CS3211 Cheatsheet AY22/23 Sem 2

by Richard Willie

Concurrent Programming

Concurrency and Parallelism

- **Concurrency:**
  - Two or more tasks can start, run, and complete in overlapping time periods.
  - They might not be running (executing on the CPU) at the same time.
  - Two or more execution flows make *progress* at the same time by interleaving their executions or by executing instructions (on CPU) at exactly the same time
- **Parallelism:**
  - Two or more tasks can run (execute) simultaneously, at the exact same time.
  - Tasks do not only make progress, they actually execute at *simultaneously*.

Race Condition

- Criteria:
  1. Two concurrent threads access a shared resources without any synchronization.
  2. At least one thread modifies the shared resources.
- A solution to the critical-section problem must satisfy the following three requirements:
  1. **Mutual exclusion**
     - If one thread is in the critical section, than no other is.
  2. **Progress**
     - If thread T is not in the critical section, then T cannot prevent other threads from entering the critical section.
     - A thread in the critical section will eventually leave it.
  3. **Bounded waiting (no starvation)**
     - There exists a bound on the number of times that other threads are allowed to enter the critical section after a thread has made a request to enter the critical section.
     - In other words, if thread T is waiting to enter its critical section, then T will eventually enter the critical section.

Deadlock

- We say that a set of processes is in a deadlocked state when every processes in the set is waiting for an event that can be caused only by another process in the set.
- Deadlock can exist if and only if the following four conditions hold simultaneously:
  1. **Mutual exclusion** – at least one resource must be held in a non-sharable mode.
  2. **Hold and wait** – there must be one process holding one resource and waiting for another resource.
  3. **No preemption** – resources cannot be preempted, i.e. critical sections cannot be aborted externally.
  4. **Circular wait** – there must exist a set of processes $\{P_1, P_2, \ldots, P_n\}$ such that P_1 is waiting for P_2, P_2 for P_3, etc.

Starvation

- Starvation is a situation where a process is prevented from making progress because some other process has the resource it requires.
- Starvation is a side effect of the scheduling algorithm:
  - A high priority process always prevents a low priority process from running on the CPU.
  - One thread always beats another when acquiring a lock.

Blocking Algorithm

- Algorithms that use mutexes, condition variables, and futures to synchronize data are called *blocking algorithms*.
  - Note: A spinlock is non-blocking, as it spins until the `test_and_set` is successful.

Non-blocking Algorithm

- An algorithm is called *non-blocking* if failure or suspension of any thread cannot cause failure or suspension of another thread.
- Disadvantages of locks:
  - **Contention**: Some threads/processes have to wait until a lock (or a whole set of locks) is released. If one of the threads holding a lock dies, stalls, blocks, or enters an infinite loop, other threads waiting for the lock may wait indefinitely until the computer is power cycled.
  - **Priority inversion**: A low-priority thread/process holding a common lock can prevent high-priority threads/processes from proceeding.

- **Convoying**: All other threads have to wait if a thread holding a lock is descheduled due to a time-slice interrupt or page fault.
- **Composability**: It is hard to combine small, correct lock-based modules into equally correct larger programs without modifying the modules or at least knowing about their internals

- **Wait-freedom**:
  - Wait-freedom is the strongest non-blocking guarantee of progress, combining guaranteed system-wide throughput with starvation-freedom.
  - An algorithm is wait-free if every operation has a bound on the number of steps the algorithm will take before the operation completes.

- **Lock-freedom**:
  - Lock-freedom allows individual threads to starve but guarantees system-wide throughput.
  - An algorithm is lock-free if, when the program threads are run for a sufficiently long time, at least one of the threads makes progress (for some sensible definition of progress).
    - In particular, if one thread is suspended, then a lock-free algorithm guarantees that the remaining threads can still make progress.
    - Hence, if two threads can contend for the same mutex lock or spinlock, then the algorithm is *not* lock-free.
  - All wait-free algorithms are lock-free.

- **Obstruction-freedom**:
  - Obstruction-freedom is the weakest natural non-blocking progress guarantee.
  - An algorithm is obstruction-free if at any point, a single thread executed in isolation (i.e., with all obstructing threads suspended) for a bounded number of steps will complete its operation.
  - All lock-free algorithms are obstruction-free.

C++

Memory Model

Modification Orders

- Every object in C++ has a *modification order* composed of all the writes to that object from all threads in the program.
- Modification order varies between runs, but in any given execution of the program, we must ensure that all threads agree on the modification order.
- This requirements means that:
  - Once a thread has seen a particular entry in the modification order,
    - subsequent reads from that thread must return later values, and
    - subsequent writes from that thread to that object must occur later in the modification order.
  - A read of an object that follows a write to that object in the same thread must either return the value written or another value that occurs later in the modification order of that object.
  - Although all threads must agree on the modification orders of each individual object in a program, they don't necessarily have to agree on the relative order of operations on separate objects.

Sequentially-consistent Ordering

- If all operations on instances of atomic types are sequentially consistent, the behavior of the multithreaded program is *as if* all these operations were performed in some particular sequence by a single thread.
- In other words, all threads must see the same order of operations.
- We can reason about sequentially consistent ordering with the following mental model:
  - There's a single global order of events that is agreed by all threads.
  - The operations from different threads are neatly interleaved one after another.
  - Once a side-effect is visible to a thread, it is also visible to all other threads.

Non-sequentially Consistent Ordering

- For non-sequentially consistent memory orderings, we *cannot* use the same mental model as the one we used for sequential consistency. This means that:
  - Operations from different threads don't neatly interleaved one after another.
  - Threads don't have to agree on the order of events (operations).
- In the absence of other ordering constraints, the only requirement is that all threads agree on the modification order of each individual variable.

Acquire-release ordering

- Under this ordering model, atomic loads are *acquire* operations (`std::memory_order_acquire`), and atomic stores are *release* operations (`std::memory_order_release`).

- If an atomic store in thread A is tagged `memory_order_release`, an atomic load in thread B from the same variable is tagged `memory_order_acquire`, and the load in thread B reads a value written by the store in thread A, *then* the store in thread A *synchronizes-with* the load in thread B.

- 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.
  - That is, once the atomic load is completed, thread B is guaranteed to see everything thread A wrote to memory.
  - This promise only holds if B actually returns the value that A stored, or a value from later in the release sequence.

- The synchronization is established only between the threads *releasing* and *acquiring* the same atomic variable. Other threads can see different order of memory accesses than either or both of the synchronized threads.

Relaxed Ordering

- Operations on atomic types performed with relaxed ordering don't participate in *synchronizes-with* relationships.

- Operations on the same variable on within a single thread still obey *happens-before* relationship, but there's almost no requirement on ordering relative to other threads.

Reordering Constraints on std::memory_order

- `std::memory_order_relaxed`: No reordering constraints.

- `std::memory_order_acquire`: No reads or writes in the current thread can be reordered *before* this load.

- `std::memory_order_release`: No reads or writes in the current thread can be reordered *after* this store.

- `std::memory_order_acq_rel`: No memory reads or writes in the current thread can be reordered *before* the load, nor *after* the store.

Fences

- `std::atomic_thread_fence` establishes memory synchronization ordering for non-atomic and relaxed atomic accesses.

- Note however, that at least one atomic operation is required to set up the synchronization, as described below.

- **Fence-atomic synchronization**:
  - A release fence F in thread A *synchronizes-with* atomic *acquire operation* Y in thread B, if
    * there exists an atomic store X (with any memory order),
    * Y reads the value written by X (or the value would be written by release sequence headed by X if X was a release operation), and
    * F is *sequenced-before* X in thread A.
  - In this case, all non-atomic and relaxed atomic stores that are *sequenced-before* F in thread A will *happen-before* all non-atomic and relaxed atomic loads from the same locations made in thread B after Y.

- **Atomic-fence synchronization**:
  - An atomic *release operation* X in thread A *synchronizes-with* an acquire fence F in thread B, if
    * there exists an atomic read Y (with any memory order),
    * Y reads the value written by X (or by the release sequence headed by X), and
    * Y is *sequenced-before* F in thread B.
  - In this case, all non-atomic and relaxed non-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 F.

- **Fence-fence synchronization**:
  - A release fence FA in thread A *synchronizes-with* an acquire fence FB in thread B, if
    * there exists an atomic object M,
    * there exists an atomic write X (with any memory order) that modifies M in thread A,
    * FA is *sequenced-before* X in thread A,
    * there exists an atomic read Y (with any memory order) in thread B,
    * Y reads the value written by X (or the value would be written by release sequence headed by X if X were a release operation), and
    * Y is *sequenced-before* FB in thread B.

- In this case, all non-atomic and relaxed atomic stores that are sequenced-before FA in thread A will *happen-before* all non-atomic and relaxed atomic atomic loads from the same locations made in thread B after FB.

Debugging Tools for Multithreaded Programs

Valgrind

- Valgrind is an instrumentation framework for building dynamic analysis tools.
- **Memcheck:**
  - Memcheck detects memory-management problems, and is aimed primarily at C and C++ programs.
  - Memcheck implementation uses *shadow memory*, i.e. all reads and writes of memory are checked, and calls to malloc/new/free/delete are intercepted.
  - As a result, Memcheck can detect if your program:
    * Accesses memory it shouldn't.
    * Uses uninitialised values in dangerous ways.
    * Leaks memory.
    * Does bad frees of heap blocks (double frees, mismatched frees).
    * Passes overlapping source and destination memory blocks to memcpy() and related functions.
  - Memcheck runs programs about 10-30x slower than normal.
- **Helgrind:**
  - Helgrind is a Valgrind tool for detecting synchronisation errors in C and C++ programs that use the POSIX pthreads threading primitives.
  - Helgrind can detect three classes of errors:
    * Misuses of the POSIX pthreads API.
    * Potential deadlocks arising from lock ordering problems.
    * Data races – accessing memory without adequate locking or synchronisation.
  - Helgrind runs programs about 100x slower than normal.
  - Deadlock detection:
    * Helgrind builds a directed graph indicating the order in which locks have been acquired.
  - Data race detection:
    * Builds a directed acyclic graph representing the collective happens-before dependencies.
    * Monitors all memory accesses.

Sanitizers

- **AddressSanitizer:**
  - AddressSanitizer can detect the following types of bugs:
    * Out-of-bounds accesses to heap, stack and globals
    * Use-after-free
    * Use-after-return
    * Use-after-scope
    * Double-free, invalid free
    * Memory leaks
  - AddressSanitizer uses a shadow memory scheme to detect memory bugs.
  - AddressSanitizer does not detect any uninitialized memory reads (this is detected by MemorySanitizer).
  - AddressSanitizer is also not capable of detecting all arbitrary memory corruption bugs, nor all arbitrary write bugs due to integer underflow/overflows.
  - Typical slowdown introduced by AddressSanitizer is 2x.
- **ThreadSanitizer:**
  - ThreadSanitizer is a tool that detects data races and deadlocks.
  - Typical slowdown introduced by ThreadSanitizer is about 5-15x.
  - Typical memory overhead introduced by ThreadSanitizer is about 5-10x.

Go

Goroutines

- A *goroutine* is a lightweight thread managed by the Go runtime.
- Goroutines run in the same address space, so access to shared memory must be synchronized.
- Goroutines are not garbage collected.

Channels

- Philosophy of Go: "Do not communicate by sharing memory. Instead, share memory by communicating."
- Channels are a typed conduit through which you can send and receive values.

```go
ch <- v      // Send v to channel ch
v := <-ch    // Receive from ch, and assign value to v
```

- **Unbuffered channel**:

```go
ch := make(chan int)
```

  – Sends and receives block until the other side is ready.

- **Buffered channel**:

```go
ch := make(chan int, 100)
```

  – Sends to a buffered channel block only when the buffer is full.

  – Receives block when the buffer is empty.

- **Range and Close**:
  – A sender can `close` a channel to indicate that no more values will be sent.

  – Receivers can test whether a channel has been closed by assigning a second parameter to the receive expression:

```go
v, ok := <-ch
```

  `ok` is `false` if there are no more values to receive and the channel is closed.

  – The loop receives values from the channel repeatedly until it is closed.

```go
ch := make(chan int, 10)
...
for i := range c {
    ...
}
```

- **Ownership of a channel**:
  – Owner should:
    * Instantiate the channel.
    * Have write-access view into the channel. (`chan` or `chan<-`)
    * Close the channel.
    * Pass ownership to another goroutine.
    * Encapsulate the channel and expose to consumers via a read-only view into the channel. (`<-chan`)

  – Consumer should:
    * Know when a channel is closed.
    * Responsibly handle blocking for any reason.

- **Select**:
  – The `select` statement lets a goroutine wait on multiple communication operations.

  – A select `blocks` until one of its cases can run, then it executes that case. It chooses one at random if multiple are ready.

```go
for {
    select {
    case i := <-c:
        // use i
    case <-quit:
        ...
        return
    default:
        ...
    }
}
```

Rust

Ownership, Borrowing, and Lifetimes

- Ownership rules:
  – Each value in Rust has an *owner*.

  – There can only be one owner at a time.

  – When the owner goes out of scope, the value will be dropped.

- The ownership rules prevent *double free*.
- The rules of references:
  – At any given time, you can have *either* one mutable reference or any number of immutable references. This rule prevents data race.

  – Every reference in Rust has a *lifetime*, which is the scope for which that reference is valid. This rule prevents *dangling references*.

- The Rust compiler has a *borrow checker* that compares scopes to determine whether all borrows are valid.

Smart Pointers in Rust

- Smart pointers are usually implemented using structs.
- Unlike an ordinary struct, smart pointers implement the `Deref` and `Drop` traits.
  - The `Deref` trait allows you to customize the behavior of the *dereference operator*.
  - The `Drop` trait allows you to customize the code that's run when an instance of the smart pointer goes out of scope. (e.g. RAII)

Box

- Boxes don't have performance overhead.
- Use cases:
  - When you have a type whose size can't be known at compile time, and you want to use a value of that type in a context that requires an exact size.
  - When you have a large amount of data and you want to transfer ownership but ensure the data won't be copied when you do so.
  - When you want to own a value and you care only that it's a type that implements a particular trait rather than being of a specific type.
- Enabling recursive types with Boxes:
  - Recursive types pose an issue because at compile time Rust needs to know how much space a type takes up.
  - Implementing a *cons list*:

```rust
enum List {
    Cons(i32, Box<List>),
    Nil,
}
```

Rc, the Referenced Counted Smart Pointer

- To enable multiple ownership explicitly, we use `Rc<T>`, which is an abbreviation for *reference counting*.
  - The `Rc<T>` type keeps track of the number of references to a value to determine whether or not the value is still in use.
  - If there are zero references to a value, the value can be cleaned up without any references becoming invalid.

```rust
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```

- Cloning an `Rc<T>` increases the reference count.
  - We can get the reference count by calling the `Rc::strong_count` function.
  - The function is named `strong_count` rather than `count` because `Rc<T>` type also has a `weak_count`.
- `Rc<T>` allows us to share data between multiple parts of our program for reading only.

RefCell and the Interior Mutability Pattern

- *Interior mutability* is a design pattern in Rust that allows you to mutate data even when there are immutable references to that data.
  - To mutate data, the pattern uses `unsafe` code.
  - Unsafe code indicates to the compiler that we're checking the rules manually instead of relying on the compiler to check them for us.
- Similar to `Rc<T>`, `RefCell<T>` is only for use in single-threaded scenarios.
- Here is a recap of the reasons to choose `Box<T>`, `Rc<T>`, or `RefCell<T>`:
  - `Rc<T>` enables multiple owners of the same data; `Box<T>` and `RefCell<T>` have single owners.
  - `Box<T>` allows immutable or mutable borrows checked at compile time; `Rc<T>` allows only immutable borrows checked at compile time; `RefCell<T>` allows immutable or mutable borrows checked at runtime.
  - Because `RefCell<T>` allows mutable borrows checked at runtime, you can mutate the value inside the `RefCell<T>` even when the `RefCell<T>` is immutable.
- With `RefCell<T>`, we use the `borrow` and `borrow_mut` methods.
  - The `borrow` method returns the smart pointer type `Ref<T>`.
  - The `borrow_mut` method returns the smart pointer type `RefMut<T>`.

- The `RefCell<T>` keeps track of how many `Ref<T>` and `RefMut<T>` smart pointers are currently active.
  - Just like the compile-time borrowing rules, `RefCell<T>` lets us have many immutable borrows or one mutable borrow at any point in time.
- We can have multiple owners of mutable data by combining `Rc<T>` and `RefCell<T>`.

Reference Cycles Can Leak Memory

- Rust's memory safety guarantee makes it difficult, but not impossible, to accidentally create memory that is never cleaned up (known as *memory leak*).
- Preventing memory leaks entirely is not one of Rust's guarantees, meaning memory leaks are memory safe in Rust.
- We can see that Rust allows memory leaks by using `Rc<T>` and `RefCell<T>`: it's possible to create references where items refer to each other in a cycle.

Fearless Concurrency

- By leveraging ownership and type checking, many concurrency errors are compile-time errors in Rust rather than runtime errors.
- The Rust standard library uses a 1:1 model of thread implementation, whereby a program uses one operating system thread per one language thread.

Using Threads in Rust

```rust
fn main() {
    let handle = thread::spawn(|| { ... });
    handle.join().unwrap();
}
```

- The return type of `thread::spawn` is `JoinHandle`.
- A `JoinHandle` is an owned value that, when we call the `join` method on it, will wait for its thread to finish.
- We use the `move` keyword to force closure to take ownership of the values it uses from the environment.

```rust
fn main() {
    let v = vec![1, 2, 3];
    let handle = thread::spawn(move || {
        println!("{:?}", v);
    });
    handle.join().unwrap();
}
```

Message-Passing Concurrency in Rust

- Rust also supports message-passing concurrency with the use of *channels*.

```rust
fn main() {
    let (tx, rx) = mpsc::channel();
    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });
    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

  - A channel has two halves: a transmitter and a receiver.
  - A channel is said to be *closed* if either the transmitter or receiver half is dropped.
- The `send` function:
  - takes ownership of its parameter, and when the value is moved, the receiver takes ownerhsip of it.
  - returns a `Result<T, E>` type, so if the receiver has already been dropped and there's nowhere to send a value, the send operation will return an error.
- The receiver has two useful methods: `recv` and `try_recv`.
  - `recv` will block the thread's execution and wait until a value is sent down the channel.
  - `try_recv` method doesn't block, but will instead return a `Result<T, E>` immediately: an `Ok` value holding a message if one is available and an `Err` value if there aren't any messages this time.
- We can treat `rx` as in iterator when we want to receive multiple messages.

```rust
for received in rx { ... }
```

  - When all transmitters have been dropped, the channel is closed, and the iteration will end.
- `mpsc` is an acronym for *multiple producer, single consumer*.
  - We can create multiple transmitters by cloning the transmitter.

Shared-State Concurrency in Rust

- Mutexes in Rust implement a strategy called *poisoning* where a mutex is considered poisoned whenever a thread panics while holding the mutex.
  - For a mutex, this means that the `lock` and `try_lock` methods return a `Result` which indicates whether a mutex has been poisoned or not.
  - A poisoned mutex, however, does not prevent all access to the underlying data. The `PoisonError` type has an `into_inner` method which return the guard to allow access to data.
- `Mutex<T>` is a smart pointer.
  - The call to `lock` returns a smart pointer called `MutexGuard`, wrapped in a `LockResult` that we handled with the call to `unwrap`.
  - The `MutexGuard` smart pointer implements `Deref` to point to our inner data.
  - The smart pointer also has a `Drop` implementation that releases the lock automatically when a `MutexGuard` goes out of scope. (RAII)
- To share a `Mutex<T>` between multiple threads, we wrap it with `Arc<T>`, which is an *atomically reference counted smart pointer*.

```rust
fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];
    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();
            *num += 1;
        });
        handles.push(handle);
    }
    for handle in handles {
        handle.join().unwrap();
    }
    println!("Result: {}", *counter.lock.unwrap());
}
```

  - Notice that `counter` is immutable but we could get a mutable reference to the value inside it; this means that `Mutex<T>` provides interior mutability, as the `Cell` family does.
- Using `Mutex<T>` does not guarantee that we are safe from deadlocks.

Sync and Send Traits

- Allowing transference of ownership between threads:
  - The `Send` marker trait indicates that the ownership of values of the type implementing `Send` can be transferred between threads.
  - Almost every Rust type in `Send`, but there are some exceptions, including `Rc<T>`.
  - Any type composed entirely of `Send` types is automatically marked as `Send` as well.
  - Almost all primitive types are `Send`, aside from raw pointers.
- Allowing access from multiple threads:
  - The `Sync` marker trait indicates that it is safe for the type implementing `Send` to be referenced from multiple threads.
  - Any type `T` is `Sync` if `&T` is `Send`, meaning the reference can be sent safely to another threads.
  - Primitive types are `Sync`, and types composed entirely of types that are `Sync` are also `Sync`.
  - The `RefCell<T>` type and the family of related `Cell<T>` types are not `Sync`. The implementation of borrow-checking that `RefCell<T>` does at runtime is not thread-safe.
  - The smart pointer `Mutex<T>` is `Sync`.
- Manually implementing the `Send` and `Sync` traits involves implementing `unsafe` Rust code.

Asynchronous Programming in Rust

Futures

- The problem with threads:
  - Context switching cost: Each switch is expensive.
  - Memory overhead: Each thread has its own stack space that needs to get managed by the OS.
- Non-blocking I/O:
  - We could have `read()` return a special error value instead of blocking.
  - If we see that a client hasn't sent us anything yet, we can do other useful work on this thread e.g. reading from other descriptors we're managing.
  - `epoll` is a kernel-provided mechanism that notifies us of what fds are ready for I/O.
  - This allows us to have concurrent I/O with one thread!
- State management:
  - Non-blocking I/O code looks okay in theory, but in reality we need to figure out how to manage the state associated with each conversation.

- *Futures* in Rust allows us to keep track of in-progress operations along with associated state, in one package.
- Rust docs: Futures are single eventual values produced by asynchronous computations.

- The `Future` trait:

```
trait Future {
    type Output;
    fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output>;
}

enum Poll<T> {
    Ready(T),
    Pending,
}
```

  - The executor thread should call poll() on the future to start it off.
  - It will run code until it can no longer progress.
    * If the future is complete, returns `Poll::Ready(T)`.
    * If future needs to wait for some event, returns `Poll::Pending`, and allows the single thread to work on another futures.
    * When `poll()` is called, `Context` structure passed in.
    * Includes a `wake()` function that is set to be called when future can make progress again. (This is implemented internally using system calls)
    * After `wake()` called, executor can use `Context` to see which Future can be polled to make new progress.

- How executors work:
  - An executor loops over futures that can currently make progress, calling `poll()` on them to give them attention until they need to wait again.
    * When no futures can make progress, the executor goes to sleep until one or more futures calls `wake()`.
    * Once awakened, the executor goes through those futures, `poll()`ing them.

- Futures should not block:
  - If code within a future causes the thread to sleep, the executor running that code is going to sleep!
  - This defeats the purpose of the system since the executor cannot continue to other futures.
  - Asynchronous code needs to use non-blocking versions of *everything*.

Async/await

- Working with futures is not ergonomic:
  - Futures are composed with various combinators.
  - Chaining combinators may result in strange decomposition, which is bad for abstraction.
- Rust introduced `async/await` which are syntactic sugar to eliminate the issues.
- An `async` function is a function that returns a `Future`.
- `.await` waits for a future and gets its value.
- You can only use `await` in `async` functions.

```
async fn add_to_inbox(email_id: u64, recipient_id: u64)
    -> Result<(), Error> {
    let message = load_message(email_id).await?;
    let recipient = get_recipient(recipient_id).await?;
    recipient.verify_has_space(&message)?;
    recipient.add_to_inbox(message).await
}
```

  - The code gets compiled into a `Future` with `poll()` method.
  - There are 5 places where the executor might be paused, or not actively executing. Thus internally, we can use an `enum` to store the state for these possibilities.

- Implications:
  - `async` functions have no stack! (sometimes called *stackless coroutines*)
    * The executor thread still has a stack, but it isn't 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 needs to have a fixed size known at compile time.
  - `async` functions are one of Rust's *zero-cost abstractions*, meaning there is no runtime overhead.

- Async code makes sense when...
  - You need an extremely high degree of concurrency.
  - Work is primarily I/O bound.
    * Context switching overhead is expensive only if you're using a tiny fraction of the time slice.
    * If you're doing a lot of work on the CPU for an extended period of time, you might prevent the executor from running other tasks.

Concurrency Libraries in Rust

Tokio

- At a high level, Tokio provides a few major components:
  - A multi-threaded runtime for executing asynchronous code.
  - An asynchronous version of the standard library.
- When not to use Tokio:
  - Speeding up CPU-bound computations by running them in parallel on several threads. Tokio is designed for I/O-bound applications. You should be using *rayon* instead.
  - Reading a lot of files, as Tokio provides no advantage here compared to an ordinary threadpool.
  - Sending a single web request. The place where Tokio gives you an advantage is when you need to do many things at the same time.

Rayon

- A data-parallelism library for Rust.
- It is extremely lightweight and makes it easy to convert a sequential computation into a parallel one.
- It also guarantees data-race freedom.

Crossbeam

- Provides an implementation of scoped threads.
  - Scoped threads are allowed to borrow variables on stack, since its mechanism guarantee to the compiler that spawned threads will be joined before the scope ends.
  - In other words, if a variable is borrowed by a thread inside the scope, the thread must complete before the variable is destroyed.
- Provides multi-producer multi-consumer channels for message passing.
- Provides exponential backoff utility.
  - Performs exponential backoff in spin loops.
  - Rationale: It's not helpful to simply retry as fast as possible. If the resource is overloaded, requesting it more will make it even more overloaded!

Classical Synchronization Problems

Readers-writers

Solution 1

- C++ implementation:

```cpp
std::mutex mut;

void reader() {
  std::shared_lock lock{mut};
  // Critical section
}

void writer() {
  std::unique_lock lock{mut};
  // Critical section
}
```

- Issue: Starvation of writers is possible.

Solution 2 (Starvation-free)

- C++ implementation:

```cpp
std::mutex mut_write;
std::mutex mut_read;

void reader() {
  { std::scoped_lock lock1{mut_write}; }
  std::shared_lock lock2{mut_read};
  // Critical section
}

void writer() {
  std::scoped_lock lock1{mut_write};
  std::unique_lock lock2{mut_read};
  // Critical section
}
```

- Issue: Starvation of writers is possible.

Barrier

Solution

- C++ implementation:

```cpp
class Barrier {
private:
  std::ptrdiff_t expected;
  std::ptrdiff_t count;
  std::mutex mut;
  std::counting_semaphore<> turnstile1;
  std::counting_semaphore<> turnstile2;

public:
  Barrier(std::ptrdiff_t expected)
    : expected{expected}, count{0}, mut{},
      turnstile1{0}, turnstile2{1} {}

  void arrive_and_wait() {
    {
      std::scoped_lock lock{mut};
      count++;
      if (count == expected) {
        turnstile2.acquire()
        turnstile1.release()
      }
    }

    turnstile1.acquire();
    turnstile1.release();

    {
      std::scoped_lock{mut};
      count--;
      if (count == 0) {
        // Reset the barrier
        turnstile1.acquire();
        turnstile2.release();
      }
    }

    turnstile2.acquire();
    turnstile2.release();
  }
};
```

- Go implementation:

```go
type Barrier struct {
    wg1 sync.WaitGroup;
    wg2 sync.WaitGroup;
}

func (b *Barrier) Init(expected int) {
    b.wg1.Add(expected);
    b.wg2.Add(expected);
}

func (b *Barrier) Wait() {
    b.wg1.Done();
    b.wg1.Wait();

    // Reset the barrier
    b.wg1.Add(1);

    b.wg2.Done();
    b.wg2.Wait();
    b.wg2.Add(1);
}
```

Dining Philosophers

```python
def philosopher(i):
    while True:
        think()
        get_chopsticks(i)
        eat()
        put_chopsticks(i)
```

Solution 1

```python
def get_chopsticks(i):
    left_chopstick(i).lock()
    right_chopstick(i).lock()

def put_chopsticks(i):
    left_chopstick(i).unlock()
    right_chopstick(i).unlock()
```

- Issue: Deadlock
- This solution is actually fine if we implement it in C++, since `std::scoped_lock` uses a deadlock avoidance algorithm when locking multiple mutexes.

Solution 2

- Intuition: If we have one less philosophers at the table, deadlock can be avoided.
- We can control the number of philosphers at the table with a 'footman' semaphore.

```
footman = Semaphore(num_of_philosophers - 1)

def get_chopsticks(i):
    footman.acquire();
    left_chopstick(i).lock()
    right_chopstick(i).lock()

def put_chopsticks(i):
    footman.release();
    left_chopstick(i).unlock()
    right_chopstick(i).unlock()
```

- This solution is starvation-free if the `footman` semaphore has the following property:
  - If a thread is waiting at a semaphore, then the number of threads that will be woken before it is bounded.
- In C++, starvation is possible because mutexes and semaphores are not fair.

Solution 3

- Intuition: To avoid deadlock, change the order in which the philosophers pick up the chopsticks.
- In Go implementation, we can use odd-even ring communication to avoid deadlock.

```
type Chopstick struct{}

type DiningTable struct {
    numOfPhilosopher int
    chopsticks []chan Chopstick
}

func (t *DiningTable) Init(numOfPhilosopher int) {
    t.numOfPhilosopher = numOfPhilosopher
    t.chopsticks = make([]chan Chopstick, 0, numOfPhilosopher)
    for i := 0; i < numOfPhilosopher; i++ {
        chopstick := make(chan chopstick, 1)
        chopstick <- Chopstick{}
        t.chopsticks = append(t.chopsticks, chopstick)
    }
}

func (t *DiningTable) LeftChopstick(pid int) chan Chopstick {
    return t.chopsticks[pid]
}

func (t *DiningTable) RightChopstick(pid int) chan Chopstick {
    return t.chopsticks[(pid + 1) % t.numOfPhilosopher]
}

func (t *DiningTable) EvenChopstick(pid int) chan Chopstick {
    if pid % 2 == 0 {
        return t.LeftChopstick(pid)
    } else {
        return t.RightChopstick(pid)
    }
}

func (t *DiningTable) OddChopstick(pid int) chan Chopstick {
    if pid % 2 == 1 {
            return t.LeftChopstick(pid)
    } else {
        return t.RightChopstick(pid)
    }
}

func (t *DiningTable) Eat(pid int, eat_callback func(pid int)) {
    evenChopstick := t.EvenChopstick(pid)
    oddChopstick := t.OddChopstick(pid)

    <-evenChopstick
    <-oddChopstick

    eat_callback(pid)

    evenChopstick <- Chopstick{}
    oddChopstick <- Chopstick{}
}
```

Solution 4 (Tanenbaum's)

- For each philosopher there is a state variable that indicates whether the philosopher is thinking, eating, or waiting to eat ("hungry") and a semaphore that indicates whether the philosopher can start eating.

```python
state = ["thinking" for i in range(n)]
sem = [Semaphore(0) for i in range(n)]
mutex = Semaphore(1)

def get_fork(i):
    mutex.wait()
    state[i] = "hungry"
    test(i)
    mutex.signal()
    sem[i].wait()

def put_fork(i):
        mutex.wait()
        state[i] = "thinking"
        test(left(i))
        test(right(i))
        mutex.signal()

def test(i):
    if state[i] == "hungry" and
    state[left(i)] == "eating" and
    state[right(i)] == "eating":
        state[i] == "eating"
        sem[i].signal()
```

- Issue: Deadlock is not possible, but the solution is not starvation-free.

Barbershop

Solution

- C++ implementation:

```cpp
template <size_t max_customer>
class Barbershop {
private:
  size_t customer;
  std::mutex mut;
  std::counting_semaphore<> sem_customer;
  std::counting_semaphore<> sem_barber;
  std::counting_semaphore<> sem_customer_done;
  std::counting_semaphore<> sem_barber_done;

public:
  void barber() {
    sem_customer.acquire();
    sem_barber.release();
    cut_hair();
    sem_customer_done.acquire();
    sem_barber_done.release();
  }

  void customer() {
    {
      std::scoped_lock lock{mut};
      if (customer == max_customer) {
        balk();
        return;
      }
      customer++;
    }

    sem_customer.release();
    sem_barber.wait();
    get_hair_cut();
    sem_customer_done.release();
    sem_barber_done.acquire();

    {
      std::scoped_lock{mut};
      customer--;
    }
  }
};
```

- Go implementation:

```go
type Barbershop struct {
    chairs chan int
    barber chan struct{}
}

func (bs *Barbershop) Init(numOfChair int) {
    bs.chairs = make(chan int, numOfChair)
    bs.barber = make(chan struct{})
}

func (bs *Barbershop) Barber() {
    for {
        bs.barber <- struct{}{}
```

```
        <-bs.chairs
        cutHair()
    }
}

func (bs *Barbershop) Customer() {
    for {
        select {
        case bs.chairs <- 1:
            <-bs.barber
            getHairCut()
        default:
            balk()
        }
    }
}
```

H2O

Solution

- C++ implementation:

```
class WaterFactory {
private:
  std::barrier<> barrier;
  std::counting_semaphore<> hydrogen_sem;
  std::counting_semaphore<> oxygen_sem;

public:
  WaterFactory()
    : barrier{3}, hydrogen_sem{2}, oxygen_sem{1} {}

  void hydrogen(void (*bond)()) {
    hydrogen_sem.acquire();
    barrier.arrive_and_wait();
    bond();
    hydrogen_sem.release();
  }

  void oxygen(void (*bond)()) {
    oxygen_sem.acquire();
    barrier.arrive_and_wait();
    bond();
    oxygen_sem.release();
  }
};
```

- Go implementation using a daemon goroutine:

```
type WaterFactoryWithDaemon struct {
    precomH chan chan struct{}
    precomO chan chan struct{}
}

func NewFactoryWithDaemon() WaterFactoryWithDaemon {
    wfd := WaterFactoryWithDaemon{
        precomH: make(chan chan struct{}),
        precomO: make(chan chan struct{}),
    }

    // Daemon goroutine
    go func() {
        h1 := <-wfd.precomH
        h2 := <-wfd.precomH
        o1 := <-wfd.precomH

        h1 <- struct{}{}
        h2 <- struct{}{}
        o1 <- struct{}{}

        <-h1
        <-h2
        <-o1
    }()

    return wfd
}

func (wfd *WaterFactoryWithDaemon) hydrogen(bond func()) {
    commit := make(chan struct{})
    wfd.precomH <- commit
    <-commit
    bond()
    commit <- struct{}{}
}

func (wfd *WaterFactoryWithDaemon) oxygen(bond func()) {
    commit := make(chan struct{})
    wfd.precomO <- commit
    <-commit
```

```go
    bond()
    commit <- struct{}{}
}
```

- Downsides of the daemon-based approach:
  - The daemon is a bottleneck and a single point of failure.
  - The daemon goroutine does not go away and leaks memory.
- Go implementation using oxygen atoms as leader goroutines:

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

func NewFactoryWithLeader() WaterFactoryWithLeader {
    wf := WaterFactoryWithLeader{
        oxygenMutex: make(chan struct{}, 1),
        precomH: make(chan chan struct{}),
    }

    wf.oxygenMutex <- struct{}{}
    return wf
}

func (wf *WaterFactoryWithLeader) hydrogen(bond func()) {
    commit := make(chan struct{})
    wf.precomH <- commit
    <-commit
    bond()
    commit <- struct{}{}
}

func (wf *WaterFactoryWithLeader) oxygen(bond func()) {
    <-wf.oxygenMutex

    h1 := <-wf.precomH
    h2 := <-wf.precomH

    h1 <- struct{}{}
    h2 <- struct{}{}

    bond()

    <-h1
    <-h2

    wf.oxygenMutex <- struct{}{}
}
```

- Rust implementation:

```rust
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::
    std::mem::drop(s);
    std::mem::drop(r);
    join_all(join_handles).await;
}
```