Composable Concurrency in Perl 6

Jonathan Worthington

A short history of pain

I've spent a lot of time working with C# and .Net. There, multi-threaded apps are common, partly because you can't tie up the main UI thread in a GUI app, and partly due to web applications.

Sadly, correctness has been less common.

The majority of code I reviewed for thread-safety had at least one locking bug or race condition – or was oblivious to threading!

Things are improving…

In recent years, mainstream languages and frameworks have taken in that most developers will never build robust software out of locks. It's just too hard.

So, we've been seeing dozens of innovations that help make things better.

My goal: make sure Perl 6 gets the best of them, *and* makes them work together!

First, some definitions…

How should we define…

Asynchrony?
Parallelism?
Concurrency?

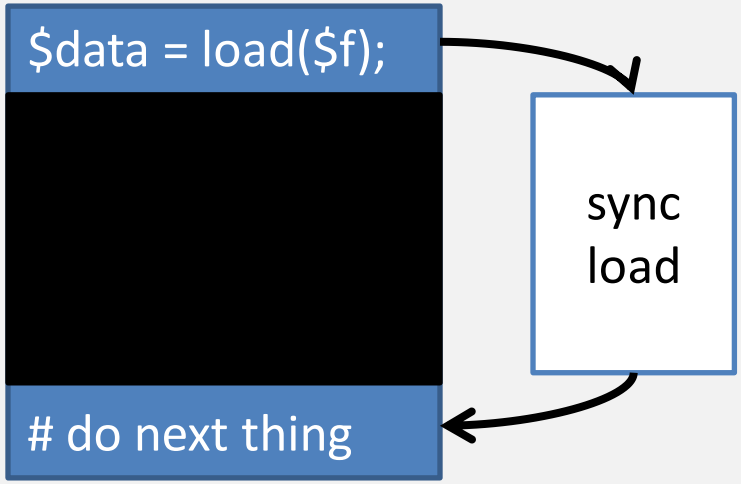
Are they the same? Different?

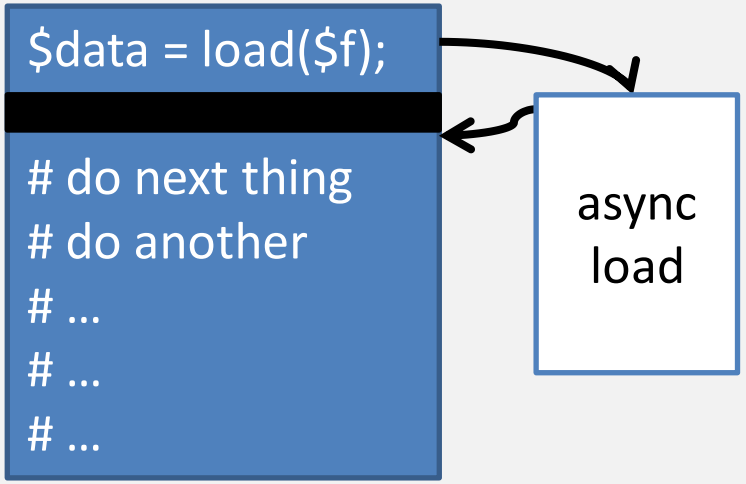
Getting everybody to agree on a definition is hard, but let me give you the ones I'll use for the duration of this talk…

Asynchrony

Synchronous is "the normal thing": we call something, it does its work, then returns a result

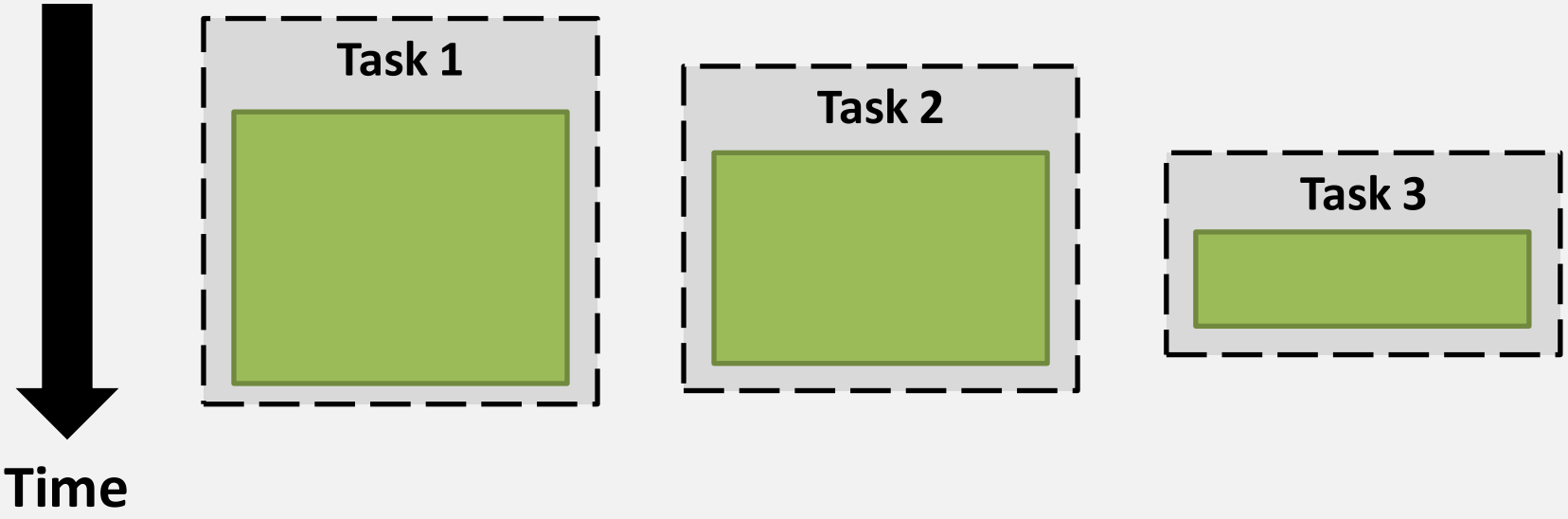
With asynchronous, we call something, it sets the work in motion, and returns





Parallelism

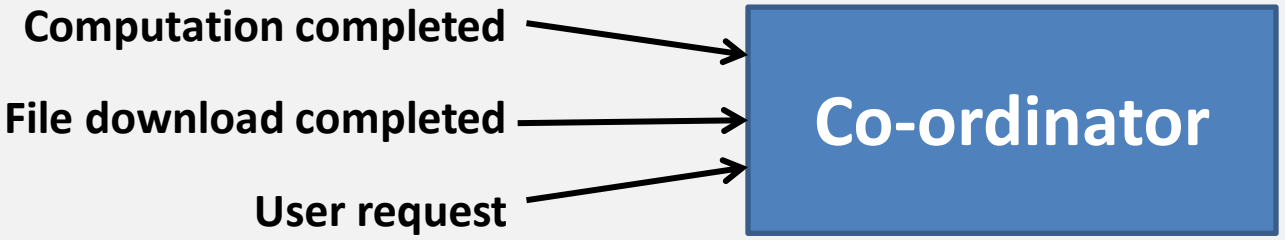
Break a problem into pieces we can do at the same time



This enables us to exploit multi-core CPUs to solve the problem faster

Concurrency

About coping with events arising whenever then please, and trying to do "the right thing"



Arises when we have asynchrony and/or parallelism at work in a system

May also arise naturally in some domains, which are inherently concurrent

Composability

It's hard to precisely define composability, at least, without resorting to category theory.

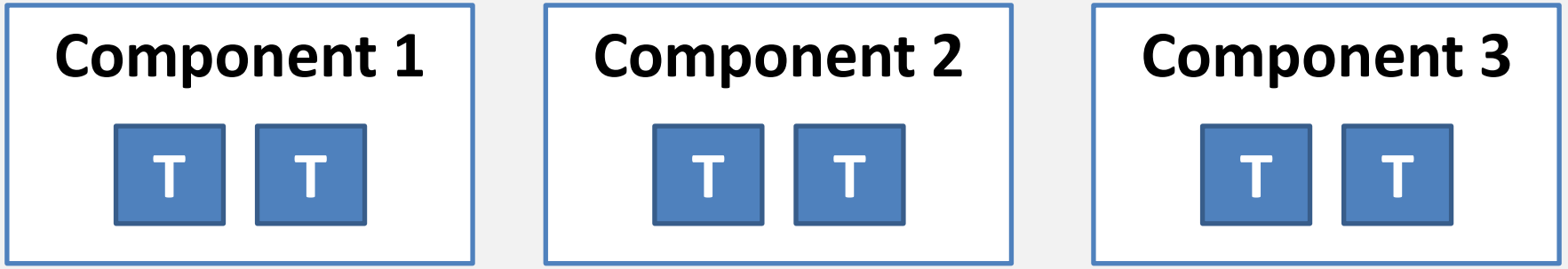
Informally, two things compose when you can put them together and they work "as expected"

Simple example: things like `map` and `grep`

```
my @chosen = @beers.grep(*.volume >= 5)\
                   .grep(*.style eq 'ale')\
                   .map(*.name);
```

Direct thread use: uncomposable

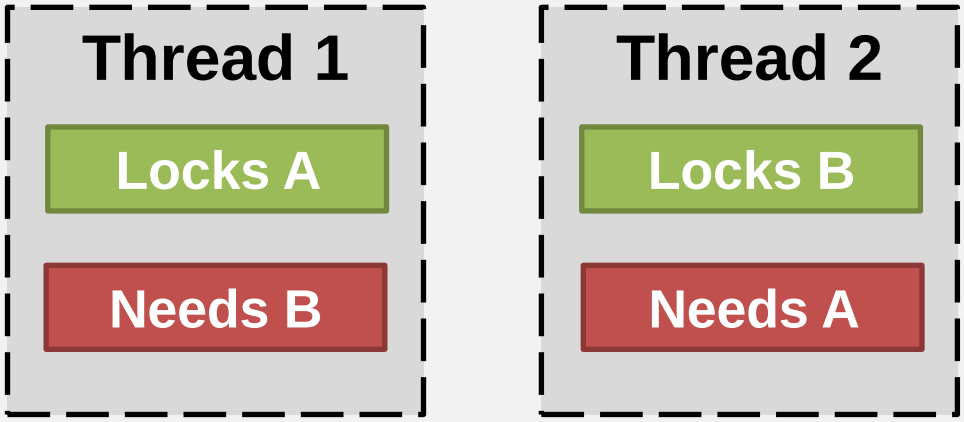
What if every component in a system that wants to do some work in parallel or asynchronously starts threads to do it?



Soon, dozens of threads! No common way to handle their failures, prioritize, etc.

Locks: uncomposable

Taking two operations that use locks and work individually , and running them together, may run into deadlock!



This circular waiting is all too easy to fall into, especially as a system grows

Callbacks: barely composable

So we asynchronously read in a file, passing two callbacks for success and failure:

```
slurp_async($filename,
    sub ($content) {
        # Do stuff
    },
    sub ($error) {
        # Handle it
    });
```

But how do we further process the outcome of these? The result is nested callbacks – the goto of asynchronous programming ☹

Factor out the synchronization

As well as being composable, combinators like `map`, `grep` and `uniq` factor out both state and flow control form our own code

For dealing with asynchrony, we'd like to define combinators that:

Are composable

and

Factor out synchronization

Promises

`Promise.start` takes a code block and schedules it to run asynchronously (on another thread)

```
my $p10000 = Promise.start({
    (1..Inf).grep(*.is-prime)[9999]
})
```

It returns a `Promise` object, which represents the ongoing piece of work

We don't spawn a thread per promise. Rather, a scheduler spreads them over a pool of threads.

The `start` function

This is a sufficiently common thing to want to do that there's a shortcut, which actually does nothing more than call `Promise.start`:

```
my $p10000 = start {
    (1..Inf).grep(*.is-prime)[9999]
}
```

Note that you should always do something with the `Promise` that comes back from a `start` block, or you risk missing errors! (Aside: we may do something with promises in sink context.)

A promise is kept or broken

If the code inside of the `start` block completes successfully, then the promise is kept. If it instead throws an exception, the promise is broken.

This can be inspected with `status`:

```
my $p10000 = start {
    (1..Inf).grep(*.is-prime)[9999]
}
say $p10000.status;  # Planned (probably)
# some time later
say $p10000.status;  # Kept
```

then

So how do we say what to do after a promise has been kept or broken? Using `then` - which returns a promise representing the combined work:

```
my $p10000 = start {
    (1..Inf).grep(*.is-prime)[9999]
}
my $base16 = $p10000.then(sub ($res) {
    $res.result.base(16)
});
my $pwrite = $base16.then(sub ($res) {
    spurt 'p10000.txt', $res.result;
    return 'p10000.txt';
});
```

Already beats callbacks!

When you call `then` on a promise, it gives you back another promise - just like `map` takes a list and gives you back another one

Furthermore, just as you can store the result of a `map` in an array and base multiple future computations off of it, you can call `then` multiple times on the same promise to base multiple future asynchronous computations off its work

Much nicer! ☺

Back to synchrony

So, `then` makes it easy for us to follow up one piece of asynchronous work with another. But what if we really want to block on a promise having a result? In this case, just call `result` or use `await` (which may take many promises):

```
say $pwrite.result; # p10000.txt
say await $pwrite;  # p10000.txt
```

If any exception is thrown by any of the steps along the way, the promise is broken and calling `result` or `await` will throw the exception.

Not just for CPU-bound

A Promise doesn't just have to represent a piece of CPU-bound work. For example, you can create a Promise kept after a delay:

```
my @a = (1..20).pick(*);
await @a.map(-> $n {
    Promise.in($n).then({ say $n })
})
```

Or after a file was read in:

```
my $data = IO::Async::File.new(path => $p).slurp;
```

Combinators: allof, anyof

`Promise.allof` and `Promise.anyof` produce a promise that is kept when all or any of the specified promises are kept.

Put this together with `in`, and we have a timeout mechanism of sorts (but lacking cancellation):

```
await Promise.anyof($p10000, Promise.in(5));
say $p10000.status == Kept
    ?? $p10000.result
    !! 'Timed out';
```

Make your own promises

The built-in promise makers aren't particularly special. In fact, you can put *anything* that will later produce a value or exception behind a promise

Simply create a new `Promise`…

```
my $p = Promise.new;
```

…and then call `keep` or `break` some point later:

```
$p.keep($value);
```

Example: `nth_or_timeout` (1)

Making our own `Promise` objects is useful for implementing new combinators

Our timeout mechanism earlier sucked because the computation continued even after the timeout

We'd prefer to have just written:

```
say await nth_or_timeout(
    (1..Inf).grep(*.is-prime),
    2000,
    10);
```

Example: nth_or_timeout (2)

```
sub nth_or_timeout(@source, $n, $timeout) {
    my $p = Promise.new;
    my $t = Promise.in($timeout);
    ...
    $p
}
```

Example: nth_or_timeout (3)

```
sub nth_or_timeout(@source, $n, $timeout) {
    my $p = Promise.new;
    my $t = Promise.in($timeout);
    start {
        my $result;
        ...
        $p.keep($result);
    }
    $p
}
```

Example: nth_or_timeout (4)

```
sub nth_or_timeout(@source, $n, $timeout) {
    my $p = Promise.new;
    my $t = Promise.in($timeout);
    start {
        my $result;
        for ^$n {
            if $t.status == Kept {
                $p.break('Timed out');
                last;
            }
            $result = @source[$n];
        }
        $p.keep($result);
    }
    $p
}
```

Only I may keep or break…

The built-in promise makers go a step further, and get exclusive access to keep or break the promise, using a Vow. You can also do this:

```
# Get/store the keeper
my $p = Promise.new;
my $k = $p.vow;

# Some time later...
$k.keep($result);
# ...or...
$k.break($exception);
```

The scheduler

If you look at the inside of `Promise`, you will find various bits of synchronization logic, but nothing that introduces actual asynchrony

That is the role of a scheduler

The current default scheduler is located through the `$*SCHEDULER` dynamic variable

It defaults to `ThreadPoolScheduler`, which schedules work over a pool of threads

Example: Promise.start

With all we've covered so far, you can now understand how `Promise.start` is implemented:

```
method start(Promise:U: &code,
                :$scheduler = $*SCHEDULER) {
    my $p   = Promise.new(:$scheduler);
    my $vow = $p.vow;
    $scheduler.cue(
        { $vow.keep(code()) },
        :catch(-> $ex { $vow.break($ex) }) );
    $p
}
```

Channels

Provide a thread-safe synchronization mechanism based around a queue

A channel is created like this:

```
my $c = Channel.new;
```

One or more threads can send:

```
$c.send($result);
```

Meanwhile, one or more threads can receive:

```
my $val = $c.receive;
```

Producer / Consumer

Channels are a great fit in scenarios where something is producing values for something else to consume and process

Since many parallel workers may produce or consume at each point, they lend themselves well to scaling each stage as needed

The stages are each single-threaded on the inside, meaning there's little synchronization to care for

Example: parse all the configs

Take a bunch of INI files, read each one, parse them, collect all the config into a single hash

Reading the files puts contents into a channel

Parsing receives, parses, and sends a hash of the parsed output on another channel

A combiner brings them together

<code>

The weakness of channels

If promises are about synchronizing scalar results, are channels about synchronizing a list of results as they become available?

While channels are great in producer/consumer scenarios, they differ from promises:

Unlike `then`, you can only `receive` once ☹

Additionally, `receive` is blocking; heavy use of channels means lots of sync/async boundaries ☹

"But you showed us…"

Those at my first Perl 6 concurrency session, given at YAPC::Europe 2013, will remember some examples showing building channel combinators.

You can still do that, but when you start doing it a lot, and for fine-grained things, it leads to a lot of thread pool threads sitting blocked on receive.

We could indeed make that case yield, but that means taking a continuation, which we'll struggle to make perform on all platforms. So…

Supplies

`Supplies` are an alternative mechanism for dealing with streams of items being produced over time

Unlike channels, they are push-based. It's a little bit like having a `Promise` whose `then` method fires multiple times with different values.

Instead of `then`, we call it `more`

A simple example

It is possible to directly create a `Supply`:

```
my $s = Supply.new;
```

We can then `tap` it, specifying something to do every time a value is supplied:

```
$s.tap(&say);
```

The tappers can then be notified:

```
$s.more('cow bell');
```

Push, not pull

We push the values out to the subscribers

No asynchrony unless you ask for it

However, once you start working asynchronously then you stay asynchronous

Can do fine-grained things with the stream of values without the blocking `receive` that channels give, so it can be much more efficient

Supply.interval

While using a `Supply` directly doesn't introduce any asynchrony (though of course, you can put it in a `start` to do so), some `Supply` factories do

For example, tapping a `Supply.interval` supplies an incrementing integer at a regular interval. This is done asynchronously.

```
Supply.interval(1).tap(&say);
```

Familiar combinators

It turns out that we can define lots of familiar combinators from list-y things on supplies.

```
my $interval = Supply.interval(5);
```

Familiar combinators

It turns out that we can define lots of familiar combinators from list-y things on supplies.

```
my $interval = Supply.interval(5);
my $slurped  = $interval.map({ slurp('foo.ini') });
```

Familiar combinators

It turns out that we can define lots of familiar combinators from list-y things on supplies.

```
my $interval = Supply.interval(5);
my $slurped  = $interval.map({ slurp('foo.ini') });
my $changed  = $slurped.grep({
    state $last = '';
    LEAVE $last = $_;
    $last ne $_
});
```

Familiar combinators

It turns out that we can define lots of familiar combinators from list-y things on supplies.

```
my $interval = Supply.interval(5);
my $slurped  = $interval.map({ slurp('foo.ini') });
my $changed  = $slurped.grep({
    state $last = '';
    LEAVE $last = $_;
    $last ne $_
});
$changed.tap(&say);
```

A more real example

A recent example from my work involved a conveyor belt of agricultural product (maybe wheat) having moisture content readings taken

We read them from a sensor mounted above a belt once per second

Every 5 seconds (in this example - longer in reality) we take a sample of the product for a closer analysis

Modelling the belt/samples scenario

```
my $seconds  = Supply.interval(1);
```

Modelling the belt/samples scenario

```
my $seconds  = Supply.interval(1);
my $belt_raw = $seconds.map({ rand xx 100 });
```

Modelling the belt/samples scenario

```
my $seconds  = Supply.interval(1);
my $belt_raw = $seconds.map({ rand xx 100 });
my $belt_avg = $belt_raw.map(sub (@values) {
    ([+] @values) / @values
});
```

Modelling the belt/samples scenario

```
my $seconds  = Supply.interval(1);
my $belt_raw = $seconds.map({ rand xx 100 });
my $belt_avg = $belt_raw.map(sub (@values) {
    ([+] @values) / @values
});
my $belt_label = $belt_avg.map({ "Belt: $_" });
```

Modelling the belt/samples scenario

```
my $seconds  = Supply.interval(1);
my $belt_raw = $seconds.map({ rand xx 100 });
my $belt_avg = $belt_raw.map(sub (@values) {
    ([+] @values) / @values
});
my $belt_label = $belt_avg.map({ "Belt: $_" });

my $samples = Supply.interval(5).map({ rand });
my $samples_label = $samples.map(
                          { "Sample: $_" });
```

Modelling the belt/samples scenario

```
my $seconds  = Supply.interval(1);
my $belt_raw = $seconds.map({ rand xx 100 });
my $belt_avg = $belt_raw.map(sub (@values) {
    ([+] @values) / @values
});
my $belt_label = $belt_avg.map({ "Belt: $_" });

my $samples = Supply.interval(5).map({ rand });
my $samples_label = $samples.map(
                          { "Sample: $_" });

my $merged = $belt_label.merge($samples_label);
```

Modelling the belt/samples scenario

```
my $seconds  = Supply.interval(1);
my $belt_raw = $seconds.map({ rand xx 100 });
my $belt_avg = $belt_raw.map(sub (@values) {
    ([+] @values) / @values
});
my $belt_label = $belt_avg.map({ "Belt: $_" });

my $samples = Supply.interval(5).map({ rand });
my $samples_label = $samples.map(
                          { "Sample: $_" });

my $merged = $belt_label.merge($samples_label);
$merged.tap(&say);
```

The on meta-combinator

Implementing combinators like `merge` is non-trivial because different threads may be sending the values - forcing us to deal with synchronization

The on combinator exists to help build other combinators involving multiple subscriptions, factoring out the synchronization

<study merge, zip>

Back to synchrony

All this asynchrony is great, but what if you somewhere need to return to the world of synchronous programming?

Any supply can be coerced to a `Channel` (sending each value)

It can also be coerced to a lazy `list`, so you can iterate over the results (for example, just using a `for` loop) - but of course, you block!

Composing composable paradigms

In general, the goal is to provide paradigms for working asynchronously that have good composability properties

Furthermore, each of those paradigms (promises, channels, supplies) should be composable with each other

Channel ⇔ Supply
Promise ⇔ Supply
Promise ⇔ Channel

Closing thoughts

Perl 6 has a big opportunity to support asynchronous and parallel programming well

Already, using Rakudo on JVM, you can put multiple CPU cores to use in your Perl 6 programs

For those of you not wanting to work on the JVM, we'll provide support for these features in Rakudo on MoarVM (target: Q1 2014)

Try it out, provide feedback, have fun ☺

Thank you!

Questions?

Blog: 6guts.wordpress.com
Twitter: @jnthnwrthngtn
Email: jnthn@jnthn.net