Inside Perl 6
Concurrency
The guts beneath the goodness
Make the easy things easy...

...and the hard things possible
Make the easy things easy...

...and the hard things possible
Make the easy things easy...

...and the hard things possible

- start, await
- .hyper.map(...)
- supply/react/
- whenever
- Threads
- Mutexes
- Condition Variables
- Semaphores
- Atomic Operations
In this session, we'll work from the hardware up to the higher-level constructs available in Perl 6.

We'll build simplified versions of those constructs, to understand something about how they work.
In this session, we'll work from the hardware up to the higher-level constructs available in Perl 6.

We'll build simplified versions of those constructs, to understand something about how they work.
Of course, the ones provided by Perl 6 have been engineered for better...

Speed
Memory use
Error reporting
Debaggability
Robustness
The CPU
Multiple levels of cache memory, some per core, some shared

Intel Core i7 has per-core L1 and L2, and shared L3 cache
Caches play a critical role in multi-threaded program performance.

Whenever data held by more than one core's cache is updated, all other cores with that data cached must invalidate it.

This is expensive!
Therefore...

Prefer thread-local, unshared data

When sharing data, share immutable data (for the CPU's and your sanity!)

Try to avoid contention over data (remember that locks are data too)
A thread is an OS-provided mechanism for running code on a CPU core.

In Perl 6, a thread is represented by the Thread class.
What will the output of this code be?

```perl
my @threads = do for 1..5 -> $id {
    Thread.start: {
        say "Hi from thread $id";
        sleep 1;
        say "Bye from thread $id"
    }
}
@threads>>.join;
```
How about this?

```perl
my int $i = 0;
my @threads = do for 1..5 -> $id {
    Thread.start: {
        $i++ for ^100000;
    }
}
@threads>>.join;
say $i;
```
Always remember:

There is no execution ordering between threads except that which you explicitly arrange for.

Nothing a thread does is atomic or uninterruptible unless you explicitly arrange for it.
CPUs provide atomic operations. Perl 6 provides access to them.

```perl
my atomicint $i = 0;
my @threads = do for 1..5 -> $id {
    Thread.start: {
        $i++ for ^100000;
    }
}
@threads>>.join;
say $i;
```
Far more powerful, however, is the atomic compare and swap operation, commonly known as "CAS"
CAS is provided by the hardware, but we can imagine it like this - with the guarantee that it is atomic

sub cas($reference is rw, $expected, $new) {
    my $seen = $reference;
    $reference = $new if $seen == $expected;
    return $seen;
}
Amazingly, we can make any data structure we want atomically updateable using CAS.*

* If we follow the rules. Very, very carefully.
Let's build a concurrent stack.

One that we can push to and pop from multiple threads "at once".

Without locks!

class ConcurrentStack {
    ...
}

It's a linked list of Node objects. They're immutable. The only mutable thing will be $!head.

class ConcurrentStack {
    my class Node {
        has $.value;
        has Node $.next;
    }
    has Node $!head;

    method push($value --> Nil) { ... }

    method pop() { ... }
}

method push($value --> Nil) {
    loop {
        my $next = $!head;
        my $new = Node.new: :$value, :$next;
        last if cas($!head, $next, $new) === $next;
    }
}
method pop() {
    loop {
        my $cur = $!head;
        fail "Stack is empty" without $cur;
        if cas($!head, $cur, $cur.next) === $cur {
            return $cur.value;
        }
    }
}

The pop method is similar, except it can fail due to an empty stack
This "loop" structure is so common, Perl 6 provides a form of CAS that takes a block computing the new value based on the current one, and does the retry loop for you.
method push($value --&gt; Nil) {  
cas $!head, -&gt; $next {  
    Node.new: :$value, :$next  
  }  
}

method pop() {  
    my $taken;  
    cas $!head, -&gt; $current {  
      fail "Stack is empty" without $current;  
      $taken = $current.value;  
      $current.next  
    }  
    return $taken;  
}
Did you ever think about how a lock is implemented?
Using CAS!

Well, at least, somewhat.
class SpinLock {
    has atomicint $!held = 0;

    method lock(|-- -> Nil) {
        while cas($!held, 0, 1) != 0 {  }  
    }

    method unlock(|-- -> Nil) {
        cas($!held, 1, 0) or die "Lock was not held";
    }
}
And yes, it really works...

```perl
my int $i = 0;
my $lock = SpinLock.new;
my @threads = do for 1..5 -> $id {
  Thread.start: {
    for ^100000 {
      $lock.lock();
      $i++;
      $lock.unlock();
    }
  }
}

@threads>>.join;
say $i;
```
Unfortunately, for many cases, this kind of lock also *really* sucks.

Why?
Observe the CPU usage of this:

```perl
my int $i = 0;
my $lock = SpinLock.new;
my @threads = do for 1..5 -> $id {
    Thread.start: {
        $lock.lock();
        $i++ for ^10000000;
        $lock.unlock();
    }
}
@threads>>.join;
say $i;
```
A spinlock is only good when we are really sure that blocking will last for a very short amount of time.

Normally, we want to get the OS scheduler involved.

Just like Perl 6's Lock class does.
This has far lower CPU utilization:

```perl
my int $i = 0;
my $lock = Lock.new;
my @threads = do for 1..5 -> $id {
    Thread.start: {
        $lock.lock();
        $i++ for ^10000000;
        $lock.unlock();
    }
}
@threads>>.join;
say $i;
```
This has far lower CPU utilization:

```perl
my int $i = 0;
my $lock = Lock.new;
my @threads = do for 1..5 -> $id {
    Thread.start: {
        $lock.lock();
        $i++ for ^10000000;
        $lock.unlock();
    }
}
@threads>>.join;
say $i;
```

But never write code like this! Why?
This form won't "leak" the lock should an exception occur:

```perl
my int $i = 0;
my $lock = Lock.new;
my @threads = do for 1..5 -> $id {
    Thread.start: {
        $lock.protect: {
            $i++ for ^10000000;
        }
    }
}
@threads>>.join;
say $i;
```
But Lock is still hard to use correctly:

Must remember to acquire the lock

Must not leak lock-protected data

Risk of deadlocks due to circular lock dependencies
It turns out that **OO done right** (which it too rarely is, alas) **can help!**
Use a Lock to protect object state:

class Index {
    has $!lock = Lock.new;
    has %!index{Str};

    method add(Str $word, Str $document --> Nil) {
        $!lock.protect: { ... }
    }

    method lookup(Str $word --> List) {
        $!lock.protect: { ... }
    }

    method elems(---> Int) {
        $!lock.protect: { ... }
    }
}
Methods that only mutate, or that return immutable values, are easy:

```ruby
method add(Str $word, Str $document --> Nil) {
  !$lock.protect: {
    %!index{$word}{$document} = True;
  }
}

method elems() {
  !$lock.protect: {
    %!index.elems
  }
}
```
Those returning more interesting data must ensure it is completely independent of the object's state, which the lock is there to protect.

```ruby
method lookup(Str $word) {
    !$lock.protect: {
        with %!index{$word} { .keys.eager }
        else { () }
    }
}
```
But surely we can do better than wrapping a `protect` call around all of our method bodies?

Indeed we can. `OO::Monitors` gives us a `monitor` keyword to use in place of `class`, and enforces the locking for us.
use OO::Monitors;

monitor Index {
    has %!index{Str};

    method add(Str $word, Str $document --> Nil) {
        %!index{$word}{$document} = True;
    }

    method lookup(Str $word) {
        with %!index{$word} { .keys.eager } else { () }
    }

    method elems() {
        %!index.elems
    }
}
Some more problems:

A thread is a pretty heavyweight unit of parallel work

Leaves us to convey results or errors back to the code that wants them
Let's build a thread pool!

Work is put into a queue

Workers in the pool compete to take tasks out of the work queue and complete them
Condition variables efficiently block a thread until a condition is met

class WorkQueue {
    has Callable @!work;
    has !$lock = Lock.new;
    has !$not-empty = !$lock.condition();

    method enqueue(&task --> Nil) {
        ...
    }

    method dequeue(---> Callable) {
        ...
    }
}
method enqueue(&task --&gt; Nil) {
  !$lock.protect: {
    my $was-empty = @$work == 0;
    push @$work, &task;
    !$not-empty.signal if $was-empty;
  }
}

method dequeue(--&gt; Callable) {
  !$lock.protect: {
    while @$work == 0 {
      !$not-empty.wait;
    }
    @$work.shift
  }
}
A worker sits in a loop, taking work from the queue and doing it

```
sub start-worker(WorkQueue $queue) {
    Thread.start: {
        loop {
            my &task = $queue.dequeue;
            task();
        }
    }
}
```
What output will this produce?

```perl
my $queue = WorkQueue.new;
start-worker($queue) xx 4;

for 1..10 -> $i {
    $queue.enqueue: {
        say "Task $i starting";
        sleep 0.5;
        say "Task $i done"
    }
}

sleep;
```
And here's how we use the built-in Perl 6 thread pool scheduler instead:

```perl
for 1..10 -> $i {
    $SCHEDULER.cue: {
        say "Task $i starting";
        sleep 0.5;
        say "Task $i done"
    }
}
sleep;
```
In reality...

Number of workers scaled by CPU core count and demand

Separate queues for stream-y data (to give thread affinity), time-sensitive events, and general work
And also...

The work queue has separate head and tail locks to reduce contention. Queue is implemented at VM level, such that we can push I/O events, timer events, signals, etc. into it.
But how can we more conveniently convey completion and a result, or the failure of, queued work?

A Promise is one way.

Let's build one!
A Promise starts out Planned, and can either be Kept or Broken

class SimplePromise {
    enum State <Planned Kept Broken>;
    has State $.state = Planned;
    has $!result;
    has $!lock = Lock.new;
    has $!completed = $!lock.condition();

    method keep($result --> Nil) { ... }
    method break(Exception $cause --> Nil) { ... }
    method result() { ... }
}
Keeping the Promise (note that we signal_all as many things may wait on its completion):

```ruby
method keep($result --> Nil) {
    !$lock.protect: {
        unless !$!state == Planned {
            die "Too late to keep";
        }
        !$result = $result;
        !$state = Kept;
        !$completed.signal_all();
    }
}
```
The result method blocks on the Promise being kept or broken:

```plaintext
method result() {
    $!lock.protect: {
        while $!state == Planned {
            $!completed.wait();
        }
        if $!state == Kept {
            $!result
        } else {
            $!result.rethrow
        }
    }
}
```
We can now implement `start`:

```perl
sub simple-start(&code) {
    my $p = SimplePromise.new;
    $*SCHEDULER.cue: {
        $p.keep(code());
        CATCH {
            default {
                $p.break($_);
            }
        }
    }
    return $p;
}
```
In reality...

Protects against double keep/break

Some tricks to reduce locking

Fancier error reporting

But the biggest difference is await...
The problem:

If calling `result` blocks a pool thread, it can't do anything else.

Can spawn extra threads, but this won't scale to tens of thousands of outstanding awaits.
sub merge-sort(@values, $from = 0, $elems = @values.elems) {
    if $elems > 1 {
        my $divide = ($elems / 2).ceiling;
        merge
        merge-sort(@values, $from, $divide),
        merge-sort(@values, $from + $divide, $elems - $divide)
    } elsif $elems == 1 {
        (@values[$from],)
    } else {
        Empty
    }
}
Parallelize it!

```perl
sub parallel-merge-sort(@values, $from = 0, $elems = @values.elems) {
    if $elems > 500 {
        my $divide = ($elems / 2).ceiling;
        my ($left, $right) = await
            (start parallel-merge-sort(@values, $from, $divide)),
            (start parallel-merge-sort(@values, $from + $divide, $elems - $divide));
        merge $left, $right
    }
    else {
        merge-sort @values, $from, $elems
    }
}
```
Perl 6.c vs. Perl 6.d

In 6.c, this spawns a ton of threads. If there's really a lot of elements, it could reach the pool's upper limit.

And Perl 6.d, it spawns threads up to the number of CPU cores. No risk of deadlocking due to running out.
An `await` on a thread pool worker takes a continuation. Schedules it to be resumed - quite possibly on a different real thread - once the result is available.
Finally...

A Promise is fine for a single value produced asynchronously.

But what about streams of asynchronous values, like timer ticks, GUI events, or data from a socket?
That's what a Perl 6 Supply is for.

It's just the observer pattern, really.
The Three Events

**Emit:** an event (packet, timer tick...)

**Done:** successful end of stream

**Quit:** exception end of stream

```plaintext
role SimpleTappable {
  method tap(&emit, &done, &quit) { ... }
}
```
A Tap

A subscription, with an optional callback upon close (unsubscription)

class SimpleTap {
    has &.on-close;
    method close(--> Nil) {
        .() with &!on-close;
    }
}

The Supply wrapper

Holds a Tappable implementation and delegates to it

class SimpleSupply {
    has SimpleTappable $.tappable is required;

    my constant DISCARD = -> $ {{}};
    my constant NOP = -> {{}};
    my constant DEATH = -> $ex { $ex.throw };
    method tap(&emit = DISCARD, &:done = NOP, &:quit = DEATH) {
        $!tappable.tap(&emit, &done, &quit)
    }

    # Many built-in methods here
}
my class Interval does SimpleTappable {
    has $.scheduler;
    has $.interval;
    has $.delay;

    method tap(&emit, &, &) {
        my $i = 0;
        my $cancellation = (!$!scheduler.cue(
            { emit($i++) },
            :every($!interval), :in($!delay)
        );
        SimpleTap.new(on-close => { $cancellation.cancel });
    }
}
method interval($interval, $delay = 0, :-$scheduler = $*SCHEDULER) {
    SimpleSupply.new:
        tappable => Interval.new(:$interval, :$delay, :$scheduler)
}
my class Map does SimpleTappable {
    has $.source;
    has &.mapper;
    method tap(&emit, &done, &quit) {
        my $source-tap = $!source.tap: &:done, &:quit, {
            emit(&!mapper($$_));
            CATCH {
                default {
                    $source-tap.close;
                    quit($$_);
                }
            }
        };
        SimpleTap.new(on-close => { $source-tap.close })
    }
}
method map(&mapper) {
    SimpleSupply.new:
        tappable => Map.new(source => self, &:mapper)
}
That's enough for reactive fizzbuzz

```perl
sub fizzbuzz($v) {
    $v %% 3 && $v %% 5 ?? 'fizzbuzz' !
    $v %% 3 ?? 'fizz' !
    $v %% 5 ?? 'buzz' !
    $v
}
my $tap = SimpleSupply
    .interval(0.3)
    .map(*+1)
    .map(&fizzbuzz)
    .tap(&say);
sleep 5;
$tap.close;
```
In reality...

Supply concurrency control is complex enough we'd need another talk this length to cover its implementation in detail.

Lots of trickiness around recursive and synchronous messaging.
In closing...
Perl 6 provides access to concurrency and parallelism primitives

However, most of the time, we're better off building our applications using the high-level things built in terms of them.
Building those higher-level things isn't simple. But it's complexity that we take out of your code.

At the same time, a basic idea of what they are doing can be helpful.

I hope this talk has provided that.
Thank you!

Questions?