out of the way. Now you've just got to do almost the same thing in a few more configurations.

Take those two benchmarks you wrote previously, and just copy-and-past them three times. In all of them, change `SharedQueueThreadPool` to `RayonThreadPool`.

The third and fourth, name `read/write_rayon_kvstore` (or whatever). These you will compare to the first two `SharedQueueThreadPool` implementations, to see the difference between yours and `RayonThreadPool`.

The fourth and fifth, name `read/write_rayon_sledkvengine`, and change the engine to `SledKvsEngine`. These you will compare to the previous two to see how your `KvsEngine` compares to sled's in a multithreaded environment.

As before, run and chart all these benchmarks. Compare them to each other as described above. How does your scheduler compare to rayon under various thread counts? How does your storage engine compare to sled under various thread counts? Are the results surprising? Can you imagine why the differences exist?

Extension 1: Comparing functions

Now you have identical benchmarks for three different thread pools, and you have run them and compared their performance yourself. Criterion has built-in support for comparing multiple implementations. Check out "comparing functions" in the Criterion User Guide and modify your benchmarks so that criterion does the comparison itself. Check out those gorgeous graphs.

Background: The limits of locks

Earlier in this project, we suggested making your `KvsEngine` thread-safe by putting its internals behind a lock, on the heap. You probably realized immediately that this wasn't going to improve throughput because it traded one type of blocking for another — instead of *maybe* blocking on disk access, it is now *definitely* blocking on mutex access.

So all we've achieved so far is this:

```
thread
          +  +---------+
   T1     |  |   R1    |
          |  +------------------+
   T2     |            |   R2    |
          |            +------------------+
   T3     |                      |   W1    |
          |                      +------------------+
   T4     |                                |   W2    |
          +                                +---------+
             --> read/write reqs over time -->
```

In the previous section you benchmarked the multithreaded throughput of your engine vs. the `SledKvsEngine`. Hopefully what you discovered is that your multithreaded implementation performed significantly worse than the `sled` multithreaded implementation (if not, well, either you are super-awesome or `sled` has some problems). Adding multithreading has so far resulted in performance that is strictly worse than the single-threaded implementation — now you've got the added work of context switching between threads, and the guaranteed blocking imposed by the mutex.

So for this part of the project, you are going to get a bit more sophisticated. Protecting the entire state behind a lock is easy — the entire state is always read and written atomically because only one client at a time has access to the entire state. But that also means that two threads that want to access the shared state must wait on each other. In other words, when `KvsEngine` is protected by a mutex then there is very little actual concurrency in the server, despite being multithreaded.

High-performance, scalable, parallel software tends to avoid locks and lock contention as much as possible. Rust makes sophisticated and high-performance concurrency patterns easier than most languages (because you don't need to worry about data races and crashes), but it *does not* protect you from making logical mistakes that would result in incorrect behavior.

So you've still got to do some hard thinking about concurrency. Fortunately there are many sophisticated parallel-programming tools in the Rust crate ecosystem, so your task is usually just to understand what they are and how to put them together, not to understand how to write complex lock-free data structures of your own.

Let's look at some progressively more sophisticated examples. We'll take an example single-threaded `KvStore` and consider how to make it thread-safe.

Here's an example single-threaded `KvStore` like you might have created in earlier projects (this is a simplified version of the one in the course example project):

```rust
pub struct KvStore {
    /// Directory for the log and other data
    path: PathBuf,
    /// The log reader
    reader: BufReaderWithPos<File>,
    /// The log writer
    writer: BufWriterWithPos<File>,
    /// The in-memory index from key to log pointer
    index: BTreeMap<String, CommandPos>,
    /// The number of bytes representing "stale" commands that could be
    /// deleted during a compaction
    uncompacted: u64,
}
```

And here's the simple multithreaded version, protecting everything with a lock. Hopefully what you've already written for this project looks something like this:

```rust
#[derive(Clone)]
pub struct KvStore(Arc<Mutex<SharedKvStore>>);

#[derive(Clone)]
pub struct SharedKvStore {
    /// Directory for the log and other data
    path: PathBuf,
    /// The log reader
    reader: BufReaderWithPos<File>,
    /// The log writer
    writer: BufWriterWithPos<File>,
    /// The in-memory index from key to log pointer
    index: BTreeMap<String, CommandPos>,
    /// The number of bytes representing "stale" commands that could be
    /// deleted during a compaction
    uncompacted: u64,
}
```

This `Arc<Mutex<T>>` solution is trivial, correct, and common:

- The [`Arc`] puts the value on the heap so it can be shared between threads, and provides a `clone` method to create a "handle" to it for each thread.
- The [`Mutex`] provides a way to regain write access to the value without having an existing `&mut` reference.

This is a perfectly reasonable solution for many cases. But in this case that mutex will be a source of *contention* under load: the `Mutex` doesn't only serialize write access to `SharedKvStore`, but read access as well. Any thread that wants to work with `KvStore` needs to wait for the `Mutex` to be unlocked by another thread. Any requests will block any other concurrent request.

What we *really* want is to not have to take locks, or — if locks are necessary — for them to rarely contend with other threads.

The next step up in sophistication from a `Mutex` is the [`RwLock`], the "reader-writer lock". This is another common type of lock that every programmer of parallel software must know. The improvement that a reader-writer lock makes over a mutex is that it allows *either* any number of readers, *or* a single writer. So in Rust terms, a `RwLock` will hand out any number of `&` pointers simultaneously, or a single `&mut` pointer. Readers still block on writers and writers still block on readers and other writers.

In our database that means that all read requests can be satisfied concurrently, but when a single write request comes in, all other activity in the system stops and waits for it. Implementing this is basically as simple as swapping the `Mutex` for `RwLock`.

And, considering our multi-threading diagram again, the resulting process flow looks like:

```
    thread
           +  +---------+
      T1   |  |   R1    |
           |  +---------+
      T2   |  |   R2    |
           |  +------------------+
      T3   |            |   W1    |
           |            +------------------+
      T4   |                      |   W2    |
           +                      +---------+
              --> read/write reqs over time -->
```

It's better, since the readers never block each other, but you can do better than that still.

Part 8: Lock-free readers

for this project you are challenged to create readers that never take a lock, even with a concurrent writer. A read request can always be serviced, regardless of write requests. (Writers can still block on other writers for now — besides being a challenging parallel programming problem, the question of whether it even makes sense to write in parallel is a difficult one to answer).

You want to end up with

```
thread
       +  +--------+
  T1   |  |   R1   |
       |  +--------+
  T2   |  |   R2   |
       |  +--------+
  T3   |  |   W1   |
       |  +-----------------+
  T4   |           |   W2   |
       +           +--------+
          --> read/write reqs over time -->
```

If we can acheive that then our readers will be lock-free: even while a single reader is blocked waiting for data from the filesystems all types of other operations, reads and writes, can proceed. Unfortunately, this will still not be enough to guarantee that the system can always service read requests. Think about what would happen if there are `N` blocked write requests on our thread pool of size `N`. You'll have to solve that later. For now, you are focused on removing the locks from readers.

Unlike `Mutex` and `RwLock`, there is no single wrapper type that we can apply to the entire arbitrary shared state to achieve the goal of reading and writing concurrently (at least, not while also being performant).

That means that we need to consider how each one of the fields of `SharedKvStore` is used, and pick the right synchronization scheme to allow all threads to make as much progress as we can, while still maintaining logical consistency of the data.

This is where the difficult reasoning with multithreading really begins. If you remove that big lock, Rust is still going to protect you from [*data races*], but it is not going to help you maintain the logical consistency between the fields necessary to maintain the invariants of your data store.

So before thinking about the solution, let's think about our requirements. We need to:

- Read from the index and from the disk, on multiple threads at a time;
- Write commands to disk, while maintaining the index;
- Read in parallel with writing, thus
- In general, to guarantee that readers will always see a consistent state while reading in parallel with a writer, which means,
  - Maintaining an invariant that log pointers in the index always point to a valid command in the log,
  - Maintaining appropriant invariants for other bookkeeping, like the `uncompacted` variable in the

following example;
- Periodically compact our on-disk data, again while maintaining invariants for readers.

The rest of this section is background on a variety of subjects that will be helpful to achieve the above, but that is the entire goal for the remaining project: modify `KvStore` to perform reads concurrently with writes.

Explaining our example data structure

In order to talk about this concretely, we're going to need an example of the data we're trying to protect and the invariants we're trying to maintain. So here's an example of a `KvStore` implementation and its fields.

```
pub struct KvStore {
    /// Directory for the log and other data
    path: PathBuf,
    /// The log reader
    reader: BufReaderWithPos<File>,
    /// The log writer
    writer: BufWriterWithPos<File>,
    /// The in-memory index from key to log pointer
    index: BTreeMap<String, CommandPos>,
    /// The number of bytes representing "stale" commands that could be
    /// deleted during a compaction
    uncompacted: u64,
}
```

This is a simplified version of the example for this project.

The purpose of the fields should be fairly clear:

`path: PathBuf` is just the path to the directory where logs are stored. It never changes — it is immutable, and immutable types are `Sync` in Rust, so it doesn't even need any protection at all. Every thread can read it at once through a shared reference.

`readers: HashMap<u64, BufReaderWithPos<File>>` is the read handle to the current log file. It needs to change to a new log file after compaction.

`writer: BufWriterWithPos<File>` is the write handle to the current log file. So any write needs mutable access to `writer`, and the compaction process needs to change the `writer` and the `current_gen`.

`index: BTreeMap<String, CommandPos>` is the in-memory index of every key in the database to its location in the index file. It is read from every reading thread, and written from every writing thread, potentially including during compaction.

`uncompacted: u64` simply counts the number of "stale" commands in the logs that have been superceded by subsequent write commands, to know when to trigger compaction.

In previous projects we didn't have to worry much about the interaction between writing, reading, and compaction producing inconsistent results, since they all happened on the same thread. Now if you are not careful with your data structure selection and their usage, it will be easy to corrupt the state of your database.

Strategies for breaking up locks

The key to advanced parallel programming is to know the tools available and when to use them. Here are some techniques we found useful while implementing this project, some of which you will need as well. They are discussed in the context of the example data structure presented above.

Understand and maintain sequential consistency

(Note that "sequential consistency" has a precise meaning, but here we're just talking generally about ensuring that things that need to happen in a specific sequence do so).

Reasoning about parallel programs is mostly about understanding the "happens-before" relationships in your code. In this thread, what changes to shared data structures do I need to see before others? What changes to shared data structures do I need to expose to other threads before others? How do I ensure that?

In single-threaded code reasoning about what happens before any particular line of code is trivial — if the code is written to happen before, then it happens before, if not, it happens after. But this isn't actually true at all, even in single-threaded code: both the CPU and the compiler will reorder code to make it run faster, the CPU operating on machine code, and the compiler operating on its internal representation prior to generating the machine code. In reality the actual code executed happens in a different order than you wrote it to execute, and it only appears to run the way you wrote it because both the CPU and compiler track *data dependencies* and don't reorder any operations that depend on another.

In multi-threaded code the compiler and CPU will *still* reorder code under the same assumptions as single-threaded code, and your code will break entirely unless you tell the compiler via synchronized types and operations that it must not allow reordering.

Any operation that must occur before or after another must be exlicitly arranged arranged to do so with synchronized types or operations, whether they be locks, atomics or otherwise.

In our example it's clear that the write to the file and the write to the index must be seen to occur in a specific order — what would happen if the index was updated before the file? Likewise, our example contains another bit of state, `uncompacted`. What's the impact of miscalculating the uncompacted size? It may not be so bad if the value of `uncompacted` can be seen to change before the data is committed to file, but it's a decision that has to be made for each value that is synchronized independently.

Identify immutable values

You've probably read a lot about immutability in Rust, and about how immutable values can be shared trivially between theads (they are `Sync`). Immutable values are the best for concurrency — just throw them behind an `Arc` and don't think about them again.

In our example, `PathBuf` is immutable.

Duplicate values instead of sharing

Cloning sometimes has a bad reputation in Rust, particularly cloning types with arbitrary size, like `String`, and `Vec`. But cloning is often perfectly reasonable: it can be quite difficult to avoid clones in some situations, and CPUs are *very good* at copying buffers of memory. Furthermore, considering our use case, the number of state copies needed to support the server is bounded by the number of threads in the threadpool.

In our example, again `PathBuf` is easily clonable.

Less obviously though, consider how to share access to files across threads. The [`File`] type requires mutable access for both reads and writes. So to share it across threads would require a lock that grants that mutable access. What is a `File` though? It's not actually a file — it's just a handle to the physical resource on disk, and it's fine to have multiple handles to the same file open at once. Note the API for `File` though — it doesn't implement `Clone`, and while it does have this enticing [`try_clone`] method, its semantics have some complex implications for multi-threaded applications.

Break up data structures by role

In our use case we have two clear roles: readers, and writers (and maybe a third for compactors). The logical separation of readers and writers into their own concurrent types is a common in Rust. Readers have their own data set to work with, and writers their own, and that provides a good opportunity for encapsulation, with all read operations beloning to one type and all write operations another.

Making this distinction will further make it very obvious which resources are accessed by both, since the reader and writer will both carry shared handles to those resources.

Use specialized concurrent data structures

Just knowing what tools are available and in which scenarios to use them may be the most difficult part of parallel programming. Beyond the basic lock types tought to everybody in school, synchronized data types become increasingly specialized.

In this project, since the in-memory index is some type of associative data structure (a.k.a. a "map"), like a tree or hash table, it's natural to ask whether there exist concurrent associative data structures.

There are, and using them is key to completing this project.

But how can you know that? The first step is ask whether concurrent maps exist. You could do this in `#beginners` on the [Rust Discord], but in this case searching "concurrent map" on the web will definitely give the answer.

That's the easy part, finding the right concurrent map *in Rust* is harder. A good first step is to learch [libs.rs]. libs.rs is like crates.io but where crates.io contains all published libraries, libs.rs is curated to contain only libraries that are well-regarded by ... well, somebody. So if it's on libs.rs then that's one indication that the library is usable, another is the download count on [crates.io] — in general, more downloaded crates are more tested than less downloaded crates. The download count can be seen as a rough proxy for the number of people who "vouch" for the crate. And finally, asking in chat is always a good idea.

Postpone cleanup until later

Like cloning, garbage collection is often frowned upon in Rust — avoiding GC is almost the entire reason Rust exists. But it's no secret that, actually, garbage collection can't be avoided, "garbage collection" and "memory reclaimation" are practically synonymous, and every language uses a mixture of garbage collection strategies. One one end of the GC spectrum, in languages with no automatic memory management, like C, the garbage collection is left entirely up to the programmer, e.g. via `malloc` and `free`. On the other end are garbage collected languages, like Java, where all memory is collected by a single general-purpose garbage collector.

In practice though, neither is all memory management and reclaimation in C done with `malloc`/`free`, nor is all memory management in Java done with the GC. Just as a trivial example, it is common for high-performance applications in both to rely on specialized [arenas], in which allocations can both be reused as well as deallocated in large batches, to optimize their memory access patterns.

Likewise in Rust, not all memory is freed deterministically. Trivial examples are in the [`Rc`] and [`Arc`] types that implement [resource counting], a simple kind of GC.

One of the greatest benefits of global garbage collectors is that they make many lock-free data structures possible. Many of the lock-free data structures described in academic literature rely on the GC for their operation. The need to adapt lock-free algorithms to not rely on a GC is the original motivation for the [`crossbeam`] library and its [`epoch`] type.

All this is to say that garbage collection comes in many forms, and its basic strategy of delaying the cleanup of resources until some future time is powerful in many scenarios.

When you can't figure out how to perform some concurrent work *right now*, it can be useful to ask "can I just do this later?"

Share flags and counters with atomics

Under the hood, most concurrent data structures are implemented using [atomic operations], or "atomics". Atomics operate on a single cell of memory, usually between 8 and 128 bytes, commonly word size (the same number of bytes as a pointer, and as the Rust `usize` type). If two threads use atomics correctly then the result of a write in one thread is visible immediately to a read in the other thread. In addition to making reads or writes immediately visible, atomic operations also constrain how the compiler and CPU may reorder instructions, in Rust via the [`Ordering`] flag.

When moving from the course-grained parallelism of locks to more fine-grain parallelism, its often necessary to augment off-the-shelf concurrent data structures with atomics.

Implement lock-free readers

That's a lot of background. Hopefully there is a lot there to think about and guide you in the right direction. Now it's your turn:

Modify `KvStore` to perform reads concurrently with writes.

And afterward…

Nice coding, friend. Enjoy a nice break.

<!--

Some ideas for sharing data without big locks

- TODO: https://gitlab.redox-os.org/redox-os/chashmap
- TODO: https://github.com/jonhoo/rust-evmap
- https://github.com/4lDO2/evc
- crossbeam-skiplist
- atomics
- invariants
- concurrent maps https://gitlab.nebulanet.cc/xacrimon/rs-hm-bench

Some of the data types here have equivalent concurrent types: for example, a `u64` can be replaced with an `

OK, I hope you are prepared. Go remove as much locks and contention from this type as you can.

There are no new test cases to complete here, but some of the earlier ones will stress this new data structure in challenging ways, your previously-written benchmarks will stress this implementation hard.

Part 9: Benchmarking lock-free data structures

TODO: just do a read-write benchmark in the earlier section, verify sum of keys TODO: make sure benchmark section always mentions to assert the results -->

<!--

Extension 1: Background compaction

- discuss issues with files and concurrency
- move compaction to a background thread
- need to refactor previous projects to use multiple logs
- this should be fairly challenging -->

<!--

Background reading ideas

- scheduling strategies
- shared mutable state, especially in multithreaded context
- threadpools
- something about parallelism and concurrency
- something that explains Arc
- something about the distinction between interior and exterior mutability, bonus if it includes parallelism
- concurrent map comparison https://gitlab.nebulanet.cc/xacrimon/rs-hm-bench

TODOs

- a concurrent map or skiplist would be better than a mutexed hashmap but there doesn't seem to be a prod-quality crate for it
- is there some new kind of measurement we can do for thread pools in addition to criterion benchmarks?
- panic handling for threads in the threadpool
- In `KvStore(Arc<SharedKvStore>)` example discuss patterns for accessing types, particular not to be tempted to use `Deref`.
- mention condvars somewhere

-->