The Art of

Concurrent Scripting

with Raku

by Brian Duggan

bduggan

FOSDEM 2024

outline

- Motivation
- Concurrency in Raku
- From Bash to Raku
- Thinking Concurrently

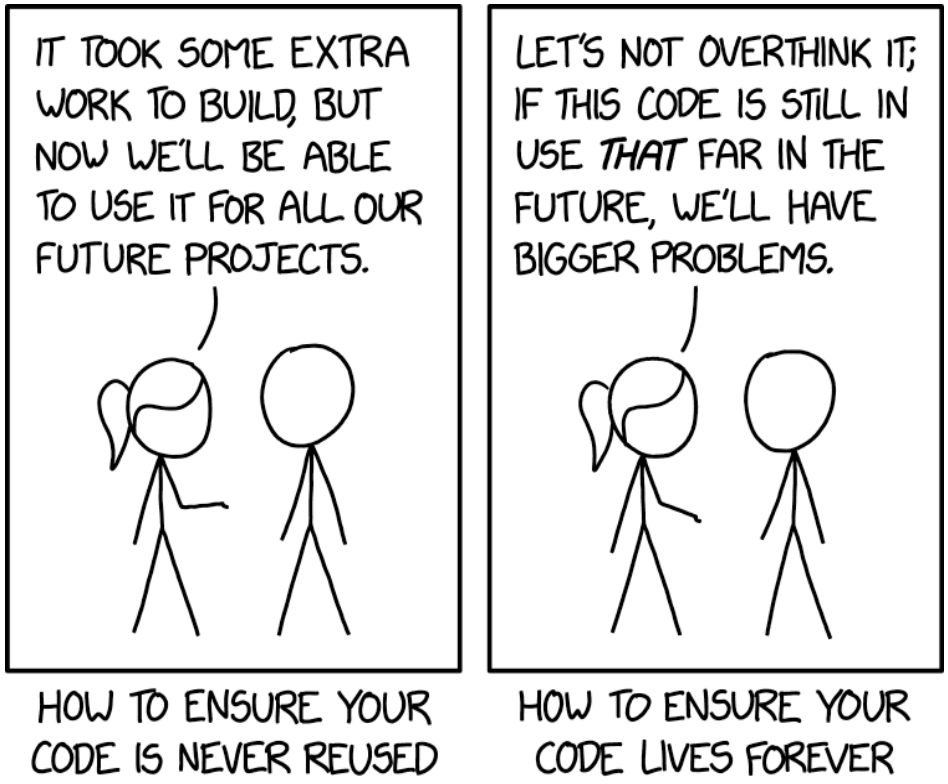
Motivation

Motivation

Shell Scripts

should ...

- be easy to write quickly
- have few or no dependencies
- be easy to understand
- not require tons of maintenance
- be reliable in case they last for a long time



Motivation

Shell Scripts

Seen often

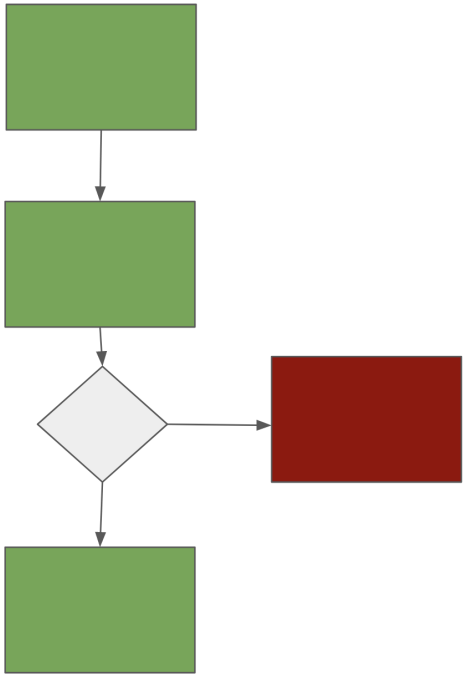
- Run commands, check exit statuses
- Simple control flow; loops, if-then
- stdin, stdout, stderr, redirects
- Atomic write-and-rename

Seen sometimes

- Lock files, pid files for concurrency control
- Parallel execution (wait)
- Receiving signals (trap), sending signals (kill)
- Timing out commands (timeout)
- Progress indicators

Seen rarely or never

- Message queues
- Event loops
- Async/await
- Threads for concurrency
- Shared memory
- Mutexes



motivation

Common Scripting Assumptions

- scripts are just doing some thing
- no need for fancy programming techniques
- Concurrency techniques are for programming not scripting
- With scripting, "real" languages are not appropriate. bash is enough!
- The world is **not that complicated**

Reality

- Scripts can do more
- Easy things are hard in bash
- There is a great language for scripting with concurrency
- Scripting languages are a limiting factor
- The world is that complicated

motivation

Common Scripting Assumptions

Mondriaanmode door Yves Saint Laurent (1966)

Reality

Pieter Brueghel the Younger - The Kermesse of St George

Better languages for scripting can help deal with reality.

Concurrency in Raku

Concurrency, Asynchrony, and Parallelism

Definitions

- Parallelism "choosing to do multiple things at once"
- Asynchrony "reacting to things that will happen"
- Concurrency "competition to access and mutate some shared resource"

See this talk by Jonathan Worthington

Raku

- was designed to support all three
- does not impose a one paradigm;
- like events, threading, message-passing, or actors
- provides tools, not rules

Raku provides tools to

- avoid race conditions
- avoid data contention
- choose your own paradigm, or mix and match
- use concurrency wisely

Getting started

Let's make a race condition!

```
start say "hello";
say "world";
```

```
hello
world
```

```
world
hello
```

```
world
```

Use `start` to schedule code for execution (in a separate thread).

The return type is a `Promise`.

Let's avoid this race condition!

```
await start say "hello";
say "world";
```

```
hello
world
```

Concurrency in Raku

from docs.raku.org/language/concurrency :

High level APIs

- **Promises :** represent execution that may not yet have completed.
- **Channels :** are one-to-one message queues.
- **Supplies :** are one-to-many message queues.
- **Proc::Async :** represents an external processes.

Low level APIs

- **Threads :** An OS thread of execution
- **Locks :** Allow synchronization across threads
- **atomic types :** atomic ints, native 32 or 64-bit ints
- **atomic operations :** fetch + increment/decrement/add/assign, compare-atomic-swap (CAS)
- **Scheduler :** Manages concurrent execution ($*SCHEDULER by default is a ThreadPoolScheduler)

Some built-in event sources:

- **IO::Notification** - file system changes
- **IO::Socket::Async** -- tcp or udp sockets
- **Supply.interval** - time changing
- **IO::Pipe** -- UNIX pipes (stdout, stderr)

Other async/concurrent-ish things

- **race** and **hyper** can schedule parallel execution
- Phasers run things out of order (more on that later)

From Bash to Raku

Turn any bash script into Raku by using "shell"

```
#!/bin/bash
```

```
echo "starting database dump!"
date
pg_dump bigdb -f bigdb.dump
date
echo "done!"
```

```
starting database dump!
```

```
#!/usr/bin/env raku
```

```
shell <<echo "starting database dump!">>;
shell "date";
shell 'pg_dump bigdb -f bigdb.dump';
shell q:to/BASH/;
date
echo "done!"
BASH
```

```
starting database dump!
```

Raku supports single quotes, double quotes, word quoting (with nested quotes), heredocs and more.

Easy things are easy

```
#!/bin/bash
```

```
echo "starting database dump!"
date
pg_dump bigdb -f bigdb.dump
date
echo "done!"
```

```
starting database dump!
```

```
#!/usr/bin/env raku
```

```
say "starting database dump!";
shell 'pg_dump bigdb -f bigdb.dump';
say now - INIT now;
say "done!"
```

```
starting database dump!
10.004250187
done
```

Code following INIT runs during the initialization phase; "now - INIT now" is the number of seconds that have passed since the program started.

INIT is a phaser. Other phasers: BEGIN, CHECK, END, ENTER, LEAVE

LEAVE is equivalent to "deferred execution" in Go.

Can we watch the seconds in real time?

```
say "starting database dump!";
my $clock = Supply.interval(1);
my $timer = $clock.tap: { .say }
shell 'pg_dump bigdb -f bigdb.dump';
$timer.close;
say "done!";
```

```
starting database dump!
1
2
3
4
5
6
7
done!
```

Use **Supply.interval(1)** to create an on-demand supply that emits a new value every 1 second. Add a **Tap** to the supply, with **tap.** Then use **close** to close the tap.

Can we watch the seconds in real time?

```
say "starting database dump!";
my $clock = Supply.interval(1).map: { .polymod(60).reverse.fmt('%02d',':'); }
my $timer = $clock.tap: { print "\r" ~ $^time }
shell 'pg_dump bigdb -f bigdb.dump';
$timer.close;
say "done!";
```

```
starting database dump!
00:07
```

You can use **map** on supplies (or lists, arrays, sequences or other iterables).

The **polymod** method returns a sequence of successive div/mod operations (mod 60, then div 60, etc). **fmt** uses printf strings to format numbers. **print** prints without a newline.

Can we do this for all shell commands?

```
my $clock = Supply.interval(1).map: { .polymod(60).reverse.fmt('%02d',':'); }

&shell.wrap: -> $cmd {
  my $timer = $clock.tap: { print "\r$cmd ... [$^time]" }
  callsame;
  $timer.close;
  say "$cmd ... done!";
}

shell 'pg_dump bigdb -f bigdb.dump';
```

```
pg_dump bigdb -f bigdb.dump ... 00:07
```

Use **wrap** to wrap a function in another one ("decorators" in python), and **callsame** to dispatch to the original.

timeouts

Run a command that might need to be stopped.

```
#!/bin/bash
timeout 1 host example.com || \
echo "DNS seems okay!"
```

```
#!/usr/bin/env raku
await Promise.anyof(
   start { shell <<host example.com>> },
   start sleep 1
)
```

Note the shell command will continue after the Raku program exits.

We want to send a TERM signal to it.

Use **start** to make a **Promise.**

Use **Promise.anyof** to make a promise that resolves when any one of several promises resolve.

Use **await** to wait for a promise to resolve. Note! there is no async, only **await!**

timeouts

Run a command that might need to be stopped: better way!

```
my $timeout = Promise.in(1);
my $proc = Proc::Async.new(<<host example.com>>);
await Promise.anyof($proc.start,$timeout);
$proc.kill( SIGTERM ) if $timeout;
```

Use **Promise.in(1)** to make a promise that resolves one second later.

Create a **Proc::Async** object, and call **start** to spawn the process, and **kill** to send a signal.

Thinking Concurrently

react-whenever vs taps

These are equivalent :

```
$supply.tap: -> $event {
  say $event
}
```

```
start react whenever $supply -> $event {
  say $event
}
```

Use **react** to make an event loop, and then add taps with **whenever.** And **start** schedules it in another thread.

example: generate HTML from markdown

Watch a directory run md2html when a file ending in ".md" is changed.

```
my $supply = $*CWD.watch.grep({ .path.ends-with('md')
$supply.tap: {
  shell "md2html {.path} > {.path}.html"
}
sleep;
```

```
my $supply = $*CWD.watch.grep({ .path.ends-with('md')
react whenever $supply {
  shell "md2html {.path} > {.path}.html"
}
```

Without **sleep** the main thread exits.

Use $*CWD to get the current working directory.

Call **watch** on an **IO::Path** object to generate a **Supply** that emits **IO::Notification** events.

Using **react** plus **whenever** is equivalent to adding a **Tap** to a Supply.

Without **start** it will block.

Example 2: calculate the median ping time

```
$ ping google.com
PING google.com (142.250.65.238): 56 data bytes
64 bytes from 142.250.65.238: icmp_seq=0 ttl=118 time=9.407 ms
64 bytes from 142.250.65.238: icmp_seq=1 ttl=118 time=6.956 ms
64 bytes from 142.250.65.238: icmp_seq=2 ttl=118 time=8.537 ms
64 bytes from 142.250.65.238: icmp_seq=3 ttl=118 time=8.535 ms
64 bytes from 142.250.65.238: icmp_seq=4 ttl=118 time=10.714 ms
^C
--- google.com ping statistics ---
5 packets transmitted, 5 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 6.956/8.830/10.714/1.230 ms
```

Let's write a script to...

- start a ping process
- stop if it is interrupted or after 10 seconds
- keep track of the times in the output, and
- print the median time (missing from the stats above)

Then,

- make a little graph with the times

React to multiple events

```
my $proc = Proc::Async.new(<<ping google.com>>, :stdout);
LEAVE $proc.kill;
my $timeout = Promise.in(10);
my @times;

react {
  whenever $timeout { done; }
  whenever $proc.stdout { /time '=' (.*) ms / and do { @times.push($0); say "$0" } }
  whenever signal(SIGINT) { done; }
  whenever $proc.start { say "ping finished" }
}

say "median ping time: " ~ @times.sort[ @times.elems div 2 ] ~ " ms";
```

```
$ ./pinggoogle.raku
15.902
15.512
15.340
^Cmedian ping time: 15.512 ms
```

Use **signal** to make a Supply and react to signals.

Note that the **@times** array is being mutated by another thread!

Locks

What if we had multiple hosts?

```
my @procs = @hosts.map: { Proc::Async.new: <<ping $^host>> }
```

Then this would be unsafe

```
      /time '=' <time> / and @times.push($0)
```

We could use a lock to protect the access to this shared data structure.

```
my $lock = Lock.new;
```

and then

```
      /time '=' <time> / and $lock.protect: { @times.push($0) }
```

A **Lock** is a low-level construct that blocks other threads. See also **Lock::Async** for a lighter-weight lock. But, in this case, another option is to use a Channel.

React to multiple events

Let's write multiping!

```
$ ./multiping.raku -h
Usage:
  ./multiping.raku [<hosts> ...]

$ ./multiping.raku google.com google.co.uk google.be
    google.com: 6.877 ******
  google.co.uk: 7.340 *******
     google.be: 7.243 *******
    google.com: 7.143 *******
  google.co.uk: 7.357 *******
     google.be: 7.146 *******
    google.com: 8.399 ********
     google.be: 6.995 ******
  google.co.uk: 7.186 *******
    google.com: 8.222 ********
     google.be: 9.567 *********
 google.co.uk: 10.485 **********
    google.com: 6.373 ******
  google.co.uk: 7.533 *******
     google.be: 7.293 *******
    google.com: 6.446 ******
  google.co.uk: 7.320 *******
     google.be: 7.011 *******
   google.com: 16.386 ****************
    google.be: 14.021 **************
 google.co.uk: 14.052 **************
    google.com: 6.332 ******
     google.be: 7.813 *******
  google.co.uk: 7.834 *******
```

React to multiple events

multiping.raku

```
#!/usr/bin/env raku
unit sub MAIN(*@hosts);
my $channel = Channel.new;

start loop {
  given $channel.receive -> % ( :$host, :$time )  {
    say "$host: $time ".fmt('%25s') ~ ("*" x ($time.Int));
  }
}

my @procs = @hosts.map: { Proc::Async.new: <<ping $^host>> }
my regex time { <[0..9.]>+ }
react {
  for @procs Z, @hosts -> ($proc,$host) {
    whenever $proc {
      /time '=' <time> / and $channel.send: %( :$host, :$<time> );
    }
    whenever $proc.start { }
  }
}
```

Make a channel.

Receive, destructure and process data.

Spawn external processes.

Construct data and send.

React to multiple events

```
$ ./multiping.raku -h
Usage:
   ./multiping.raku [<hosts> ...]

$ ./multiping.raku google.com google.co.uk google.be
     google.com: 6.877 ******
   google.co.uk: 7.340 *******
      google.be: 7.243 *******
     google.com: 7.143 *******
   google.co.uk: 7.357 *******
      google.be: 7.146 *******
     google.com: 8.399 ********
      google.be: 6.995 ******
   google.co.uk: 7.186 *******
     google.com: 8.222 ********
      google.be: 9.567 *********
  google.co.uk: 10.485 **********
     google.com: 6.373 ******
   google.co.uk: 7.533 *******
      google.be: 7.293 *******
     google.com: 6.446 ******
   google.co.uk: 7.320 *******
      google.be: 7.011 *******
    google.com: 16.386 ****************
     google.be: 14.021 **************
  google.co.uk: 14.052 **************
     google.com: 6.332 ******
      google.be: 7.813 *******
   google.co.uk: 7.834 *******
```

How about pg_multidump?

Let's write a script to dump multiple databases at the same time.

```
race for (1..10).race(batch => 1, degree => 10) { sleep 1 }
say now - INIT now;
```

```
1.125074767
```

```
#!/usr/bin/env raku
unit sub MAIN(*@databases);
&shell.wrap: { say "starting $^cmd"; callsame; say "done with $cmd" }
race for @databases.race(batch => 1, degree => 10) {
  shell "pg_dump $^db > $db.sql";
}
```

```
./pg_multidump.raku one two three
starting pg_dump one > one.sql
starting pg_dump two > two.sql
starting pg_dump three > three.sql
done with pg_dump one > one.sql
done with pg_dump three > three.sql
done with pg_dump two > two.sql
```

Call the method **race** on a sequence to turn it into a **HyperSeq**. Then use the statement prefix to parallelize execution.

Conclusions

Using concurrency in Raku is fun and easy, and is a practical way to write versatile scripts.

We have seen examples of

- tracking progress of a command in another thread
- timing out a command using a Promise
- using asynchronous techniques to respond to filesystem events
- using asynchronous techniques to respond to lines emitted from a command
- instant parallelism -- spawning multiple processes at once and running them in batches
- using locks (mutexes) to manage concurrency

For further reading, check out

- ecosystem modules **OO::Actors** and **OO::Monitors** for nice ways to encapsulate concurrency in classes
- other modules in the **Concurrent::** namespace on https://raku.land
- The raku docs -- https://docs.raku.org/language/concurrency -- which has many more examples.

Thank You!