ė - Goal and Overview

The goal of this project is to program computers using pluggable units of software.

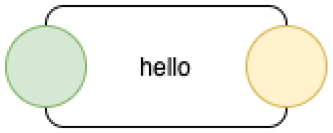
To do this we need:

- micro-concurrency
- 0D
- layers, relative sofware units
- multiple languages.

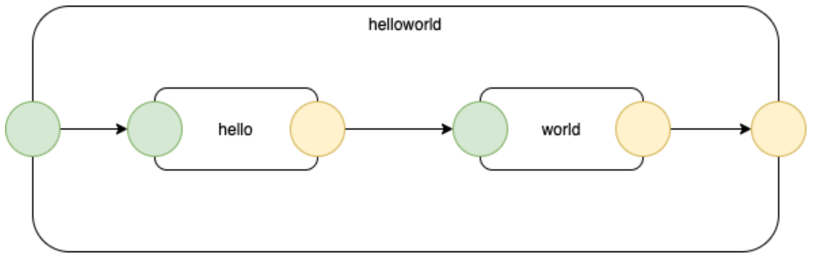
Hello World

Very simple example

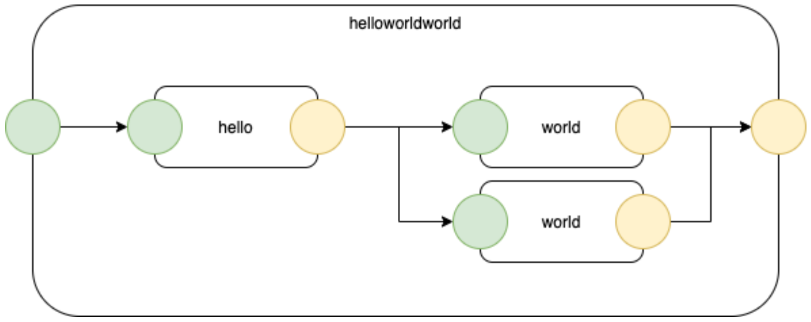
Leaf



Container



Re-Architecting



Scalability

Pluggability is necessary for scalability, but, more elaborate (complicated) examples would be needed.

Benefits

- anti-bloatware
- technical drawings come "for free"
- concurrency comes "for free"
- "build and forget" development
- distributed programming comes "for free"
- multiple-CPU paradigm
- ability to plug together software components to create mimimal set of functionality

further discussion...Eh - Benefits

Key Insights

- 0D - No Dependencies
- FIFOs and LIFOs
- Pipelines
- Structured Message Passing
- "First Principles Thinking"
- Closures
- "Parallelism" is composed of more than one concept

FIFOs vs LIFOs

- LIFOs are implemented in most modern programming languages (i.e. function call and call-stack)

- LIFOs make implementing Operating Systems and pluggable components more difficult than necessary
- both LIFOs and FIFOs are useful, but only LIFOs are inherently supported by modern programming languages
    - e.g. function calling employs a LIFO, and does not employ a FIFO
    - FIFOs *can* be modeled as classes, but that is not the same as being inherently supported by the notation / programming language

Pipelines

Pipelines are useful for plumbing software units together, but, functions calling functions do not implement pipelines (due to the LIFO issue).

Structured Message Passing

structured message passing.png

First Principles Thinking

WIP/First Principles Thinking

Various Random Issues

My notes about Operating Systems, in screencast form: https://beta-share.descript.com/view/gdfwt4MfKjF

- divide-and-conquer - developers vs. end-users
- divide-and-conquer - 2 main technical aspects of operating systems
- bloatware
- processes

Divide and Conquer

- divide and conquer is understood by most programmers
- but, most programmers do not employ *enough* divide-and-conquer
    - e.g. what is commonly call "parallelism" can be broken into 2 categories
        1. 0D
        2. Simultaneity

Closures

- Operating Systems are Greenspun's 10th Rule applied to Closures

Parallelism: Redux

1. 0D
2. Simultaneity

Further Discussion

Eh - Key Insights

How Is This Different From What We Already Have?

State-of-the-art operating systems, like Linux, Windows, MacOS, have two (2) main functions:

1. control multitasking and blocking
2. provide a rich set of device drivers.

Wrapper operating systems cannot control blocking when the programming units perform ad-hoc, unstructured blocking on their own. "Functions" *block* when calling other functions. State-of-the-art Operating Systems need to resort to brute-force methods to pry blocking away from programming units, e.g. preemption.

This project does not *directly* address the issue of providing a rich set of device drivers (2), since, each device driver represents "hard work" - specific knowledge about the internal workings of each device, where each device might have wildly different operations. Programming such devices is made harder by the over-use of synchronization, etc. As such, writing device drivers in a stand-alone style should be "easier". Incorporating already-existing device drivers is discussed below.

Status

This project is essentially about joining the pieces of the project into a coherent whole.

- Most of the pieces are on the floor (see sub-projects) and getting simpler with each iteration.
- Currently working on Python version of Leaf components using HSMs (hierarchical state machines), trying to simplify HSM P.O.C.

Github

eh

Future

- Containers
- Examples
- DaS

    - draw.io, SVG
- using existing O/Ss
- using existing Software
- emerging technologies
    - Robotics
    - Games - NGPs, Actors
- anti-bloatware using Lambda wrappers
    - Lambdas, λs, are stack-oriented (LIFO), trying to use λs for non-stack-oriented programs causes bloat
    - functions that call other functions perform ad-hoc blocking, which O/Ss must fight to maintain control of blocking (e.g. using preemption)

Sub-Projects

0D - No Dependencies

- use FIFOs, don't use function calls for component-to-component messaging
- deferred message sending
    - don't deliver message immediately, just stick it in an output queue
    - parent wrapper component ("Container") walks output queue of Child and routes messages to other children (or to its own output queue)

FIFOs and LIFOs

- building FIFOs is a no-brainer in most modern languages

Pipelines

- UNIX is an example of pipelining put to good use
    - UNIX uses "lines" (terminated with newline) as the substrate for communication
    - this is too complicated
        - restricts UNIX tools to processing text
    - use bytes instead of lines
        - include "tag" string/byte in message
- Communicating HSMs will be much smaller units of software than full-blown UNIX commands

Structured Message Passing

0D makes it possible to choose how to structure a system composed of Components.

Structured Message Passing is like Structured Programming. Nothing new added to the toolbox.

Good design principles based on what we already know, e.g. ORG Charts in business.

Don't colour outside of the lines.

Contain, contain, contain.

"First Principles Thinking"

Survey of current problems.

Survey summarized in screencast (above).

Conclusions

- over-synchronization
    - it's difficult to do some simple things (like multitasking), because of presence of pervasive synchronization
- FIFOs not LIFOs
- O/S processes are just clumsy, big closures with FIFOs for input (often called "mailboxes")
- FIFOs for output are often implemented under-the-hood by Operating Systems
    - could be easily implemented using Classes (don't need Operating Systems to handle output FIFOs for apps)

Closures

No-brainer using most modern programming languages.

Diagrams - DaS (Diagrams as Syntax)

- done at least once with Prolog+Lisp+Haskell
    - theory: could be much simpler using Ohm-JS
    - previous version used backtracking to figure out (inference) containment relationships
    - theory: SVG grouping can embed containment relationships directly in the data without further inferencing
    - containment relationships are:
        - nested boxes
        - ports belonging to boxes
    - connection relationships are:
        - line begins at element
        - line ends at element
- i.e. containment and connection represent slightly more information than what is alreadycontained in program text files now
    - i.e. easy to implement, once you know that you want these relationships

    - ASCII Art for "box" is `{...}`
    - the "global variable problem" exists because containment is not strictly adhered-to
        - ASCII-Art for box doesn't make it "as obvious" that a variable is not contained in a box, or crosses the edges of a box
- draw.io creates compressed .drawio files which contain enough info for parsing
- SVG contains almost enough info
    - rectangles
    - ellipses
    - lines
    - text
    - grouping
    - missing: source and target elements for lines
        - draw.io outputs source and target information
- Ohm-JS can parse .drawio files
    - Ohm-JS is based on PEG, includes backtracking and easy parsing
- currently blocked on creating a JS program to uncompress .drawio format
    - just a technicality, done more than once in the past
    - needs to be made stand-alone or into a library function (probably 1-line in JS)

Hierarchical State Machines

Use hierarchy to conquer the "state explosion problem".

First suggested by Harel in original StateChart paper Harel StateCharts

Parent overrides Child, which is opposite to what is considered to be *class inheritance* in programming.

If a Parent changes state, it exits all Child states (recursively, to any depth).

A Child cannot change the behaviour of a Parent, hence, each unit of code (a diagram or text) "make sense" in a stand-alone manner, hence, no hidden dependencies.

Operating System *processes* are State Machines. HSM results in State Machines that are smaller than Operating System state machines.

Approach

Formulate questions

Questions such as...

- Why do hardware designs tend to work while software designs fail and become more complicated?

- Pipelines are different from functions. How are pipelines different?
- Message passing. Is message-passing possible in the synchronous paradigm?
- Message passing - asynchronous - has a bad rep because it is often ad-hoc. Is there an equivalent to "structured programming" for async message-passing?
- Closures - are closures the same as "processes" in operating systems?
- DaS - Diagrams as Syntax. Why are most programming languages textual?
- Tells - what is currently considered difficult? Multitasking, async, callbacks, mutation, sequencing, history, state ... Can these be improved? Are they difficult because they're difficult or because our notation makes them difficult?
- Is "something" the same across all programming languages?
- What is parallelism?
- Operating Systems and Programming Languages were invented in the 1950's under the single-cpu assumption. Is the single-cpu assumption still valid?
- Are end-users forced to use the "same" operating systems / computer environments as developers?
- Can the goal (pluggable components) be sub-divided into smaller sub-goals?
  - Which properties must components have to be pluggable?
  - Which properties inhibit pluggability?

Synthesize

- upon answering the above questions, it is possible to synthesize a new programming environment?

Discussion

micro-concurrency
0D
2022-07-11-0D
2022-08-21-Layers

References

Harel StateCharts
call/return spaghetti
name
- no Greek
- no ASCII
Dependency essay
Functions vs. dependencies article
Movable Feast Machine not related, but, indication of "steam engine time" - similar issues being addressed differently
2022-07-10-Layered Programming - New-Breed Structured Programming
2022-05-30-Fire and Forget
2022-06-04-Fire and Forget

Tools / POCs

Tools and Proofs of Concept that are related to this project but not part of the 1-week Jam

- https://github.com/guitarvydas/das2json
  - transpiles diagram testbench.drawio to JSON
  - transpiles diagram helloworld.drawio to JSON
  - uses hybrid diagram elements containing Python code
    - boxes, connections, ports written as a diagram
    - "Leaf" code written as Python (embedded in the diagrams)
    - the diagrams are hybrids - boxes + ports + lines + Python code
      - i.e. micro-concurrency written in diagram form, the rest written as regular Python code
      - do the minimum necessary, and let Python do the rest of the heavy lifting
  - uses PROLOG inferencing to create a factbase, including containment and connection relationships
  - could be simplified, probably with Ohm-JS
- https://github.com/guitarvydas/das
  - similar to above, except slightly more ambitious
  - transpiles helloworld.drawio to Python, then runs the Python
  - transpiles d2py.drawio to Python, then runs the Python
    - d2py.drawio is a "script" for self-compiling the das-to-python transpiler to Python
    - the resulting Python script calls "make" for building the transpiler
    - the diagram shows one way to trap errors and quit
    - the UX (the diagram language syntax) could be more humane - at the moment, it consists of boxes and lines which might look "too complicated" to non-programmers (thought: maybe rely on box containment to "inherit" messages, eliding some of the ports and lines)
- https://github.com/guitarvydas/vsh

    - checkout afa53b8
    - contains very old (pre-2013) Visual SHell experiment
    - diagram compiler written in itself (using yEd editor)
    - generates .gsh files
    - .gsh is *grash* a minimal (approx 8 instruction) assembler for invoking UNIX C system functions (e.g. dup2(), exec(), etc.)
    - (I ran out of time during this Jam to make this work again)
- pipelines
    - see UNIX shells
    - see Aho's paper on using pipeline syntax as a replacement for Denotational Semantics syntax
    - (I ran out of time to dig up this reference during this Jam)
- micro-concurrency
    - partially based on implementing Harel StateCharts
    - partially based on Holt's book Concurrent Euclid, UNIX, Tunis
    - partially based on designing hardware using state machine diagrams
    - "sequencing" and "concurrency" is "easy" with state diagrams, less easy with functional notation
- conclusions about 0D and "parallelism"
    - 0D
- HSM - Hierarchical State Machines
    - HSM P.O.C.

Notes to Self

2022-08-21-Notes on Eh