Jin Wei CS3211 Finals Notes

Concurrency vs Parallelism

Parallelism $\subseteq$ Concurrency

Require hardware for true concurrency.

Processes vs Threads

- processes are expensive to create
  - overhead of syscalls
  - all data structures must be allocated, initialise, copied
- inter-process communication is costly
  - overhead of syscalls through OS

Threads share the *address space* of the process (* still need to synchronise).

- user-level threads
  - mapped onto kernel threads
  - POSIX threads are 1:1 to kernel threads
- kernel threads

Race Condition, Data Race

Race Condition

Depends on relative ordering of the execution. Not always undesirable.

Invariants that are broken during an *update* of a data structure is also referred to as a race condition. This is normally bad!

- dangling pointers
- random memory corruption - due to reading inconsistent values from a partial update
- double free, e.g. two threads pop the same value and both delete it

Lifetime issues:

- thread outlives the data it accesses

Data Race

Data race for two accesses to a *single memory location* from separate threads (sharing)

- no enforced ordering between accesses (no ordering)
- *and* one or both are not atomic (no ordering)
- *and* one or both are writes (mutation)

Undefined behaviour (literally *anything* can happen)! Always undesirable.

Avoid using critical section, using locks, ordered atomics, transactional memory, ...

Program Execution

- Compilation and linking (done by compiler - gcc, g++, clang)
  - Preprocessor: replaces preprocessor directives (`#include` and `#define`)
  - Compiler: parses pure C++ code into assembly
  - Assembler: assembles assembly into machine code
  - Linker: produces the final compilation binary from the object files produced
- Loading
  - usually OS specific
- Execution
  - coordinated by OS
  - program gets access to system resources: CPU, memory, devices, ...

Concurrent Data Structures

Multiple threads can access the data structure concurrently, performing the same or distinct operations. Every thread sees a self-consistent view of the data structure.

- No data is lost or corrupted
- All invariants are upheld
- No problematic race condition

Constructors and destructors require exclusive access to the data structure.

Invariants

* use invariants to reason about correctness

- ensure that no thread can see a state where the invariants are broken
- ensure invariants are not broken even when there are exceptions

Protect with mutex

- Forces serialisation, try to minimise these regions.
  - Lock at an appropriate granularity, e.g. per-node instead of whole-data-structure
- Possibly use shared mutexes (allows concurrent reads)

Blocking Data Structures

Uses blocking constructs: mutexes, condition variables and futures

Execution of the thread is suspended until the block is removed (e.g. mutex unlocked, cond_var notified, future is done)

Non-blocking Data Structures

Does not use blocking constructs: instead, uses spin-lock, atomics

- Obstruction-free ("progress if no interference"): if all other threads are paused, then any given thread will complete its operation in a bounded number of steps
- Lock-free ("someone makes progress"): if multiple threads are operating on a data structure, then after a bounded number of steps, one of them will complete its operation
  - allows individual threads to starve, but guarantees system-wide throughput
  - more than one thread can access the data structure concurrently
- Wait-free ("no one ever waits"): every thread operating on a data structure will complete its operation in a bounded number of steps, even if other threads are also operating on the data structure
  - Wait-free => Lock-free
  - avoids lock-free's problem of a thread starving
  - Very hard!

2 definitions of completed:

1. if the function returns
2. if its effects become visible

Pros:

- high concurrency: some thread makes progress with each step
- robustness
  - if a thread dies halfway, it won't affect the data structure
  - cannot block threads from accessing the data structure

Cons:

- livelocks are possible
- decreases actual performance (even though the time each thread spends waiting has been reduced)
  - atomic operations are slower
  - memory content + write propagation
  - cache ping-pong with multiple threads accessing the same atomic variables
  - false sharing

Classic Synchronisation Problems

- producer consumer problem
- reader writer problem
- (reusable) barrier
  - all threads must stop at this point and wait until all other threads reach this barrier

- dining philosophers
- sleeping barber / barbershop

Reusable Barrier

```
int expected;
std::counting_semaphore turnstile{0};
std::counting_semaphore turnstile2{1};

void arrive_and_wait() {
    {
        std::scoped_lock lk{mu};
        count++;
        if (count == expected) {
            // close waiter turnstile
            turnstile2.acquire();
            // let everyone into critical section
            turnstile.release(expected);
        }
    }

    turnstile.acquire();

    {
        std::scoped_lock lk{mu};
        count--;
        if (count == 0) {
            // close turnstile to reset barrier
            turnstile.acquire();
            // let waiters through
            turnstile2.release(expected);
        }
    }

    turnstile2.acquire();
}
```

```
func New(expected int) *Barrier {
    var wg sync.WaitGroup
    var wg2 sync.WaitGroup

    wg.Add(expected)
    wg2.Add(expected)

    return &Barrier{
        wg: wg,
        wg2: wg2,
    }
}
func (b *Barrier) Wait() {
    b.wg.Done()
    b.wg.Wait()
    // now all threads have arrived

    // reset barrier
    b.wg.Add(1)
    b.wg2.Done()
    b.wg2.Wait() // wait for barrier to be fully reset

    b.wg2.Add(1) // reset wg2
}
```

C++

C++ Memory Model

As-if rule

> Only the observed behaviour needs to be correct.

C++ compiler can do anything as long as:

- order of accesses (reads + writes) to *volatile* objects stay the same, i.e. they are not reordered with respect to other volatile accesses
- at program termination, data written to files is exactly the same as intended
- prompting text sent to interactive devices is shown before program waits for input

A program with undefined behaviour can do anything.

Atomics

`<atomic>` header, `std::atomic<T> value`

Used to enforce *modification order*

Modification Order (only for atomics!)

Once a thread has seen a particular entry in the modification order:

- subsequent reads from the thread must return the same or later values in the order
- subsequent writes from the thread must occur later in the modification order

All threads must agree on the modification order of *each individual object*, but not the relative order of operations on separate objects.

- `std::memory_order_seq_cst`
  - can find a total ordering (for that particular object) equivalent to a sequential program
- `std::memory_order_acquire`, `std::memory_order_release`, `std::memory_order_acq_rel`
  - no total ordering, but has synchronises-with
- `std::memory_order_relaxed`
  - no synchronises-with; has happens-before; has monotonic reads

If the hardware doesn't support the consistency level, the compiler must still guarantee it (might end up being more expensive).

Sequenced-before

Within a thread, A can be *sequenced-before* B.

If A is sequenced-before B, evaluation of A must finish before evaluation of B starts.

Synchronises with

Occurs between atomic load and store operations.

Synchronises two threads.

Simply happens-before

Regardless of threads, A *simply happens-before* B if any are true:

- A is sequenced-before B
- or A synchronises with B
- or transitive: A -> C, C-> B => A -> B

Strongly happens-before

Regardless of threads, A *strongly happens-before* B if any are true:

- A is sequenced-before B
- or A synchronises with B, and both are sequentially consistent
- or A is sequenced-before X, X simply happens before Y, and Y is sequenced-before B
- or transitive: A -> C, C-> B => A -> B

Informally: A appears to be evaluated before B in all contexts.

* Visible side effects

The write A is seen by read B if:

- A happens-before B
- and there is no other side effect "between them" (between in terms of happens-before)

Compare and Swap Pattern

- `compare_exchange_weak` VS `compare_exchange_strong`
- `_weak` may spuriously fail (sometimes fail even if `value == old_value`)
- `_strong` won't spuriously fail but is more expensive
- use `_strong` only if computation of `new_value` is more expensive

```
int old_value = value.load();

while (true) {
    int new_value = old_value + 5;
    // be careful of the ABA problem!
    if (value.compare_exchange_weak(old_value, new_value)) {
        return true; // success
    }
    // failed, keep looping
}
```

ABA Problem

A location is read twice, has the same value for both reads, thus, it concludes that nothing happened in-between.

BUT, another thread could have interleaved, did some work, then reset the value back. Especially a problem because addresses might be reused!

Compare the object `(Node*, uintptr_t)` (where `uintptr_t` is some unique generation counter) instead of just `Node*`.

Use after free

Pointer points to memory that has been freed.

Solutions:

- recycle objects
- never free anything, i.e. leak all the memory
- mark objects for deletion while there are threads in functions that *could* read the objects; free all objects when no one is trying to read them
- use reference counting, i.e. `shared_ptr`
- use hazard pointers
  - each thread keeps a list of objects that it is modifying
  - these objects *cannot* be modified / freed by anyone

Compiling C++ Code

Remember to pass the flag -std=c++20 and enable POSIX threads (-pthread) when compiling your code. Other useful flags are:

- -O3: Enable optimisations
- -g: Enable debugging symbols
- -Wpedantic -Wconversion -Wall -Werror: Enable a reasonable set of warnings
- -c: Compile to object file instead of executable (useful for code snippets)
- -S: Compile to assembly
- -fsanitize=address: Compile with AddressSanitizer
- -fsanitize=thread: Compile with ThreadSanitizer
- -fsanitize=memory: Compile with MemorySanitizer (only works on clang)

```
clang++ -g -std=c++20 -pthread -O3 \
  -Wpedantic -Wall -Wextra -Wconversion -Werror \
  queue.cpp main.cpp \
  -o main
```

Debugging Action Plan

1. Run Valgrind memcheck
2. Run ThreadSanitizer (TSan)
3. Run AddressSanitizer (ASan)
4. Run Helgrind

Rule of 3

If you need a custom destructor, copy constructor or copy assignment operator, you probably need all 3.

Rule of 5 = Rule of 3 + Move Operations

NOTE: when a class only defines a **move** stuff, but not **copy** stuff => it can only be moved and not coped (e.g. `std::thread` and `std::unique_ptr`)

C++ Ownership

Owner is an **object** containing a pointer to an object allocated by *new* for which a *delete* is required.

Every object *should* have exactly one owner. Only the owner should destruct the data!

RAII - Resource Acquisition is Initialisation

- binds the life cycle of a resource (e.g. fd, socket, ...) to the lifetime of an object
- alloc in constructor
- dealloc in destructor

C++ Lifetime

- Begins when:
  - storage is obtained
  - initialisation is complete (constructor call is done)
- Ends when:
  - if non-class type: object is destroyed
  - if class type: destructor call starts
  - storage is released, or reused by an object that is not nested within it
- Lifetime is equal to or nested within the lifetime of its storage
- If the lifetime of a reference outlives the lifetime of the object, it is a dangling reference (pointing to an invalid object)

Lambda Expressions (Lambdas)

```
// lambda capture expressions
// & ⇒ capture by reference
// = ⇒ capture by value
// this ⇒ must be after &, =
cond.wait(lk, [=, this]{ return 4; })
```

C++ References, Pointers

- pointers
  - creates a new variable (the value is the address of the other variable)
    - new address, different from the original variable
  - can be reassigned
  - can have null value
  - can be used to "mutate" the original variable when passed into a function
- references
  - just an alias to a variable (nothing new is created!)
    - shares the same address as the original variable
  - cannot be reassigned
  - cannot be null
  - can be used to "mutate" the original variable when passed into a function
    - does not copy the object

std::thread

`get_id()` to identify a thread

> if `join()` and `detach()` are not called on the thread, main function will throw an exception!

- `join()` must be called exactly once
  - use `joinable()` to check
- `detach()`: take note of local variables passed into the thread (lifetime issue)

Use `std::thread:hardware_concurrency()` to obtain the number of threads. Don't run more threads than the hardware can support, this is called *oversubscription* (for CPU-bound tasks). Context switching will decrease performance. NOTE: you might need to keep one thread for the *master* process.

- `std::thread` is movable, but *not* copyable
- `std::thread` synchronises-with on creation and joining

```
// start a thread: using a function
void do_some_work();
std::thread t(do_some_work);

// start a thread: using a function object
class background_task {
public:
    void operator()() const {
        do_something();
    }
};
background_task f;
std::thread t(f); // works
std::thread t(background_task()); // DOES NOT works
std::thread t((background_task())); // works
std::thread t{background_task()}; // recommended

// start a thread: using a lambda expression
std::thread t([]{ // works
    do_something();
});

/// passing variables

// pass by value copies the string
void f(int i, std::string const& s);
std::thread t(f, 3, "ello"); // ok

char buffer[1024];
populate the buffer here
std::thread t(f, 3, buffer); // !! buffer might go out of scope before it is converted to std::string

std::thread t(f, 3, std::string(buffer)); // ok: explicitly convert -> placed on `general store`

void g(some_type& data);
std::thread t(g, std::ref(data)) // ok: explicitly cast to general store
```

Locks & Condition Variables & Pointers

Avoiding deadlock:

- Avoid nested locks (use `std::lock` to lock multiple mutexes)
- Avoid calling user-supplied code while holding a lock
- Acquire locks in the same fixed order

Some common patterns:

- hand-over-hand locking

Locks

- `std::mutex mu` - must be unlocked in the same thread!
  - `mu.lock()`
  - `mu.unlock()` - must call on every path (including exceptions!)
- `std::shared_mutex mu` (extension of `std::mutex` that allows concurrent reads)
  - `mu.lock()`, `mu.unlock()`, `mu.lock_shared()`, `mu.unlock_shared()`
- `std::lock_guard<std::mutex> lk{mu}`
  - RAII wrapper around mutex - acquires ownership of the mutex
  - `std::lock_guard lk{mu, std::adopt_lock}` - steals ownership of the lock (won't lock again)
- `std::unique_lock<std::mutex> lk{mu}`
  - can be deferred (and locked with `std::lock` later on): `std::unique_lock lk {mu, std::defer_lock}; std::lock(mu);`

    - movable to another unique_lock: `std::unique_lock lk1{mu}; std::unique_lock lk2{lk1};`
- `std::lock(mu1, mu2, ...)` - lock multiple locks with a deadlock avoidance algorithm
- `std::scoped_lock<std::mutex, std::mutex> lk{mu1, mu2}` - "strictly superior" version of `std::lock_guard`
    - RAII wrapper around `std::lock`
- `std::binary_semaphore mu{0}` - can be used across threads

Condition Variables

- `std::condition_variable cond` - only for `std::mutex`
    - `std::unique_lock lk{mu}` - use a unique_lock first (unlock method is required)
    - `cond.wait(lk, []{return is_ready();})` - waits until `is_ready()` returns *true*
    - `while (!is_ready()) cond.wait(lk)` - equivalent alternative
    - `cond.notify_one()`, `cond.notify_all()`
    - `cond.wait()`, `cond.wait_for(lk, duration, predicate)` (predicate or duration is up), `cond.wait_until(lk, end_time, predicate)` (predicate or end time is reached)
- `std::condition_variable_any cond` - for any mutex-like objects

While waiting, a condition variable can check the condition any number of times.

The waiting thread can spuriously wake up to reacquire the mutex to check the condition.

Smart Pointers

- `std::shared_ptr<T> data{std::make_shared<T>(std::move(value))}`
    - uses reference-counting
    - destroys the object when reference == 0
    - note: uses copy-semantics
    - note: the object inside is **not** thread-safe
    - note: use `std::weak_ptr` to avoid circular references (objects can never be deallocated)
- `std::unique_ptr<T> p{new T}`
- `std::weak_ptr<T> ptr{std::make_shared<T>(...)}` - smart pointer that holds a weak reference to an object managed by `std::shared_ptr` (must be converted to a `std::shared_ptr` to access the references object)
    - `ptr.lock()` - returns a `std::shared_ptr` object

Barriers

- `std::barrier b{3}` - reusable
    - associated with a thread (each thread can only call `arrive` *once*)
    - `b.arrive_and_wait()`, `b.arrive()`, `b.wait()`
- `std::latch l{3}` - NOT reusable
    - associated with an "item" (threads can call `arrive` multiple times)
    - note: the number is of type `std::ptrdiff_t`
- `std::atomic_thread_fence(std::memory_order_acquire)` - pair with an acquire in another thread
    - An atomic_thread_fence with memory_order_release ordering prevents all preceding reads and writes from moving past all subsequent stores
    - Useful for when there aren't any atomic writes

Go

Go channels internal design: https://docs.google.com/document/d/1yIAYmbvL3JxOKOjuCyon7JhW4cSv1wy5hC0ApeGMV9s/pub

Channels = thread-safe MPMC queue

`select` psuedo randomly picks the case to avoid starvation.

Communicating Sequential Processes - CSP

- concurrency: structure a program by breaking it into pieces that can be executed independently
- communication: coordinate the independent executions

Safe Concurrency

- Immutable data
- Data protected by confinement

- Ad-hoc confinement: data is modified only from **one** goroutine, even though it is accessible from multiple goroutines
  - not very good, requires static analysis (compiler doesn't help)
- Lexical confinement: restrict the access to shared locations
  - expose only the reading / writing handle of the channel (compiler can type check)
  - expose different slices of the array to different goroutines
    - recall that slicing a slice is basically free (data isn't copied!)

Goroutine Scheduling

runtime multiplexes goroutines onto OS threads with automatic *M:N* mapping

G (goroutine), M (OS thread), P (CPU process)

concurrency is decoupled from parallelism!

Goroutine is a special class of coroutine (concurrent subroutine)

- when a goroutine blocks (e.g. because of a syscall), other goroutines are not blocked
- go's runtime can suspend goroutines (preemptable)
- Goroutines are really cheap compared to OS threads
  - couple of kilobytes, grows when needed
  - prefer finer grain concurrency
- Goroutines are **not** garbage collected!
- Go runtime will transfer data from the stack to the heap if a reference to the data is passed to another goroutine
  - there is no guarantee on whether a particular data is on the stack or the heap

Each P has:

- its own decentralised work queues (deques)
  - centralised queue has problems with locality and imbalances
- at a fork point, add tasks to the tail of deque associated with the thread
  - when thread is idle, steal work from the head of some other P's deque
- at a join point that cannot be realised yet (the synchronised-with goroutine has not completed), pop work off tail of own deque

Channel Blocking Rules

Operation	Channel State	Result
Read	nil	block
	open and not empty	Value, true
	open and empty	block
	closed	ZeroValue, false
	write only	compilation error
write	nil	block
	open and full	block
	open and not full	writes value
	closed	panic
	receive only	compilation error
close	nil	panic
	open and not empty	closes channel; reads succeed until channel is drained (then, ZeroValue is returned)
	open and empty	closes channel; reads return ZeroValue
	closed	panic
	receive only	compilation error

Iterating over a channel (using `x := range ch`) only ends when the channel is closed.

```
// psuedo-randomly selects any channel with an item
select {
case ← c:
    ...
case ← time.After(1 * time.Second):
```

```
    ...
}
```

Channel Ownership

Not enforced by the compiler, but recommended to follow these rules. Utilise unidirectional channels (read-only, write-only) when possible for compiler to assist.

Single owner should:

- instantiate the channel
- perform writes
- close the channel
- pass ownership to another goroutine
- ^ only expose the channel to consumers via a reader channel

Multiple consumers should:

- know when a channel is closed
- responsibly handle blocking for any reason

To avoid goroutine leaking: the owner should be responsible for ensuring that the goroutine can / will stop.

Go Memory Model

Go Happens Before

Within a goroutine, process order is maintained (sequenced before).

Execution order observed by different goroutines are different.

To guarantee that a particular write w is read by a read r:

- w happens before r
- all other writes to that variable either happens before w or after r

Happens before = sequenced before (process order) + synchronised before

Go Synchronised Before

- `go` statement is synchronised before the goroutine's execution begins
- exit of a goroutine is **not** synchronised before anything
- a *send* on a channel is synchronised before the completion of the corresponding *receive*
- *closing* of a channel is synchronised before a receive that returns a zero value (because the channel is closed)
- receive from a unbuffered channel is synchronised before send on the channel
- k-th receive on a channel with capacity C is synchronised before (k+C)-th send from that channel completes
  - otherwise, there's "no space" in the buffer to send

Go Patterns / Channel Patterns

- Serialiser
  - if in-order: forward it
  - if not: buffer and send when it is supposed to be sent
- Fan-in (necessary for centralising results), fan-out (useful for distributing intense work to many threads)
  - difficult to handle fan-in if there's dependencies
  - fanout should use `runtime.NumCPU()` number of goroutines.
- Higher Order Channels
  - channel of channels
- Pipelining
  - each stage is a group of goroutines running the same function
  - in each stage, the functions take input from upstream via inbound channels, then send the results downstream via outbound channels
  - useful for resource-constrained parallelism while maintaining separation of concerns
- for-select

Confinement

- ad-hoc confinement: data is modified only from one goroutine, but accessible from multiple
  - requires static analysis to ensure safety

- lexical confinement: restricts access to shared locations
  - e.g. expose only read or write handles of channel
  - e.g. expose only array slices

Go Stack vs Heap

As of Go 1.17, Go runtime will allocate the elements of slice x on stack if the compiler proves they are only used in the current goroutine and N <= 64KB:

```
var x = make([]byte, N)
```

And Go runtime will allocate the array y on stack if the compiler proves it is only used in the current goroutine and N <= 10MB:

```
var y [N]byte
```

Then how to allocated (the elements of) a slice which size is larger than 64KB but not larger than 10MB on stack (and the slice is only used in one goroutine)?

Just use the following way:

```
var y [N]byte
var x = y[:]
```

In fact, we could allocate slices with arbitrary sum element sizes on stack.

```
const N = 500 * 1024 * 1024 // 500M
var v byte = 123

func createSlice() byte {
    var s = []byte{N: 0}
    for i := range s { s[i] = v }
    return s[v]
}
```

Changing 500 to 512 makes the program crash.

https://stackoverflow.com/a/69187698

Goroutine Scheduler: Work Stealing

- G: goroutine (task)
- M: thread
  - maintains a local runnable queue of Gs
- P: processor
- there is a global runnable queue of Gs

Rules:

- sometimes (with a certain probability): check the global runnable queue
- if not found: check local queue
- if not found:
  - try to steal from other Ps (takes half the tasks)
  - if not, check global runnable queue
  - if not found, poll network

Rust

Two goals:

- Safety in systems programming
  - systems programming languages: C / C++ / Fortran
- Painless / fearless concurrency

Strong safety guarantees: no segmentation faults, no data races, expressive type system

- without performance compromise: no garbage collector and runtime (runs directly on hardware)
- same level of performance as C / C++

Rust Memory Safety

Mutating aliased pointers (pointers that point to the same memory, i.e. element + whole array) is a problem. Rust avoids this through borrow checking (many shared borrows, 1 mutable borrow / owner)

- Ownership does not allow aliasing!
- Immutable borrow allows aliasing, but no mutation
- Deep copying of data is explicit using `clone()`
  - note: C++ copy constructor does a deep copy
- Ownership prevents double-free
  - The owner frees
- Borrowing prevents use-after-free

Data races are prevented because sharing + mutation is prevented.

```
fn main() {
    publish(&book);
}

// immutable borrow (shared borrow)
// book is a immutable reference
fn publish(book: &Vec<String>) { ... }

// mutable borrow (only one mutable borrow is allowed at one time)
fn publish_mut(book: &mut Vec<String>) { ... }
```

`move` keyword: closure takes ownership of the values **it uses**

Rust references = C++ pointers (with some asterisks)

- rust references are never invalid, i.e. never null
  - always safe to dereference!
- rust variables cannot be read before initialisation
  - note: C++ throws a warning

Rust analyses lifetime with usage of the variable. The lifetime of a variable ends when it is last used. (live-variable analysis)

Interior mutability

Unsolvable

- deadlock, livelock, etc
  - cannot be detected at compile time
- some memory leaks
  - rust only solves some memory leaks
  - circular references can still leak memory
  - https://tiemoko.com/blog/blue-team-rust/

Mutex & Reference Counting & Atomics

- Reference counting `Rc<T>` - `Rc::new(vec![1,2,3])`
  - single thread
- Atomic reference counting `Arc<T>` - `Arc::new(vec![1,2,3])`
  - allows shared (immutable!) references, can be used across threads
  - `Arc::clone(&data)` where `data` is an `Arc<T>`
- `Mutex<T>` - `Mutex::new(0)`
  - usually use with `Arc<T>`: `Arc::new(Mutex::new(0))`
    - clone it, then, move it into spawned thread
  - `let data = Arc::new(Mutex::new(0));`
  - `let mut data = data.lock().unwrap();`
  - `*data += 1;`

- `AtomicI32`, `AtomicU32`, ... (not generic!)
  - `let number = AtomicUsize::new(10);`
  - `let prev = number.fetch_add(1, SeqCst);`
  - `number.load(SeqCst);`
  - `number.store(2, SeqCst);`
  - `let prev = number.swap(2, SeqCst);`
  - ordering: SeqCst, Release, Acquire, AcqRel, Relaxed
- `Semaphore`
  - `let sem: Semaphore::new(1);`
  - `let _lk = sem.acquire();` - RAII style acquire
    - name cannot be `_` to avoid it being dropped immediately

Traits

- Send - transferred across thread boundaries
- Sync - safe to share references between threads
  - Type T is Sync <=> &T is Send
- Copy - safe to memcpy (for built in types)

Multi-Producer, Single-Consumer FIFO Queue

```
let (tx, rx) = mpsc::channel();
let tx2 = tx.clone();
// rx cannot be cloned

thread.spawn(move || tx.send(4));
thread.spawn(move || tx2.send(5));

// will print 4 and 5, in either order
println!("{:?}", rx.recv);
println!("{:?}", rx.recv);
```

Rust Libraries

Crossbeam

- scoped threads (now in std)
  - scoped threads can borrow non-`'static` data because the scope guarantees all threads are joined at the end of the scope
  - all threads spawned within the scope (that weren't manually joined) will be automatically joined before the function returns
- message passing with multiple-producer & multiple-consumer
  - with exponential backoff
  - with bounded channels

```
let mut a = vec![1, 2, 3];
let mut x = 0;

thread::scope(|s| {
    s.spawn(|| {
        println!("hello from the scoped thread");
        // can borrow `a` and mutably borrow `x` here
        dbg!(&a);
        x += a[0] + a[2];
    });
    println!("hello from the main thread");
});
```

Rayon

Data-parallelism library, similar to OpenMP (but not with compilation directives)

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

// no mutation
```

```rust
fn sum_of_squares(input: &[i32]) -> i32 {
    input.par_iter()
         .map(|i| i * i)
         .sum()
}

// mutation
fn increment_all(input: &mut [i32]) {
    input.par_iter_mut()
         .for_each(|p| *p += 1);
}
```

Aynsc Rust

Non-blocking I/O

- requires multiplexing of sockets
  - using `epoll` syscall (requires support from OS!)
- Async functions have no stacks (also called stackless coroutines)
  - Executor thread still has a stack (but not used to store state when switching between async tasks)
  - All states are contained in the Future
- Rust bans recursion in async functions
  - Future returns needs to have a fixed size known at compile time
    - the size is the `union` of all the possible states
  - Not technically impossible to implement, but it isn't supported
- Nearly optimal in terms of memory usage + allocations
  - lower overhead compared to hand-written non-assembly code
  - zero cost abstractions: abstraction layers disappear at compile time (don't exist at runtime)

```
epoll ⇒ [7, 12, 15]

read(7)  ⇒ 0101010011... (more data but we return early)
read(12) ⇒ 0101010011...
read(15) ⇒ 0101010011... X (no more data to read from fd 15)
```

State Management

need to maintain state for each conversation

- using *Futures* (JavaScript uses Promises)
  - Futures oversees a specific operation and its state (not the steps!)
  - execution is done by the executor thread
  - represents a value that will exist sometime in the future
- Event loop = runtime for Futures
  - keeps polling Future until it is ready
  - user-space scheduler for futures
  - can have one or more threads polling from the event loop

Future Trait

- executor thread should call `poll(self, ctx Context)` on the Future to init it
  - `ctx` includes a `wake()` function which will wake up the executor that polled (implemented using syscalls)
  - after the executor is woken, it can use `ctx` to see which Future can be polled again
- will run code until it can no longer progress
  - if the future is done, returns `Poll:Ready(T)`
  - if the future needs to wait, return `Poll::Pending`

> do NOT sleep / block the thread!! (block the future instead)

Executors

Tokio is an executor runtime

- responsible for calling `poll()` on futures -> otherwise, no progress is made!
- if no futures can make progress, the executor goes to sleep until it is woken with `wake()`

- can be single-threaded or multi-threaded
  - so, shared data must be protected

Future Composition

```
// bad example
let future = placeOnStove(meat)
    .then(|meat| cookOneSide(meat))
    .then(|meat| flip(meat))
    .then(|meat| cookOneSide(meat));
```

- need to pass variables to every function in the chain even though we only need it at the very end

```
// use async keyword
async fn addToInbox(email_id: u64, recipient_id: u64) -> Result<(), Error> {
    // this sequentially computes this 4 steps
    // use "join" to do things in parallel
    let message = loadMessage(email_id).await?;
    let recipient = get_recipient(recipient_id).await?;
    recipient.verifyHasSpace(&message)?;
    recipient.addToInbox(message).await
    // .await waits for a future and gets the value
}
```

- not yet started (future is created, but not yet `poll()`-ed)
- awaiting for `loadMessage`
- awaiting for `get_recipient`
- awaiting for `addToInbox`
- future has completed

`.await` can only be called in an async function or async block

`let a, b, ... = tokio::join!(future1, future2, ...)` - join macro that joins all the (variable number of) futures provided

- can be destructed into the results
- stdlib version of `join!` is still in nightly

Async Main Function

```
#[tokio::main]
async fn main() {}
```

`tokio::main` macro sets up the executor and polls the future returned from the main function.

When to use Async?

- extremely high degree of concurrency
  - just use threads if the number of tasks is small
- primarily I/O bound
  - context switching is extensive only if you're using a bit of the time slice
    - aka I/O bound stuff
  - CPU bound code would prevent the executor from running other tasks

Async in other languages

- Python
  - still fairly new
  - single-threaded :/
  - executor and Futures are separate
    - possible to optimise / tune them separately in the future!
- JavaScript
  - similar concept to Rust, but less efficient because of dynamic memory allocation
- Go
  - goroutines are asynchronous tasks, but not stack-less!
  - uses resizable stacks, because Go is garbage collected

- C++20
  - just got stack-less coroutines (super new)

https://reberhardt.com/cs110l/spring-2021/

C10K problem (handling 10 thousand network connections): https://en.wikipedia.org/wiki/C10k_problem

---

Checking Concurrent Programs

Model Checking / Formal Methods

- build a mathematical model using a special Domain Specific Language (DSL)
- check the model for problems (manual or automatic)
  - are all the constraints met?
  - does anything unexpected happen?
  - does it deadlock?
- why?
  - check things make sense before starting the (costly) implementation
  - prove *certain properties* for existing code
  - *aggressive optimise* the code without compromising correctness
- pros & cons
  - pros
    - rigorous
    - verify all traces exhaustively
    - produce a system run that violates the requirement
  - cons
    - the specification used might be faulty
    - tedious in coming up with a complete specification
      - might be too complex
    - time consuming

once it's ok, *then* write the code

Model Checkers for Concurrent Programs

TLA+ (TLC) - Temporal Logic of Actions+

- T: temporal (time)
- L: boolean logic
- A: actions (state machines)
- +: (extra stuff)

Proposed by Leslie Lamport in 1999

- focuses on temporal properties
- good for modelling concurrent systems + distributed systems
- can't generate code
- uses "simple mathematics" (because engineers can't do math)
- finds all possible states up to some number of steps
  - examines the states for violations of certain invariances, e.g. safety (bad things won't happen), liveness (good things eventually happen)
  - checks all possible concurrent interleaving!!

```
// TLA definition
operator(a,b,c) == (a /\ b) \/ (a /\ ~c)

// C equivalent: (a && b) || (a && !c)
```

```
// `Next` transitions `state` from "hello" to "goodbye" (`state'`)
Next == state  = "hello" /\ state' = "goodbye"
```

```
var x = 1;
x = 2;
x = 3;

Init == x=1
Step1 == x=1 /\ x'=2
Step2 == x=2 /\ x'=3
Done == x=3 /\ UNCHANGED x  // required to prevent deadlock
Next == Step1 \/ Step2 \/ Done

// overall specification that guarantees always Next or stutter (when in Init)
// note: /\ (not \/)
Spec == Init /\ [](Next \/ UNCHANGED x)
Spec == Init /\ [][Next]_x  // equivalent syntax
Spec == Init /\ [][Next]_x /\ WF_x(Next)  // with "weak" fairness
```

> all variables **must** be accounted for in an action

`UNCHANGED x` is known as *stuttering*

Properties

TLA+ can determine if these properties hold

1. Always true
   - `[] (x > 0)`
2. Eventually true
   - `<> (x = 2)`
3. Eventually always (eventually true + stays there)
   - `<>[] (x = 3)`
4. Leads to (if it becomes LHS, eventually will be RHS)
   - `(x=2) ~> (x=3)`

Set Theory

Can be used to define concurrency, e.g. multiple producers (in producer-consumer problem)

```
// define a set
{1,2,3}

// define a set by predicate p
{e \in S : p}

// for all `e` in `S`, predicate `p` is true
\A e \in S : p

// exists `x` in `S` where predicate `p` is true
\E x \in S : p
```

Coq Proof Assistant

- generates oCaml, Haskell, Scheme
- good for interactive proof methods

Alloy (alloy analyser)

- focuses on relational logic
- good for modelling structures

https://en.wikipedia.org/wiki/TLA%2B

Distributed System Challenges

- no global clock / ordering of events
- CAP theorem (consistency, availability, partition tolerance)

Stuff Implemented in Class

- concurrent queue with coarse-grained mutexes (tutorial 1)
- lock-free (non-linearisable) MPMC queue (tutorial 4)
  - non-linearisable => cannot create a serial ordering of the operations on the queue
  - `push`, `try_pop`
- lock-free stack (tutorial 4)

---

io_uring (2023 PYP)

```
struct Request {
    int client_fd;
    data_t data;
    res_t result;
}

class ConcurrentRing {
private:
    LockFreeQueue<Request> queue;
    std::counting_semaphore write{size};
    std::counting_semaphore read{0};

public:
    void submit_request(Request req) {
        write.acquire();
        queue.push(req);
        read.release();
    }

    Request retrieve_request() {
        read.acquire();

        std::optional<Request> req;
        while (true) {
            req = queue.try_pop();
            if (req) break;
        }

        write.release();
        return req;
    }
}

// point D (init)
// W worker threads
for (int i = 0; i < W; i++) {
    std::thread([&]() {
        while (true) {
            Request req = SQ.retrieve_request();
            req.process();
            CQ.submit_request(req);
        }
    });
    std::thread([&]() {
        while (true) {
            Request req = CQ.retrieve_request();
            send(req.client_fd, req);
        }
    });
}

// point E (on receiving a client)
std::thread([&]() {
    data_t data = read(client_fd);
    while (data) {
        Request req{client_fd, data};
        SQ.submit_request(req);
        data = read(client_fd);
```

```
    }
});
```

```
struct Request {
    conn net.Conn
    resp Response
}
struct Response {
    conn net.Conn
    resp Data
}

// point F (init)
submitQueue := make(chan Request)
completionQueue := make(chan Response)

// spawn `SIZE` worker goroutines
for i := 0; i < SIZE; i++ {
    go func() {
        req := <-submitQueue
        completionQueue <- req.Process()
    }()
    go func() {
        res := <-completionQueue
        res.conn.Write(resp)
    }()
}

// point G
go func() {
    req := Request{conn: conn, resp: nil}
    submitQueue<-req
}()
```

```
// point H
loop {
    // for each client
    let (stream, _) = listener.accept().await?;

    // thread: handles incoming requests
    tokio::spawn(async move {
        loop {
            let mut buf = Vec::new();
            stream.read(&mut buf).await;
            sq_tx.send(IoOperation::Read(stream, buf)).await;
        }
    });
    //
    tokio::spawn(async move {
        while let Some(op) = sq_rx.recv().await {
            tokio::spawn(async move {
                let res = process(op.data).await;
                cq_tx.send(IoOperation::Write(op.stream, res)).await;
            });
        }
    });
    // thread: sending back the result
    tokio::spawn(async move {
        while let Some(op) = cq_rx.recv().await {
            op.stream.write_all(op.data).await;
        }
    });
}
```

Advantages:

- perform computation in parallel, maximising the use of multiple cores

Disadvantages:

- coding complexity
- context switching
- inter-thread communication
- increased memory usage

Quantification:

- measuring execution time (or speedup), compared against sequential program