What the heck are Coroutines?

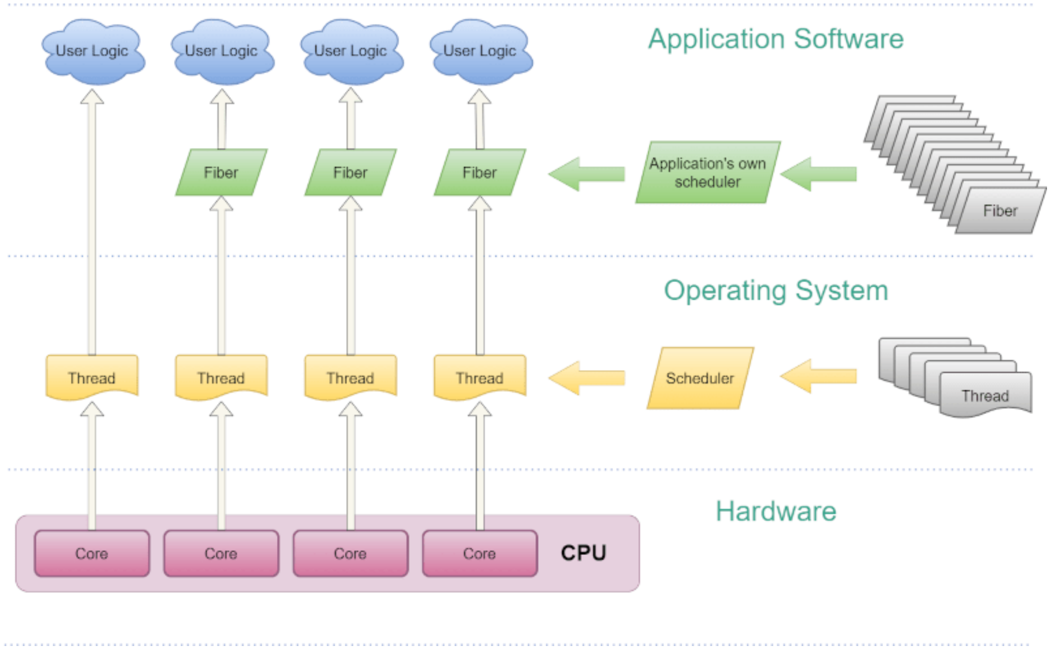
Delphi Dev Days 2024
Paul McGee

Computing Hierarchy

System Land
- CPU
- Cores

OS
- Processes
- Threads

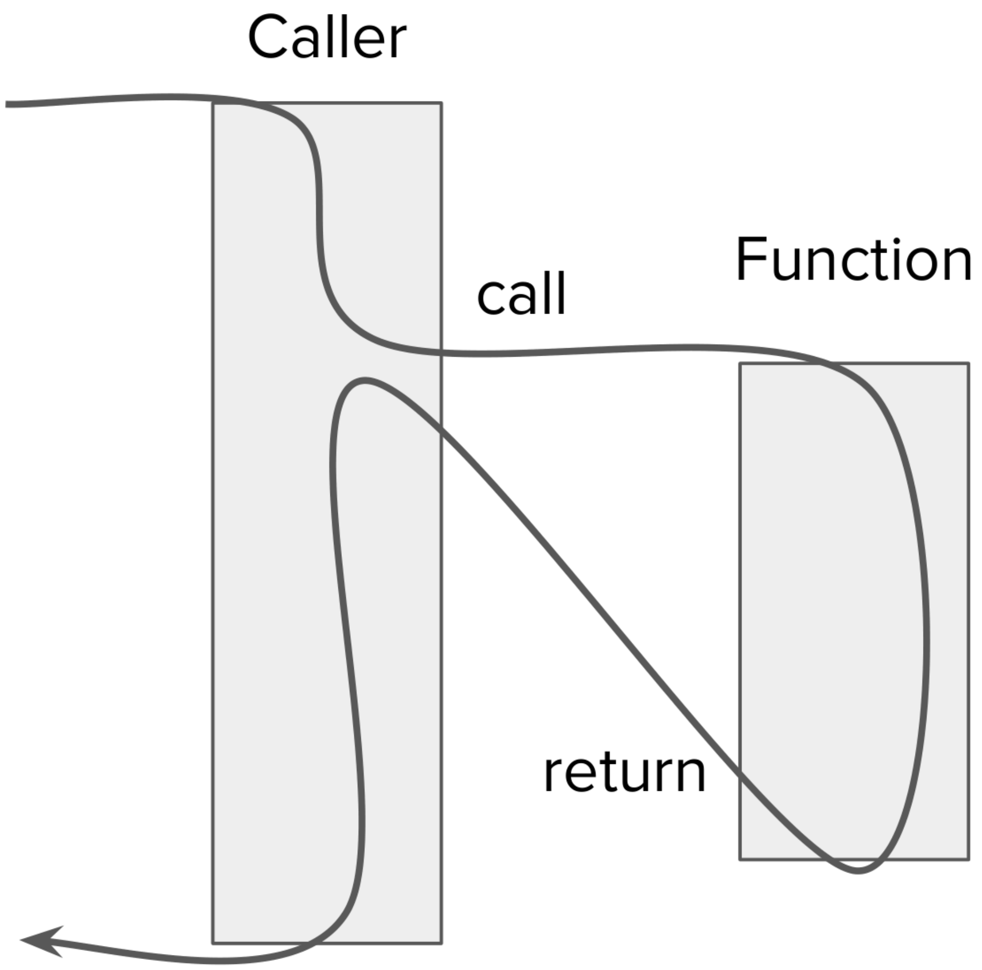
User Land
- Fibers
- Coroutines



https://hackaday.com/2023/09/25/processes-threads-and-fibers/

How do functions work?

- The stack is prepared for the call
- Execution jumps to the function
- The function completes, with a value
- The stack contains the return address
- Execution returns to the caller



Toby Allsopp—CppCon 2017

How can functions be generalised?

- CPU/GPU Threads - We can initiate computation on other threads, and join results back into the main thread.

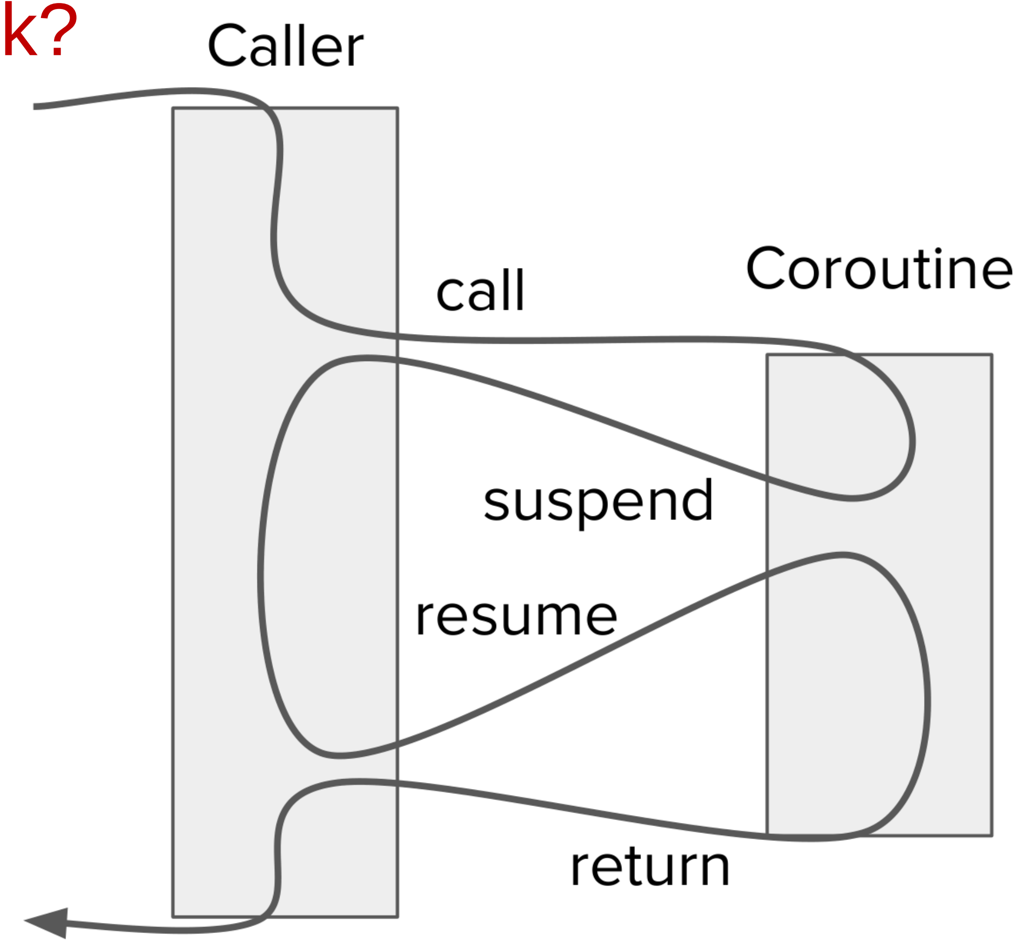
- Recursion - functions that can call themselves without destroying the existing execution state.

- Continuation – (Related to 1st Class functions)
  Functions can exit as data objects, and be used
  as input to, and output from, other functions

- Coroutines – functions that can be Interrupted and Resumed

How do resumable functions work?

- The stack is prepared for the call
- Execution jumps to the function
- The function suspends, maybe with a value
- The stack and program counter are saved
- Execution returns to the caller
- On next entry, the stack and PC are restored, and the function resumes



Toby Allsopp—CppCon 2017

Computing Speed

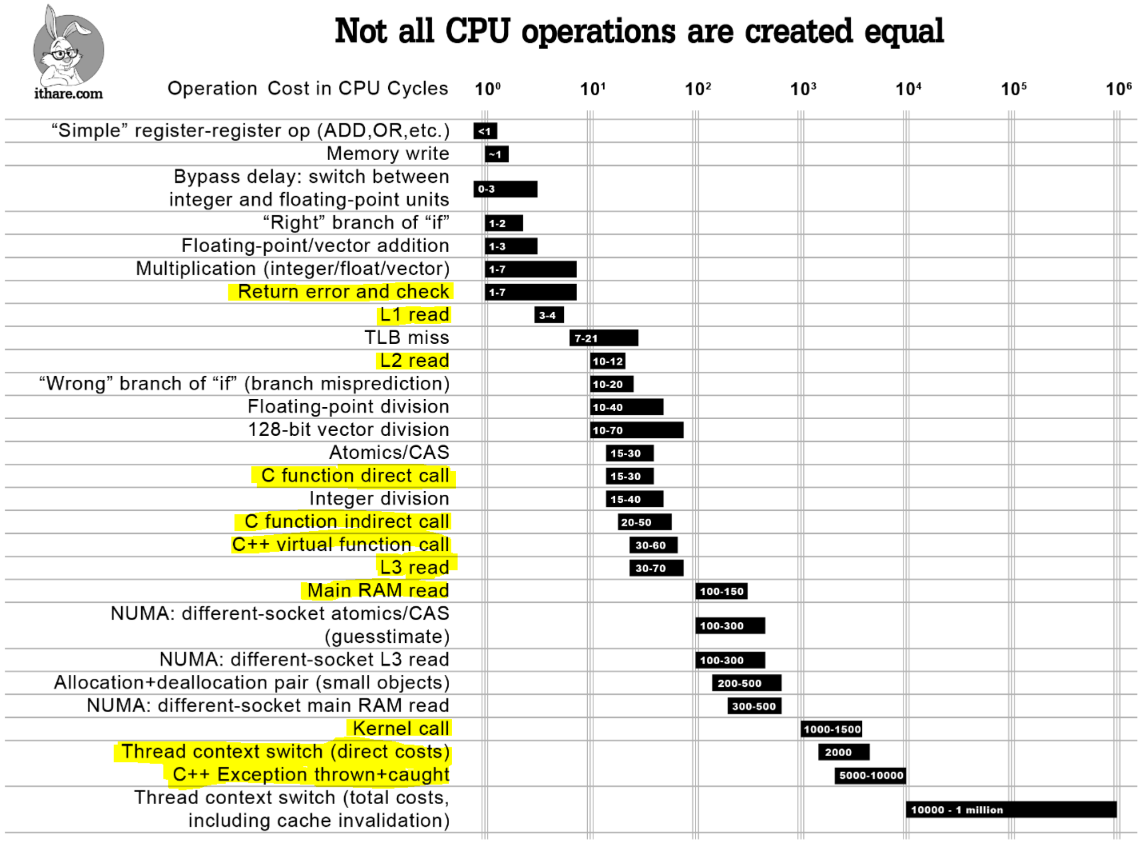
User Land

- Fibers
- Coroutines

Calling Coroutines is on the order of magnitude of a function call.

Thread stack size might be in the order of 2-5 Mb. Limit: maybe 10,000 ?

Coroutine stack size might be 64 Kb. Limit: maybe 1,000,000 ?



http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/

Computing Speed

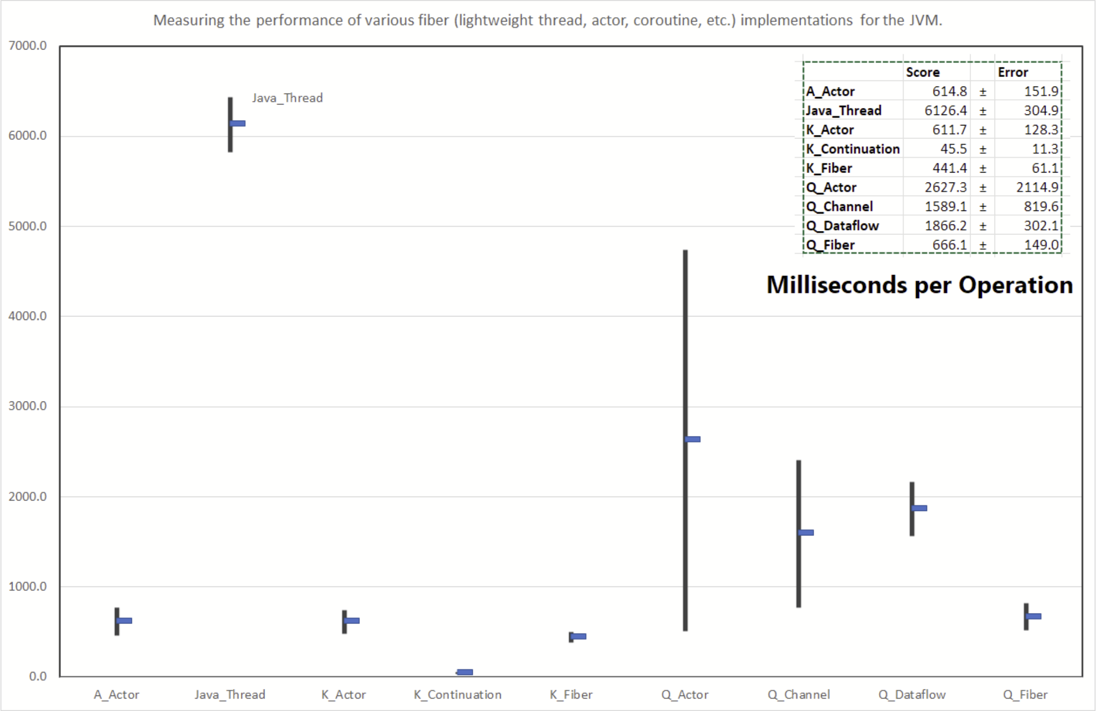
User Land

- Fibers
- Coroutines

Calling Coroutines is on the order of magnitude of a function call.

Thread stack size might be in the order of 2-5 Mb. Limit: maybe 10,000 ?

Coroutine stack size might be 64 Kb. Limit: maybe 1,000,000 ?



https://github.com/vy/fiber-test

Coroutine Characteristics

Resumable functions

- Generalise 'normal' functions
- Retain local state
- Suspend and Resume operation
- Single-threaded control flow
- Cooperative multitasking
- Yield/Return back to Caller, or another specified Coroutine
- Share common ground with State Machines

- May resume on a different thread than started on

Coroutine Jargon

Asymmetric

- Yield/Return back to Caller

Symmetric

- May yield to another specified coroutine

Stackful

- Maintain their own stack
- Support nested and recursive coroutines

Stackless

- Store state on the heap
- Non-recursive
- More compatible with Structured Concurrency

Coroutine Uses

Can be used to implement

- Iterators/Generators (IEnumerable)
- Lazy datatypes
- Task switching
- Actors
- Design clarity – hiding persistent state, separable code
- Exceptions/Error flow
- Pipes

Coroutine History

- 1958-1963 Melvin Conway coins the term, working with COBOL

Design of a Separable Transition-Diagram Compiler*

Melvin E. Conway
Directorate of Computers, USAF
L. G. Hanscom Field, Bedford, Mass.

A COBOL compiler design is presented which is compact enough to permit rapid, one-pass compilation of a large subset of COBOL on a moderately large computer. Versions of the same compiler for smaller machines require only two working tapes plus a compiler tape. The methods given are largely applicable to the construction of ALGOL compilers.

Introduction

This paper is written in rebuttal of three propositions widely held among compiler writers, to wit: (1) syntax-directed compilers [1] suffer practical disadvantages over other types of compilers, chiefly in speed; (2) compilers should be written with compilers; (3) Cobol [2] compilers must be complicated. The form of the rebuttal is to describe a high-speed, one-pass, syntax-directed Cobol compiler which can be built by two people with an assembler in less than a year.

Coroutines and Separable Programs

That property of the design which makes it amenable to many segment configurations is its *separability*. A program organization is separable if it is broken up into processing modules which communicate with each other according to the following restrictions: (1) the only communication between modules is in the form of discrete items of information; (2) the flow of each of these items is along fixed, one-way paths; (3) the entire program can be laid

Coroutine History

- 1980 Christopher Marlin

WIKIPEDIA
The Free Encyclopedia

There is no single precise definition of coroutine. In 1980 Christopher D. Marlin[4] summarized two widely-acknowledged fundamental characteristics of a coroutine:

1. the values of data local to a coroutine persist between successive calls;
2. the execution of a coroutine is suspended as control leaves it, only to carry on where it left off when control re-enters the coroutine at some later stage.

Coroutine PASCAL

Coroutine PASCAL[91,92] is the result of the addition of a Simula-like coroutine facility to Pascal. The definition of this facility, presented in terms of the Control Definition and Implementation Language (CDIL), is based on Wang and Dahl's description[152] of a set of primitives "resembling" those of Simula. Wang and Dahl's model, which

[91] M.J.Lemon, "Coroutine PASCAL: A Case Study in Separable Control", Technical Report No. 76-13, Dept. of Computer Science, University of Pittsburgh, Pittsburgh, Pennsylvania (December 1976).

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

95

Christopher D. Marlin

Coroutines

A Programming Methodology, a Language Design and an Implementation

Springer-Verlag
Berlin Heidelberg New York 1980

Coroutines in Languages

- 1967 Simula
- 1978 Modula-2
- 1992 Turbo Pascal 7.0 (with µThreads library)
  https://github.com/nickelsworth/blaise/blob/master/src/fundamentals3/microthreads/cMicroThreads.pas
- 2003 Lua
- 2005 C# 2.0 iterator/yield
- 2006 Python 2.5
- 2007 Go
- 2009 C++ Boost.Coroutine
- 2011 D
- 2011 Kotlin
- 2012 C# 5.0 async/await
- 2017 Java Project Loom begins
- 2020 C++20
- 2022 Java 19 Loom = preview feature

How did I understand coroutines?

I had trouble thinking about implementing coroutines, over quite a period of time.

The first thing I imagined was hacking into stack-frames, storing data to the heap.

I made an approximating step …

How did I understand coroutines?

I made a poor approximation …

```
{$APPTYPE CONSOLE}
program ContinuationExample;

type
  TContinuation = procedure;
        TState = (first, second);
var
  CurrentContinuation : TContinuation;
              state : TState;


procedure CaptureContinuation;
begin
  case state of
       first :
           begin
              writeln('Capturing continuation...');
              { Capture the current state by setting CurrentContinuation to the current procedure }
              CurrentContinuation := @CaptureContinuation;
           end;
       second :
           writeln('Back to First procedure');
  end;
end;


procedure ResumeContinuation;
begin
  writeln('Resuming continuation...');
  { Call the captured continuation to jump back to the captured state }
  CurrentContinuation;
end;


procedure ExampleCoroutine;
begin
  writeln('Start');   state := first;   CaptureContinuation;   writeln;
  writeln('Middle');  state := second;  ResumeContinuation;    writeln;
  writeln('End');
end;

begin
  writeln('Before coroutine');
  ExampleCoroutine;
  writeln('After coroutine');
  readln;
end.
```

Accurate? Useful?

No.

Maybe it could behave like a python generator.

But it was definitely not elegant.

```python
def square_generator(n):
    """
    A generator function that yields the squares of numbers up to n
    """
    i = 0
    while i <= n:
        yield i**2
        i += 1

# Using the generator to print the first 10 squares
for num in square_generator(10):
    print(num)


```

You won't believe what happened next !

That was in : January 2024

Then in : April 2024

You won't believe what happened next !

This C code was posted as a dig on Twitter.

Look closely at this chaos. 🙂

The Moisrex
@the_moisrex

You just ruined switch-statements for me.

```
struct range_struct
{
    int i;
    int stop;
    int step;
    enum {
        At_start,
        In_loop,
        Done
    } state;

    explicit range_struct(int stop, int step = 1)
        : stop(stop), step(step), state(At_start)
    {
    }

    int resume()
    {
    switch (state)
    {
    case At_start:  for (i = 0; i < stop;)
                    {
                        state = In_loop;
                        return i;
    case In_loop:       i += step;
                    }
                    state = Done;
    case Done:      return 0;
    }
    }
};
```

ALT

1:23 AM · Apr 7, 2024 · **249.1K** Views

90 217 1.9K 699

You won't believe …

Look closely at this chaos. 🙂

I thought it was interesting,
just to figure out what it was saying.

THEN

I wondered … could we maybe do the same thing in Delphi ??

We have GOTO. We have Labels.
(I had to look up how they work)

You just ruined switch-statements for me.

```
struct range_struct
{
    int i;
    int stop;
    int step;
    enum {
        At_start,
        In_loop,
        Done
    } state;

    explicit range_struct(int stop, int step = 1)
        : stop(stop), step(step), state(At_start)
    {
    }

    int resume()
    {
    switch (state)
    {
    case At_start:  for (i = 0; i < stop;)
                    {
                        state = In_loop;
                        return i;
    case In_loop:       i += step;
                    }
                    state = Done;
    case Done:      return 0;
    }
    }
};
```

ALT

We can ! And it is quite elegant. … And it uses GOTO !!

```
{$APPTYPE CONSOLE}
program Project1;

type
  range_struct = record
    type
      Tstate = (at_start, in_loop, done);
    var
      stop, step,
         i, start : integer;
      state       : Tstate;

    function    resume()  : integer;
    constructor λ ( _start,
                    _stop : integer;
                    _step : integer = 1);
  end;

constructor range_struct.λ( _start, _stop, _step:integer);
begin
    start := _start;
    stop  := _stop;
    step  := _step;
    state := at_start;
end;

function range_struct.resume() : integer;
label α,β,δ;
begin
         case state of
              at_start : goto α;
              in_loop  : goto β;
              done     : goto δ;
         end;

   α :   i:=start;
         repeat
             begin
                 state := in_loop;
                 exit(i);
   β :           i     := i + step;
             end
         until i >= stop;
         state := done;
   δ :   exit(0);
end;

begin
   var ρ := range_struct.λ(1,20,2);
   var π := range_struct.λ(2,20,2);

   for var i in [1..5] do begin
       writeln('a   ',i:2, 'th call : ', ρ.resume);
       writeln('   b ',i:2, 'th call : ', π.resume);
   end;

   writeln;
   writeln('Mwah ha ha. GOTO is back, baby!');
   readln;
end.
```

Output :

a... 1th call : 1
...b 1th call : 2
a... 2th call : 3
...b 2th call : 4
a... 3th call : 5
...b 3th call : 6
a... 4th call : 7
...b 4th call : 8
a... 5th call : 9
...b 5th call : 10

Mwah ha ha. GOTO is back, baby!

The twist ending …

Initially I was just hyped to have fun with translating the C code.

Then I realised … this code gives off big coroutine vibes.

THEN I found out why!

The same exact code !!

- C++ Coroutines Do Not Spark Joy - by Malte Skarupke
  https://probablydance.com/2021/10/31/c-coroutines-do-not-spark-joy/
- Coroutines in C – by Simon Tatham
  https://www.chiark.greenend.org.uk/~sgtatham/coroutines.html
- The Art of Computer Programming: Fundamental Algorithms, Volume 1 – by Donald Knuth

So where does that take us?

- Example 1983 – 'Coroutines in Pascal' Run length encoding

```
      CD3E6
      F64A7P4A.

produces the printout

      CDE  EEE  FFF  FFF  F44
      444  44A  PPP  PPP  PPA
      AAA  A
```

```
procedure  NEXT(var  CH: Char);        { Flush blanks and <CR>'s }

   begin  repeat  Read(CH)  until  (CH <> ' ')  end;

procedure  INCHAR;

   begin
   case  XX  of
      AA:  begin
           NEXT(CH);
           if  CH = '.'  then  DONE := True;
           if  CH in ['0'..'9']  then
              begin
              REPL := Ord(CH) - 48;      { Note:  Ord('0') = 48 }
              if  REPL > 0  then  XX := BB;
              NEXT(CH);
              if  CH = '.'  then              { Save final digit. }
                 begin  CH := Chr(REPL + 48);  DONE := True  end
              end
           end;     { AA }
      BB:  begin
           REPL := REPL - 1;  if  REPL <= 0  then  XX := AA
           end    { BB }
      end    { case }
   end;     { INCHAR }

procedure  OUTCHAR;

   begin
   case  YY  of
      CC:  begin  Write(PRINTER, CH);  YY := DD  end;
      DD:  begin  Write(PRINTER, CH);  YY := EE  end;
      EE:  begin  Write(PRINTER, CH);
           BLOCKNO := (BLOCKNO + 1) mod 5;
           if  BLOCKNO = 0  then  Writeln(PRINTER)
           else  Write(PRINTER, '  ');
           YY := CC
           end
      end    { case }
   end;     { OUTCHAR }
```

So where does that take us?

- Conway's Game of Life
- Localising state – "Fortified functions"
- Spring4D coroutines
- C++20 coroutines
- Python
- Go 1.23 – release notes this week

So where does that take us?

- Conway's Game of Life
- Localising state – "Fortified functions"
- Spring4D coroutines
- C++20 coroutines
- Python
- Go 1.23 – release notes this week

So where does that take us?

- Conway's Game of Life
- Localising state – "Fortified functions"
- Spring4D coroutines
- C++20 coroutines
- Python
- Go 1.23 – release notes this week

So where does that take us?

- Conway's Game of Life
- Localising state – "Fortified functions"
- Spring4D coroutines
- C++20 coroutines
- Python
- Go 1.23 – release notes this week

So where does that take us?

- Conway's Game of Life
- Localising state – "Fortified functions"
- Spring4D coroutines
- C++20 coroutines
- Python
- Go 1.23 – release notes this week

So where does that take us?

- Conway's Game of Life
- Localising state – "Fortified functions"
- Spring4D coroutines
- C++20 coroutines
- Python
- Go 1.23 – release notes this week

Finally

- Also: Fibers, AIO, Isaac Pinheiro,
- Coroutines, Fibers, & Effects – ADUG Forum
  https://forums.adug.org.au/t/coroutines-fibers-and-effects/59752
- Code repo - Github
- Referenced material – Watch this space
- Past & Future Talks?
    - Category Theory for Delphians
    - Nullable & Optional Types
    - The Sex Life of Functions
    - Async/Await in C# and Javascript
    - Functional Programming (See Dr Kevin Bond's talk TODAY)