CS3211 - Parallel and Concurrent Programming

Overview

Concurrency vs Parallelism

- Concurrency: 2 or more tasks can start, run, and complete in **overlapping time periods**. They might or might not be executing on CPU at the same instant, in which case they can make progress by interleaving their executions (switching).
- Parallelism: 2 or more tasks can execute on CPU simultaneously **at the same time**.

Hardware (multicore, multiprocessor, SMT) enables TRUE concurrency (a.k.a. parallelism) and the number of hardware threads dictate the amount of TRUE concurrency.

We can enable concurrency by making use of:

- multiple process: overhead of creation (system calls, data structures allocation, initialization, copy) and communication between processes (via OS)
- multiple threads: cheaper as it share the address space of the process (no copy)

Process interaction with OS can come as exceptions or interrupts.

- Exceptions: execution of a machine level instruction causes exception. It is synchronous (due to program execution). Execute an exception handler.
- Interrupts: external events, usually hardware-related (timer, mouse movement) interrupts program execution. It is asynchronous (independent of program execution). Execute an interrupt handler.

Shared Memory Paradigm

Race Condition vs Data Race

- Race Condition happens when the outcome depends on the relative ordering of execution of operations on 2 or more threads. *Not all race conditions are bad*.
- Data Race happens when: 2 concurrent threads access a shared non-atomic resource without sync ops. and at least 1 thread modifies the shared resource.

Critical section reequirements:

- Mutual exclusion (mutex): if one thread is in the critical section, then no other is.
- Progress: If some thread T is not in the critical section, then T cannot prevent some other thread S from entering the critical section. A thread in the critical section will eventually leave it.
- Bounded waiting (no starvation): If some thread T is waiting on the critical seciton, then T will eventually enter the critical section
- Performance: the overhead of entering and exiting CS is small w.r.t. work done in it.

Mechanisms: Locks, Semaphores, Monitors, Messages

Deadlock: occurs when the waiting process is still holding onto another resource that first needs before it can finish. It holds iff all 4 conditions below hold:

- Mutual exclusion: at least 1 resource must be held in a non-sharable mode
- Hold and Wait: 1 process is holding 1 resource and waiting for another resource
- No preemption: CS cannot be aborted externally
- Circular wait

Starvation: a situation where process is prevented from making progress because some other process has the resource it requires. It is a side effect of scheduling algo

Concurrency:

- Advantages: separation of concerns, performance.
- Disadvantages: concurrency issues, maintenance difficulties, threading overhead

Types of Parallelism:

- Task Parallelism (focus of this module): Do the same work faster
  - divide the work by task type
  - divide a sequence of tasks (pipeline)
  - use task pools
- Data Parallelism: embarassingly parallel algo, do same work with different parts of data.

Shared Address Space Model

- communication abstraction: by reading/writing from/to **shared variables**
- Assumes hardware support
  - any thread can load and store from any address
  - matches shared memory systems (access to memory is efficient, but contention increases with the number of threads running simultaneously)
- Assumes cache coherence and memory consistency

C++

Background (C++98)

- doesn't acknowledge the existence of threads.
- Multithreading support is only provided via compiler-specific extensions.
- No memory model: assume a sequential abstract machine

C++11 introduce a thread-aware memory model and other synchronization facilities → allows for portable multithreaded code with guaranteed behaviour (no longer rely on platform-specific extensions)

Thread creation:

```
std::thread t1([](std::string name){ do_something(); })
// wait for t1 to finish.
```

```
// make sure to join even when there is an exception
t1.join();
```

If we are detaching the thread, we need to care about the scope of local variables passed into the function.

```
void f(int i, std::string const& s);
void beware_of_scope(int some_param) {
        char buffer[1024];
        sprintf(buffer, "%i", some_param);
        // oops, might exit before buffer is converted into std::string
        // std::thread t(f, 3, buffer);
        std::thread t(f, 3, std::string{ buffer });
        t.detach();
}

void g(widget_data& data);
void beware_of_unnecessary_copy() {
        widget_data data;
        // oops, data is first copied, and then passed by ref into g
        // std::thread t(g, data);
        std::thread t(g, std::ref(data));
        t.join();
}
```

C++ Ownership

C++'s model of resource management is based on the use of constructors (ctors) and destructors (dtor). Each object on the free store (heap, dynamic store) must have exactly 1 **owner**.

RAII

RAII(Resource Acquisition is Inialization) → binds the lifecycle of a resource that must be acquired before use e.g. allocated heap memory, thread, open socket, open file, locked mutex, database connection, to the lifetime of an object.

Lifetime

- Begins when storage is obtained and initialization is complete
- Ends when the object is destroyed (non-class type) or destructor call starts (class type)
- Lifetime of a reference begins when its initialization is complete and ends as if it were a scalar object
  - No guarantee on the lifetime of the referred object: may end before the end of the lifetime of the reference → dangling references

```
void demo1() {
        A a; // Start of lifetime of a
        { // any scope, including control-flow (if) and for-loops.
```

```
            A b; // Start of lifetime of b
        } // End of lifetime of b
        A c; // Start of lifetime of c
        // End of lifetime of c
        // End of lifetime of a
}
```

Thread Ownership

Same semantics as `std::unique_ptr`. `std::thread` instances own a resource. Instances are movable but not copyable.

C++ Synchronization

Synchronization is needed to maintain invariants, i.e. statements that are always true about a particular data structure.

- invariants are often broken during an update on the data structure → should prevent other threads from working with the data.

Mutex

Mutex provides serialization.

```
std::mutex mut;

void enqueue(Job job) {
        std::scoped_lock lock{mut}; // ctor locks mutex
        // or std::unique_lock lock{mut}

        jobs.push(job);
        // dtor unlocks mutex
}
```

Use `std::shared_mutex` to allow concurrent access from readers, but exclusive access for writers.

Locks

Locks build on RAII. Using locks are safer than manually calling `.lock()` and `.unlock()` on the mutex as RAII ensures that `.unlock()` is automatically called on scope exit regardless of exceptions etc.

- `std::unique_lock` allows locking a `std::mutex` later using `std::defer_lock`.
- `std::shared_lock` allows shared locking of a `std::shared_mutex`
- `std::lock` locks one or more mutexes at once without risk of deadlock.
- `std::scoped_lock` uses less memory than `std::unique_lock`, if you don't need to call `.lock()` and `.unlock()` manually.

Best Practices

It is common to group the mutex and protected data together in a class rather than use global variables. Don't pass pointers and references to protected data outside the scope of the lock. Smells:

- returning ptrs/refs from a function
- storing data and ptrs/refs to it in externally visible memory
- passing ptrs/refs to data as arguments to user-supplied functions.

Condition Variable and Monitor

Monitor consists of a mutex, a condition variable, a condition to wait for.

During a call to `wait(lock, predicate)`, a condition variable:

- may check the predicate any number of times with the mutex locked → predicate must be pure functions
- returns immediately iff the predicate returns true
- Spuriously waken up, i.e. not as a direct response from a notification

```
std::condition_variable cond;

void enqueue(Job job) {
        {
            std::scoped_lock lock{mut};
            jobs.push(job);
        }

        cond.notify_one();
}

Job dequeue() {
        std::unique_lock lock{mut}; // cond.wait calls .unlock()

        while (jobs.empty()) {
                // unlocks lock and blocks on cond.
                cond.wait(lock);
                // can be spuriously waken up, that's why we loop again.
        }
        // Alternatively, this is exactly the same code
        // cond.wait(lock, [this]() { return !jobs.empty(); });

        Job job = jobs.front();
        jobs.pop();

        return job;
}
```

C++ Memory Model

Compiler can reorder instructions. Hardware can reorder execution. However, correcty synchronized program should behave as if:

- memory ops are executed in an order that is equivalent to some sequentially consistent interleaved execution
- each write appears to be atomic and globally visible simultaneously to all core.

So, languages need memory models too. Memory models provide a contract to programmers about how their memory operations will be reordered by the compiler. Execution reorderings by the hardware are transparent to programmer and handled by the compiler (compiler will insert necessary sync to cope with HW memory model)

> C++11 guarantee sequential consistency for data-race-free programs (without non-seq_cst atomics), a.k.a. "SC for DRF"

As-If Rule (abridged)

The C++ compiler is permitted to perform any changes to the program as long as the following remains true:

- Preserve **visible side effects**: at program termination, data written to files is exactly as if the program was executed as written
- Prompting text which is sent to interactive devices will be shown before program waits for input.
- Program has no undefined behaviour. Otherwise, compiler is free to violate this rule

Modification Orders

For every object in the program, there is a modification order. It is composed of all writes to an object from all threads.

- Once a thread has seen a particular entry in the modification order, subsequent reads must return that or later values, and subsequent writes must occur later in the modification order.
- All threads must agree on the modification orders of each individual object.
  - Threads need not necessarily agree on the relative order of operations on separate objects.

Atomics

Atomic Operations is a (language-guaranteed) indivisible operation, i.e. always observed fully done from any thread in the system. It can be used to enforce modification orders.

- Example: `load, store, exchange, compare_exchange_weak, compare_exchange_strong, fetch_add, fetch_sub, test_and_set`. Not all types support all operations.
- All operations have an optional memory-ordering argument, which defaults to seq_cst

Atomic != lock-free. Atomic types can be implemented using a mutex internally.

std::memory_order enumeration

- `relaxed`: all ops
  - no sync or ordering constraint imposed on other reads or writes, only operation's atomicity is guaranteed
- `release`: store / read-modify-write ops.
  - A store operation with this memory order performs the *release operation*: no reads or writes in the current thread can be reordered *after* this store.
  - All writes in the current thread are visible in other threads that acquire the same atomic variable.
- `acquire`: load / read-modify-write ops
  - A load operation with this memory order performs the *acquire operation*: no reads or writes in the current thread can be reordered before this load.
  - All writes in other threads that release the same atomic variable become visible in the current thread.
- `acq_rel`: read-modify-write ops
  - Is both an *acquire operation* and a *release operation* → No memory reads or write in the current thread can be reordered before the load nor after the store.
- `seq_cst`: This is the default ordering
  - All loads are *acquire operation*, all stores are *release operation* and all read-modify-write are *acquire operation* and *release operation*.

Happens-Before

To reason about modification order, we can use happens-before relationship. It specifies which operations see the effects of which other operations.

- Intra-thread happens-before (aka program order / **sequenced-before**)
- Inter-thread happens-before (aka **synchronizes-with**)

Happens-Before is transitive.

Memory Ordering for Atomic Operations

Some architectures (called weakly-ordered platforms e.g. ARM), need additional sync instructions to achieve sequential consistent ordering. In x86-64 loads by default have acquire ordering, and stores have release ordering.

Sequentially Consistent

tl;dr: A single, global (agreed by all threads) modification order of all memory operations tagged with `seq_cst`

- Using SC ordering signals that there are 2 or more atomic variables that must be consistent with one another.
- A SC store synchronizes-with a SC load of the same variable that reads the value stored
  - Does not apply to atomic operations with other memory orderings (they don't have a single global order of events) → Advice: use SC operations on all threads.

Acquire-Release

No total order of operations, but there is pairwise synchronization between a *release* operation and an *acquire* operation that *reads the value written*.

To force sync, we usually use while loops or CAS (Compare-And-Swap) loops.

```
while(x.load(std::memory_order_acquire) != intended_value));

int expected = 1;
// while I failed to change from expected -> 2
while(!y.compare_exchange_weak(expected, 2, std::memory_order_acq_rel))
        expected = 1;
```

Moreover, If an atomic store in thread A is tagged `memory_order_release` and an atomic load in thread B from the same variable is tagged `memory_order_acquire`, all memory writes (non-atomic and relaxed atomic) that happened-before the atomic store from the point of view of thread A, become visible side-effects in thread B. In other words, once the atomic load is completed, thread B is guaranteed to see everything thread A wrote to memory.

- Using Acquire-Release ordering signals that we process things in batch.
- Mutex and spinlock are examples of release (unlock) and acquire (lock) sync, decrementing shared_ptr counters is acq-rel with dtor.

Relaxed

- Using relaxed ordering signals that we only care about atomicity, but not execution orders, e.g. incrementing a counter.
- Operations performed with relaxed ordering don't participate in synchronizes-with relationships, i.e. no requirement on ordering relative to other threads, although it still obey sequenced-before relationships.

Fences `std::atomic_thread_fence`

Fences can be used to establish memory synchronization ordering of non-atomic and relaxed atomic accesses without modifying data.

Fences act as *memory barriers*, a line in the code that certain operations can't cross.

Let X be either a release fence or an atomic release operation in thread A. Let Y be either an acquire fence or an atomic acquire operation in thread B.

- If X synchronizes with Y, all non-atomic and relaxed atomic stores that are sequenced before X in thread A will happen-before all non-atomic and relaxed atomic loads from the same locations made in thread B after Y.
- The requirements for X to synchronize-with Y is to have: an atomic write operation M that is X or sequenced-after X in thread A whose value is read by an atomic operation N that is Y or sequenced-before Y in thread B.

Example for fence-fence: FA happens-before FB iff:

```
t1: W1, FA, atomic write X to M,
t2:                              atomic read Y of M (reads X), FB, R2
```

and it results in R2 reading all W1

Concurrent Data Structures

Using Locks

Locking is the easiest way to build thread-safe data structures.

- Ensure threads never see a state where invariants of data structure are broken.
- Minimize opportunities for deadlock → restrict scope of locks, avoid nested locks
- (Opt) Minimize amount of serialization → lock at an appropriate granularity
- To increase *opportunity fo concurrency*, use smaller protected region.
  - If 1 thread is accessing the DS from a particular fn, which other fns are safe to call from other thread?
  - Can some parts of operation be performed outside of lock?
  - Use different mutex to protect different part of DS?
- Ctors and dtors require exclusive access to the data. Pay attention to assignment, swap or copy construction operations

Lock-Free

CAS loop (compare and swap / compare and set)

```cpp
std::atomic<int> my_int{0};
int old_value = my_int;
while (true) {
        int new_value = old_value + 5;
        if (my_int.compare_exchange_weak(old_value, new_value)) break;
        // CAS loop failed, old_value is updated to the current value
}
```

CAS loop might suffers from **ABA problem.** (Not all CAS loop suffers this problem though, e.g. `fetch_add` implementation above only cares that it does a `+5` to previous value)

A solution to ABA problem is to use an (increasing) generation counter. This make sure that the content really haven't changed. See CS3211 > ^c1697e.

Testing and Debugging

Bugs

- Unwanted blocking: deadlock, livelock, blocking on I/O or other external input
- Data races: UB due to unsychronized concurrent access to a shared memory location
- Broken Invariants
  - dangling pointers: another thread deleted data being accessed

    - random memory corruption: thread reading inconsistent values resulting from partial updates
    - double free: 2 threads pop the same value from a queue, so both delete same associated data
- Lifetime issues
    - Thread outlives the data that it accesses (use-after-free)
    - Example: referencing local variables that go out of scope before thread completed.

Techniques

Techniques to find bug: Review the code or Testing

Example questions to use during code review:

- Which data needs to be protected from concurrent access and how do you ensure the data is protected?
- Where in the code could other threads be at this time?
- Which mutexes does this thread hold, and 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 these requirements enforced?
- Is the data loaded by this thread still valid? Could it have been modified by other threads?

Guidelines for testing:

- Run the smallest amount of code that could potentially demonstrate a problem
    - Test various scenarios e.g. multiple threads calling push()/pop() on an empty/full queue.
    - Test environments: multiple threads (3 or 4 or 1024), enough processing cores?, which processor architecture? suitable scheduling?
- Eliminate the concurrency from the test to verify that the problem is indeed concurrency-related
- Design for testability
    - Functions are short, closed-operation, clear responsibility
    - Eliminate concurrency - divide into parts that operate within a single thread, and parts responsible for communication paths between threads.
    - Watch out for lib calls that use internal variables to store state

Techniques for multithreaded testing

- Stress testing
    - Run the code many times, with many threads running at once
    - Write tests that maximize problematic circumstances
    - Use multiple hardware / architecture to run tests
    - Cover all code paths
- Special implementation of synchronization primitives library
    - Mark shared data in some way and allow the library to check that operations on a particular shared data are done with a particular mutex locked.
    - Record the sequence of locks if more than 1 mutex is held by a particular thread at once

    - Give priorities to threads to acquire a resource
- Test for performance: scalability - code get expected speedup when running with increasing number of cores. Usually not happening due to contention in accessing shared data.

Debugging Tools: help in identifying (concurrency) bugs. It needs to track the state of the memory and compute *happens-before* to find data races.

Debugging

- Valgrind: Dynamic Binary Instrumentation
    - Memcheck, a memory checker tool: use **shadow memory**, tools that shadow every byte of memory used by a program with another value in software
        - A ("Addresability") bits: 1 indicates if an addressable memory byte. detect heap buffer overflows, wild reads and writes.
        - V ("Validity") bits: 0 indicates a defined bit. detect dangerous use of undefined values
        - Heap blocks: records the location of every live heap block. detect repeated frees, memory leaks
    - Helgrind detects
        - misuses of POSIX pthreads API: intercepts calls to functions and instruments them.
        - Potential deadlocks arising from lock ordering problems: build a directed graph indicating order in which locks have been acquired.
        - Data races: Build a DAG representing the collective happens-before dependencies, and monitors all memory accesses for illegal order using a set of rules.
- Sanitizer: compilation-based approach
    - e.g. `-fsanitize=address`, ThreadSanitizer (e.g. data races), MemorySanitizer (e.g. uninitialized reads), UndefinedBehaviorSanitizer (e.g. Integer Overflow, Null Pointer), Leak Sanitizer (e.g. memory leaks)
    - Address Sanitizer: make use of **shadow byte**. Each aligned 8 bytes can have exactly 9 states → state stored in shadow byte.
    - Thread Sanitizer: runtime library
        - malloc replacement, intercepts all synchronization, reads, writes
        - **Shadow cell**: an 8-byte to represent 1 memory access (16bits: threadId, 42bits: epoch, 5 bits: position/size in 8-byte word, 1 bit: isWrite)
        - Store stack trace for previous access:
            - each thread has a cyclic buffer containing 64-bit (type + PC) events (memory access, function entry/exit)
            - Events can be replayed ot be shown on report
        - Detects normal data races; use-after-free; races on mutexes, file descriptors, barrier; leaked threads; destruction of locked mutex; potential deadlocks, etc.
- `perf c2c`: debug false sharing

Message Passing Paradigm

Task Dependency Graph

- A directed acyclic graph where the node represents tasks and edge represents control dependency between tasks.
- Properties: Critical path length (maximum completion time), degree of concurrency (Total work / critical path length)

Challenges:

- Finding enough concurrency
  - Speedup: $S_p(n) = \frac{T_{\text{best\_seq}}(n)}{T_p(n)}$.
  - Amdahl's law: $S_p(n) = \frac{1}{f+(1-f)/p}$ where f is the sequential fraction, a.k.a. fixed-workload performance.
- Granularity of tasks
- Locality
- Coordination and synchronization

Golang

Go model is based on Communicating Sequential Processes (CSP)

- Concurrency (tasks / structures): break program into pieces that can be executed independently, i.e. goroutines.
- Communication (dependencies) : coordinate between the independent executions (channels and select statements).

Goroutines

- functions running **independently** (detached mode) in a shared address space. It is cheaper than threads (overhead: 3 cheap instructions, given a few kilobytes), and follow the fork-join model.
- multiplexed onto OS threads by the Go runtime via an automatic scheduling. This decouples concurrency (program structure) from parallelism (actual execution).
- A special class of coroutine: (1) when a goroutine blocks, that thread blocks but other goroutine continues, (2) preemptable.
- Are not garbage collected → should prevent goroutine leaks.

Channels

- Channels are reference to a place in memory where the channel resides.
- Similar to pipe `|` operator in shell. Values passed along and then read out downstream. No knowledge is required about other parts of the program that work with the channel.
- It is ok to keep a channel open, if you drop all references, it would eventually be garbage collected.
  - Principle: never close channel from the receiver side and don't close channel if it has multiple concurrent senders from go - Is it OK to leave a channel open? - Stack

Overflow

- Behaviour
  - Reading or writing to `nil` channels blocks. Closing a `nil` channel panics
  - Reading from a closed channel will return its default value. Writing or closing a closed channel panics.
- Using unidirectional channels signals to programmers how the channel is intended to be used.

Select

- All channel reads and writes (case statements) are considered simultaneously to see if any of them are ready. If none is ready, it blocks. If multiple channel is ready, Go will randomly (with almost uniform probability) select one of the channels
- Can have a `default` case to prevent blocking (is usually used to send a signal to stop reading and cleanup goroutine)
  - Alternatively we can use `context` package. `<-ctx.Done()`
  - `context.WithTimeout` will overwrite the timeout declared in the parent context.

sync

A `singleflight.Group` remembers the function invoked for the same key. Any concurrent call to the same-keyed function will not invoke the function; instead, it will wait for the first function invocation of the same key to return.

```
var reqGp singleflight.Group
cat, err, _ := reqGp.Do("cat", db.ReadCat)
```

Memory Model

Happens Before

- Within a single goroutine, reads and writes must behave as if they are executed in program order
- Execution order observed by different goroutines might differ.
- To guarantee that a read r of a variable v observes a particular write w to v, ensure that (1) w happens before r, (2) any other write to shared variable v either happens before w or after r.

  Synchronization

- The `go` statement that starts a new goroutine happens-before the goroutine's execution begins
- The exit of a goroutine is not guaranteed to happen before any event in the program.
- A send on a channel happens-before the corresponding receive from the channel completes

- A receive from an unbuffered channel happens before the corresponding send on the channel completes
- The k-th receive on a channel with capacity C happens before the $(k+C)$-th send to the channel completes.
- The closing of a channel happens-before a receive that returns a zero-value because the channel is closed

Patterns

(Lexical) Confinement: Restricting access to shared locations

- Expose only the reading/writing handle of the channel
- Expose only a slice of array

For-select loop

```
for {
        select {
        case <-done:
                // do something
        default:
        }
        // do some work
}
```

- Keep the `select` statement as short as possible
- Prevent goroutines from leaking by ensuring its termination

Error Handling

- Couple potential result with potential error

Pipelines

```
goroutine A ==> (send data through channel) goroutine B ==> goroutine C
```

- Separate concerns of each stage (independently modified and combined)
  - enable us to use more goroutines for portions of the pipeline.
- In designing, we should aim for roughly similar processing time / stage
- Pipeline is better than task pool when there is a cap on a specific resource that can be obtained

Fan-out, Fan-in

- Stages in a pipeline might be slower than the other, and might benefit from parallelism, e.g. computationally intensive work
- Fan out: start multiple goroutines to handle input from pipeline
  - We should do this if it doesn't rely on values that the stage had calculated before

    - No guarantee on the order concurrent copies run, nor in what order they return
- Fan-In: combine multiple streams of data into 1 channel.
    - Spin up 1 goroutine for each incoming channel with task of transferring data from its stream into the multiplexed stream.

Rust

Rust's key innovations are : compile-time enforced ownership of values and shared xor mutable. These prevents use-after-frees (via borrowing and lifetimes), double-frees (via ownership), dangling pointers, data races (not allowing sharing and mutation) etc.

- By default, variables in Rust are moved. Deep copy of data needs to be made explicit using `clone`. Variables are destroyed upon (owner's) scope exit.
- Borrowing
    - Shared borrow: can only read data, shared.
    - Mutable borrow: exclusive, can modify data.
- Concurrency support in Rust is library-based, e.g. `std::thread`
- Traits: an interface that you can implement.
    - Send : safe to send between threads, e.g. `String, u32, Arc<String>`
    - Copy : safe to memcpy, e.g. `u32, f32` but not for Strings
    - Clone
- Atomically Reference Counter `Arc`: similar to C++'s `std::shared_ptr`
    - Only allows shared references
    - Allocates data on the heap, track the reference count internally, and only free the allocation when there are no more referrers.

```
use std::thread;
use std::sync::{Mutex, Arc};

fn main() {
  let counter = Arc::new(Mutex::new(0));

  let t0 = {
    let counter = counter.clone();
    thread::spawn(move || {
      *counter.lock().unwrap() += 1;
    })
  };

  let t1 = {
    let counter = counter.clone();
    thread::spawn(move || {
      *counter.lock().unwrap() += 1;
    })
  };

  t0.join().unwrap();
  t1.join().unwrap();
```

```
    println!("{}", *counter.lock().unwrap());
}
```

Synchronization in Rust

- Shared memory
  - Mutex is based on ownership. The data is protected by the mutex. Locking returns a guard through which we can access the data.
  - Atomics is similar to modern C++ with all its memory ordering.
- Message-passing: mpsc channel (multi-producer, single-consumer FIFO queue)

```
fn sync_inc(counter: &Mutex<i32>) {
        let mut data: MutexGuard<i32> = counter.lock().unwrap();
        *data += 1;
}
```

Interior Mutability

Note that the `counter` in the example above is a shared borrow. Despite that, we are able to get a mutable reference to the data within mutex using `lock()`. This pattern is called *interior mutability*. It is internally implemented using unsafe code.

> Users of libraries (like us) can generally rely on the fact that as long as we do not have `unsafe` directly in our code, we are safe from data races and memory bugs.

This is useful for global variables (static items) that are immutable and must implement the `Sync` trait. Other types that exhibit interior mutability in Rust are `Cell, RefCell`

```
let (tx, rx) = std::sync::mpsc::channel();
let tx2 = tx.clone();
std::thread::spawn(move || tx.send(5));
// thread::spawn(move || tx.clone().send(4)) don't work as tx is moved.
std::thread::spawn(move || tx2.send(4));

// prints 4 and 5 in an unspecified order
println!("{:?}", rx.recv());
println!("{:?}", rx.recv());
```

Asynchronous Programming

Closures

- can be called like a function, but are not just simple functions
- can be passed as parameters, returned to and from functions.

Motivation: Overhead of threads

- Context switching cost: threads give up the rest of the CPU time slice when blocking functions are called → context switch happens: register restored, virtual address space switched, cache stepped on.
- Memory overhead: Each thread has its own stack space that needs to get managed by the OS.

Non-Blocking I/O

- Example of blocking syscall: `read()` . Instead of blocking, would it be possible to return a special error value and do other useful work, e.g. reading from other descriptors?
- `epoll` is a kernel-provided mechanism that notifies us of what file descriptors are ready for I/O, some kind of event loop.
- Key problem: manage the state associated with each task.

Futures

- Allows us to keep track of in-progress operations along with associated state.
- Represents a value that will exist sometime in the future
- Calculation hasn't happened yet, or is probably ongoing, we can just keep asking until it is ready.

For this, Rust provides us the `Future` trait (a stripped-down version shown below)

```
trait Future {
        type Output;
        fn poll(&mut self, ctx: &mut Context) -> Poll<Self::Output>;
}
enum Poll<T> {
        Ready(T),
        Pending,
}
```

- Rust futures are **poll-based**. The executor thread should call `poll()` on the future to start it (otherwise no computation happen). `Future` is the computation.
- The executor will then run code until it can no longer progress. If the future is complete, it will return `Poll::Ready(T)` , otherwise it returns `Poll::Pending`
- The `Context` structure includes a `wake()` function that will be called when the future can make progress again.
- Executor loops over futures that can currently make progress.
  - Calls `poll()` on them to give them attention until they need to wait again
  - When no futures can make progress, the executor can sleep until 1 or more future calls `wake()`
- Note that code within a future should not block. Otherwise, the executor running that code is going to sleep.
- Asynchronous code needs to use non-blocking version of everything, including mutexes, blocking syscalls, etc.

Async/Await

To increase ergonomics, Rust introduce syntactic sugar: `async` and `await` which will transformed into the same code as Future with `poll()`

- An `async` function is a function that returns a Future.
- `.await` waits for a future and get its value. It can only be called on an `async` function or block.

Implications:

- Async functions have no stack (stackless coroutines)
  - The executor thread still has a stack (used to run normal/synchronous functions), but it is not used to store state when switching between async tasks.
  - All state is self contained in the generated Future
- No recursion: The Future returned by an async function must have compile-time-known size.
- Rust async functions are nearly optimal in terms of memory usage and allocations.
- Async functions makes sense when (1) you need an extremely high degree of concurrency and (2) work is primarily I/O bound, e.g. server; otherwise the executor might be prevented from running other tasks.

Libraries

Rayon

- A drop-in data parallelism library, similar in functionality with OpenMP.
- Can be as simple as changing `iter()` to `par_iter()`.
- Implement's its version of iterator adapter, e.g. `map, filter, for_each` and terminal, e.g. `sum`

```
use rayon::prelude::*;

fn magic_sum(from: u128, to: u128) -> u128 {
  (from..to).into_par_iter().filter(|i| i % 7 == i % 5).sum() // change here
}

fn main() {
  let (from, to) = {
    let mut args = std::env::args();
    args.next(); // skip argv[0]
    (args.next().unwrap(), args.next().unwrap())
  };
  println!("{}", magic_sum(from.parse().unwrap(), to.parse().unwrap()));
}
```

Crossbeam

- Ability to create "scoped" threads
  - Scope is like a little container to put our threads in
  - You cannot borrow variables mutably into 2 threads in the same scope

    - `crossbeam::thread::scope` will join all threads created in its scope before it returns → allowing threads to capture references to values owned by `main` that don't live for `'static`.
- Provides message passing paradigm of multiple-producer-mutiple-consumer (MPMC) channel with exponential backoff.

```rust
use std::sync::atomic::{AtomicI32, Ordering};

fn main() {
  let counter = AtomicI32::new(0);

  crossbeam::thread::scope(|s| {
    s.spawn(|_| { counter.fetch_add(1, Ordering::Relaxed); });
    s.spawn(|_| { counter.fetch_add(1, Ordering::Relaxed); });
  }).unwrap();

  println!("{}", *counter.lock().unwrap());
}
```

Tokio

Tokio is one of the popular async runtime in Rust-land.

```rust
use tokio::io::{AsyncReadExt, AsyncWriteExt};
use tokio::net::TcpListener;

#[tokio::main] // this macro submits Future to the executor
async fn main() {
        let listener = TcpListener::bind("127.0.0.1:8000").await.unwrap();
        loop {
                let (socket, _) = listener.accept().await?;
                tokio::spawn(async move { // move moves socket.
                        let mut buf = [0; 1024];
                        let n = socket.read(&mut buf).await.unwrap();
                        socket.write_all(&buf[0..n]).await.unwrap();
                });
        }
}
```

Concurrency Problems

H2O

```
struct WaterFactory3 {
  std::counting_semaphore<> oxygenSem;
  std::counting_semaphore<> hydrogenSem;
  std::barrier<> barrier;
```

```
  WaterFactory3() : oxygenSem{1}, hydrogenSem{2}, barrier{3} {}

  void oxygen(void (*bond)()) {
    oxygenSem.acquire();  // Lets at most one oxygen through
    barrier.arrive_and_wait();
    bond();
    oxygenSem.release();  // We are done, let the next oxygen in
  }

  void hydrogen(void (*bond)()) {
    hydrogenSem.acquire();  // Lets at most two hydrogen through
    barrier.arrive_and_wait();
    bond();
    hydrogenSem.release();  // We are done, let the next hydrogen in
  }
};
```

```
type WaterFactoryWithLeader struct {
        oxygenMutex chan struct{}
        precomH     chan chan struct{}
        commit      chan chan struct{}
}

func NewFactoryWithLeader() WaterFactoryWithLeader {
        wf := WaterFactoryWithLeader{
                oxygenMutex: make(chan struct{}, 1),
                precomH:     make(chan chan struct{}),
                commit:      make(chan chan struct{}),
        }
        wf.oxygenMutex <- struct{}{}
        return wf
}

func (wf *WaterFactoryWithLeader) hydrogen(bond func()) {
        commit := make(chan struct{}) // Step 1: Create private
communication channel
        wf.precomH <- commit          // Step 2: (Precommit)
        <-commit                      // Step 3: (Commit)
        bond()                        // Step 4: Bond
        commit <- struct{}{}          // Step 5: (Postcommit)
}

func (wf *WaterFactoryWithLeader) oxygen(bond func()) {
        // Step 1: Become leader
        <-wf.oxygenMutex // For fun, we can use a channel as a mutex

        // Step 2: (Precommit) Receive arrival requets from 2 hydrogen atoms
        h1 := <-wf.precomH
        h2 := <-wf.precomH

        // Step 3: (Commit) Tell the 2 hydrogen atoms to start bonding
        h1 <- struct{}{}
```

```
        h2 <- struct{}{}

        // Step 4: Bond
        bond()

        // Step 5: (Postcommit) Wait until the 2 hydrogen atoms to finish
        // We re-use the same communication channel as (Commit)
        <-h1
        <-h2

        // Step 6: Step down from being leader
        wf.oxygenMutex <- struct{}{}
}
```

```
use std::sync::Arc;

use futures::future::join_all;
use tokio::sync::{mpsc, Barrier, Mutex, Semaphore};


async fn hydrogen(id: usize, barrier: Arc<Barrier>,
        sem: Arc<Semaphore>, chan: mpsc::Sender<usize>) {
    let _permit = sem.acquire().await.unwrap();
    barrier.wait().await;

    chan.send(id).await.unwrap();
}

async fn oxygen(id: usize, barrier: Arc<Barrier>, chan:
Arc<Mutex<mpsc::Receiver<usize>>>) {
    let mut chan_guard = chan.lock().await;
    barrier.wait().await;

    let h1 = chan_guard.recv().await.unwrap();
    let h2 = chan_guard.recv().await.unwrap();
    println!("H {} - O {} - H {}", h1, id, h2);
}

#[tokio::main]
async fn main() {
    let barrier = Arc::new(Barrier::new(3));
    let h_sem = Arc::new(Semaphore::new(2));

    let (s, r) = mpsc::channel(2);
    let r = Arc::new(Mutex::new(r));

    let hydrogens =
        (0..200).map(|i| tokio::spawn(hydrogen(i, barrier.clone(),
h_sem.clone(), s.clone())));
    let oxygens = (0..100).map(|i| tokio::spawn(oxygen(i, barrier.clone(),
r.clone())));

    let join_handles = Iterator::chain(hydrogens, oxygens).collect::<Vec<_>>
```

```
();

    std::mem::drop(s);
    std::mem::drop(r);

    join_all(join_handles).await;
}
```

FIFO Semaphore

We can implement a queue of waiters by using a ticket queue (e.g. like in a clinic). Arrivals obtain a queue number from the receptionist, and when they're ready to be released, the "now serving" number is incremented to their queue number or higher.

```
struct FIFOSemaphore2 {
  alignas(128) std::atomic<std::ptrdiff_t> next_ticket;
  alignas(128) std::atomic<std::ptrdiff_t> now_serving;

  // Initialise with `initial_count` requests "pre-served"
  FIFOSemaphore2(std::ptrdiff_t initial_count)
      : next_ticket{1}, now_serving{initial_count} {}

  void acquire() {
    // When a thread arrives, get a ticket
    std::ptrdiff_t my_ticket =
        next_ticket.fetch_add(1, std::memory_order_relaxed);
    // Wait until the now serving number reaches or passes us
    while (now_serving.load(std::memory_order_acquire) < my_ticket) {}
  }

  void release() {
    // Increment the now serving number
    now_serving.fetch_add(1, std::memory_order_acq_rel);
  }
};
```

Using a queue of semaphores

```
struct FIFOSemaphore5 {
  struct Waiter {
    std::binary_semaphore sem{0};
  };

  std::mutex mut;
  std::queue<std::shared_ptr<Waiter>> waiters;
  std::ptrdiff_t count;

  FIFOSemaphore5(std::ptrdiff_t initial_count)
      : mut{}, waiters{}, count{initial_count} {}
```

```
  void acquire() {
    auto waiter = std::make_shared<Waiter>();
    {
      std::scoped_lock lock{mut};
      if (count > 0) {
        count--;  // Positive count,
        return;   // simply decrement without blocking
      }
      waiters.push(waiter);  // Zero count, add to waiters
    }
    waiter->sem.acquire();  // and block on the semaphore
  }

  void release() {
    std::shared_ptr<Waiter> waiter;
    {
      std::scoped_lock lock{mut};
      if (waiters.empty()) {
        count++;  // No waiters, simply increment count
        return;
      }

      waiter = waiters.front();  // Pop a waiter
      waiters.pop();
    }
    waiter->sem.release();  // and signal it
  }
};
```

Go - daemon goroutine

```
type Semaphore2 struct {
        acquireCh chan chan struct{}
        releaseCh chan struct{}
}

func NewSemaphore2(initial_count int) *Semaphore2 {
        sem := new(Semaphore2)
        sem.acquireCh = make(chan chan struct{}, 100)
        sem.releaseCh = make(chan struct{}, 100)

        go func() {
                count := initial_count
                // The FIFO queue that stores the channels used to unblock
waiters
                waiters := NewChanQueue()

                for {
                        select {
                        case <-sem.releaseCh: // Increment or unblock a
waiter
                                if waiters.Len() > 0 {
                                        ch := waiters.Pop()
```

```
                                        ch <- struct{}{} // Unblocks the
oldest waiter
                                } else {
                                        count++
                                }

                        case ch := <-sem.acquireCh: // Decrement or add a
waiter
                                if count > 0 {
                                        count--
                                        ch <- struct{}{} // Don't keep
waiter blocked
                                } else {
                                        waiters.PushBack(ch) // Add waiter
to back of queue
                                }
                        }
                }
        }()

        return sem
}

func (s *Semaphore2) Acquire() {
        ch := make(chan struct{})
        // Send daemon a channel that can be used to unblock us
        s.acquireCh <- ch
        // Block until daemon decides to unblock us
        <-ch
}

func (s *Semaphore2) Release() {
        s.releaseCh <- struct{}{}
}
```

Appendix

Simple Shared Pointer

```
template <typename T>
class simple_shared_ptr {
  T* ptr;
  std::atomic<std::size_t>* count;

 public:
  simple_shared_ptr(T* ptr) : ptr{ptr}, count{new std::atomic<size_t>{1}} {
    std::cout << "Ptr ctor" << std::endl;
  }

  simple_shared_ptr(const simple_shared_ptr& other)
```

```
      : ptr{other.ptr}, count{other.count} {
    std::cout << "Copy ctor" << std::endl;
    count->fetch_add(1, std::memory_order_relaxed);
  }

  // Prevent simple_shared_ptr from being mutated after construction
  simple_shared_ptr& operator=(const simple_shared_ptr& other) = delete;

  T* get() {
    std::cout << "get" << std::endl;
    return ptr;
  }

  ~simple_shared_ptr() {
    std::cout << "Dtor" << std::endl;
    if(count->fetch_sub(1, std::memory_order_acq_rel) == 1) {
      delete ptr;
      delete count;
    }
  }
};
```

Lock-Free Multi-Producer Multi-Consumer Queue

```
struct Job {
  int id;
  int data;
};

// Unbounded lock free queue with enqueue and non-blocking dequeue
// Attempts to adapt fine grained locking design (JobQueue7)
//
// Avoids ABA using generation counter
// Avoids UAF using node recycling
//
// Note that now that nodes can be reused, we can actually have a race
// where the write happens after the read (so the read sees a FUTURE value)
//
// To fix this, we need synchronization between consumer to producer, not
// just producer to consumer
class JobQueue11 {
  using stdmo = std::memory_order;
  struct Node;
  // A node is a dummy node if its next pointer is set to QUEUE_END
  // We use the next ptr to establish the synchronizes-with relationship
  // next is in charge of "releasing" job
  struct Node {
    std::atomic<Node *> next = QUEUE_END;
    Job job;
  };

  static inline Node *const QUEUE_END = nullptr;
  static inline Node *const STACK_END = QUEUE_END + 1;
```

```
  struct GenNodePtr {
    Node *node;
    std::uintptr_t gen;
  };

  alignas(64) std::atomic<Node *> jobs_back;       // producer end
  alignas(64) std::atomic<GenNodePtr> jobs_front;  // consumer end
  alignas(64) std::atomic<GenNodePtr> stack_top;   // recycling centre

public:
  // Queue starts with a dummy node
  JobQueue11()
      : jobs_back{new Node{}},
        jobs_front{GenNodePtr{jobs_back.load(stdmo::relaxed), 1}},
        stack_top{GenNodePtr{STACK_END, 1}} {}
  ~JobQueue11() {
    // Assumption: no other threads are accessing the job queue
    Node *cur_queue = jobs_front.load(stdmo::relaxed).node;
    while (cur_queue != QUEUE_END) {
      Node *next = cur_queue->next;
      delete cur_queue;
      cur_queue = next;
    }

    Node *cur_stack = stack_top.load(stdmo::relaxed).node;
    while (cur_stack != STACK_END) {
      Node *next = cur_stack->next;
      delete cur_stack;
      cur_stack = next;
    }
  }

private:
  // Get node from recycling centre if possible,
  // else allocate a new one using `new`
  //
  // deallocate_node needs to synchronize with this function
  //
  // We'll synchronize with the corresponding deallocate_node using the
  // next pointer in the node that we get from the stack.
  Node *allocate_node() {
    // Try to splice a node from the stack, and return it
    //
    // Standard CAS loop with generation counter to avoid ABA.
    GenNodePtr cur_stack = stack_top.load(stdmo::relaxed);

    while (true) {
      if (cur_stack.node == STACK_END) {
        // If the recycling centre is empty, we'll allocate a new node
        return new Node{};
      }
      Node *cur_stack_next = cur_stack.node->next.load(stdmo::acquire);
      GenNodePtr new_stack{cur_stack_next, cur_stack.gen + 1};
      if (stack_top.compare_exchange_weak(cur_stack, new_stack,
```

```
stdmo::relaxed)) {
        // Successfully spliced out a node from recycling centre
        return cur_stack.node;
      }
    }
  }

  // Put node in recycling centre
  // This function needs to synchronize with allocate_node
  //
  // The corresponding allocate_node will synchronize with us using the
  // next pointer in the node that we put in the stack.
  void deallocate_node(Node *node) {
    // Splice the node into the stack
    // Standard CAS loop with generation counter to avoid ABA.
    GenNodePtr cur_stack = stack_top.load(stdmo::relaxed);

    while (true) {
      node->next.store(cur_stack.node, stdmo::release);
      GenNodePtr new_stack{node, cur_stack.gen + 1};
      if (stack_top.compare_exchange_weak(cur_stack, new_stack,
stdmo::relaxed)) {
        break;
      }
    }
  }

public:

  void enqueue(Job job) {
    Node *new_node = allocate_node();

    // Since we might be getting a recycled node, we reset the next
    // pointer so it'll be a dummy node when we append it to the LL.
    new_node->next.store(QUEUE_END, stdmo::relaxed);

    // Use jobs_back.exchange to establish a global order of all enqueues
    //
    // We use acq_rel because:
    // - Release contents of `new_node`
    // - Similarly acquire contents of `old_dummy`
    // We need this because the contents may have been `new`d in a
    // different thread, or it's a recycled node and we need to wait for
    // pending reads to finish.
    //
    // So this is how we "share" the guarantee in the comment above with
    // other producers.
    Node *old_dummy = jobs_back.exchange(new_node, stdmo::acq_rel);

    // old_dummy is unique for every enqueue
    old_dummy->job = job;

    // Now "release" the job to consumers,
    // and also append to LL at the same time
    old_dummy->next.store(new_node, stdmo::release);
```

```
  }

  std::optional<Job> try_dequeue() {
    // Splice node from the front of queue, but only if it's not dummy
    // Successfully splicing a node establishes global order of dequeues

    // Relaxed ordering because we don't need to synchronize with other
    // consumers, we need to synchronize with the producer that made the
    // node, which is done via the `next` ptr.
    GenNodePtr old_front = jobs_front.load(stdmo::relaxed);

    while (true) {
      // "Acquire" job if it exists
      // Also use next pointer to know what to update jobs_front to
      Node *old_front_next = old_front.node->next.load(stdmo::acquire);

      if (old_front_next == QUEUE_END) {
        // Observed dummy node, so we can abort as the queue is empty
        return std::nullopt;
      }

      GenNodePtr new_front{old_front_next, old_front.gen + 1};

      // Node is not dummy, so we try to update jobs_front
      // Relaxed because we don't need to acquire on failure, nor release
      // on success, as we don't need to synchronize with other consumers.
      if (jobs_front.compare_exchange_weak(old_front, new_front,
stdmo::relaxed)) {
        // Node now belongs to us
        break;
      }
      // We couldn't update jobs_front, so someone else successfully
      // dequeued a node. We'll just loop.
    }
    Job job = old_front.node->job;
    deallocate_node(old_front.node);
    return job;
  }
};
```

Go - Context

```
func handleSigs() chan struct{} {
        done := make(chan struct{})
        go func() {
                sigs := make(chan os.Signal, 1)
                signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM)
                <-sigs
                close(done)
        }()
        return done
}
```

```
//ctx (timeout at 4s)
// |
// |---> ctx3 (cancellable, cancellation don't affect ctx/ctx2)
// |
// v
//ctx2 (timeout at 2s)
func main() {
        startTime := time.Now()

        ctx, _ := context.WithTimeout(context.Background(), 4 * time.Second)

        ctx2, _ := context.WithTimeout(ctx, 2 * time.Second)
        go func() {
                <-ctx2.Done()
                fmt.Printf("ctx2 done at %v\n", time.Now().Sub(startTime))
        }()

        ctx3, cancel := context.WithCancel(ctx)
        go func() {
                <-ctx3.Done()
                fmt.Printf("ctx3 done at %v\n", time.Now().Sub(startTime))
        }()

        go func() {
                <-handleSigs()
                cancel()
                fmt.Printf("signal in at %v\n", time.Now().Sub(startTime))
        }()

        <-ctx.Done()

        fmt.Printf("ctx %v\n", time.Now().Sub(startTime))
}
```

Rust - Quick Sort

```
fn quick_sort<T:PartialOrd+Send>(v: &mut [T]) {
    if v.len() > 1 {
        let mid = partition(v);
        let (lo, hi) = v.split_at_mut(mid);
        rayon::join(|| quick_sort(lo),
                    || quick_sort(hi));
    }
}
```

Rust - Mio

```
// adapted from https://github.com/tokio-
rs/mio/blob/master/examples/tcp_server.rs

use mio::event::Event;
```

```rust
use mio::net::{TcpListener, TcpStream};
use mio::{Events, Interest, Poll, Registry, Token};

use std::collections::HashMap;
use std::io::{BufRead, BufReader, ErrorKind, Write};
use std::net::SocketAddr;

const SERVER: Token = Token(0);

struct ClientState {
    reader: BufReader<TcpStream>,
    line_buffer: Vec<u8>,
    written_until: usize,
    read_complete: bool,
}

fn main() -> std::io::Result<()> {
    let mut poll = Poll::new()?;
    let mut events = Events::with_capacity(128);

    // Setup the TCP server socket.
    let port = std::env::args()
        .nth(1)
        .map(|s| s.parse().unwrap())
        .unwrap_or(50000u16);
    let mut listener = TcpListener::bind(SocketAddr::from(([127, 0, 0, 1],
port)))?;

    // Register the server with poll we can receive events for it.
    poll.registry()
        .register(&mut listener, SERVER, Interest::READABLE)?;

    // Map of `Token` -> `ClientState`.
    let mut connections = HashMap::new();
    // Unique token for each incoming connection.
    let mut next_token = Token(SERVER.0 + 1);

    loop {
        poll.poll(&mut events, None)?;

        for event in events.iter() {
            match event.token() {
                SERVER => loop {
                    // Received an event for the TCP server socket, which
                    // indicates we can accept an connection.
                    let (mut connection, address) = match listener.accept()
{
                        Ok((connection, address)) => (connection, address),
                        Err(e) if e.kind() == ErrorKind::WouldBlock => {
                            // If we get a `WouldBlock` error we know our
                            // listener has no more incoming connections
queued,
                            // so we can return to polling and wait for some
                            // more.
                            break;
```

```
                        }
                        Err(e) => {
                            // If it was any other kind of error, something
went
                            // wrong and we terminate with an error.
                            return Err(e);
                        }
                    };

                    println!("Accepted connection from: {}", address);
                    let token = next_token;
                    next_token.0 += 1;
                    poll.registry()
                        .register(&mut connection, token,
Interest::READABLE)?;
                    connections.insert(
                        token,
                        ClientState {
                            reader: BufReader::new(connection),
                            line_buffer: Vec::new(),
                            written_until: 0,
                            read_complete: false,
                        },
                    );
                },
                token => {
                    // Maybe received an event for a TCP connection.
                    let done = if let Some(connection) =
connections.get_mut(&token){
                        handle_connection_event(poll.registry(), connection,
event)?
                    } else {
                        // Sporadic events happen, we can safely ignore
them.
                        false
                    };
                    if done {
                        if let Some(mut connection) =
connections.remove(&token) {

poll.registry().deregister(connection.reader.get_mut())?;
                        }
                    }
                }
            }
        }
    }
}

/// Returns `true` if the connection is done.
fn handle_connection_event(
    registry: &Registry,
    connection: &mut ClientState,
    event: &Event,
) -> std::io::Result<bool> {
```

```
    if connection.read_complete && event.is_writable() {
        // We can (maybe) write to the connection.
        match connection
            .reader
            .get_mut()
            .write(&connection.line_buffer[connection.written_until..])
        {
            Ok(n) => {
                connection.written_until += n;
                if connection.written_until >= connection.line_buffer.len()
{
                    // After we've written a full line, reset the buffer and
go back to reading.
                    connection.line_buffer.clear();
                    connection.written_until = 0;
                    connection.read_complete = false;
                    registry.reregister(
                        connection.reader.get_mut(),
                        event.token(),
                        Interest::READABLE,
                    )?;
                }
            }
            // Would block "errors" are the OS's way of saying that the
            // connection is not actually ready to perform this I/O
operation.
            Err(ref err) if err.kind() == ErrorKind::WouldBlock => {}
            // Got interrupted (how rude!), we'll try again.
            Err(ref err) if err.kind() == ErrorKind::Interrupted => {
                return handle_connection_event(registry, connection, event)
            }
            // Other errors we'll consider fatal.
            Err(err) => return Err(err),
        }
    }
    if !connection.read_complete && event.is_readable() {
        // We can (maybe) read from the connection.
        match connection
            .reader
            .read_until(0xA, &mut connection.line_buffer)
        {
            Ok(0) => {
                // Reading 0 bytes means the other side has closed the
                // connection or is done writing, then so are we.
                println!("Connection closed");
                return Ok(true);
            }
            Ok(_) => {
                connection.read_complete = true;
                registry.reregister(
                    connection.reader.get_mut(),
                    event.token(),
                    Interest::WRITABLE,
                )?;
            }
```

```
            // Would block "errors" are the OS's way of saying that the
            // connection is not actually ready to perform this I/O
operation.
            Err(ref err) if err.kind() == ErrorKind::WouldBlock => {},
            Err(ref err) if err.kind() == ErrorKind::Interrupted => {
                return handle_connection_event(registry, connection, event)
            },
            // Other errors we'll consider fatal.
            Err(err) => return Err(err),
        }
    }

    Ok(false)
}
```

Rust - Rayon

```
use std::ops::Add;

pub trait IntoParallelRefIterator<'data> {
    type Iter: ParallelIterator<Item = Self::Item>;
    type Item: Send + 'data;
    fn par_iter(&'data self) -> Self::Iter;
}

impl<'data, T: Sync + 'data> IntoParallelRefIterator<'data> for [T] {
    type Item = &'data T;
    type Iter = SliceIter<'data, T>;

    fn par_iter(&'data self) -> Self::Iter {
        SliceIter { slice: self }
    }
}

pub struct SliceIter<'data, T: Sync> {
    slice: &'data [T],
}

fn run_consumer<'data, T: Sync, C: Consumer<&'data T>>(
    slice: &'data [T],
    consumer: C,
) -> C::Result {
    if slice.len() > 32 {
        let mid = slice.len() / 2;
        let reducer = consumer.to_reducer();
        let left = consumer.split_off_left();
        let right = consumer;
        let (left_result, right_result) = rayon_core::join(
            || run_consumer(&slice[..mid], left),
            || run_consumer(&slice[mid..], right),
        );
        reducer.reduce(left_result, right_result)
    } else {
```

```
        let mut folder = consumer.into_folder();
        for item in slice {
            folder = folder.consume(item);
        }
        folder.complete()
    }
}

impl<'data, T: Sync + 'data> ParallelIterator for SliceIter<'data, T> {
    type Item = &'data T;

    fn drive<C>(self, consumer: C) -> C::Result
    where
        C: Consumer<Self::Item>,
    {
        run_consumer(self.slice, consumer)
    }
}

pub trait Folder<Item>: Sized {
    type Result;

    fn consume(self, item: Item) -> Self;
    fn complete(self) -> Self::Result;
}

pub trait Reducer<Result> {
    fn reduce(self, left: Result, right: Result) -> Result;
}

pub trait Consumer<Item>: Send + Sized {
    type Folder: Folder<Item, Result = Self::Result>;
    type Reducer: Reducer<Self::Result>;
    type Result: Send;

    fn split_off_left(&self) -> Self;
    fn to_reducer(&self) -> Self::Reducer;
    fn into_folder(self) -> Self::Folder;
}

pub trait ParallelIterator: Sized + Send {
    type Item: Send;

    fn drive<C>(self, consumer: C) -> C::Result
    where
        C: Consumer<Self::Item>;

    fn sum(self) -> Self::Item
    where
        Self::Item: Add<Output = Self::Item> + Default,
    {
        self.drive(SumConsumer)
    }

    fn map<TMapFn, TMapResult>(self, map_op: TMapFn) -> Map<Self, TMapFn>
```

```
    where
        TMapFn: Fn(Self::Item) -> TMapResult + Sync + Send,
        TMapResult: Send,
    {
        Map { prev: self, map_op }
    }

    fn filter<TPredFn>(self, filter_op: TPredFn) -> Filter<Self, TPredFn>
    where
        TPredFn: Fn(&Self::Item) -> bool + Sync + Send,
    {
        Filter {
            prev: self,
            filter_op,
        }
    }
}

#[derive(Clone)]
pub struct Map<TPrevIter: ParallelIterator, TMapFn> {
    prev: TPrevIter,
    map_op: TMapFn,
}

impl<TPrevIter, TMapFn, TMapResult> ParallelIterator for Map<TPrevIter,
TMapFn>
where
    TPrevIter: ParallelIterator,
    TMapFn: Fn(TPrevIter::Item) -> TMapResult + Sync + Send,
    TMapResult: Send,
{
    type Item = TMapFn::Output;

    fn drive<C>(self, consumer: C) -> C::Result
    where
        C: Consumer<Self::Item>,
    {
        self.prev.drive(MapConsumer {
            next: consumer,
            map_op: &self.map_op,
        })
    }
}

struct MapConsumer<'f, TNextConsumer, TMapFn> {
    next: TNextConsumer,
    map_op: &'f TMapFn,
}

impl<'f, T, TMapResult, TNextConsumer, TMapFn> Consumer<T>
    for MapConsumer<'f, TNextConsumer, TMapFn>
where
    TNextConsumer: Consumer<TMapFn::Output>,
    TMapFn: Fn(T) -> TMapResult + Sync,
    TMapResult: Send,
```

```
{
    type Folder = MapFolder<'f, TNextConsumer::Folder, TMapFn>;
    type Reducer = TNextConsumer::Reducer;
    type Result = TNextConsumer::Result;

    fn split_off_left(&self) -> Self {
        MapConsumer {
            next: self.next.split_off_left(),
            map_op: self.map_op,
        }
    }

    fn to_reducer(&self) -> Self::Reducer {
        self.next.to_reducer()
    }

    fn into_folder(self) -> Self::Folder {
        MapFolder {
            next: self.next.into_folder(),
            map_op: self.map_op,
        }
    }
}

struct MapFolder<'f, TNextConsumer, TMapFn> {
    next: TNextConsumer,
    map_op: &'f TMapFn,
}

impl<'f, T, TMapResult, TNextFolder, TMapFn> Folder<T> for MapFolder<'f,
TNextFolder, TMapFn>
where
    TNextFolder: Folder<TMapFn::Output>,
    TMapFn: Fn(T) -> TMapResult,
{
    type Result = TNextFolder::Result;

    fn consume(self, item: T) -> Self {
        let mapped_item = (self.map_op)(item);
        MapFolder {
            next: self.next.consume(mapped_item),
            map_op: self.map_op,
        }
    }

    fn complete(self) -> TNextFolder::Result {
        self.next.complete()
    }
}

#[derive(Clone)]
pub struct Filter<TPrevIter: ParallelIterator, TPredFn> {
    prev: TPrevIter,
    filter_op: TPredFn,
}
```

```rust
impl<TPrevIter, TPredFn> ParallelIterator for Filter<TPrevIter, TPredFn>
where
    TPrevIter: ParallelIterator,
    TPredFn: Fn(&TPrevIter::Item) -> bool + Sync + Send,
{
    type Item = TPrevIter::Item;

    fn drive<C>(self, consumer: C) -> C::Result
    where
        C: Consumer<Self::Item>,
    {
        self.prev.drive(FilterConsumer {
            next: consumer,
            filter_op: &self.filter_op,
        })
    }
}

struct FilterConsumer<'p, C, P> {
    next: C,
    filter_op: &'p P,
}

impl<'p, T, TNextConsumer, TPredFn: 'p> Consumer<T> for FilterConsumer<'p,
TNextConsumer, TPredFn>
where
    TNextConsumer: Consumer<T>,
    TPredFn: Fn(&T) -> bool + Sync,
{
    type Folder = FilterFolder<'p, TNextConsumer::Folder, TPredFn>;
    type Reducer = TNextConsumer::Reducer;
    type Result = TNextConsumer::Result;

    fn split_off_left(&self) -> Self {
        FilterConsumer {
            next: self.next.split_off_left(),
            filter_op: self.filter_op,
        }
    }

    fn to_reducer(&self) -> Self::Reducer {
        self.next.to_reducer()
    }

    fn into_folder(self) -> Self::Folder {
        FilterFolder {
            next: self.next.into_folder(),
            filter_op: self.filter_op,
        }
    }
}

struct FilterFolder<'p, TNextFolder, TPredFn> {
    next: TNextFolder,
```

```
    filter_op: &'p TPredFn,
}

impl<'p, TNextConsumer, TPredFn, T> Folder<T> for FilterFolder<'p,
TNextConsumer, TPredFn>
where
    TNextConsumer: Folder<T>,
    TPredFn: Fn(&T) -> bool + 'p,
{
    type Result = TNextConsumer::Result;

    fn consume(self, item: T) -> Self {
        let filter_op = self.filter_op;
        if filter_op(&item) {
            let next = self.next.consume(item);
            FilterFolder { next, filter_op }
        } else {
            self
        }
    }

    fn complete(self) -> Self::Result {
        self.next.complete()
    }
}

struct SumConsumer;

impl<T> Consumer<T> for SumConsumer
where
    T: Send + Add<Output = T> + Default,
{
    type Folder = SumFolder<T>;
    type Reducer = Self;
    type Result = T;

    fn split_off_left(&self) -> Self {
        SumConsumer
    }

    fn to_reducer(&self) -> Self::Reducer {
        SumConsumer
    }

    fn into_folder(self) -> Self::Folder {
        SumFolder {
            sum: Default::default(),
        }
    }
}

impl<T> Reducer<T> for SumConsumer
where
    T: Send + Add<Output = T>,
{
```

```rust
    fn reduce(self, left: T, right: T) -> T {
        left + right
    }
}

struct SumFolder<T> {
    sum: T,
}

impl<T> Folder<T> for SumFolder<T>
where
    T: Add<Output = T>,
{
    type Result = T;

    fn consume(self, item: T) -> Self {
        SumFolder {
            sum: self.sum + item,
        }
    }

    fn complete(self) -> Self::Result {
        self.sum
    }
}

fn main() {
    let numbers = (0..10000).collect::<Vec<usize>>();
    let sum: usize = numbers
        .par_iter()
        .filter(|&&i| i & 1 == 1)
        .map(|i| i - 1)
        .sum();
    println!("{}", sum);
}
```